Virtual University of Pakistan
Federal Government University

Advance Operating System

Course - Code: CS703

Chapter	Subject	Lectures	Page
1	Introduction and Basic Concepts	01-03	02
2	Process and Threads	04-22	20
3	Memory Management	22-28	118
4	File Systems	29-34	152
5	Input / Output	35-38	170
4	Security	39-45	188

Presented by:
Mr. Farhan Zaidi
MS Computer Science
University of Southern California, USA

Reference Book 1:
Modern Operating Systems by Tanenbaum

Download Link:
http://www.vumultan.com/Books/CS703-Modern-Operating-Systems.pdf

Reference Book 2:
Operating Systems Concepts by Siberschatz, Galvin, Gagne

Download Link:
http://www.vumultan.com/Books/Operating-Systems-Concepts.pdf

A World Class Education at Your Door Step

ورچوئل یونیورسٹی آف پاکستان

Lecture 1:

Overview of today's lecture

- Course Objectives and Pre-requisites
- Introduction to what an operating system is?
- Issues involved in the design of an operating system
- Different types of operating systems
- Re-cap of the lecture

Course objectives

- In depth treatment of topics from design perspective.
- Consider why things are done in a certain way instead of just how they are done.
- Consider design alternatives.
- Be able to tell the difference between a good design and a bad design.
- Also consider engineering tradeoffs involved in the design process.
- Practical aspects from a computer scientist's perspective.

These include:

1. A case study throughout the course of a real commercial operating system kernel. We will present Linux kernel for this purpose. We will also make comparisons to other operating systems e.g. Windows where applicable.
2. Hands on training as a programmer on using operating system services in relatively complex programs. I will assign 4 to 5 programming assignments for this purpose.

Research perspective

- We will also look at some contemporary and classical research on operating system topics in the research literature. For this purpose, I will assign a few readings from the literature on operating systems.

Course Pre-requisites

- C/C++ programming. This is an essential pre-requisite since without this; you won't be able to do the assignments.
- An undergraduate first course on data structures. This should include implementation of elementary data structures e.g. lists, stack, queues, trees etc. in a high level language like C or C++ etc.
- A first course on operating systems would be helpful but not strictly required since we will cover the basics of each topic before diving into the deeper stuff.

What is an operating system?

Top-down view

- Provides an extended or virtual machine abstraction to user programs
- Easier to program than the underlying hardware.
- All services are invoked and accomplished through system calls.

Bottom-up view

- Acts as a resource manager of a complex system
- Resources consist of processors, memories, timers, disks, mice, keyboard, network interfaces, printers etc.
- OS manages allocation of these resources to user programs in an orderly and controlled manner

Resource multiplexing

- OS multiplexes resources in two ways:
- In time, In space
- Time multiplexing involves different programs taking turns in using the resource. Example: CPU scheduling, printer sharing.
- Space multiplexing involves different program getting part of the resource possibly at the same time. Example: memory is divided into several running programs.

The major OS issues

- Structure: how is the OS organized?
- Sharing: how are resources shared across users?
- Naming: how are resources named (by users or programs)?
- Security: how is the integrity of the OS and its resources ensured?
- Protection: how is one user/program protected from another?
- Performance: how do we make it all go fast?
- Reliability: what happens if something goes wrong (either with hardware or with program)?
- Extensibility: can we add new features?
- Communication: how do programs exchange information, including across a network?

More OS issues

- Concurrency: how are parallel activates (computation and I/O created and controlled?
- Scale: what happens as demands or resources increase?
- Persistence: how do you make data last longer than program executions?
- Distribution: how do multiple computers interact with each other?
- Accounting: how do we keep track of resources usage, and perhaps charge for it?

Protection and security as an example

- None
- OS from my program
- Your program from my program
- My program from my program
- Access by intruding individuals
- Access by intruding programs
- Denial of service
- Distributed denial of service
- Spoofing
- Spam
- Worms
- Viruses
- Stuff you download and run knowingly (bugs, Trojan horses)
- Stuff you download and run unknowingly (cookies, spyware)

Type of Operating Systems

- Main frame operating systems
  - ✓ Huge amounts of I/O activity.
  - ✓ 1000s of disks not unusual provide batch,
  - ✓ Transaction and time sharing services.
  - ✓ Batch processing is routine non-interactive jobs. e.g. claims processing, sales reporting etc. Transaction processing systems handle large number of small requests e.g. check processing in banks, air line reservations etc.
  - ✓ Time-sharing systems allow multiple remote users to run jobs at the same time e.g. querying a database. OS Optimized for these tasks. Example is OS/390
- Server operating systems
  - ✓ Run on very large PCs, workstations or even main-frames. They serve multiple users over a network simultaneously and allow them to share hardware and software. Examples are web servers, database transaction servers etc. Examples of OS in this class are Win2K, XP and flavors of UNIX.
- Multiprocessor operating systems
  - ✓ OS is basically a variation to server operating systems with special provisions for connectivity and communication management between different CPUs.
- PC operating systems
  - ✓ OS provides a nice interface to a single user.
  - ✓ Typically used for word processing, spread sheets, Internet access etc.
- Real-time operating systems
  - ✓ Characterized by time as the key parameter. Real-time response to internal and external events is more important than any other design goal. Classified into two sub-categories: Hard and Soft real-time.
  - ✓ Example applications include Industrial process control, robotics and assembly lines, air traffic control, network routers and telecommunication switches, multi-media systems etc.
- Embedded operating systems
  - ✓ Embedded in small devices e.g. palm-top computers e.g. PDA, TV sets, micro-wave ovens, mobile phones. They have characteristics of real-time systems (mainly soft real-time) but also have restraints on power consumption, memory usage etc.
  - ✓ Examples include PalmOS and Windows CE. Height of this type is smart-card systems.

Distributed Systems

Distribute the computation among several physical processors.

- Loosely coupled system
  - ✓ Each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.
- Advantages of distributed systems:
  - ✓ Resources Sharing, Computation speed up – load sharing, Reliability, Communications

Parallel Systems

Multiprocessor systems with more than on CPU in close communication.

- Tightly coupled system
  - ✓ Processors share memory and a clock; communication usually takes place through the shared memory.
- Advantages of parallel system
  - ✓ Increased throughput, Economical, Increased reliability, graceful degradation, fail-soft systems

Lecture 2

Overview of today's lecture

- Major components of an operating system
- Structure and internal architecture of an operating system
- Monolithic Vs Micro-kernels
- Virtual Machine Monitors
- Re-cap of the lecture

Major OS Components

- Process management
- Memory management
- I/O
- Secondary Storage
- File System
- Protection
- Accounting
- Shell (OS UI)
- GUI
- Networking

Process Operation

- The OS provides the following kinds operations on processes (i.e. process abstraction interface)
  - ✓ Create a process
  - ✓ Delete a process
  - ✓ Suspend a process
  - ✓ Resume a process
  - ✓ Clone a process
  - ✓ Inter-process communication
  - ✓ Inter-process synchronization
  - ✓ Create / delete a child process

I/O

- A Big Chunk Of OS Kernel deals with I/O
  Millions of Lines in windows XP (including drivers)
- The OS provides standard interface between programs and devices
- Device drivers are the routines that interact with specific device types:
  Encapsulates device specific knowledge
  - ✓ E.g. how to initialize a device, how to request the I/O, how to handle interrupts and errors
  - ✓ E.g. SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers.
- Note: windows has ~35000 device drivers.

Secondary Storage

- Secondary storage (disk, tape) is persistent memory
  - ✓ Often magnetic media survives power failures (hopefully)
- Routines that interact with disks are typically at a very low level in the OS Used by many components
  - ✓ Handle scheduling of disk operations, head movement,
  - ✓ Error handling and often management of space on disk

- Usually independent of file system
  - ✓ Although there may be cooperation
  - ✓ File system knowledge of device details can help optimize performance. E.g. place related files close together on disk

File System

- Secondary storage device are crude and awkward. E.g. write 4096 byte block to a sector
- File system are convenient abstraction
- A file is a basic long term storage unit
- A directory is just a special kind of file

Command interpreter (shell)

- A particular program that handles the interpretation of users commands and helps to manage processes
- On some systems, command interpreter may be a standard part of the OS
- On others, its just not privileged code that provides an interface to the user
- On others there may be no command language

File system operations

- The file system interface defines standard operations
  - ✓ File (or directory) creation and deletion
  - ✓ Manipulating of files and directories
  - ✓ Copy
  - ✓ Lock
- File system also provide higher level services
  - ✓ Accounting and quotes
  - ✓ Backup
  - ✓ Indexing or search
  - ✓ File versioning

Accounting

- Keeps track of resource usage
- Both to enforce quotas ¯you‘re over the disk limit‖ Or to produce bills
- Important for time shared computers like mainframes

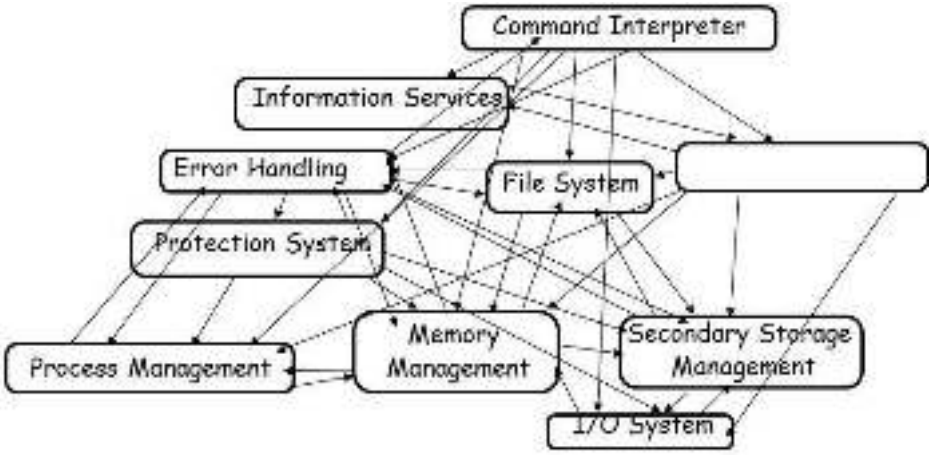
Networking

An OS typically has a built-in communication infra-structure that implements:

a. A network protocol software stack
b. A route lookup module to map a given destination address to a next hop.
c. A name lookup service to map a given name to a destination machine.

OS structure

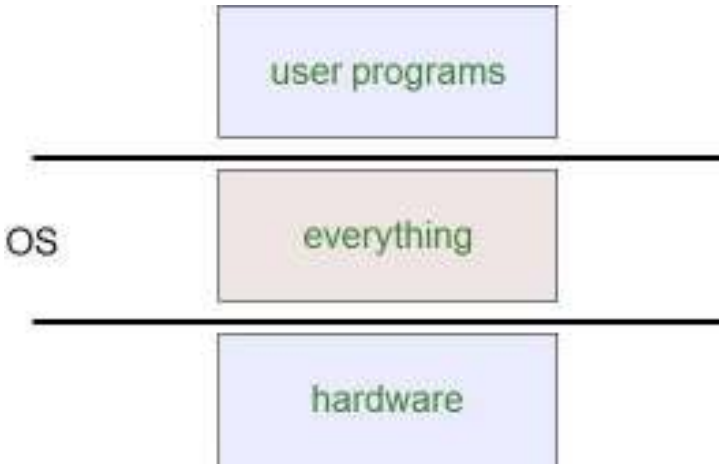
- It's not always clear how to stitch OS modules together:
- An OS consists of all of these components, plus:
  - ✓ Many other components
  - ✓ System programs (e.g. boot strap code, the init program).



- Major issues:
  - ✓ How do we organize all this?
  - ✓ What are all the code modules, and where do they exist?
  - ✓ How do they cooperate?
- Massive software engineering and design problem
  - ✓ Design a large complex program that:
    - Performs well, is reliable, is extensible, is backwards compatible…

Early structure: Monolithic

- Traditionally, OS's (like UNIX, DOS) were built as a monolithic entity:



Monolithic Design

- Major Advantages:
  - ✓ Cost of module interaction is low
- Disadvantages
  - ✓ Hard to understand
  - ✓ Hard to modify
  - ✓ Unreliable
  - ✓ Hard to maintain
- What id alternative?
  - ✓ Find ways to organize the OS in order to simplify its design and implementation.

Layering

- The traditional approach is layering
  - ✓ Implement OS as a set of layers
  - ✓ Each layer presents an enhanced virtual machine to the layer above.
- The first description of the system was Djakarta's THE system.
  - ✓ Layer 5: job managers
  - ✓ Layer 4: device managers
  - ✓ Layer 3: console manager
  - ✓ Layer 2: pager manager
  - ✓ Layer 1: Kernel
  - ✓ Layer 0: Hardware

Problems with layering

Imposes hierarchical structure

- Real system are more complex:
  - ✓ File system requires VM services (buffers).
  - ✓ VM would like to use files for its backing store
- Strict layering isn‘t flexible enough
  - ✓ Poor performance
- Each layer crossing has overhead associated with it
  - ✓ Disjunction between model and reality
- Systems modeled as layers, but not really built that way

Microkernel’s

- Popular in the late 80‘s, early 90‘s
  - ✓ recent resurgence of popularity for small devices
- Goal
  - ✓ Minimum functionality in the kernel. Most of the OS functionality in user level servers.
  - ✓ Examples of servers are file servers, terminal servers, memory servers etc.Each part becomes more manageable. Crashing of one service doesn‘t bring the system down.

- This results in
  - ✓ Better reliability (isolation between components)
  - ✓ Ease of extension and customization
  - ✓ Poor performance (user/kernel boundary crossing)
- Minimum functionality in the kernel. Most of the OS functionality in user level servers. Examples of servers are file servers, terminal servers, memory servers etc.
- Each part becomes more manageable. Crashing of one service doesn't bring the system down.
- Distribution of the system becomes transparent.
- Kernel provides basic primitives e.g. transport of messages, loading programs into memory, device handling.
- Policy decisions are made in the user space while mechanisms are implemented in micro-kernel.
- Micro-kernel lends itself well to OO design principles. Components based design possible.
- Disadvantage: Performance
- Solutions:
- Reduce micro-kernel size
- Increase micro-kernel size

Virtual Machine Monitors

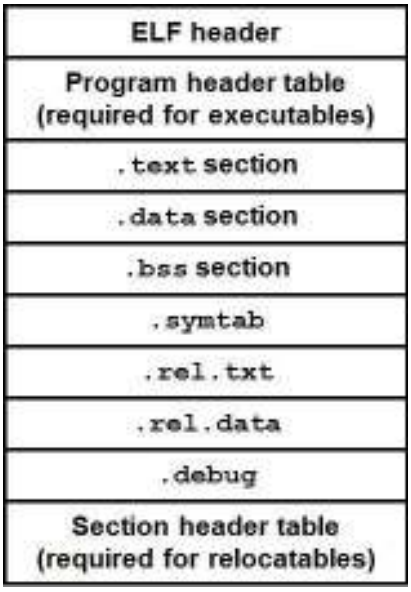
- Export a virtual machine to user programs that resembles hardware.
- A virtual machine consists of all hardware features e.g. user/kernel modes, I/O, interrupts and pretty much everything a real machine has.
- A virtual machine may run any OS.
- Examples: JVM, VM Ware, User-Mode Linux (UML).
- Advantage: portability
- Disadvantage: slow speed

Lecture 3

Overview of today's lecture

ELF Object File Format

- Elf header
  - ✓ Magic number, type (.o, exec, .so), machine, byte ordering, etc.
- Program header table
  - ✓ Page size, virtual addresses memory segments (sections), segment sizes.
- .text section
  - ✓ Code
- .data section
  - ✓ Initialized (static) data
- .bss section
  - ✓ Uninitialized (static) data
  - ✓ ¯Block Started by Symbol‖
  - ✓ ¯Better Save Space‖
  - ✓ Has section header but occupies no space



- symtab section
  - ✓ Symbol table
  - ✓ Procedure and static variable names
  - ✓ Section names and locations

- .rel.text section
  - ✓ Relocation info for .text section
  - ✓ Addresses of instructions that will need to be modified in the executable
  - ✓ Instructions for modifying.

- .rel.data section
  - ✓ Relocation info for .data section
  - ✓ Addresses of pointer data that will need to be modified in the merged executable

- .debug section
  - ✓ Info for symbolic debugging (gcc -g)

Example C Program

m.c

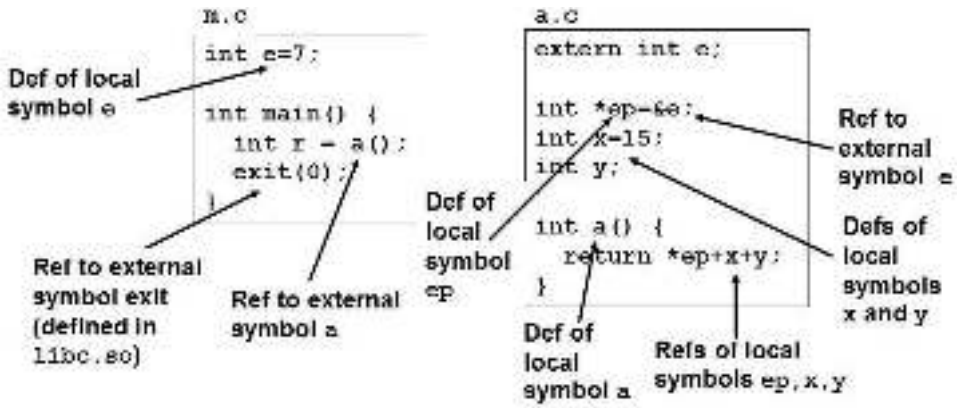
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Relocating Symbols and Resolving External References

- Symbols are lexical entities that name functions and variables.
- Each symbol has a value (typically a memory address).
- Code consists of symbol definitions and references.
- References can be either local or external.



m.o Relocation Info

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

```
Disassembly of section .text:

00000000 <main>: 00000000 <main>:
   0:   55                  pushl  %ebp
   1:   89 e5               movl   %esp,%ebp
   3:   e8 fc ff ff ff      call   4 <main+0x4>

   8:   6a 00               pushl  $0x0
   a:   e8 fc ff ff ff      call   b <main+0xb>

   f:   90                  nop
```

```
Disassembly of section .data:

00000000 <e>:
   0:   07 00 00 00
```

a.o Relocation Info (.text)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .text:

00000000 <a>:
   0:   55                  pushl  %ebp
   1:   8b 15 00 00 00      movl   0x0,%edx
   6:   00
                            3: R_386_32     ep
   7:   a1 00 00 00 00      movl   0x0,%eax
                            8: R_386_32     x
   c:   89 e5               movl   %esp,%ebp
   e:   03 02               addl   (%edx),%eax
  10:   89 ec               movl   %ebp,%esp
  12:   03 05 00 00 00      addl   0x0,%eax
  17:   00
                            14: R_386_32    y
  18:   5d                  popl   %ebp
  19:   c3                  ret
```

a.o Relocation Info (.data)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

00000000 <ep>:
   0:   00 00 00 00
                            0: R_386_32     e
 00000004 <x>:
   4:   0f 00 00 00
```

Executable after Relocation and External Reference Resolution (.text)

```
08048530 <main>:
 8048530:       55                      pushl  %ebp
 8048531:       89 e5                   movl   %esp,%ebp
 8048533:       e8 08 00 00 00          call   8048540 <a>
 8048538:       6a 00                   pushl  $0x0
 804853a:       e8 35 ff ff ff          call   8048474 <_init+0x94>
 804853f:       90                      nop

08048540 <a>:
 8048540:       55                      pushl  %ebp
 8048541:       8b 15 1c a0 04          movl   0x804a01c,%edx
 8048546:       08
 8048547:       a1 20 a0 04 08          movl   0x804a020,%eax
 804854c:       89 e5                   movl   %esp,%ebp
 804854e:       03 02                   addl   (%edx),%eax
 8048550:       89 ec                   movl   %ebp,%esp
 8048552:       03 05 d0 a3 04          addl   0x804a3d0,%eax
 8048557:       08
 8048558:       5d                      popl   %ebp
 8048559:       c3                      ret
```

Executable After Relocation and External Reference Resolution(.data)

m.c

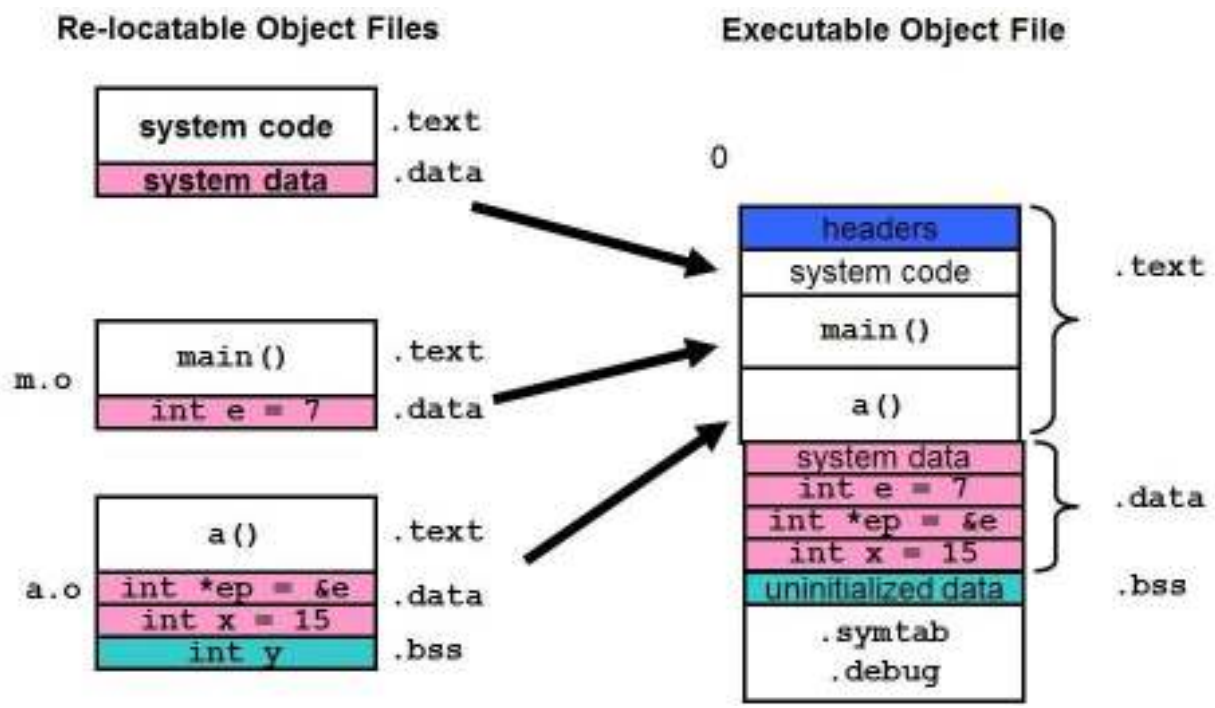
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

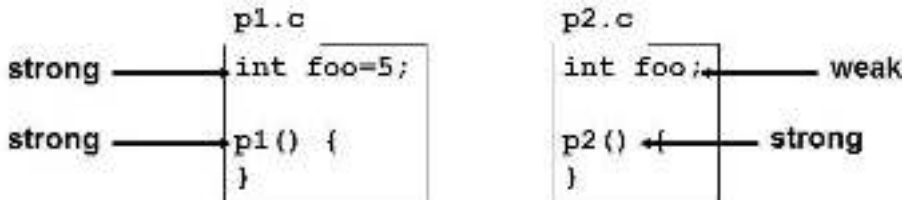
```
Disassembly of section .data:

0804a018 <e>:
 804a018:       07 00 00 00

0804a01c <ep>:
 804a01c:       18 a0 04 08

0804a020 <x>:
 804a020:       0f 00 00 00
```

Merging Re-locatable Object Files into an Executable Object File



Strong and Weak Symbols

- Program symbols are either strong or weak
  - ✓ strong: procedures and initialized globals
  - ✓ weak: uninitialized globals



Linker's Symbol Rules

- Rule 1. A strong symbol can only appear once.
- Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
  - ✓ References to the weak symbols resolve to the strong symbol.
- Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

Linker Puzzles

`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (p1)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to x will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to x in p2 might overwrite y! Evil!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to x in p2 will overwrite y! Nasty!

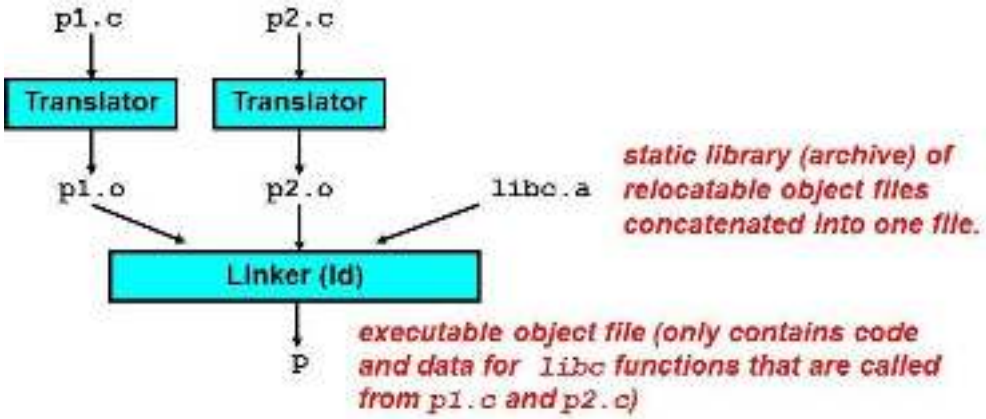
`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Packaging Commonly Used Functions

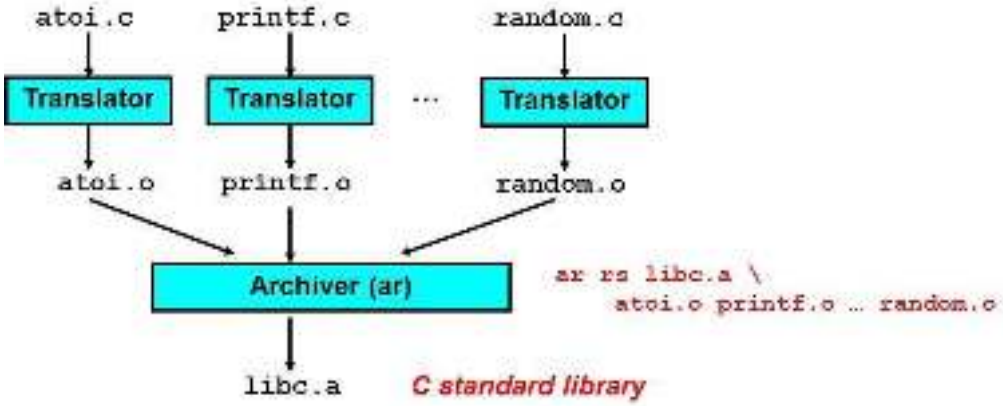
- How to package functions commonly used by programmers?
  - ✓ Math, I/O, memory management, string manipulation, etc.
- Awkward, given the linker framework so far:
  - ✓ Option 1: Put all functions in a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - ✓ Option 2: Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer
- Solution: static libraries (.a archive files)
  - ✓ Concatenate related re-locatable object files into a single file with an index (called an archive).
  - ✓ Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - ✓ If an archive member file resolves reference, link into executable.

Static Libraries (archives)



- Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (libc), math library (libm)]
- Linker selects only the .o files in the archive that are actually needed by the program.

Creating Static Libraries



Archiver allows incremental updates:

- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

- libc.a (the C standard library)
  - ✓ 8 MB archive of 900 object files.
  - ✓ I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- libm.a (the C math library)
  - ✓ 1 MB archive of 226 object files.
  - ✓ floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

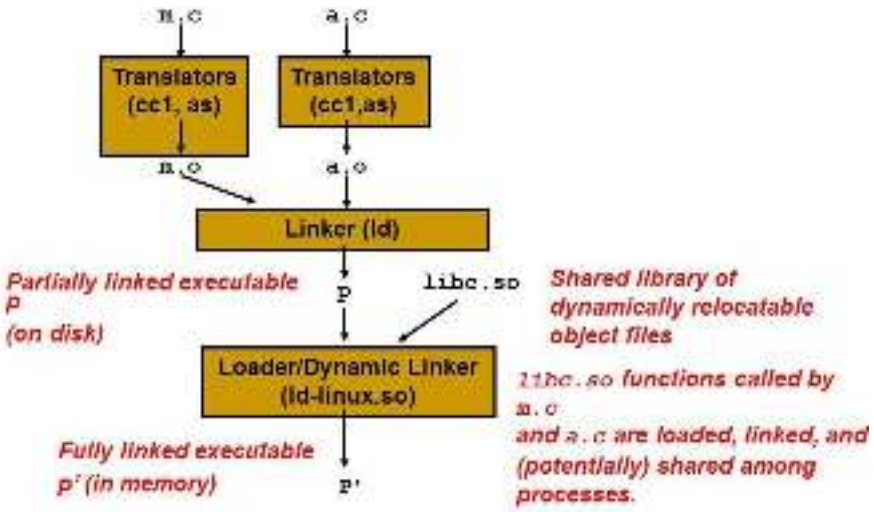
Using Static Libraries

- Linker's algorithm for resolving external references:
  - ✓ Scan .o files and .a files in the command line order.
  - ✓ During the scan, keep a list of the current unresolved references.
  - ✓ As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
  - ✓ If any entries in the unresolved list at end of scan, then error.
- Problem:
  - ✓ Command line order matters!
  - ✓ Moral: put libraries at the end of the command line.

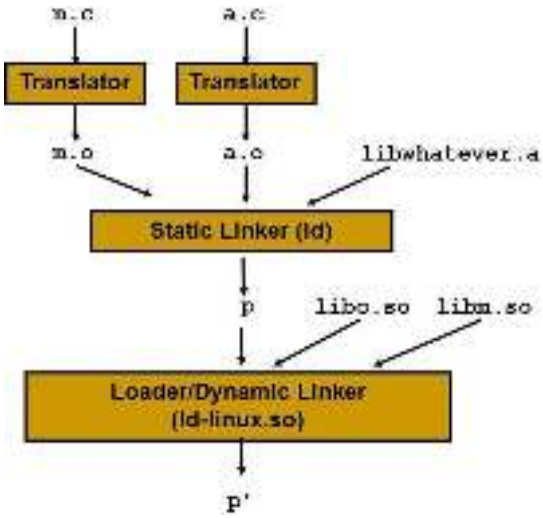
Shared Libraries

- Static libraries have the following disadvantages:
  - ✓ Potential for duplicating lots of common code in the executable files on a filesystem.
    - e.g., every C program needs the standard C library
  - ✓ Potential for duplicating lots of code in the virtual memory space of many processes.
  - ✓ Minor bug fixes of system libraries require each application to explicitly relink
- Solution:
  - ✓ Shared libraries (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
- Dynamic linking can occur when executable is first loaded and run.
  - ✓ Common case for Linux, handled automatically by ld-linux.so.
- Dynamic linking can also occur after program has begun.
  - ✓ In Linux, this is done explicitly by user with dlopen().
  - ✓ Basis for High-Performance Web Servers.
- Shared library routines can be shared by multiple processes

Dynamically Linked Shared Libraries



The Complete Picture



Start-up code in init segment

Same for all C programs

```
0x080480c0 <start>:
call __libc_init_first  /* startup code in .text */
call _init              /* startup code in .init */
atexit                  /* startup code in .text */
call main               /* application's entry point */
call _exit              /* return control to OS */
```

Note: The code that pushes the arguments for each function is not shown

Lecture 4

Overview of today's lecture

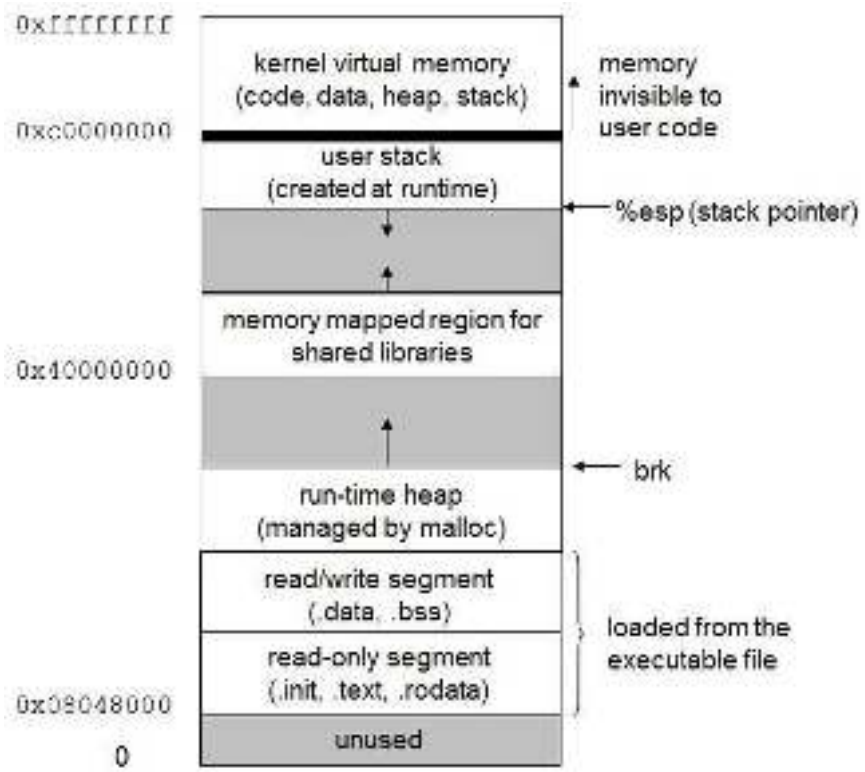
- Process definition
- What are Address spaces
- Components of an address space
- Methods of altering control flow of a CPU
- Interrupts, Traps and faults
- How does a process enter into the operating system
- Context switching
- Introduction to process management models and state machines
- Re-cap of the lecture

Process

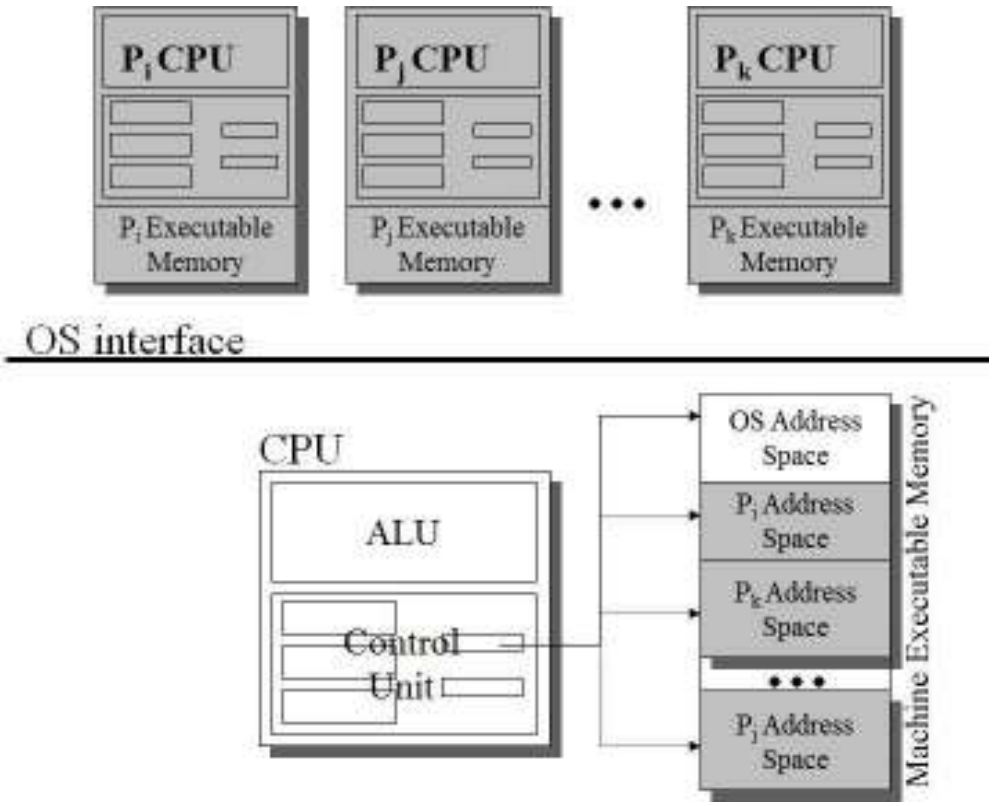
- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions

Private Address Spaces

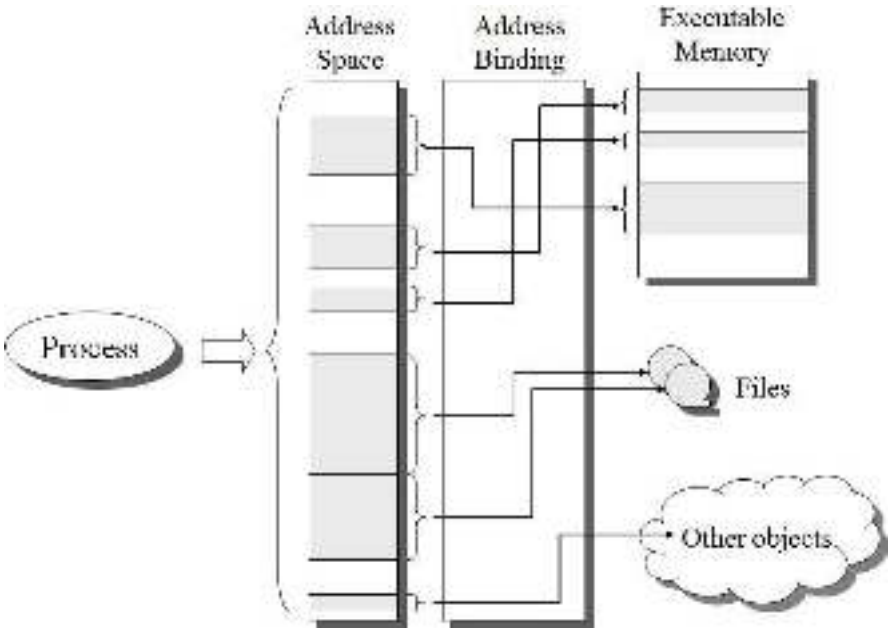
- Each process has its own private address space.



Implementing the Process Abstraction



The Address Space

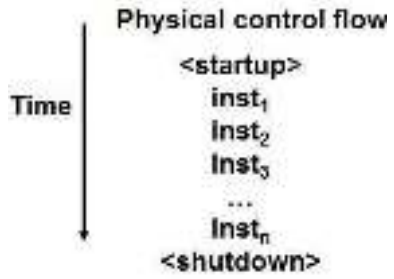


Execution of the Operating System

- Non-process Kernel
  - ✓ Execute kernel outside of any process
  - ✓ Operating system code is executed as a separate entity that operates in privileged mode
- Execution Within User Processes
  - ✓ Operating system software within context of a user process
  - ✓ Process executes in privileged mode when executing operating system code
- Process-Based Operating System
  - ✓ Implement operating system as a collection of system processes
  - ✓ Useful in multi-processor or multi-computer environment

Control Flow

- Computers do Only One Thing
  - ✓ From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
  - ✓ This sequence is the system's physical control flow (or flow of control).



Altering the Control Flow

- Two basic mechanisms available to the programmer for changing control flow:
  - ✓ Jumps and branches
  - ✓ Function call and return using the stack discipline.
  - ✓ Both react to changes in program state.

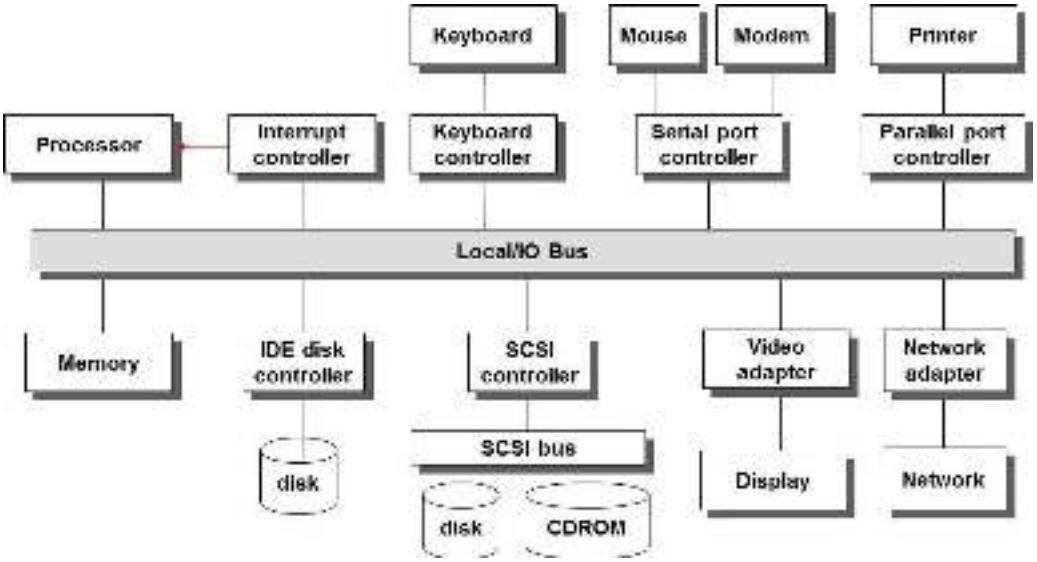
- Insufficient for a useful system
  - ✓ Difficult for the CPU to react to changes in system state.
    - o data arrives from a disk or a network adapter.
    - o Instruction divides by zero
    - o User hits ctl-c at the keyboard
    - o System timer expires

- System needs mechanisms for ¯exceptional control flow‖

Exceptional Control Flow

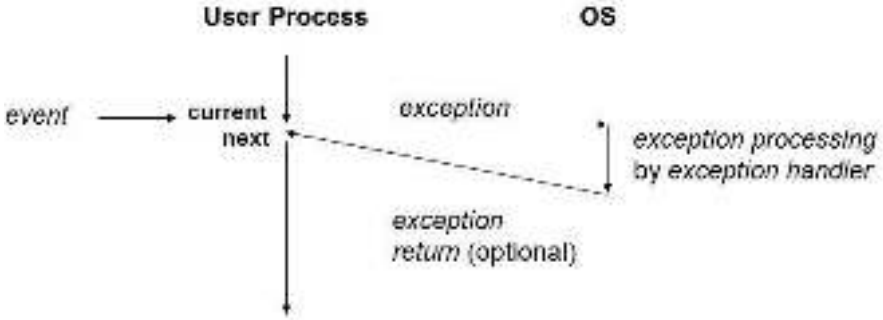
- Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level Mechanism
  - ✓ exceptions
    - change in control flow in response to a system event (i.e.,change in system state)
  - ✓ Combination of hardware and OS software
- Higher Level Mechanisms
  - ✓ Process context switch
  - ✓ Signals
  - ✓ Nonlocal jumps (setjmp/longjmp)
  - ✓ Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions



Exceptions

- An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)

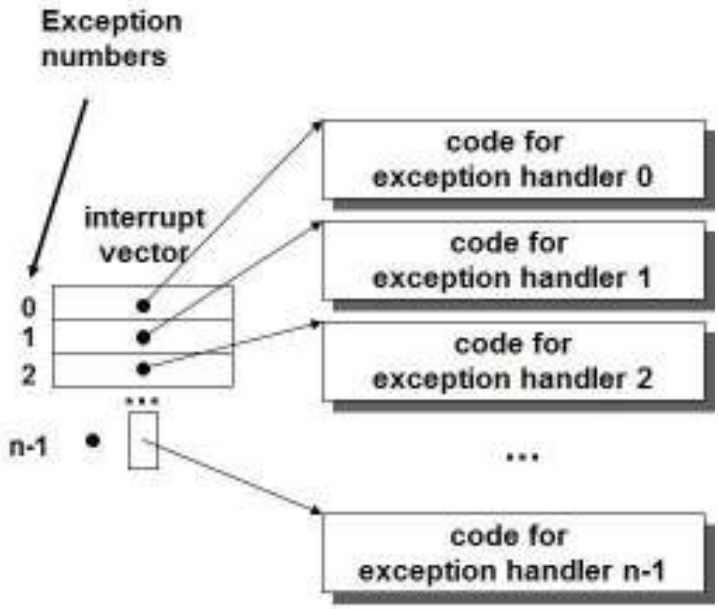


Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - ✓ Indicated by setting the processor‘s interrupt pin
  - ✓ handler returns to ¯next‖ instruction.
- Examples:
  - ✓ I/O interrupts
    - hitting ctl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - ✓ Hard reset interrupt
    - hitting the reset button
  - ✓ Soft reset interrupt
    - hitting ctl-alt-delete on a PC

Interrupt Vectors

- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler for k is called each time exception k occurs.



Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ✓ Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to ¯next‖ instruction
  - ✓ Faults

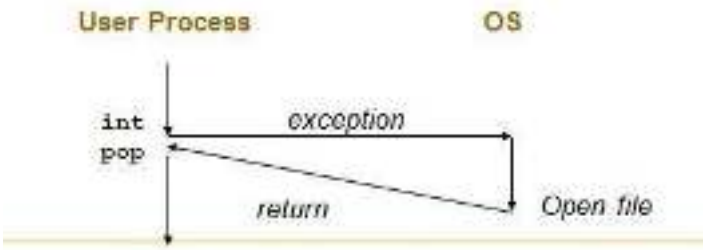
- Unintentional but possibly recoverable
- Examples: page faults (recoverable), protection faults (unrecoverable).
- Either re-executes faulting (¯current‖) instruction or aborts.

✓ Aborts
- unintentional and unrecoverable
- Examples: parity error, machine check.
- Aborts current program

Trap Example

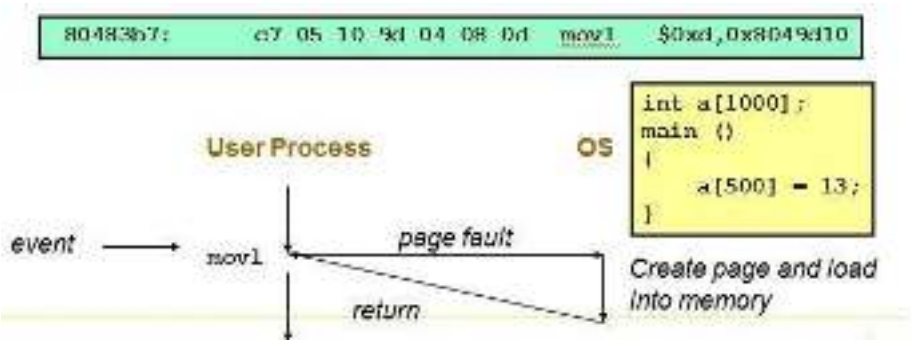
- Opening a File
  - ✓ User calls open(filename, options)
  - ✓ Function open executes system call instruction int
  - ✓ OS must find or create file, get it ready for reading or writing
  - ✓ Returns integer file descriptor

```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80                    int    $0x80
 804d084:      5b                       pop    %ebx
 . . .
```



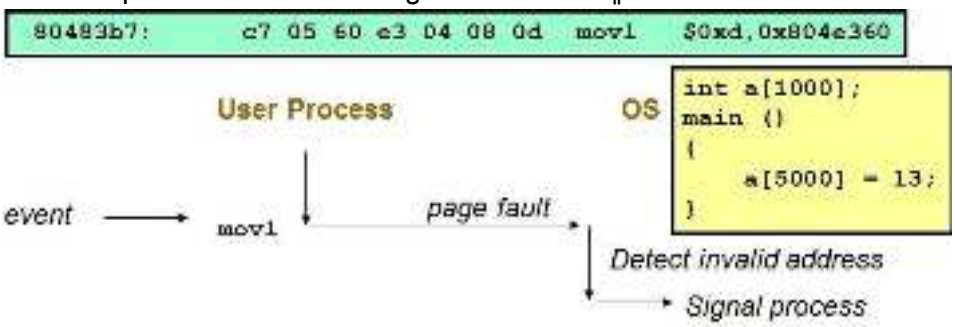
Fault Example # 1

- Memory Reference
  - ✓ User writes to memory location
  - ✓ That portion (page) of user's memory is currently on disk
  - ✓ Page handler must load page into physical memory
  - ✓ Returns to faulting instruction
  - ✓ Successful on second try

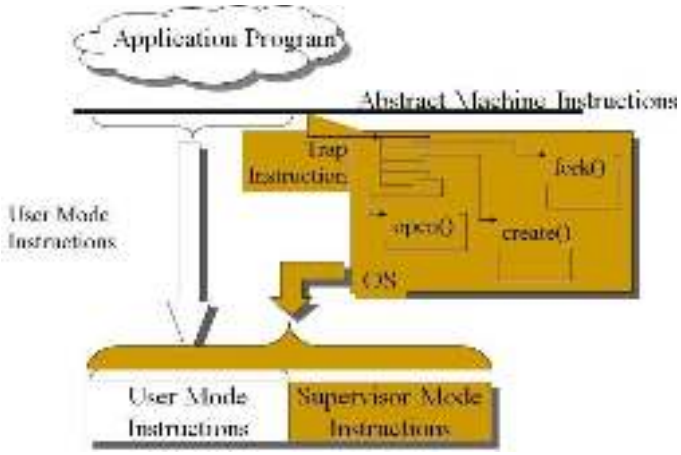


Fault Example # 2

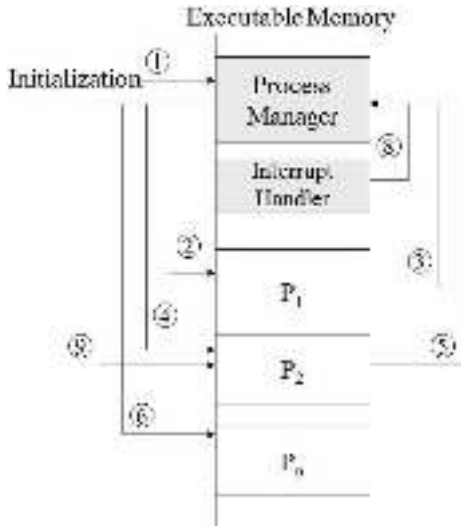
- Memory Reference
  - ✓ User writes to memory location
  - ✓ Address is not valid
  - ✓ Page handler detects invalid address
  - ✓ Sends SIGSEG signal to user process
  - ✓ User process exits with segmentation fault



The Abstract Machine Interface



Context Switching



When to Switch a Process

- Clock interrupt
  - ✓ process has executed for the maximum allowable time slice
- I/O interrupt
- Memory fault
  - ✓ memory address is in virtual memory so it must be brought into main memory
- Trap
  - ✓ error or exception occurred
  - ✓ may cause process to be moved to Exit state
- Supervisor call
  - ✓ such as file open

Process Creation

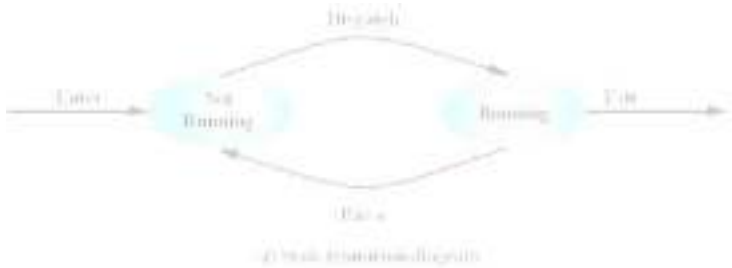
- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block
- Set up appropriate linkages
  - ✓ Ex: add new process to linked list used for scheduling queue
- Create or expand other data structures
  - ✓ Ex: maintain an accounting file

Change of Process State

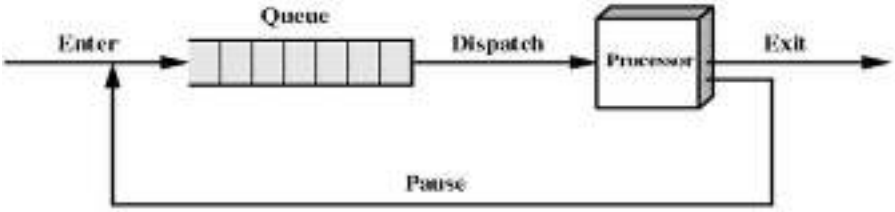
- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently in the Running state
- Move process control block to appropriate queue – ready; blocked; ready/suspend
- Select another process for execution
- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process

Two-State Process Model

- Process may be in one of two states
  - ✓ Running
  - ✓ Not-running



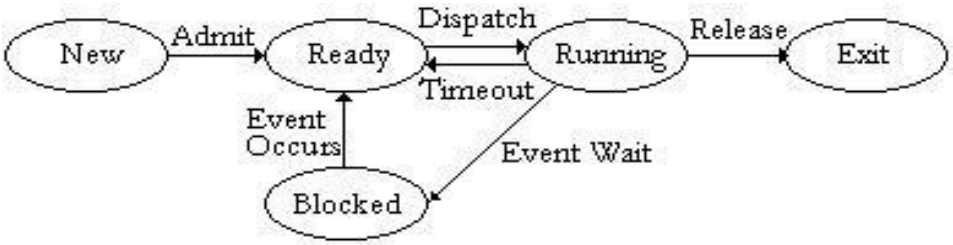
Not-Running Process in a Queue

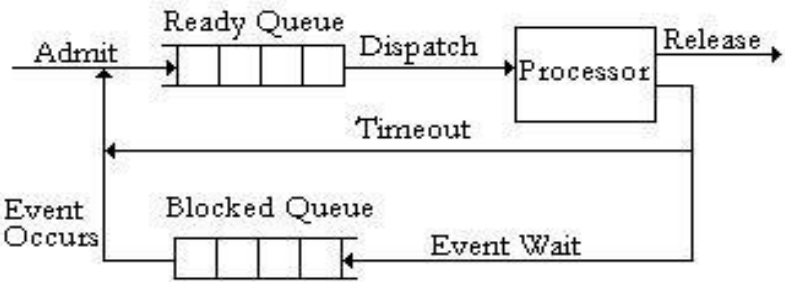


(b) Queuing diagram

Five-state Model

- Processes may be waiting for I/O
- Use additional states:
  - ✓ Running: currently being run
  - ✓ Ready: ready to run
  - ✓ Blocked: waiting for an event (I/O)
  - ✓ New: just created, not yet admitted to set of runnable processes
  - ✓ Exit: completed/error exit





- May have separate waiting queues for each event Transitions:
  - ✓ Null → New - Process is created
  - ✓ New → Ready - O.S. is ready to handle another process (Memory, CPU)
  - ✓ Ready → Running - Select another process to run
  - ✓ Running → Exit - Process has terminated
  - ✓ Running → Ready - End of time slice or higher-priority process is ready
  - ✓ Running → Blocked - Process is waiting for an event (I/O, Synchronization)
  - ✓ Blocked → Ready - The event a process is waiting for has occurred, can continue
  - ✓ Ready → Exit - Process terminated by O.S. or parent
  - ✓ Blocked → Exit - Same reasons

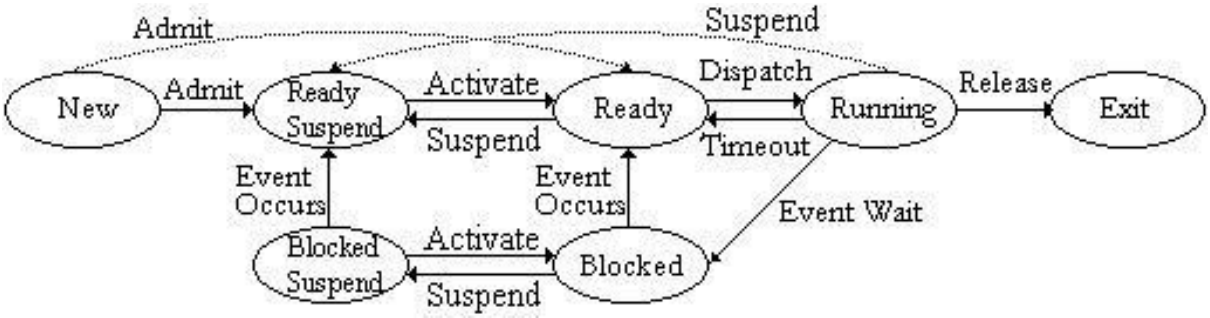
Lecture 5

Overview of today's lecture

- Re-view of the previous lecture
- Process management models and state machines (cont'd from the previous lecture)
- What is in a process control block
- Operating system calls for process management in UNIX family of systems
- Example programs invoking OS services for process management
- Re-cap of lecture

Suspending Processes

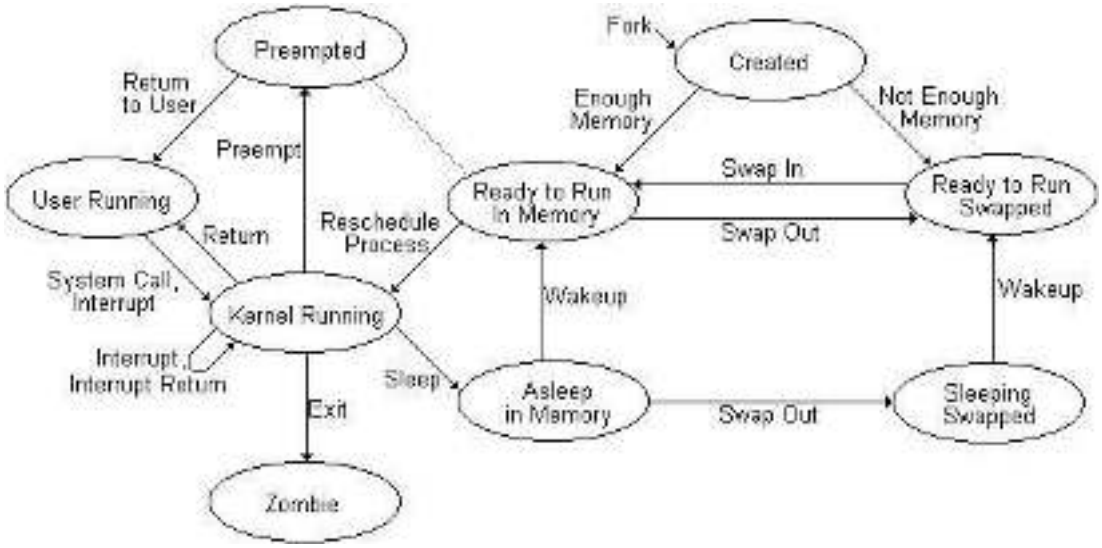
- May suspend a process by swapping part or all of it to disk
  - ✓ Most useful if we are waiting for an event that will not arrive soon (printer, keyboard)
  - ✓ If not done well, can slow system down by increasing disk I/O activity
- State Transition Diagram



- Key States:
  - ✓ Ready - In memory, ready to execute
  - ✓ Blocked - In memory, waiting for an event
  - ✓ Blocked Suspend - On disk, waiting for an event
  - ✓ Ready Suspend - On disk, ready to execute

Unix SVR4 Processes

- Uses 9 processes states
- Preempted and Ready to run, in memory are nearly identical
  - ✓ A process may be preempted for a higher-priority process at the end of a system call
- Zombie - Saves information to be passed to the parent of this process
- Process 0 - Swapper, created at boot
- Process 1 - Init, creates other processes



Modes of Execution

- User mode
  - ✓ Less-privileged mode
  - ✓ User programs typically execute in this mode
- System mode, control mode, or kernel mode
  - ✓ More-privileged mode
  - ✓ Kernel of the operating system

Operating System Control Structures

- Information about the current status of each process and resource
- Tables are constructed for each entity the operating system manages

Memory Tables

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

I/O Tables

- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer

File Tables

- Existence of files
- Location on secondary memory
- Current Status
- Attributes
- Sometimes this information is maintained by a file management system

Process Control Block

- Process identification
  - ✓ Identifiers
    - Numeric identifiers that may be stored with the process control block include
      - Identifier of this process
      - Identifier of the process that created this process (parent process)
      - User identifier
- Processor State Information
  - ✓ User-Visible Registers
    - A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.
  - ✓ Control and Status Registers
    - These are a variety of processor registers that are employed to control the operation of the processor. These include
    - Program counter: Contains the address of the next instruction to be fetched
    - Condition codes: Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
    - Status information: Includes interrupt enabled/disabled flags, execution mode
  - ✓ Stack Pointers
    - Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

- Process Control Information
  - ✓ Scheduling and State Information
    - This is information that is needed by the operating system to perform its scheduling function. Typical items of information:
    - Process state: defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
    - Priority: One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
    - Scheduling-related information: This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
    - Event: Identity of event the process is awaiting before it can be resumed

- ✓ Data Structuring
  - A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.
- ✓ Inter-process Communication
  - Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.
- ✓ Process Privileges
  - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

fork: Creating new processes

- int fork(void)
  - ✓ creates a new process (child process) that is identical to the calling process (parent process)
  - ✓ returns 0 to the child process
  - ✓ returns child's pid to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Fork Example #1

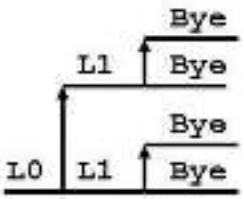
- Key Points
  - ✓ Parent and child both run same code
    - Distinguish parent from child by return value from fork
  - ✓ Start with same state, but each has private copy
    - Including shared output file descriptor
    - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- Key Points
  - ✓ Both parent and child can continue forking

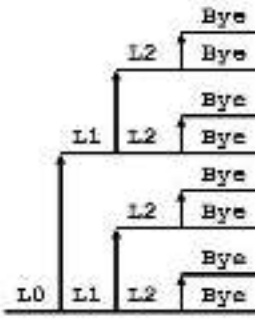
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Key Points
  - ✓ Both parent and child can continue forking

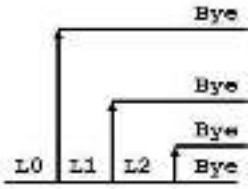
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - ✓ Both parent and child can continue forking

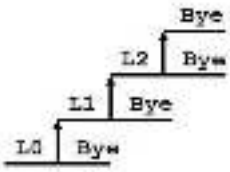
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Key Points
  - ✓ Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



exit: Destroying Process

- void exit(int status)
  - ✓ exits a process
    - Normally return with status 0
  - ✓ atexit() registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- Idea
  - ✓ When process terminates, still consumes system resources
    - Various tables maintained by OS
  - ✓ Called a ¯zombie‖
    - Living corpse, half alive and half dead
- Reaping
  - ✓ Performed by parent on terminated child
  - ✓ Parent is given exit status information
  - ✓ Kernel discards process

- What if Parent Doesn't Reap?
  - ✓ If any parent terminates without reaping a child, then child will be reaped by init process
  - ✓ Only need explicit reaping for long-running processes
    - E.g., shells and servers

Zombie Example

- ps shows child process as ¯defunct‖
  - ✓ Killing parent allows child to be reaped

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID =
%d\n",
                getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
                getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Non-terminating Child Example

- Child process still active even though parent has terminated
  - ✓ Must kill explicitly, or else will keep running indefinitely

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

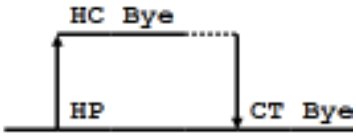
```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID =
%d\n", getpid());
        exit(0);
    }
}
```

wait: Synchronizing with children

- int wait(int *child_status)
  - ✓ suspends current process until one of its children terminates
  - ✓ return value is the pid of the child process that terminated
  - ✓ if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
       if ((pid[i] = fork()) == 0)
           exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                 wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminate abnormally\n", wpid);
    }
}
```

Waitpid

- waitpid(pid, &status, options)
  - ✓ Can wait for specific process
  - ✓ Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
       if ((pid[i] = fork()) == 0)
           exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
       pid_t wpid = waitpid(pid[i], &child_status, 0);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                 wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
    }
```

Wait/Waitpid Example Outputs

- Using wait (fork10)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

- Using waitpid (fork11)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

exec: Running new programs

- int execl(char *path, char *arg0, char *arg1, …, 0)
  - ✓ loads and runs executable at path with args arg0, arg1, …
    - path is the complete path of an executable
    - arg0 becomes the name of the process
      - typically arg0 is either identical to path, or else it contains only the executable filename from path
    - ¯real‖ arguments to the executable start with arg1, etc.

- list of args is terminated by a (char *)0 argument

✓ returns -1 if error, otherwise doesn't return!

```
main() {
   if (fork() == 0) {
      execl("/usr/bin/cp", "cp", "foo", "bar",
0);
   }
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```

Lecture No. 6

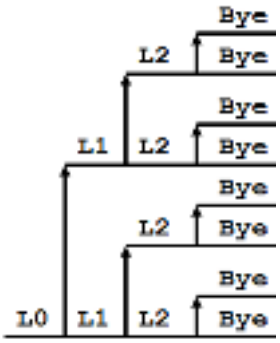
Overview of today's lecture

- Fork examples (cont'd from previous lecture)
- Zombies and the concept of Reaping
- Wait and waitpid system calls in Linux
- Concurrency–The need for threads within processes
- Threads–Introduction
- Re-cap of lecture

Fork Example #3

- Key Points
  - ✓ Both parent and child can continue forking

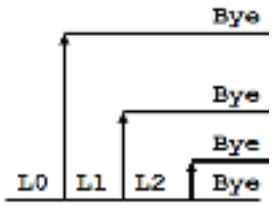
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - ✓ Both parent and child can continue forking

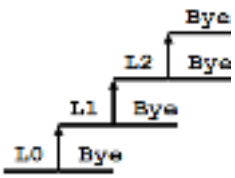
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Key Points
  - ✓ Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



exit: Destroying Process

- void exit(int status)
  - ✓ exits a process
    - Normally return with status 0
  - ✓ atexit() registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- Idea
  - ✓ When process terminates, still consumes system resources
    - Various tables maintained by OS
  - ✓ Called a ¯zombie‖
    - Living corpse, half alive and half dead
- Reaping
  - ✓ Performed by parent on terminated child
  - ✓ Parent is given exit status information
  - ✓ Kernel discards process
- What if Parent Doesn't Reap?
  - ✓ If any parent terminates without reaping a child, then child will be reaped by init process
  - ✓ Only need explicit reaping for long-running processes
    - E.g., shells and servers

Zombie Example:

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID =
%d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- ps shows child process as ¯defunct‖
- Killing parent allows child to be reaped

Non-terminating Child Example:

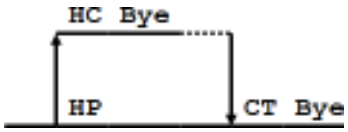
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID =
%d\n", getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

wait: Synchronizing with children

- int wait(int *child_status)
  - ✓ suspends current process until one of its children terminates
  - ✓ return value is the pid of the child process that terminated
  - ✓ if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated




```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```

Wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status
%d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

Waitpid

- waitpid(pid, &status, options)
  - ✓ Can wait for specific process
  - ✓ Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```

exec: Running new programs

- int execl(char *path, char *arg0, char *arg1, …, 0)
  - ✓ loads and runs executable at path with args arg0, arg1, …
    - path is the complete path of an executable
    - arg0 becomes the name of the process
      - typically arg0 is either identical to path, or else it contains only the executable filename from path
    - ¯real‖ arguments to the executable start with arg1, etc.
    - list of args is terminated by a (char *)0 argument
  - ✓ returns -1 if error, otherwise doesn‘t return!

```
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

Summarizing

- Exceptions
  - ✓ Events that require nonstandard control flow
  - ✓ Generated externally (interrupts) or internally (traps and faults)

- Processes
  - ✓ At any given time, system has multiple active processes
  - ✓ Only one can execute at a time, though
  - ✓ Each process appears to have total control of processor + private memory space
- Spawning Processes
  - ✓ Call to fork
    - o One call, two returns
- Terminating Processes
  - ✓ Call exit
    - o One call, no return
- Reaping Processes
  - ✓ Call wait or waitpid
- Replacing Program Executed by Process
  - ✓ Call execl (or variant)
    - o One call, (normally) no return

Concurrency

- Imagine a web server, which might like to handle multiple requests concurrently
  - ✓ While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information
- Imagine a web client (browser), which might like to initiate multiple requests concurrently
- Imagine a parallel program running on a multiprocessor, which might like to employ ¯physical concurrency‖
  - ✓ For example, multiplying a large matrix - split the output matrix into k regions and compute the entries in each region concurrently using k processors

What's in a process?

- A process consists of (at least):
  - ✓ an address space
  - ✓ the code for the running program
  - ✓ the data for the running program
  - ✓ an execution stack and stack pointer (SP)
    - o traces state of procedure calls made
  - ✓ the program counter (PC), indicating the next instruction
  - ✓ a set of general-purpose processor registers and their values
  - ✓ a set of OS resources
    - o open files, network connections, sound channels, …
- That's a lot of concepts bundled together!
- decompose …
  - ✓ an address space
  - ✓ threads of control
  - ✓ (other resources…)

What's needed?

- In each of these examples of concurrency (web server, web client, parallel program):
  - ✓ Everybody wants to run the same code
  - ✓ Everybody wants to access the same data
  - ✓ Everybody has the same privileges
  - ✓ Everybody uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
  - ✓ an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - ✓ the program counter (PC), indicating the next instruction
  - ✓ a set of general-purpose processor registers and their values

How could we achieve this?

- Given the process abstraction as we know it:
  - ✓ fork several processes
  - ✓ cause each to *map* to the same physical memory to share data
- It's really inefficient
  - ✓ space: PCB, page tables, etc.
  - ✓ time: creating OS structures, fork and copy addr space, etc.

Can we do better?

- Key idea:
  - ✓ separate the concept of a process (address space, etc.)
  - ✓ …from that of a minimal ¯thread of control‖ (execution state: PC, etc.)
- This execution state is usually called a thread, or sometimes, a lightweight process

Threads and processes

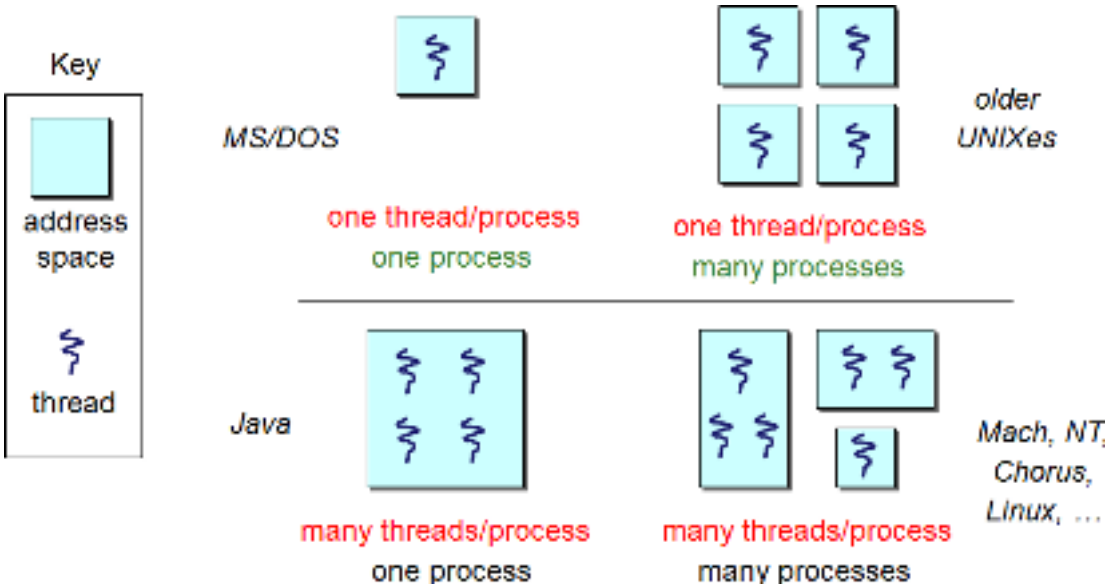
- Most modern OS's (Mach, Chorus, NT, modern UNIX) therefore support two entities:
  - ✓ the process, which defines the address space and general process attributes (such as open files, etc.)
  - ✓ the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - ✓ address spaces, however, can have multiple threads executing within them
  - ✓ sharing data between threads is cheap: all see the same address space
  - ✓ creating threads is cheap too!
- Threads become the unit of scheduling
  - ✓ processes / address spaces are just containers in which threads execute

Lecture No. 7

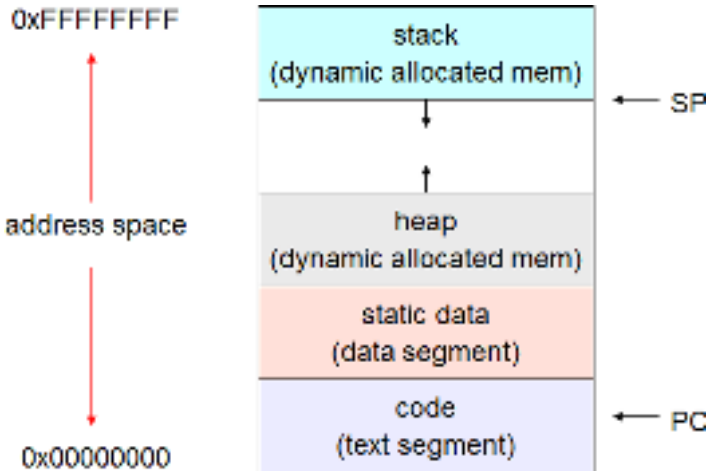
Overview of today's lecture

- The design space for threads
- Threads illustrated and view in an address space
- User level and kernel level thread implementations
- Problems and advantages of user level thread implementations
- Problems and advantages of kernel level thread implementations
- Re-cap of lecture

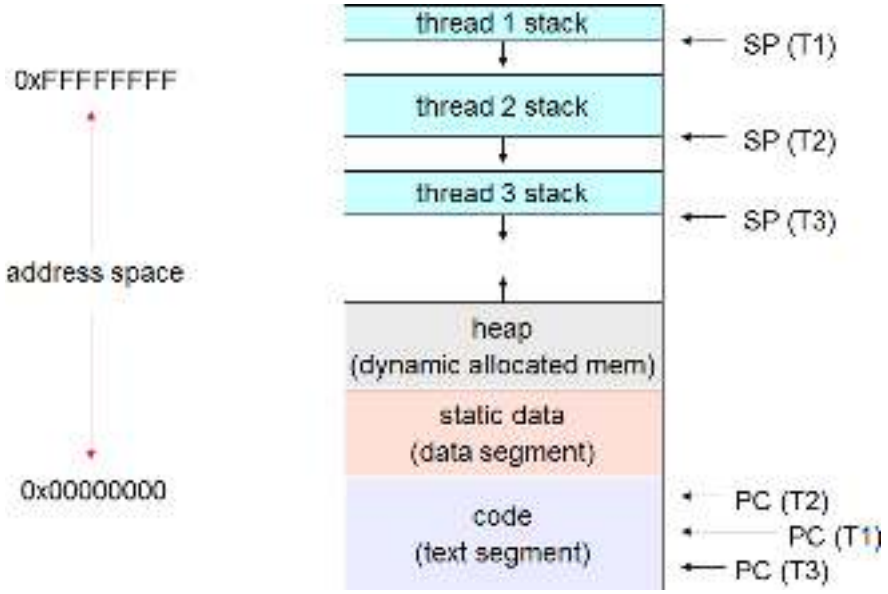
The design space



(old) Process address space



(new) Process address space with threads



Process/thread separation

- Concurrency (multithreading) is useful for:
  - ✓ handling concurrent events (e.g., web servers and clients)
  - ✓ building parallel programs (e.g., matrix multiply, ray tracing)
  - ✓ improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
  - ✓ even though only one thread can run at a time
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - ✓ creating concurrency does not require creating new processes
  - ✓ ‗faster / better / cheaper‖

Kernel threads

- OS manages threads *and* processes
  - ✓ all thread operations are implemented in the kernel
  - ✓ OS schedules all of the threads in a system
    - o if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - o possible to overlap I/O and computation inside a process

- Kernel threads are cheaper than processes
  - ✓ less state to allocate and initialize

- But, they‘re still pretty expensive for fine-grained use (e.g., orders of magnitude more expensive than a procedure call)

- ✓ thread operations are all system calls
  - context switch
  - argument checks
- ✓ must maintain kernel state for each thread

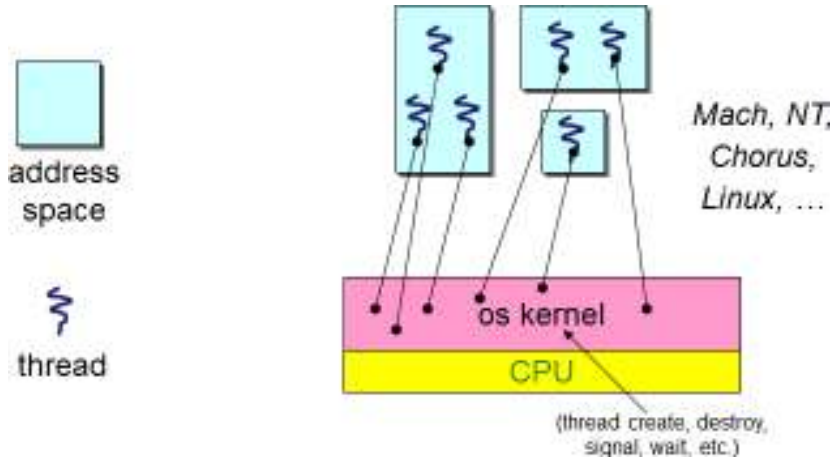
User-level threads

- To make threads cheap and fast, they need to be implemented at the user level
  - ✓ managed entirely by user-level library, e.g., libpthreads.a
- User-level threads are small and fast
  - ✓ each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - ✓ creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
  - ✓ user-level thread operations can be 10-100x faster than kernel threads as a result

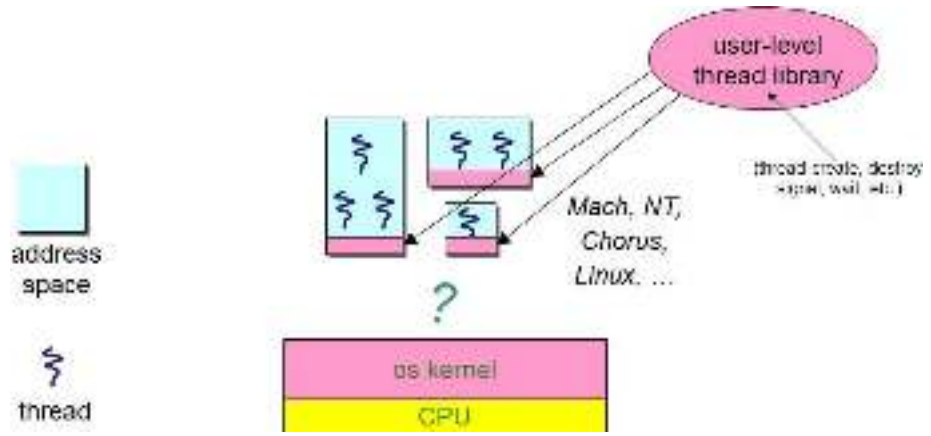
User-level thread implementation

- The kernel believes the user-level process is just a normal process running code
  - ✓ But, this code includes the thread support library and its associated thread scheduler
- The thread scheduler determines when a thread runs
  - ✓ it uses queues to keep track of what threads are doing: run, ready, wait
    - just like the OS and processes
    - but, implemented at user-level as a library

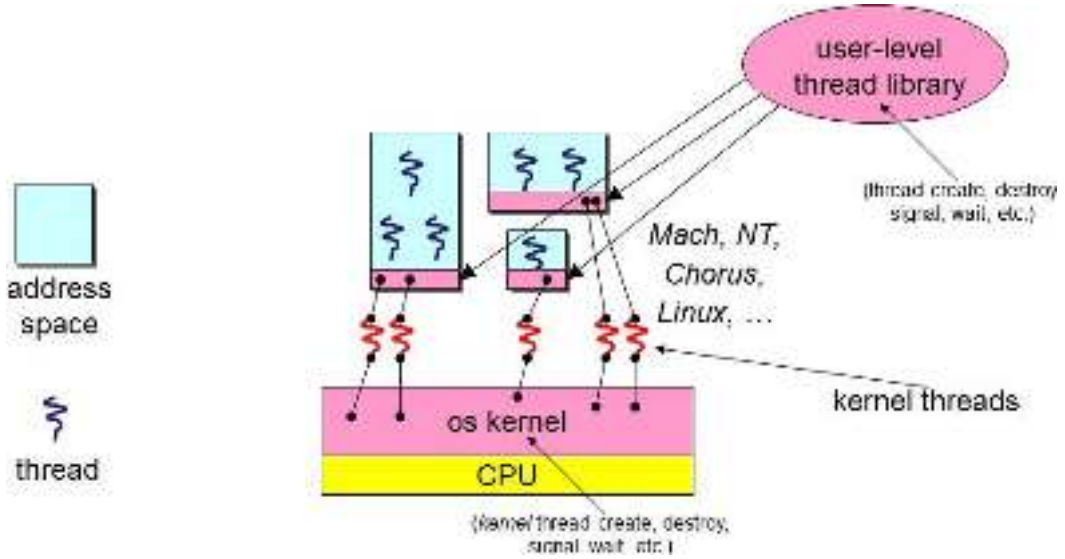
Kernel threads



User-level threads, conceptually



Multiple kernel threads "powering" each address space



How to keep a user-level thread from hogging the CPU?

- Strategy 1: force everyone to cooperate
  - ✓ a thread willingly gives up the CPU by calling **yield()**
  - ✓ **yield()** calls into the scheduler, which context switches to another ready thread
  - ✓ what happens if a thread never calls **yield()**?
- Strategy 2: use preemption
  - ✓ scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a UNIX signal (man signal)
    - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - ✓ at each timer interrupt, scheduler gains control and context switches as appropriate

Thread context switch

- Very simple for user-level threads:
  - ✓ save context of currently running thread
    - push machine state onto thread stack
  - ✓ restore context of the next thread
    - pop machine state from next thread‘s stack
  - ✓ return as the new thread
    - execution resumes at PC of next thread
- This is all done by assembly language
  - ✓ it works at the level of the procedure calling convention
    - thus, it cannot be implemented using procedure calls
    - e.g., a thread might be preempted (and then resumed) in the middle of a procedure call
    - C commands setjmp and longjmp are one way of doing it

What if a thread tries to do I/O?

- The kernel thread ¯powering‖ it is lost for the duration of the (synchronous) I/O operation!
- Could have one kernel thread ¯powering‖ each user-level thread
  - ✓ no real difference from kernel threads – ¯common case‖ operations (e.g., synchronization) would be quick
- Could have a limited-size ¯pool‖ of kernel threads ¯powering‖ all the user-level threads in the address space
  - ✓ the kernel will be scheduling these threads, obliviously to what‘s going on at user-level

What if the kernel preempts a thread holding a lock?

- Other threads will be unable to enter the critical section and will block (stall)
  - ✓ tradeoff, as with everything else
- Solving this requires coordination between the kernel and the user-level thread manager
  - ✓ ¯scheduler activations‖
    - each process can request one or more kernel threads
      - process is given responsibility for mapping user-level threads onto kernel threads
      - kernel promises to notify user-level before it suspends or destroys a kernel thread

Pros and Cons of User Level threads

Pros

- Procedure invocation instead of system calls results in fast scheduling and thus much better performance
- Could run on exisiting OSes that don‘t support threads
- Customized scheduling. Useful in many cases e.g. in case of garbage collection
- Kernel space not required for thread specific data. Scale better for large number of threads

Cons

- Blocking system calls affect all threads
- Solution1: Make the system calls non-blocking. Is this a good solution?
- Solution2: Write Jacket or wrapper routines with select. What about this solution? Requires re-writing parts of system call library.
- Similar problem occurs with page faults. The whole process blocks
- Voluntary yield to the run-time system necessary for context switch
- Solution: Run time system may request a clock signal.
- Programs that use multi-threading need to make system calls quite often. If they don't then there is usually no need to be multi-threaded.

Pros and cons of kernel level threads

- Blocking system calls cause no problem
- Page faults can be handled by scheduling another thread from the same process (if one is ready)
- Cost of system call substantially greater
- Solution: thread recycling; don't destroy thread data structures when the thread is destroyed.

Lecture No. 8

Overview of today's lecture

- POSIX threads (pthreads) standard interface and calls
- Simple pthreads Hello World program
- Linux processes and threads
- The clone() system call
- Fields in task_struct in Linux
- Process/threads states and FSM in Linux
- Looking ahead into the next lecture
- Re-cap of lecture

Posix Threads (Pthreads) Interface

- Pthreads: Standard interface for ~60 functions that manipulate threads from C programs.
  - ✓ Creating and reaping threads.
    - pthread_create
    - pthread_join
  - ✓ Determining your thread ID
    - pthread_self
  - ✓ Terminating threads
    - pthread_cancel
    - pthread_exit
    - exit [terminates all threads] , ret [terminates current thread]
  - ✓ Synchronizing access to shared variables
    - pthread_mutex_init
    - pthread_mutex_[un]lock
    - pthread_cond_init
    - pthread_cond_[timed]wait

The Pthreads "hello, world" Program

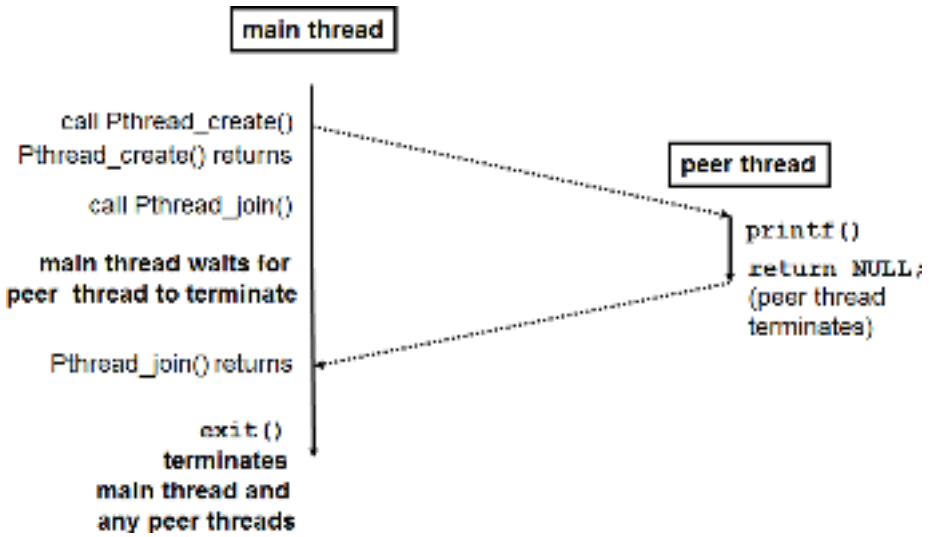
```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of Threaded"hello, world"



Processes and threads in Linux

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.
- A distinction is only made when a new thread is created by the **clone** system call.
  - ✓ **fork** creates a new process with its own entirely new process context
  - ✓ **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

Main Flags for Linux clone

- CLONE_CLEARID Clear the task ID.
- CLONE_DETACHED The parent does not want a SIGCHLD signal sent on exit.
- CLONE_FILES Shares the table that identifies the open files.
- CLONE_FS Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
- CLONE_IDLETASK Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
- CLONE_PARENT Caller and new task share the same parent process.
- CLONE_PTRACE If the parent process is being traced, the child process will also be traced.
- CLONE_SIGHAND Shares the table that identifies the signal handlers.
- CLONE_THREAD Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
- CLONE_VFORK If set, the parent does not get scheduled for execution until the child invokes the *execve()* system call.
- CLONE_VM Shares the address space (memory descriptor and all page tables).

clone()

- fork() is implemented as a wrapper around clone() with specific parameters
- __clone(fp, data, flags, stack)
  - ✓ "__" means "don't call this directly"
  - ✓ fp is thread start function, data is params
  - ✓ flags is of CLONE_ flags
  - ✓ stack is address of user stack
  - ✓ clone() calls do_fork() to do the work

Internal Kernel threads

- Linux has a small number of kernel threads that run continuously in the kernel (daemons)
  - ✓ No user address space (only kernel mapped)
- Creating: kernel_thread()
- Process 0: idle process
- Process 1
  - ✓ Spawns several kernel threads before transitioning to user mode as /sbin/init
  - ✓ kflushd (bdflush) – Flush dirty buffers to disk under "memory pressure"
  - ✓ kupdate – Periodically flushes old buffers to disk
  - ✓ kswapd – Swapping daemon

Linux processes

- In Linux terminology, processes are called tasks.
- Linux has a list of process descriptors
- (which are of type task_struct defined in include/linux/sched.h in Linux source tree).
- The maximum number of threads/processes allowed is dependent upon the amount of memory in the system.
- Check /proc/sys/kernel/threads_max for the current limit. By writing to that file, the limit can be changed on the fly (by the superuser).

Linux Process Identity

- Users: pid; Kernel: address of descriptor
  - ✓ Pids dynamically allocated, reused
    - 16 bits – 32767, avoid immediate reuse
  - ✓ Pid to address hash
  - ✓ static task_array
    - Statically limited # tasks
    - This limitation removed in 2.4
- current->pid (macro)

do_fork()

- Highlights
  - ✓ alloc_task_struct()
  - ✓ find_empty_process()
  - ✓ get_pid()
  - ✓ Update ancestry
  - ✓ Copy components based on flags
  - ✓ copy_thread()
  - ✓ Link into task list, update nr_tasks
  - ✓ Set TASK_RUNNING
  - ✓ wake_up_process()

Linux data structures for processes

- The process control blocks are called descriptors
- They are kept in circular linked lists
- The linked list implementation in the kernel uses the following structure as a node:
- struct list_head {

```
struct list_head *next, *prev;
};
```

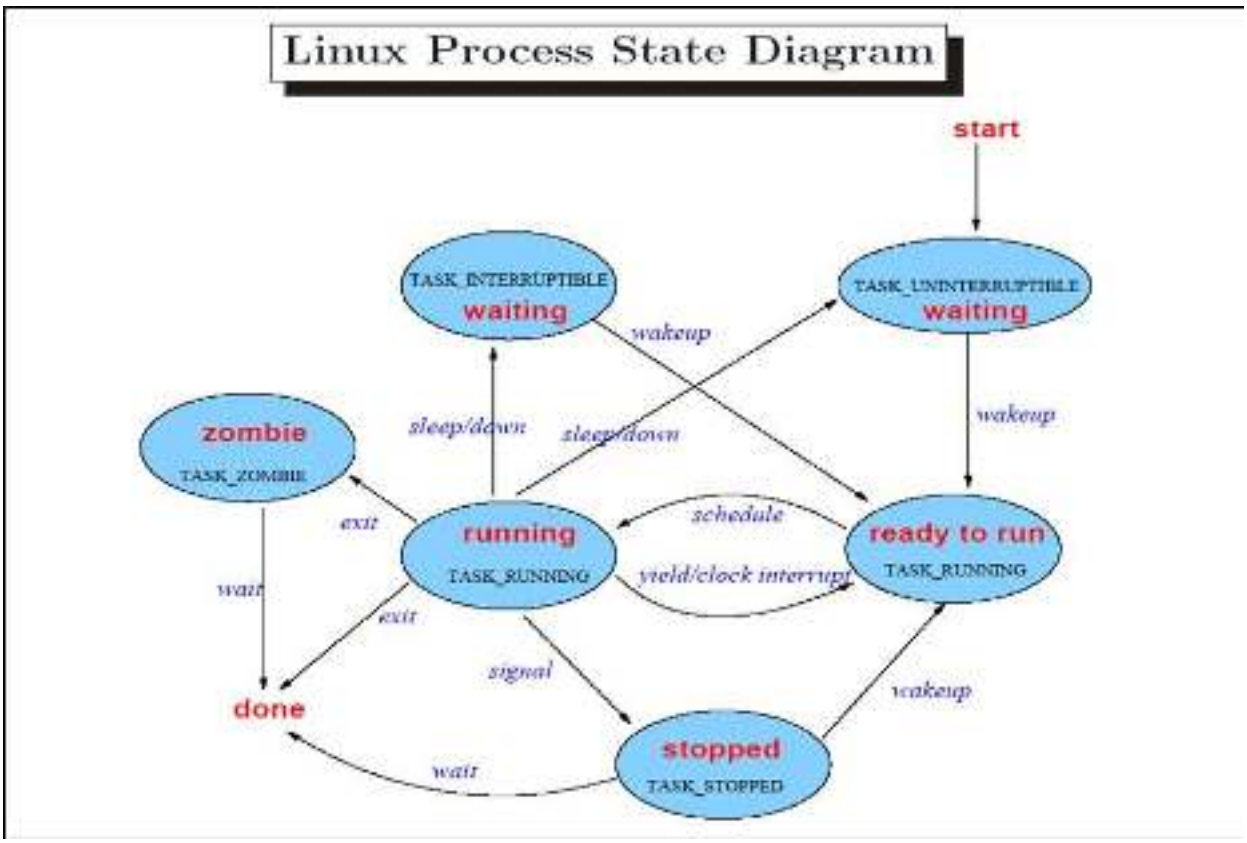
- The linked lists are circular, so there is no head or tail node. You can start at any node and traverse the whole list.

Task struct contents

- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.
- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.
- **Address space:** Defines the virtual address space assigned to this process.
- **Processor-specific context:** The registers and stack information that constitute the context of this process.

- **State:** The execution state of the process. These include:
  - ✓ **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
  - ✓ **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
  - ✓ **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
  - ✓ **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
  - ✓ **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

See /include/linux/sched.h for process states in the Linux kernel



Lecture No. 9

Overview of today's lecture

- Shared variable analysis in multi-threaded programs
- Concurrency and synchronization
- Critical sections
- Solutions to the critical section problem
- Concurrency examples
- Re-cap of lecture

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared variables?
  - ✓ The answer is not as simple as ¯global variables are shared‖ and ¯stack variables are private‖.
- Requires answers to the following questions:
  - ✓ What is the memory model for threads?
  - ✓ How are variables mapped to memory instances?
  - ✓ How many threads reference each of these instances

Threads Memory Model

- Conceptual model:
  - ✓ Each thread runs in the context of a process.
  - ✓ Each thread has its own separate thread context.
    - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers.
  - ✓ All threads share the remaining process context.
    - Code, data, heap, and shared library segments of the process virtual address space.
    - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - ✓ While register values are truly separate and protected....
  - ✓ Any thread can read and write the stack of any other thread.
- Mismatch between the conceptual and operation model is a source of confusion and errors.

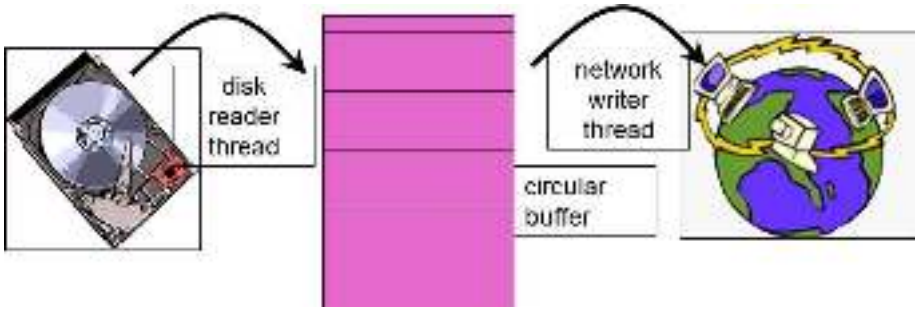
What resources are shared?

- Local variables are not shared
  - ✓ refer to data on the stack, each thread has its own stack
  - ✓ never pass/share/store a pointer to a local variable on another thread's stack!
- Global variables are shared
  - ✓ stored in the static data segment, accessible by any thread
- Dynamic objects are shared
  - ✓ stored in the heap, shared if you can name it
    - in C, can conjure up the pointer
      - ➢ e.g., void *x = (void *) 0xDEADBEEF

- in Java, strong typing prevents this
  - must pass references explicitly

Synchronization

- Threads cooperate in multithreaded programs
  - ✓ to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - ✓ also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



- For correctness, we have to control this cooperation
  - ✓ must assume threads interleave executions arbitrarily and at different rates
    - scheduling is not under application writers' control
- We control cooperation using synchronization
  - ✓ enables us to restrict the interleaving of executions
- Note: this also applies to processes, not just threads
- It also applies across machines in a distributed system

Example of Threads Accessing Another Thread's Stack

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++svar);
}
```

Peer threads access main thread's stack indirectly through global ptr variable

Shared Variable Analysis

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
ptr	yes	yes	yes
svar	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

- A variable x is shared iff multiple threads reference at least one instance of x. Thus:
  - ✓ ptr, svar, and msgs are shared.
  - ✓ i and myid are NOT shared.

badcnt.c: An Improperly Synchronized Threaded Program

```
unsigned int cnt = 0; /* shared */
#define NITERS 100000000
int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                   count, NULL);
    Pthread_create(&tid2, NULL,
                   count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
                cnt);
    else
        printf("OK cnt=%d\n",
                cnt);
}
```

```
/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

cnt should be equal to 200,000,000. What went wrong?!

Assembly Code for Counter Loop

C code for counter loop

```
for (i=0; i<NITERS; i++)
    cnt++;
```

Corresponding asm code
(gcc -O0 -fforce-mem)

Head (H_i)

Load cnt (L_i)
Update cnt (U_i)
Store cnt (S_i)

Tail (T_i)

```
.L9:
        movl -4(%ebp),%eax
        cmpl $99999999,%eax
        jle .L12
        jmp .L10
.L12:
        movl cnt,%eax        # Load
        leal 1(%eax),%edx    # Update
        movl %edx,cnt        # Store
.L11:
        movl -4(%ebp),%eax
        leal 1(%eax),%edx
        movl %edx,-4(%ebp)
        jmp .L9
.L10:
```

Concurrent Execution

- Key idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!
  - ✓ I_i denotes that thread i executes instruction I
  - ✓ $\%eax_i$ is the contents of %eax in thread i's context

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

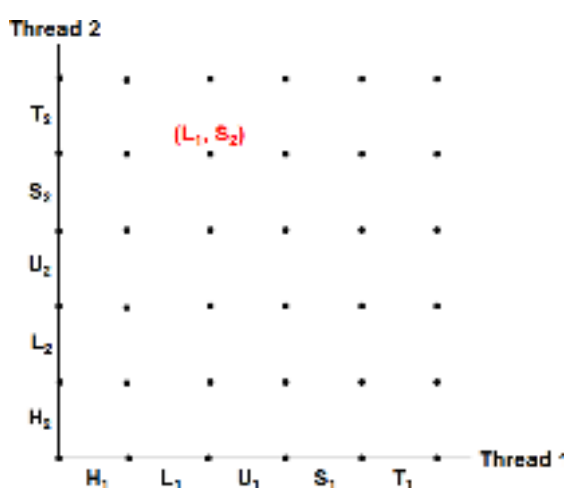
- How about this ordering?

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1			
1	L_1			
2	H_2			
2	L_2			
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

We can clarify our understanding of concurrent execution with the help of the progress graph

Progress Graphs

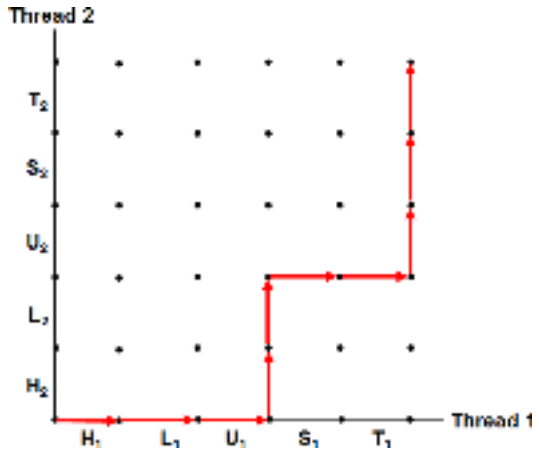
- A progress graph depicts the discrete execution state space of concurrent threads.
- Each axis corresponds to the sequential order of instructions in a thread.
- Each point corresponds to a possible execution state ($Inst_1$, $Inst_2$).
- E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.



Trajectories in Progress Graphs

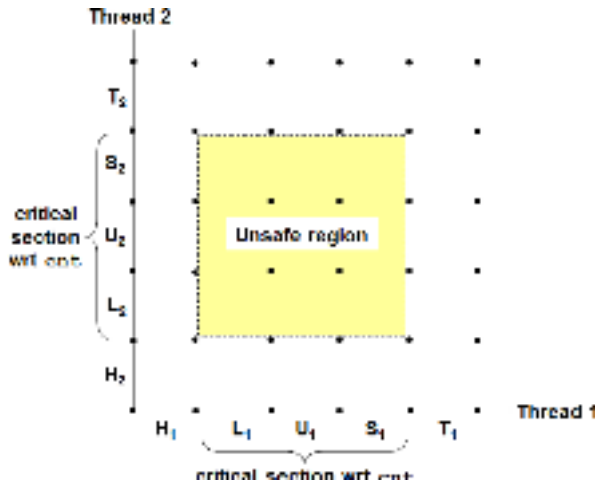
- A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:
H1, L1, U1, H2, L2,
S1, T1, U2, S2, T2



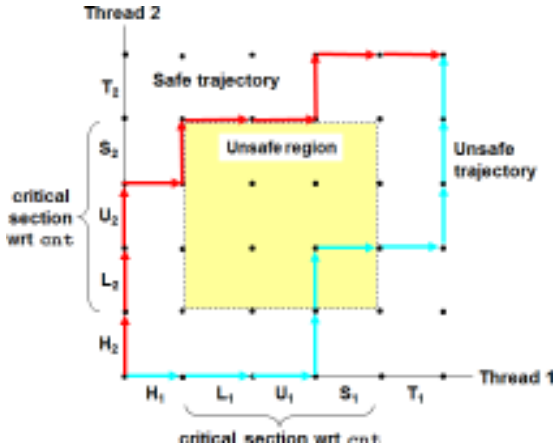
Critical Sections and Unsafe Regions

- L, U, and S form a critical section with respect to the shared variable cnt.
- Instructions in critical sections (wrt to some shared variable) should not be interleaved.
- Sets of states where such interleaving occurs form unsafe regions.



Safe and Unsafe Trajectories

- Def: A trajectory is safe iff it doesn't touch any part of an unsafe region.
- Claim: A trajectory is correct (wrt cnt) iff it is safe.



The classic example

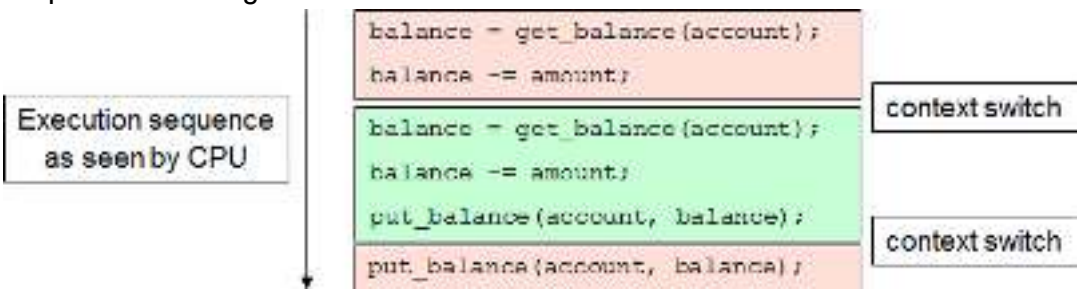
- Suppose we have to implement a function to withdraw money from a bank account:
  int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
  }
- Now suppose a husband and wife share a bank account with a balance of $100.00
  - ✓ what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?
- Represent the situation by creating a separate thread for each person to do the withdrawals
  - ✓ have both threads run on the same bank mainframe:

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



- What's the account balance after this sequence?
  - ✓ who's happy, the bank or you?
- How often is this unfortunate sequence likely to occur?

Race conditions and concurrency

- Atomic operation: operation always runs to completion, or not at all. Indivisible, can't be stopped in the middle. On most machines, memory reference and assignment (load and store) of words, are atomic.
- Many instructions are not atomic. For example, on most 32-bit architectures, double precision floating point store is not atomic; it involves two separate memory operations.

The crux of the problem

- The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
  - ✓ creates a **race condition**
    - output is non-deterministic, depends on timing
- We need mechanisms for controlling access to shared resources in the face of concurrency
  - ✓ so we can reason about the operation of programs
    - essentially, re-introducing determinism
- Synchronization is necessary for any shared data structure
  - ✓ buffers, queues, lists, hash tables, scalars, …

Synchronization related definitions

- Mutual exclusion: ensuring that only one thread does a particular thing at a time. One thread doing it excludes all others.
- Critical section: piece of code that only one thread can execute at one time. All other threads are forced to wait on entry. when a thread leaves a critical section, another can enter. Mutual exclusion is required inside the critical section.
- Key idea -- all synchronization involves waiting

Critical section solution requirements

- Critical sections have the following requirements
  - ✓ mutual exclusion
    - at most one thread is in the critical section
  - ✓ progress
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - ✓ bounded waiting (no starvation)
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - ➢ assumes threads eventually leave critical sections
  - ✓ performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

Synchronization: Too Much Milk

Person A	Person B
3:00 Look in fridge. Out of milk.	
3:05 Leave for store.	
3:10 Arrive at store.	Look in fridge. Out of milk.
3:15 Buy milk.	Leave for store.
3:20 Arrive home, put milk in fridge	Arrive at store.
3:25	Buy milk.
3:30	Arrive home, put milk in fridge.
Oops!! Too much milk	

Too much milk: Solution 1

- Too Much Milk: Solution #1
- What are the correctness properties for the too much milk problem?
- Never more than one person buys; someone buys if needed.
- Restrict ourselves to only use atomic load and store operations as building blocks.
- Solution #1:

```
if (noMilk) {
  if (noNote){
      leave Note;
      buy milk;
      remove note;
  }
}
```

- Why doesn't this work?

Too much milk: Solution 2

Actually, solution 1 makes problem worse -- fails only occasionally. Makes it really hard to debug. Remember, constraint has to be satisfied, independent of what the dispatcher does -- timer can go off, and context switch can happen at any time.

Solution #2:

```
Thread A                Thread B
leave note A            leave note B
if (no Note from B)     if (no Note from A)
                        {
{
    if (noMilk)             if(noMilk)
      buy Milk                  buy Milk;
}                         }
Remove Note A           remove Note B
```

Lecture No. 10

Overview of today's lecture

- Concurrency examples (cont'd from previous lecture)
- Locks
- Implementing locks with disabling interrupts
- Implementing locks with busy waiting
- Implementing locks with test and set like low-level hardware instructions
- Semaphores–Introduction and definition
- Re-cap of lecture

Too Much Milk Solution 3

```
Thread A                Thread B
leave note A            leave note B
while (note B)// X       if (no note A) { //Y
  do nothing;                if( no Milk)
    if (noMilk)                  buy milk
      buy milk;             }
    remove note A           remove note B
```

Does this work? Yes. Can guarantee at X and Y that either
(i) safe for me to buy (ii) other will buy, ok to quit

At Y: if noNote A, safe for B to buy (means A hasn't started yet); if note A, A is either buying, or waiting for B to quit, so ok for B to quit

At X: if no note B, safe to buy;
if note B, don't know, A hangs around.
Either: if B buys, done; if B doesn't buy, A will.

Too Much Milk Summary

- Solution #3 works, but it's really unsatisfactory:
    1. Really complicated -- even for this simple an example, hard to convince yourself it really works
    2. A's code different than B's -- what if lots of threads? Code would have to be slightly different for each thread.
    3. While A is waiting, it is consuming CPU time (busywaiting)
- There's a better way: use higher-level atomic operations; load and store are too primitive.

Locks

- Lock: prevents someone from doing something.
    1. Lock before entering critical section, before accessing shared data
    2. unlock when leaving, after done accessing shared data
    3. wait if locked
        - Lock::Acquire -- wait until lock is free, then grab it
        - Lock::Release -- unlock, waking up a waiter if any
- These must be atomic operations -- if two threads are waiting for the lock, and both see it's free, only one grabs it! With locks, the too much milk problem becomes really easy!

```
lock->Acquire();
if (nomilk)
   buy milk;
lock->Release();
```

Ways of implementing locks

- All require some level of hardware support
- Directly implement locks and context switches in hardware Implemented in the Intel 432.
- Disable interrupts (uniprocessor only) Two ways for dispatcher to get control:
- internal events -- thread does something to relinquish the CPU
- external events -- interrrupts cause dispatcher to take CPU away
- On a uniprocessor, an operation will be atomic as long as a context switch does not occur in the middle of the operation. Need to prevent both internal and external events. Preventing internal events is easy.

Prevent external events by disabling interrupts, in effect, telling the hardware to delay handling of external events until after we're done with the atomic operation.

A flawed, but very simple implementation

```
Lock::Acquire() { disable interrupts;}
Lock::Release() { enable interrupts; }
```

1. Critical section may be in user code, and you don't want to allow user code to disable interrupts (might never give CPU back!). The implementation of lock acquire and release would be done in the protected part of the operating system, but they could be called by arbitrary user code.
2. Might want to take interrupts during critical section. For instance, what if the lock holder takes a page fault? Or does disk I/O?
3. Many physical devices depend on real-time constraints. For example, keystrokes can be lost if interrupt for one keystroke isn't handled by the time the next keystroke occurs. Thus, want to disable interrupts for the shortest time possible. Critical sections could be very long running.

Busy-waiting implementation

```
class Lock {
        int value = FREE;

        Lock::Acquire() {
          Disable interrupts;
          while (value != FREE) {
            Enable interrupts; // allow interrupts
            Disable interrupts;
          }
        value = BUSY;
        Enable interrupts;
        }
        Lock::Release() {
          Disable interrupts;
          value = FREE;
          Enable interrupts;
        }
}
```

Problem with busy waiting

- Thread consumes CPU cycles while it is waiting. Not only is this inefficient, it could cause problems if threads can have different priorities. If the busy-waiting thread has higher priority than the thread holding the lock, the timer will go off, but (depending on the scheduling policy), the lower priority thread might never run.
- Also, for semaphores and monitors, if not for locks, waiting thread may wait for an arbitrary length of time. Thus, even if busy-waiting was OK for locks, it could be very inefficient for implementing other primitives.

Implementing without busy-waiting (1)

```
        Lock::Acquire()
        {
          Disable interrupts;
          while (value != FREE) {
                put on queue of threads waiting for lock
                 change state to sleeping or blocked
          }
          value = BUSY;
          Enable interrupts;
        }
        Lock::Release()
          {
            Disable interrupts;
            if anyone on wait queue {
                    take a waiting thread off
```

```
                    put it on ready queue
                change its state to ready
        }
    value = FREE;
    Enable interrupts;
}
```

Implementing without busy-waiting (2)

- When does Acquire re-enable interrupts :
- In going to sleep?
- Before putting the thread on the wait queue?
- Then Release can check queue, and not wake the thread up.
- After putting the thread on the wait queue, but before going to sleep?
- Then Release puts thread on the ready queue, When thread wakes up, it will go to sleep, missing the wakeup from Release.
- In other words, putting the thread on wait queue and going to sleep must be done atomically before re-enabling interrupts

Atomic read-modify-write instructions

- On a multiprocessor, interrupt disable doesn't provide atomicity.
- Every modern processor architecture provides some kind of atomic read-modify-write instruction. These instructions atomically read a value from memory into a register, and write a new value. The hardware is responsible for implementing this correctly on both uniprocessors (not too hard) and multiprocessors (requires special hooks in the multiprocessor cache coherence strategy).
- Unlike disabling interrupts, this can be used on both uniprocessors and multiprocessors.
- Examples of read-modify-write instructions:
  - ✓ test&set (most architectures) -- read value, write 1 back to memory
  - ✓ exchange (x86) -- swaps value between register and memory
  - ✓ compare&swap (68000) -- read value, if value matches register, do exchange

Implementing locks with test&set

Test&set reads location, sets it to 1, and returns old value.

```
Initially, lock value = 0;
Lock::Acquire {
        while (test&set(value) == 1)
        ; // Do nothing
        }
Lock::Release {
value = 0;
}
```

- If lock is free, test&set reads 0 and sets value to 1, so lock is now busy. It returns 0, so Acquire completes. If lock is busy, test&set reads 1 and sets value to 1 (no change), so lock stays busy, and Acquire will loop. This is a busy-wait loop, but as with the discussion above about disable interrupts, you can modify it to sleep if lock is BUSY.

Semaphores

- semaphore = a synchronization primitive
  - ✓ higher level than locks
  - ✓ invented by Dijkstra in 1968, as part of the THE OS
- A semaphore is:
  - ✓ a variable that is manipulated atomically through two operations, signal and wait
  - ✓ wait(semaphore): decrement, block until semaphore is open also called P(), after Dutch word for test, also called down()
  - ✓ signal(semaphore): increment, allow another to enter also called V(), after Dutch word for increment, also called up()

Blocking in Semaphores

- Each semaphore has an associated queue of processes/threads
  - when wait() is called by a thread,
    if semaphore is ¯available‖, thread continues
    if semaphore is ¯unavailable‖, thread blocks, waits on queue
  - signal() opens the semaphore
    if thread(s) are waiting on a queue, one thread is unblocked
    if no threads are on the queue, the signal is remembered for next time a wait() is called
- In other words, semaphore has history
  - this history is a counter
  - if counter falls below 0 (after decrement), then the semaphore is closed
- wait decrements counter
- signal increments counter

A pseudocode implementation

```
type semaphore = record
    value: integer;
    L: list of processes;
end

wait(S):
    S.value = S.value - 1;
    if S.value < 0
    then begin
            add this process to S.L;
            block;
            end;

signal(S):
    S.value = S.value + 1;
    if S.value <= 0
    then begin
            remove a process P from S.L;
            wakeup P
            end;
```

wait()/signal() are critical sections! Hence, they must be executed atomically with respect to each other.

Two types of Semaphores

- Binary semaphore (aka mutex semaphore)
  - ✓ guarantees mutually exclusive access to resource
  - ✓ only one thread/process allowed entry at a time
  - ✓ counter is initialized to 1
- Counting semaphore (aka counted semaphore)
  - ✓ represents a resources with many units available
  - ✓ allows threads/process to enter as long as more units are available
  - ✓ counter is initialized to N
- N = number of units available
- Only operations are P and V -- can't read or write value, except to set it initially
- Operations must be atomic -- two P's that occur together can't decrement the value below zero.

Safe Sharing with Semaphores

- Here is how we would use P and V operations to synchronize the threads that update cnt.

```
/* Semaphore s is initially 1 */

/* Thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(s);
        cnt++;
        V(s);
    }
    return NULL;
}
```

Lecture No. 11

Overview of today's lecture

- Producer consumer problem with a bounded buffer
- Solving producer consumer problem using locks and semaphores
- Semaphores used for two purposes
- Condition variables and monitors–introduction
- Condition variables–definition
- Hoare and Mesa style monitors
- Recap of lecture

Producer-consumer with a bounded buffer

- Problem definition
- Producer puts things into a shared buffer, consumer takes them out. Need synchronization for coordinating producer and consumer.
- Multimedia processing:Producer creates MPEG video frames, consumer renders the frames
- Event-driven graphical user interface: Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer. Consumer retrieves events from buffer and paints the display.
- Don't want producer and consumer to have to operate in lockstep, so put a fixed-size buffer between them; need to synchronize access to this buffer. Producer needs to wait if buffer is full; consumer needs to wait if buffer is empty.
- Solution uses semaphores for both mutex and scheduling.

Correctness constraints for solution

1. Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
2. Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
3. Only one thread can manipulate buffer queue at a time (mutual exclusion)

Use a separate semaphore for each constraint;
Note semaphores being used in multiple ways.
Semaphore fullBuffers;
// if 0, no item in buffer
Semaphore emptyBuffers;
// if 0, nowhere to put more item
Semaphore mutex; // mutual exclusion

Semaphore solution

- Semaphore fullBuffers = 0 // initially, no item!
- Semaphore emptyBuffers = numBuffers;
- // initially, number of empty slots
- // semaphore used to count how many
- // resources there are!
- Semaphore mutex = 1; // no one using the buffer

```
Producer() {
        emptyBuffers.P();
        mutex.P(); // make sure no one else is accessing the buffer
        put 1 item in buffer
        mutex.V(); // ok for others to use buffer
        fullBuffers.V();
}
Consumer() {
        fullBuffers.P();
        mutex.P(); // make sure no one else is accessing the buffer
        take 1 item out;
        mutex.V(); // next person's turn
        emptyBuffers.V();
}
```

Questions

- Why does producer does P & V operations on different semaphores than the consumer?
- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers? Do we need to change anything?

Two uses of semaphores

- **Mutual exclusion**
  - ✓ When semaphores are used for mutual exclusion, the semaphore has an initial value of 1, and P() is called before the critical section, and V() is called after the critical section.
  - ✓ semaphore->P();
  - ✓ // critical section goes here
  - ✓ semaphore->V();
- **Scheduling constraints**
  - ✓ Semaphores can also be used to express generalized scheduling constraints -- in other words, semaphores provide a way for a thread to wait for something. Usually, in this case, the initial value of the semaphore is 0, but not always!
  - ✓ For example, you can implement Thread‘s join (reaping) using semaphores:
  - ✓ Initial value of semaphore = 0
  - ✓ ThreadJoin calls P
  - ✓ ThreadFinish calls V

Monitor Definition

a lock and zero or more condition variables for managing concurrent access to shared data

Note: Tanenbaum and Silberschatz both describe monitors as a programming language construct, where the monitor lock is acquired automatically on calling any procedure in a C++ class, for example. No widely-used language actually does this, however! So in many real-life operating systems, such as Windows, Linux, or Solaris, monitors are used with explicit calls to locks and condition variables.

Condition variables

A simple example:

```
AddToQueue() {
        lock.Acquire(); // lock before using shared data
        put item on queue; // ok to access shared data
        lock.Release(); // unlock after done with shared data
}
RemoveFromQueue() {
        lock.Acquire(); // lock before using shared data
        if something on queue // ok to access shared data
        remove it;
        lock.Release(); // unlock after done with shared data
        return item;
}
```

- How do we change RemoveFromQueue to wait until something is on the queue? Logically, want to go to sleep inside of critical section, but if we hold lock when going to sleep, other threads won't be able to get in to add things to the queue, to wake up the sleeping thread. Key idea with condition variables: make it possible to go to sleep inside critical section, by atomically releasing lock at same time we go to sleep
- Condition variable: a queue of threads waiting for something inside a critical section.
- Condition variables support three operations:
- Wait() -- release lock, go to sleep, re-acquire lock
  Note: Releasing lock and going to sleep is atomic
- Signal() -- wake up a waiter, if any
- Broadcast() -- wake up all waiters
- Rule: must hold lock when doing condition variable operations

A synchronized queue, using condition variables:

```
AddToQueue() {
        lock.Acquire();
        put item on queue;
        condition.signal();
        lock.Release();
}
RemoveFromQueue() {
        lock.Acquire();
        while nothing on queue
        condition.wait(&lock);// release lock; go to sleep; re-acquire lock
        remove item from queue;
        lock.Release();
        return item;
}
```

Mesa vs. Hoare monitors

- Need to be careful about the precise definition of signal and wait.
- **Mesa-style:** (most real operating systems) Signaller keeps lock and processor. Waiter simply put on ready queue, with no special priority. (in other words, waiter may have to wait for lock)
- **Hoare-style:** (most textbooks)
- Signaller gives up lock, CPU to waiter; waiter runs immediately Waiter gives lock and processor back to signaller when it exits critical section or if it waits again.

Readers/Writers

- **Motivation**
  Shared database (for example, bank balances, or airline seats)
  Two classes of users:
  Readers -- never modify database
  Writers -- read and modify database
  Using a single lock on the database would be overly restrictive.
  Want:
  many readers at same time
  only one writer at same time

Lecture No. 12

Overview of today's lecture

- Readers/writers problem
- Solving readers/writers problem using condition variables
- Pros and cons of the solution
- Duality of synchronization primitives
- Implementing condition variables using semaphores as building blocks
- Thread safety and reentrant functions
- Ways to solve thread un-safety problem of library functions
- Thread un-safe functions in C library
- Recap of lecture

Readers/Writers

Constraints

1. Readers can access database when no writers (Condition okToRead)
2. Writers can access database when no readers or writers (Condition okToWrite)
3. Only one thread manipulates state variables at a time.

- Basic structure of solution
- Reader
- wait until no writers
- access database
- check out -- wake up waiting writer
- Writer
- wait until no readers or writers
- access database
- check out -- wake up waiting readers or writer
- State variables:
  - ✓ # of active readers -- AR = 0
  - ✓ # of active writers -- AW = 0
  - ✓ # of waiting readers -- WR = 0
  - ✓ # of waiting writers -- WW = 0
- Condition okToRead = NIL
- Condition okToWrite = NIL
- Lock lock = FREE

- Code:

```
Reader() {
lock.Acquire();
while ((AW + WW) > 0) { // check if safe to read if any writers, wait
        WR++;
        okToRead.Wait(&lock);
        WR--;
}
```

```
            AR++;
            lock.Release();
            Access DB
            lock.Acquire();
            AR--;
            If (AR == 0 && WW > 0)//if no other readers still
            // active, wake up writer
            okToWrite.Signal(&lock);
            lock.Release();
    }
```

- Writer() { // symmetrical

```
    lock.Acquire();
    while ((AW + AR) > 0)          // check if safe to write
     {
            // if any readers or writers, wait
            WW++;
            okToWrite->Wait(&lock);
            WW--;
     }
    AW++;
    lock.Release();
    Access DB
    // check out
    lock.Acquire();
    AW--;
    if (WW > 0) // give priority to other writers
    okToWrite->Signal(&lock);
    else if (WR > 0)
    okToRead->Broadcast(&lock);
    lock.Release();
    }
```

Questions

1. Can readers or writers starve? Who and Why?
2. Why does checkRead need a while?

semaphores and monitors

Illustrate the differences by considering: can we build monitors out of semaphores? After all, semaphores provide atomic operations and queueing.

Does this work?

```
Wait() { semaphore - > P(); }
Signal() { semaphore - > V(); }
```

Condition variables only work inside of a lock.. Does this work?

```
Wait(Lock *lock) {
        lock->Release();
        Semaphore - > P();
        Lock - > Acquire();
}
Signal() {
        Semaphore - > V();
}
```

- What if thread signals and no one is waiting? No op.
- What if thread later waits? Thread waits.
- What if thread V's and no one is waiting? Increment.
- What if thread later does P? Decrement and continue.
- In other words, P + V are commutative -- result is the same no matter what order they occur.
- Condition variables are NOT commutative. That's why they must be in a critical section -- need to access state variables to do their job.
- Does this fix the problem?

```
        Signal() {
                if semaphore queue is not empty
                semaphore->V();
        }
```

- For one, not legal to look at contents of semaphore queue. But also: race condition -- signaller can slip in after lock is released, and before wait. Then waiter never wakes up! Need to release lock and go to sleep atomically. Is it possible to implement condition variables using semaphores? Yes!!!

```
Semaphore mutex = 1; // This lock is outside of the condition object
Condition {
        Semaphore lock = 1;
        Seamphore waitSem = 0;
        Int numWaiters = 0;
    }
```

```
wait(cond, mutex)
{
  P(cond.lock);
     cond.numWaiters++;
  V(cond.lock);
 V(mutex);
  P(cond.waitSem);
  P(cond.lock);
     cond.numWaiters - -;
V(cond.lock);
  P(mutex);
}
```

```
signal (cond, mutex)
 {
  P(cond.lock);
  if (cond.numWaiters > 0)
   {
        V(cond. waitSem);

   }

      V(cond.lock);
}
```

Thread Safety

- Functions called from a thread must be *thread-safe*.
- We identify four (non-disjoint) classes of thread-unsafe functions:
  - ✓ Class 1: Failing to protect shared variables.
  - ✓ Class 2: Relying on persistent state across invocations.
  - ✓ Class 3: Returning a pointer to a static variable.
  - ✓ Class 4: Calling thread-unsafe functions.

Thread-Unsafe Functions

- Class 1: Failing to protect shared variables.
  - ✓ Fix: Use P and V semaphore operations.
  - ✓ Issue: Synchronization operations will slow down code.
- Class 3: Returning a ptr to a static variable.
- Fixes:
  1. Rewrite code so caller passes pointer to struct.
     - Issue: Requires changes in caller and callee.

```
struct hostent

*gethostbyname(char name)
{
        static struct hostent h;
        <contact DNS and fill in h>
        return &h;
}
```

2. Lock-and-copy
   - Issue: Requires only simple changes in caller (and none in callee)
   - However, caller must free memory.

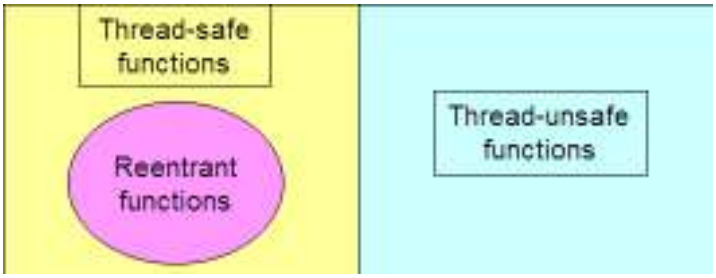
```
hostp = Malloc(...));
gethostbyname_r(name, hostp);

struct hostent
*gethostbyname_ts(char *p)
{
  struct hostent *q = Malloc(...);
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p; /* copy */
  V(&mutex);
  return q;
}
```

Thread-Unsafe Functions

- Class 4: Calling thread-unsafe functions.
  - ✓ Calling one thread-unsafe function makes an entire function thread-unsafe.
  - ✓ Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- A function is *reentrant* iff it accesses NO shared variables when called from multiple threads.
  - ✓ Reentrant functions are a proper subset of the set of thread-safe functions
  - ✓ The fixes to Class 2 and 3 thread – unsafe functions require modifying the function to make it reentrant.



Thread-Safe Library Functions

- All functions in the Standard C Library are thread-safe.
  - ✓ Examples: malloc, free, printf, scanf
- Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r

Lecture No. 13

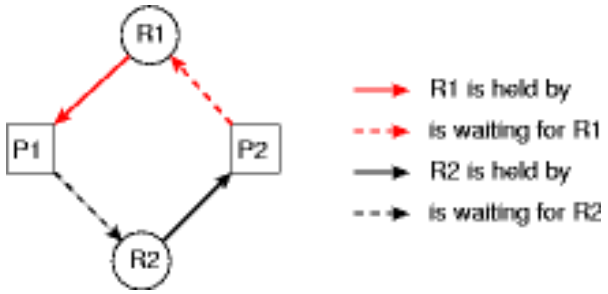
Overview of today's lecture

- Deadlocks
  - ✓ Definition
  - ✓ Four necessary and sufficient conditions
  - ✓ Examples
  - ✓ Detection
  - ✓ Avoidance
  - ✓ Prevention
  - ✓ Current practice

Formal definition of a deadlock

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is release of a currently held resource
- None of the processes can …
  - ✓ run
  - ✓ release resources
  - ✓ be awakened

Resource graph



Conditions for deadlock

Without **all** of these, can't have deadlock:

1. Mutual exclusion (for example: mutex on bounded buffer)
2. No preemption (if someone has resource, can't take it away)
3. Hold and wait
4. Circular wait

Deadlock example

- Given two threads, what sequence of calls cause the following to deadlock?

```
/* transfer x dollars from a to b */
void transfer(account *a, account *b, int x) {
        P (a - > sema);
        P (b - > sema):
        a - > balance += x;
        b - > balance -= x;
        V (a - > sema);
        V ( b - > sema);
}
```

Deadlocks don't require synch primitives

- Does deadlock require locks? No. Just circular constraints.
  - ✓ Example: consider two threads that send and receive data to each other using two circular buffers (buf size = 4096 bytes). Recall: a full buffer causes the sender to block until the receiver removes data.

```
T1:                          T2:
send n bytes to T2           while(receive 4K of data)
                              ;
while(receive data)          process  data
      display data           send 4K result to T1
  exit                       exit
```

Solutions to Deadlock

Detect deadlock and fix

- scan graph
- detect cycles
- fix them // this is the hard part!
- a) Shoot thread, force it to give up resources.
- This isn't always possible -- for instance, with a mutex, can't shoot a thread and leave world in an inconsistent state.
- b) Roll back actions of deadlocked threads (transactions)
- Common technique in databases

Transactions and two phase commit

- Acquire all resources, if block on any, release all, and retry
- Printfile:
  lock-file
  lock-printer
  lock-disk
  do work
  release all
- Pro: dynamic, simple, flexible
- Con:
  - ✓ cost with number of resources?
  - ✓ length of critical section?
  - ✓ hard to know what's needed a priori

Preventing deadlock

- Need to get rid of one of the four conditions.
  a) No sharing -- totally independent threads.
  b) Preempt resources
  c) Make all threads request and reserve everything they'll need at beginning
- Problem is -- predicting future is hard, tend to over-estimate resource needs (inefficient)

Banker's algorithm

- More efficient than reserving all resources on startup
- State maximum resource needs in advance
- Allocate resources dynamically when resource is needed
- Wait if granting request would lead to deadlock (request can be granted if some sequential ordering of requests is deadlock free)
- Bankers algorithm allows the sum of maximum resource needs of all current threads to be greater than the total resources, as long as there is some way for all the threads to finish without getting into deadlock.
- For example, you can allow a thread to proceed if the total available resources - # allocated >= max remaining that might be needed by this thread.
- In practice, Banker‘s algorithm is rarely used since ir requires predicting maximum resource requirements in advance.

Resource ordering

- Make everyone use the same ordering in accessing resources.
- For example, all threads must grab semaphores in the same order

Current practice

- Microsoft SQL Server
  - ✓ ¯The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock.‖
- Oracle
  - ✓ As Microsoft SQL Server, plus ¯Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order... For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table.‖
- Windows internals (Linux no different)
  - ✓ ¯Unless they did a huge change in Vista, the NT kernel architecture is a deadlock minefield. With the multi-threaded re-entrant kernel there is plenty of deadlock potential.‖
  - ✓ ¯Lock ordering is great in theory, and NT was originally designed with mutex levels, but they had to be abandoned. Inside the NT kernel there is a lot of interaction between memory management, the cache manager, and the file systems, and plenty of situations where memory manager acquires its lock and then calls the cache manager. This happens while the file system calls the cache manager to fill the cache which in turn goes through the memory manager to fault in its page. And the list goes on.‖

Lecture No. 14

Overview of today's lecture

- Thread usage paradigms
- Paper by Hauser et al.
- Pros and Cons of different paradigms

Uses of threads

- To exploit CPU parallelism
  - ✓ Run two CPUs at once in the same program
- To exploit I/O parallelism
  - ✓ Run I/O while computing, or do multiple I/O
  - ✓ Listen to the ¯window‖ while also running code, e.g. allow commands during an interactive game
- For program structuring
  - ✓ E.g., timers

Paradigms of thread usage (from Hauser et al paper)

- Defer work
- General pumps
- Slack processes
- Sleepers
- One-shots
- Deadlock avoidance
- Rejuvenation
- Serializers
- Encapsulated fork
- Exploiting parallelism

Defer work.

- A very common scenario for thread usage
- Client may see unexpected response something client requested is fired off to happen in the background
  - ✓ Forking off examples
    - a document print operation
    - Updating a window
    - Sending an email message
  - ✓ Issue? What if thread hangs for some reason. Client may see confusing behavior on a subsequent request!

Pumps

- Components of producer-consumer pipelines that take input in, operate on it, then output it ¯downstream‖
- Value is that they can absorb transient rate mismatches
- *Slack process*: a pump used to explicitly add delay, employed when trying to group small operations into batches

Sleepers, one-shots

- These are threads that wait for some event, then trigger, then wait again
- Examples:
  - ✓ Call this procedure every 20ms, or after some timeout
  - ✓ Can think of device interrupt handler as a kind of sleeper thread

Deadlock avoiders

- Thread created to perform some action that might have blocked, launched by a caller who holds a lock and doesn't want to wait

Bohr-bugs and Heisenbugs

- Bruce Lindsey, refers to models of the atom
- A Bohr nucleas was a nice solid little thing. Same with a Bohr-bug. You can hit it reproducibly and hence can fix it
- A Heisenbug is hard to pin down: if you localize an instance, the bug shifts elsewhere. Results from non-deterministic executions, old corruption in data structures, etc….

Task rejuvenation

- A nasty style of thread
  - ✓ Application had multiple major subactivities, such as input handler, renderer.
  - ✓ Something awful happened.
  - ✓ So create a new instance and pray

Others

- Serializers: a queue, and a thread that removes work from it and processes that work item by item
  - ✓ for (;;)

```
{
get_next_event();
handle_event();
}
```

- Concurrency exploiters: for multiple CPUs
- Encapsulated forks: Hidden threads used in library packages

Lecture No. 15

Threads considered marvelous

- Threads are wonderful when some action may block for a while
  - ✓ Like a slow I/O operation, RPC, etc
- Your code remains clean and ¯linear‖
- Moreover, aggregated performance is often far higher than without threading

Threads considered harmful

- They are fundamentally non-deterministic, hence invite Heisenbugs
- Reentrant code is really hard to write
- Surprising scheduling can be a huge headache
- When something ¯major‖ changes the state of a system, cleaning up threads running based on the old state is a pain

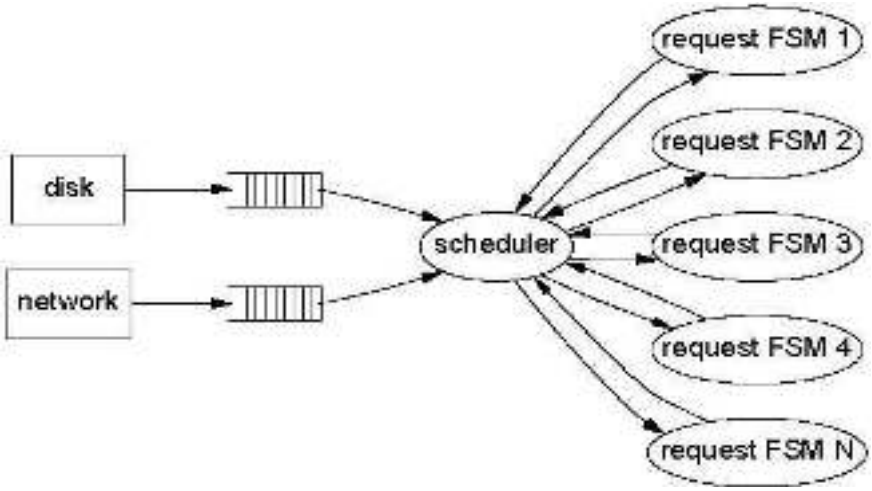
Classic issues

- Threads that get forked off, then block for some reason
  - ✓ Address space soon bloats due to number of threads increasing beyond a threshold, causing application to crash
- Erratic use of synchronization primitives. Program is incredibly hard to debug, some problems seen only now and then
- As threads grow in number, synchronization overhead becomes significant. Semaphore and lock queues become very large due to waiting threads

Bottom line?

- Concurrency bugs are incredibly common and very hard to track down and fix
- Programmers find concurrency unnatural
  - ✓ Try to package it better?
  - ✓ Identify software engineering paradigms that can ease the task of gaining high performance

Event-oriented paradigm



Classic solution?

- Go ahead and build a an event driven application, and use threads as helpers
- Connect each ¯external event source‖ to a main hand-crafted ¯event monitoring routine‖
  - ✓ Often will use signals or a kernel supplied event notification mechanism to detect that I/O is available
  - ✓ Then package the event as an *event object* and put this on a queue. Kick off the event scheduler if it was asleep
  - ✓ Scheduler de-queues events, processes them one by one… forks lightweight threads as needed (when blocking I/O is required).
  - ✓ Flash web server built on this paradigm

Problems with the architecture?

- Only works if all the events show up and none is missed
- Depends on OS not ¯losing‖ events. In other words event notification mechanism must be efficient, scalable
- Scheduler needs a way to block when no work to do and must be sure that event notification can surely wake it up
- Common event notification mechanism in Linux, Windows etc. Select() and poll()
- New much more efficient and scalable mechanisms:
- epoll (2.6 and above Linux kernel only)
- IOCompletionPorts in Windows NT, XP etc.

Goals of "the perfect scheduler"

- Minimize latency: metric = response time (user time scales ~50-150millisec) or job completion time
- Maximize throughput: Maximize #of jobs per unit time.
- Maximize utilization: keep CPU and I/O devices busy. Recurring theme with OS scheduling
- Fairness: everyone gets to make progress, no one starves

Problem cases

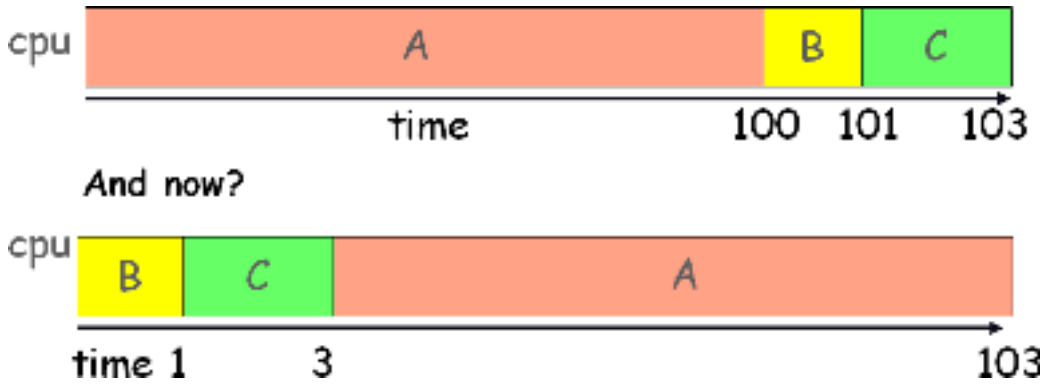
- I/O goes idle because of blindness about job types
- Optimization involves favoring jobs of type ¯A‖ over ¯B‖. Lots of A's? B's starve.
- Interactive process trapped behind others. Response time worsens for no reason.
- Priorities: A depends on B. A's priority > B's. B never runs.

First come first served (FCFS or FIFO)

- Simplest scheduling algorithm:
  - ✓ Run jobs in order that they arrive
  - ✓ Uni-programming: Run until done (non-preemptive)
  - ✓ Multi-programming: put job at back of queue when blocks on I/O (we'll assume this)
  - ✓ Advantage: Simplicity

FCFS (2)

- Disadvantage: wait time depends on arrival order
- unfair to later jobs (worst case: long job arrives first)
- Example: three jobs (times: A=100, B=1, C=2) arrive nearly simultaneously - what's the average completion time?



FCFS Convoy effect

- A CPU bound job will hold CPU until done, or it causes an I/O burst (rare occurrence, since the thread is CPU-bound)
  - ✓ long periods where no I/O requests issued, and CPU held
  - ✓ Result: poor I/O device utilization
- Example: one CPU bound job, many I/O bound
  - ✓ CPU bound runs (I/O devices idle)
  - ✓ CPU bound blocks
  - ✓ I/O bound job(s) run, quickly block on I/O
  - ✓ CPU bound runs again
  - ✓ I/O completes
  - ✓ CPU bound still runs while I/O devices idle (continues…)

Lecture No. 16

Round robin (RR)

- Solution to job monopolizing CPU? Interrupt it.
  - ✓ Run job for some ¯time slice,‖ when time is up, or it blocks, it moves to back of a FIFO queue
  - ✓ most systems do some flavor of this
- Advantage:
  - ✓ fair allocation of CPU across jobs
  - ✓ low average waiting time when job lengths vary:

- What is average completion time?

Round Robin's Big Disadvantage

- Varying sized jobs are good, but what about same-sized jobs? Assume 2 jobs of time=100 each:

- Average completion time?
- How does this compare with FCFS for same two jobs?

RR Time slice tradeoffs

- Performance depends on length of the time-slice
  - ✓ Context switching isn't a free operation.
  - ✓ If time-slice is set too high (attempting to amortize context switch cost), you get FCFS. (i.e. processes will finish or block before their slice is up anyway)
  - ✓ If it's set too low you're spending all of your time context switching between threads.
  - ✓ Time-slice frequently set to ~50-100 milliseconds
  - ✓ Context switches typically cost 0.5-1 millisecond
- Moral: context switching is usually negligible (< 1% per time-slice in above example) unless you context switch too frequently and lose all productivity.

Priority scheduling

- Obvious: not all jobs equal
  - ✓ So: rank them.
- Each process has a priority
  - ✓ run highest priority ready job in system, round robin among processes of equal priority
  - ✓ Priorities can be static or dynamic (Or both: Unix)

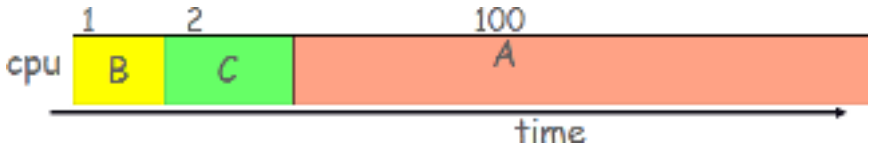
- ✓ Most systems use some variant of this
- Common use: couple priority to job characteristic
  - ✓ Fight starvation? Increase priority as time last ran
  - ✓ Keep I/O busy? Increase priority for jobs that often block on I/O
- Priorities can create deadlock.
  - ✓ Fact: high priority always runs over low priority.

Handling thread dependencies

- Priority inversion e.g., T1 at high priority, T2 at low
  - ✓ T2 acquires lock L.
  - ✓ Scene 1: T1 tries to acquire L, fails, spins. T2 never gets to run.
  - ✓ Scene 2: T1 tries to acquire L, fails, blocks. T3 enters system at medium priority. T2 never gets to run.
- Scheduling = deciding who should make progress
  - ✓ Obvious: a thread's importance should increase with the importance of those that depend on it.
  - ✓ Result: Priority inheritance

Shortest time to completion first (STCF)

- STCF (or shortest-job-first)
  - ✓ run whatever job has least amount of stuff to do
  - ✓ can be pre-emptive or non-pre-emptive
- Example: same jobs (given jobs A, B, C)
  - ✓ average completion = (1+3+103) / 3 = ~35 (vs ~100 for FCFS)



- Provably optimal: moving shorter job before longer job improves waiting time of short job more than harms waiting time for long job.

STCF Optimality Intuition

- consider 4 jobs, a, b, c, d, run in lexical order

  - ✓ the first (a) finishes at time a
  - ✓ the second (b) finishes at time a+b
  - ✓ the third (c) finishes at time a+b+c
  - ✓ the fourth (d) finishes at time a+b+c+d
  - ✓ therefore average completion = (4a+3b+2c+d)/4
  - ✓ minimizing this requires a <= b <= c <= d.

STCF – The Catch

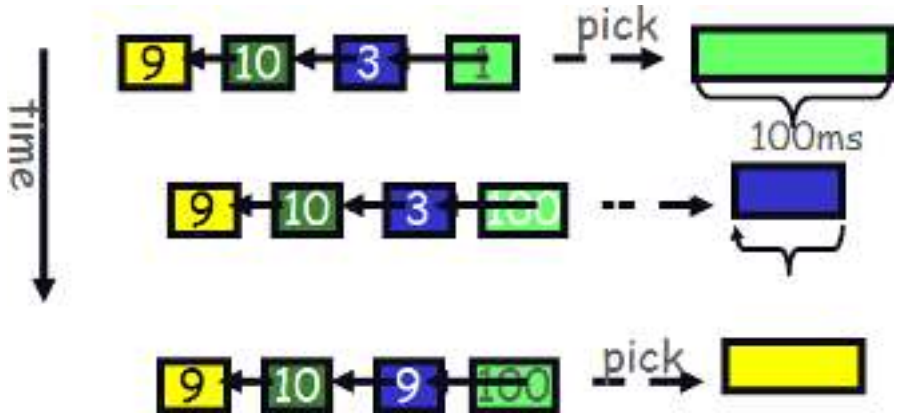
- This ¯Shortest to Completion First‖ OS scheduling algorithm sounds great! What's the catch?

How to know job length?

- Have user tell us. If they lie, kill the job.
  - ✓ Not so useful in practice
- Use the past to predict the future:
  - ✓ long running job will probably take a long time more
  - ✓ view each job as sequence of sequentially alternating CPU and I/O jobs. If previous CPU jobs in the sequence have run quickly, future ones will too (¯usually‖)
  - ✓ What to do if past != future?

Approximate STCF

- ~STCF: predict length of current CPU burst using length of previous burst
  - ✓ record length of previous burst (0 when just created)
  - ✓ At scheduling event (unblock, block, exit, …) pick smallest ¯past run length‖ off the ready Q

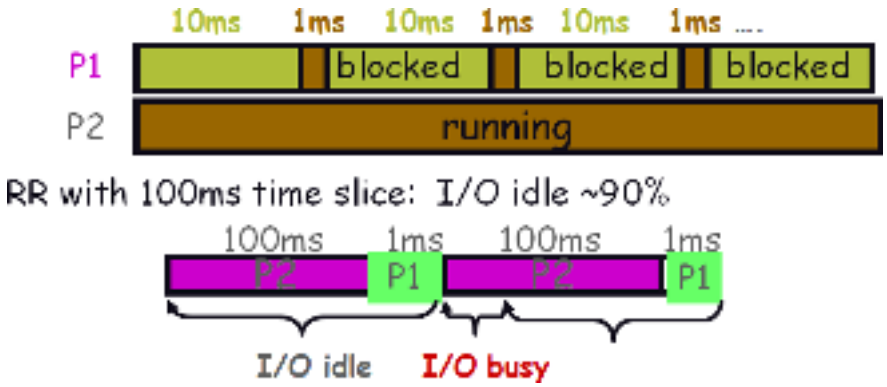


Practical STCF

- Disk: can predict length of next ¯job‖!
  - ✓ Job = Request from disk.
  - ✓ Job length ~ cost of moving disk arm to position of the requested disk block. (Farther away = more costly.)
- STCF for disks: shortest-seek-time-first (SSTF)
  - ✓ Do read/write request closest to current position
  - ✓ Pre-emptive: if new jobs arrive that can be serviced on the way, do these too.
- Problem:
  - ✓ Problem?
  - ✓ Elevator algorithm: Disk arm has direction, do closest request in that direction. Sweeps from one end to other

~STCF vs RR

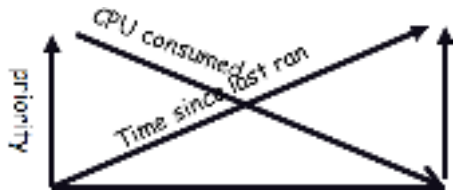
- Two processes P1, P2



  - ✓ 1ms time slice? RR would switch to P1 9 times for no reason (since it would still be blocked on I/O)
- ~STCF Offers better I/O utilization

Generalizing: priorities + history

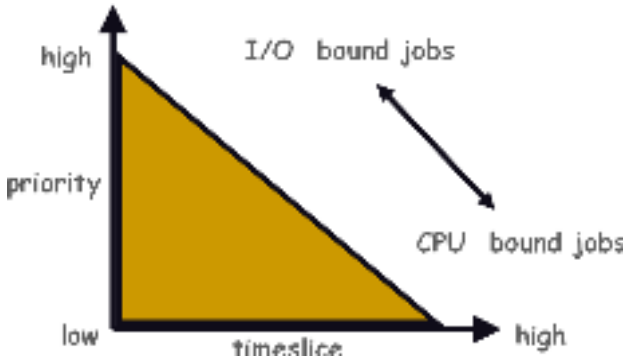
- ~STCF good core idea but doesn't have enough state
  - ✓ The usual STCF problem: starvation (when?)
  - ✓ Sol'n: compute priority as a function of both CPU time P has consumed and time since P last ran



- Multi-level feedback queue (or exponential Q)
  - ✓ Priority scheme where adjust priorities to penalize CPU intensive programs and favor I/O intensive
  - ✓ Pioneered by CTSS (MIT in 1962)

Visual aid of a multi-level system

- Priorities and time-slices change as needed depending on characteristic of process



Lecture No. 17

Some problems with multilevel queue concept

- Can‘t low priority threads starve?
  - ✓ Ad hoc: when skipped over, increase priority
- What about when past doesn‘t predict future?
  - ✓ e.g., CPU bound switches to I/O bound
  - ✓ Want past predictions to ¯age‖ and count less towards current view of the world.

Summary

- FIFO:
  - ✓ + simple
  - ✓ - short jobs can get stuck behind long ones; poor I/O
- RR:
  - ✓ + better for short jobs
  - ✓ - poor when jobs are the same length
- STCF:
  - ✓ + optimal (ave. response time, ave. time-to-completion)
  - ✓ - hard to predict the future
  - ✓ - unfair
- Multi-level feedback:
  - ✓ + approximate STCF
  - ✓ - unfair to long running jobs

Some Unix scheduling problems

- How does the priority scheme scale with number of processes?
- How to give a process a given percentage of CPU?
- OS implementation problem:
  - ✓ OS takes precedence over user process
  - ✓ user process can create lots of kernel work: e.g., many network packets come in, OS has to process. When doing a read or write system call, ….

Linux Scheduling

- Builds on traditional UNIX multi-level feedback queue scheduler by adding two new scheduling classes.
- Linux scheduling classes:
  - ✓ SCHED_FIFO: FCFS real-time threads
  - ✓ SCHED_RR: round-robin real-time threads
  - ✓ SCHED_OTHER: Other non-real-time threads
- Multiple priorities may be used within a class Priorities in real-time classes are higher than non-real-time classes.

- Rules for SCHED_FIFO
  1. The system will not interrupt a SCHED_FIFO thread except in the following cases:
     a. Another FIFO thread of higher priority becomes ready
     b. The executing FIFO thread blocks on I/O etc.
     c. The executing FIFO threads voluntarily gives up the CPU e.g. terminates or yields
  2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority
- SCHED_RR class is similar to SCHED_FIFO except that there is a time-slice associated with an RR thread. On expiry of the time slice, if the thread is still executing, it is pre-empted and another thread from either SCHED_FIFO or SCHED_RR is selected for execution.
- SCHED_OTHER class is managed by the traditional UNIX scheduling algorithms i.e. multi-level feedback queue.

Lottery scheduling: random simplicity

- Problem: this whole priority thing is really ad hoc.
  - ✓ How to ensure that processes will be equally penalized under load?
- Lottery scheduling! Very simple idea:
  - ✓ give each process some number of lottery tickets
  - ✓ On each scheduling event, randomly pick ticket
  - ✓ run winning process
  - ✓ to give process P n% of CPU, give it (total tickets)* n%
- How to use?
  - ✓ Approximate priority: low-priority, give few tickets, high-priority give many
  - ✓ Approximate STCF: give short jobs more tickets, long jobs fewer. Key: If job has at least 1, will not starve

Grace under load change

- Add or delete jobs (and their tickets):
  - ✓ affect all proportionally
- Example: give all jobs 1/n of cpu?
  - ✓ 4 jobs, 1 ticket each

  - ✓ each gets (on average) 25% of CPU.
  - ✓ Delete one job:

  - ✓ automatically adjusts to 33% of CPU!
- Easy priority donation:
  - ✓ Donate tickets to process you're waiting on.
  - ✓ Its CPU% scales with tickets of all waiters.

Classifications of Multiprocessor Systems

- Loosely coupled or distributed multiprocessor, or cluster
  - ✓ Each processor has its own memory and I/O channels
- Functionally specialized processors
  - ✓ Such as I/O processor
  - ✓ Controlled by a master processor
- Tightly coupled multiprocessing
  - ✓ Processors share main memory
  - ✓ Controlled by operating system

Granularity of parallelism

- Coarse and Very Coarse-Grained Parallelism
  - ✓ Synchronization among processes at a very gross level (after > 2000 instructions on the average)
  - ✓ Good for concurrent processes running on a multiprogrammed uniprocessor
  - ✓ Can by supported on a multiprocessor with little change

Granularity of parallelism

- Medium grained parallelism
  - ✓ Single application is a collection of threads
  - ✓ Threads usually interact frequently
- Fine-Grained Parallelism
  - ✓ Highly parallel applications
  - ✓ Specialized and fragmented area

Lecture No. 18

Assignment of Processes to Processors

- Treat processors as a pooled resource and assign process to processors on demand
- Permanently assign process to a processor
  - ✓ Known as group or gang scheduling
  - ✓ Dedicate short-term queue for each processor
  - ✓ Less overhead
  - ✓ Processor could be idle while another processor has a backlog
- Global queue
  - ✓ Schedule to any available processor
- Master/slave architecture
  - ✓ Key kernel functions always run on a particular processor
  - ✓ Master is responsible for scheduling
  - ✓ Slave sends service request to the master
  - ✓ Disadvantages
    - o Failure of master brings down whole system
    - o Master can become a performance bottleneck
- Peer architecture
  - ✓ Operating system can execute on any processor
  - ✓ Each processor does self-scheduling
  - ✓ Complicates the operating system

Process Scheduling

- Single queue for all processes
- Multiple queues are used for priorities
- All queues feed to the common pool of processors

Thread Scheduling

- Executes separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- Threads running on separate processors yields a dramatic gain in performance

Multiprocessor Thread Scheduling

- Dedicated processor assignment
  - ✓ Threads are assigned to a specific processor
- Dynamic scheduling
  - ✓ Number of threads can be altered during course of execution

Load Sharing

- Load is distributed evenly across the processors
- No centralized scheduler required
- Use global queues
- Most commonly used paradigm. Used by Linux, Windows and several UNIX flavors on multi-processor machines

Disadvantages of Load Sharing

- Central queue needs mutual exclusion
  - ✓ May be a bottleneck when more than one processor looks for work at the same time
- Preemptive threads are unlikely resume execution on the same processor
  - ✓ Cache use is less efficient
- If all threads are in the global queue, all threads of a program will not gain access to the processors at the same time

Gang Scheduling

- Simultaneous scheduling of threads that make up a single process
- Useful for applications where performance severely degrades when any part of the application is not running
- Threads often need to synchronize with each other

Dedicated Processor Assignment

- When application is scheduled, its threads are assigned to processors using a graph theoretic optimization algorithm on the control flow graph (CFG) of the application where each node in the CFG is a thread
- Disadvantages
  - ✓ Some processors may be idle
  - ✓ No multiprogramming of processors

Real-Time Systems

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
- Tasks or processes attempt to control or react to events that take place in the outside world
- These events occur in ¯real time‖ and tasks must be able to keep up with them

Hard real-time systems

- If deadlines are not met, results are catastrophic
- Examples
  - ✓ Air traffic control
  - ✓ Aircraft auto-pilot

Soft real-time systems

- The system tries to meet all deadlines in a best effort manner, and succeeds in meeting most of them. Overall %age of met deadlines is very high on the average
- Missing a few occasionally does not result in catastrophic conditions

Characteristics of Real-Time Operating Systems

- Deterministic
  - ✓ Operations are performed at fixed, predetermined times or within predetermined time intervals
  - ✓ Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time
- Responsiveness
  - ✓ How long, after acknowledgment, it takes the operating system to service the interrupt
    - Includes amount of time to begin execution of the interrupt
    - Includes the amount of time to perform the interrupt
    - Effect of interrupt nesting
- User control
  - ✓ User specifies priority
  - ✓ Specify paging
  - ✓ What processes must always reside in main memory
  - ✓ Disks algorithms to use
  - ✓ Rights of processes

.

Lecture No. 19

Characteristics of Real-Time Operating Systems

- Reliability
  - ✓ Degradation of performance may have catastrophic consequences
- Fail-soft operation
  - ✓ Ability of a system to fail in such a way as to preserve as much capability and data as possible
  - ✓ Stability

Features of Real-Time Operating Systems

- Fast process or thread switch
- Small size
- Ability to respond to external interrupts quickly
- Multitasking with inter-process communication tools such as semaphores, signals, and events
- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling base on priority
- Minimization of intervals during which interrupts are disabled
- Delay tasks for fixed amount of time
- Special alarms and timeouts

Real-Time Scheduling

- Static table-driven
  - ✓ Determines at load or even compile time when a task begins execution
- Static priority-driven preemptive
  - ✓ Traditional priority-driven scheduler is used
- Dynamic planning-based
  - ✓ Feasibility determined at run time
- Dynamic best effort
  - ✓ No feasibility analysis is performed

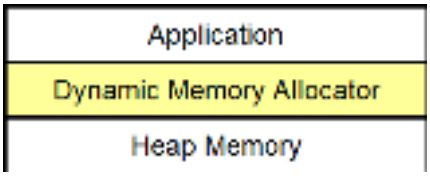
Deadline Scheduling

- Information used
  - ✓ Ready time
  - ✓ Starting deadline
  - ✓ Completion deadline
  - ✓ Processing time
  - ✓ Resource requirements
  - ✓ Priority
  - ✓ Subtask scheduler

Rate Monotonic Scheduling

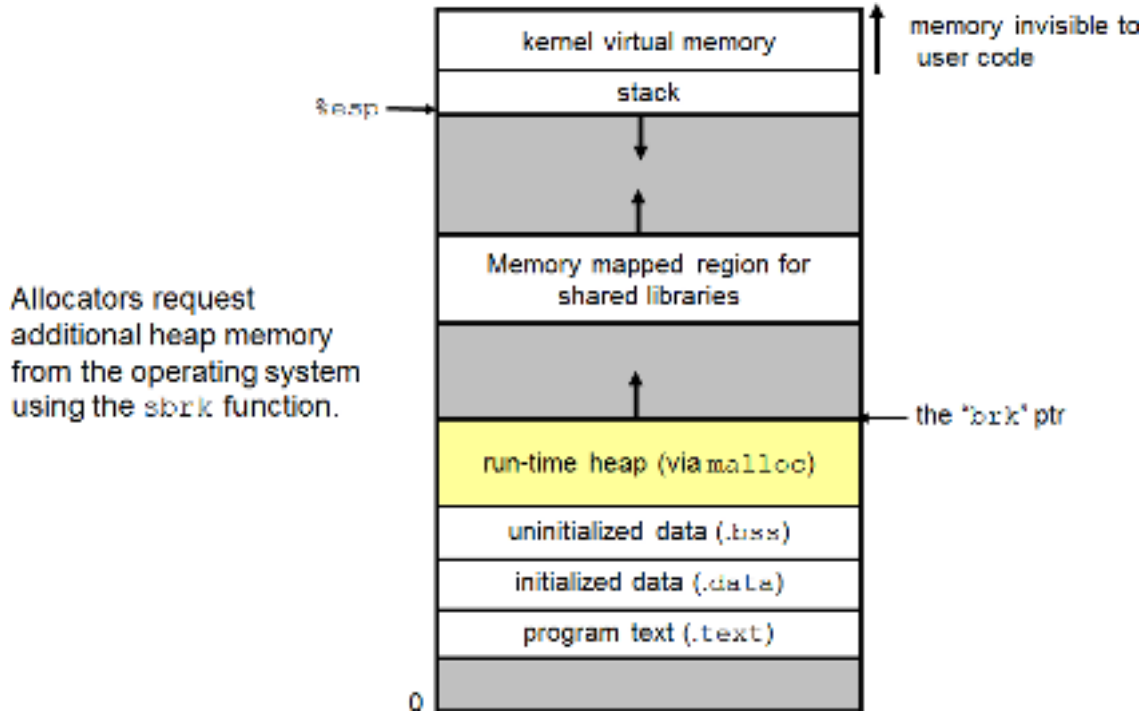
- Assigns priorities to tasks on the basis of their periods
- Highest-priority task is the one with the shortest period

Dynamic Memory Allocation



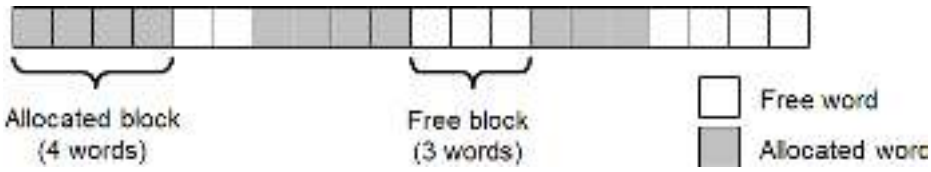
- Explicit vs. Implicit Memory Allocator
  - ✓ Explicit: application allocates and frees space
    - E.g., malloc and free in C
  - ✓ Implicit: application allocates, but does not free space
    - E.g. garbage collection in Java, ML or Lisp
- Allocation
  - ✓ In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - ✓ Gives out free memory blocks to application

Process Memory Image



Assumptions

- Memory is word addressed (each word can hold a pointer)



Constraints

- Applications:
  - ✓ Can issue arbitrary sequence of allocation and free requests
  - ✓ Free requests must correspond to an allocated block
- Allocators
  - ✓ Can't control number or size of allocated blocks
  - ✓ Must respond immediately to all allocation requests
    - o i.e., can't reorder or buffer requests
  - ✓ Must allocate blocks from free memory
    - o i.e., can only place un-allocated blocks in free memory
  - ✓ Must align blocks so they satisfy all alignment requirements
    - o 8 byte alignment for GNU malloc (libc malloc) on Linux machines

Lecture No. 20

Overview of today's lecture

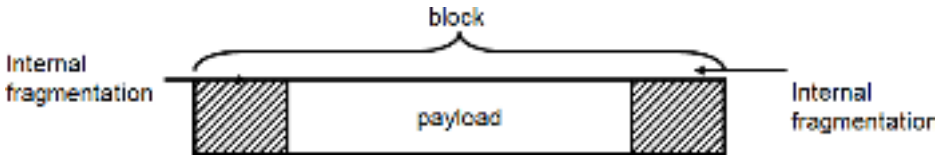
- Goals of a good allocator
- Memory Fragmentation
  - ✓ Internal and external
- Implementation issues of memory allocators
  - ✓ Knowing how much to free
  - ✓ Tracking free blocks
- Implicit list based allocator details

Goals of Good malloc/free

- Primary goals
  - ✓ Good time performance for malloc and free
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - ✓ Good space utilization
    - User allocated structures should be large fraction of the heap.
    - Want to minimize ¯fragmentation‖.
- Some other goals
  - ✓ Good locality properties
    - Structures allocated close in time should be close in space
    - ¯Similar‖ objects should be allocated close in space
  - ✓ Robust
    - Can check that free(p1) is on a valid allocated object p1
    - Can check that memory references are to allocated space

Internal Fragmentation

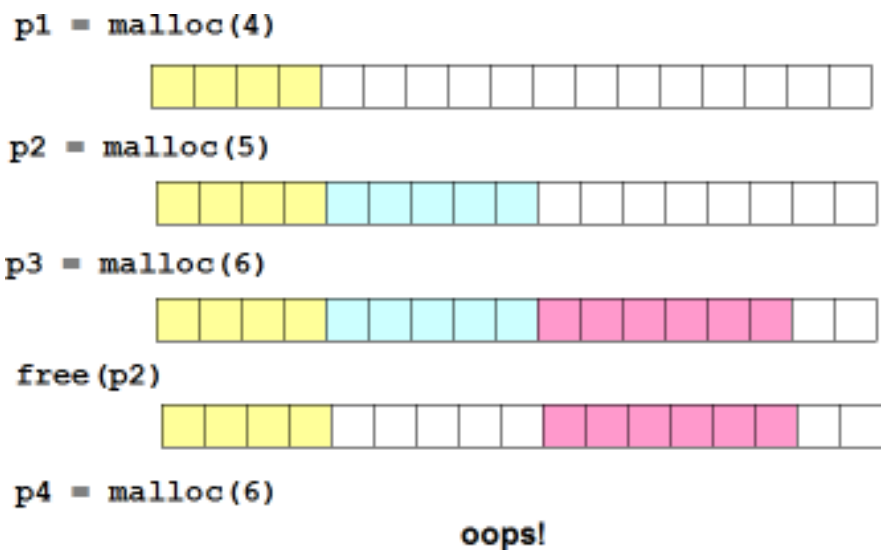
- Poor memory utilization caused by *fragmentation*.
  - ✓ Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - ✓ For some block, internal fragmentation is the difference between the block size and the payload size.



  - ✓ Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
  - ✓ Depends only on the pattern of *previous* requests, and thus is easy to measure.

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough



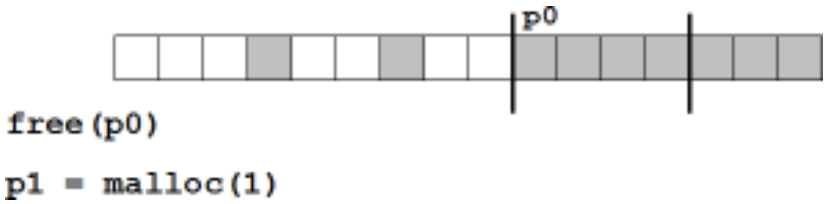
- External fragmentation depends on the pattern of future requests, and thus is difficult to measure.

Impossible to "solve" fragmentation

- If you read allocation papers or books to find the best allocator(!?!?!) it can be frustrating:
  - ✓ all discussions revolve around tradeoffs
  - ✓ the reason? There cannot be a best allocator

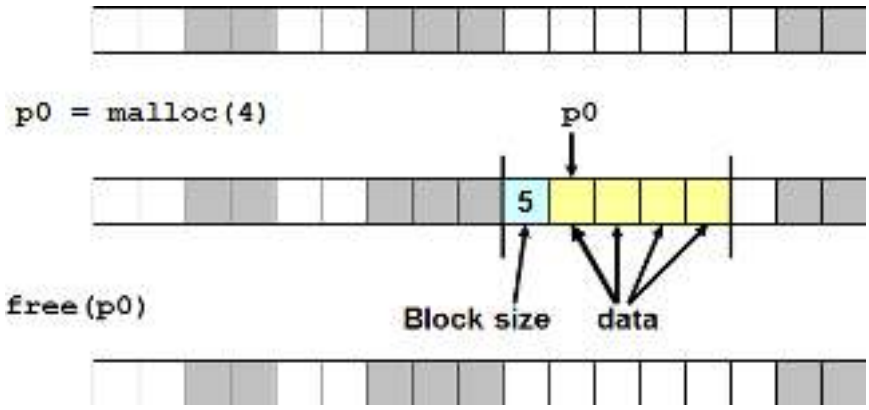
- Theoretical result:
  - ✓ for any possible allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation.

- ¯pretty well‖ = ~20% fragmentation under many workloads



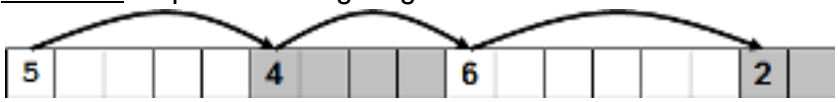
Knowing How Much to Free

- Standard method
  - ✓ Keep the length of a block in the word preceding the block.
    - This word is often called the header field or header

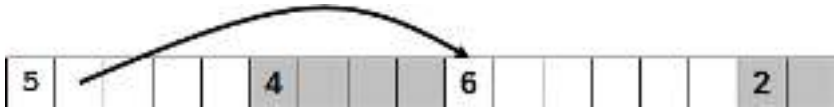


Keeping Track of Free Blocks

- Method 1: Implicit list using lengths -- links all blocks



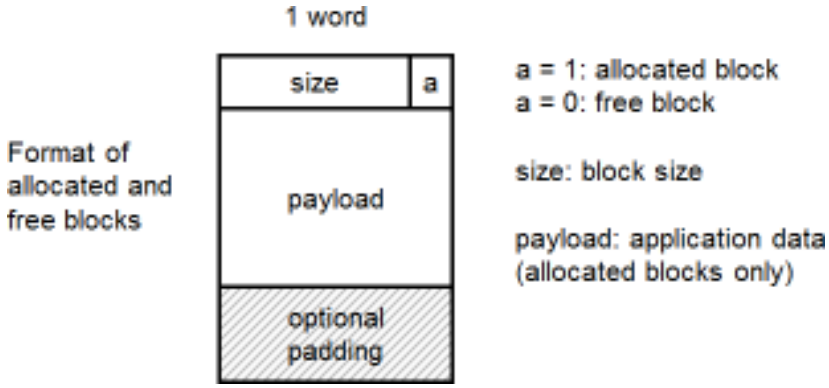
- Method 2: Explicit list among the free blocks using pointers within the free blocks



- Method 3: Segregated free list
  - ✓ Different free lists for different size classes
- Method 4: Blocks sorted by size
  - ✓ Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Method 1: Implicit List

- Need to identify whether each block is free or allocated

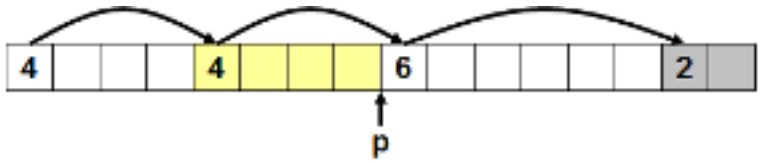


Implicit List: Finding a Free Block

- First fit:
  - ✓ Search list from beginning, choose first free block that fits
- Next fit:
  - ✓ Like first-fit, but search list from location of end of previous search
  - ✓ Best fit:
  - ✓ Search the list, choose the free block with the closest size that fits

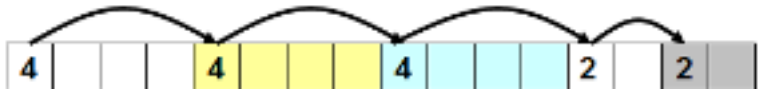
Implicit List: Allocating in Free Block

- Allocating in a free block - splitting




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

```
addblock(p, 4)
```



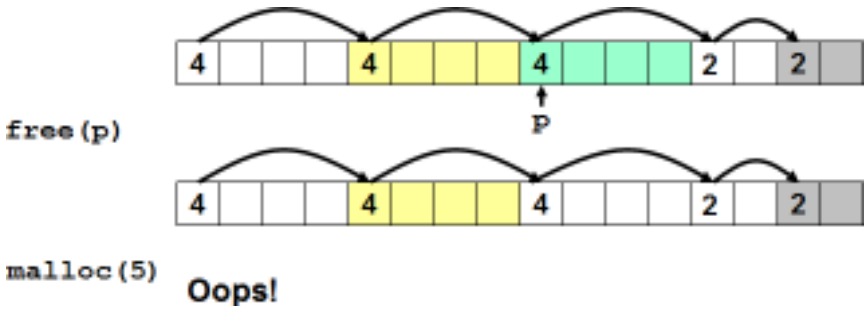
- Need to identify whether each block is free or allocated

Implicit List: Finding a Free Block

- First fit:
  - ✓ Search list from beginning, choose first free block that fits
- Next fit:
  - ✓ Like first-fit, but search list from location of end of previous search
- Best fit:
  - ✓ Search the list, choose the free block with the closest size that fits

Implicit List: Freeing a Block

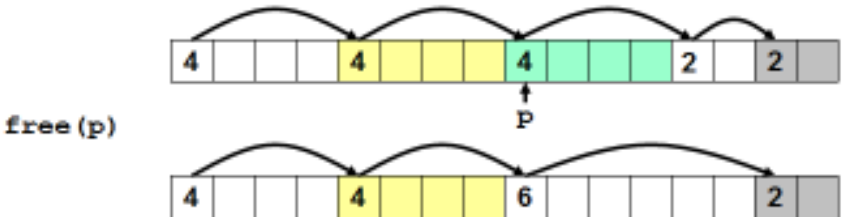
- Simplest implementation:
  - ✓ Only need to clear allocated flag
    void free_block(ptr p) { *p = *p & -2}



Implicit List: Coalescing

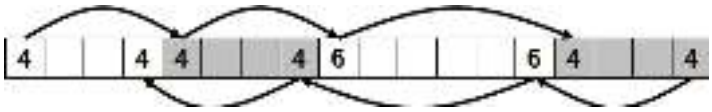
- Join (*coelesce*) with next and/or previous block if they are free
  - ✓ Coalescing with next block

```
void free_block(ptr p) {
    *p = *p & -2;          // clear allocated flag
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;    // add to this block if
}                         //    not allocated
```



Implicit List: Bidirectional Coalescing

- Boundary tags [Knuth73]
  - ✓ Replicate size/allocated word at bottom of free blocks
  - ✓ Allows us to traverse the ¯list‖ backwards, but requires extra space

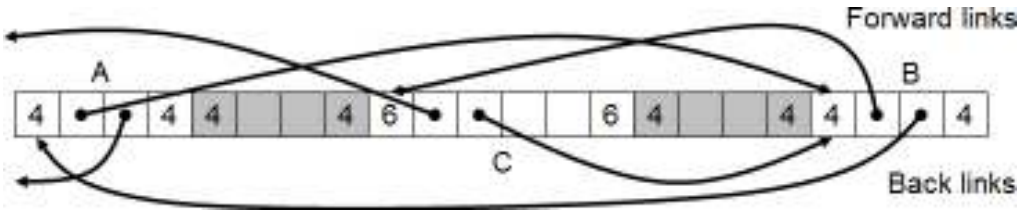


Lecture No. 21

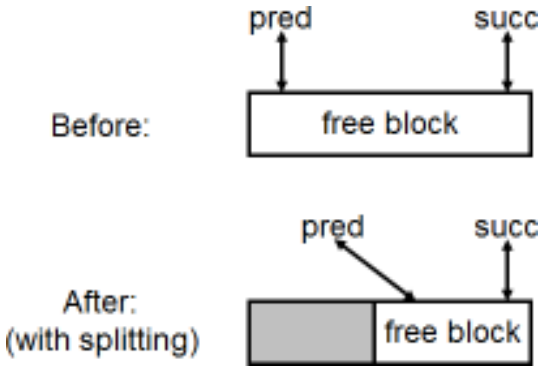
Overview of today's lecture

- Explicit free lists base allocator details
  - ✓ Freeing with LIFO policy
  - ✓ Segregated free lists
- Exploiting allocation patterns of programs
  - ✓ Exploiting peaks via arena allocators
  - ✓ Garbage collection

Explicit Free Lists



Allocating From Explicit Free Lists

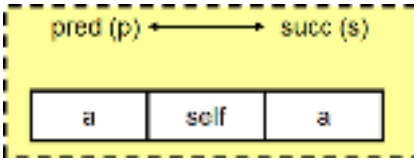


Freeing With Explicit Free Lists

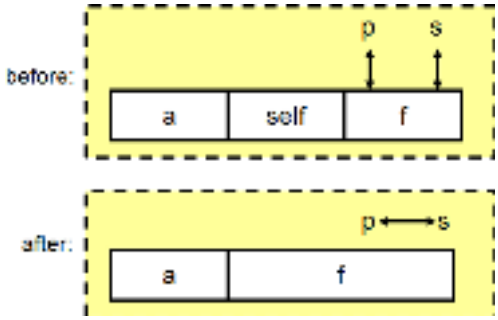
- Insertion policy: Where in the free list do you put a newly freed block?
  - ✓ LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - ✓ Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order
      - ✘ i.e. addr(pred) < addr(curr) < addr(succ)

Freeing With a LIFO Policy

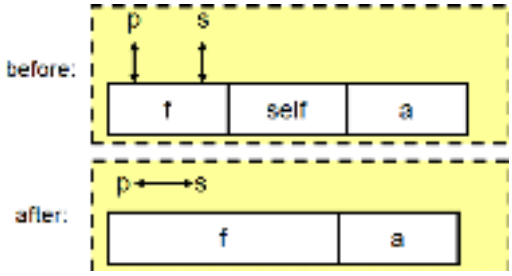
- Case 1: a-a-a



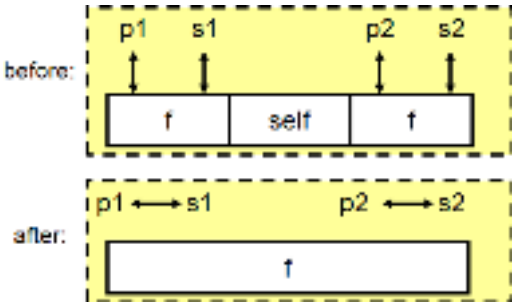
- Case 2: a-a-f



- Case 3: f-a-a



- Case 4: f-a-f



Explicit List Summary

- Comparison to implicit list:
  - ✓ Allocate is linear time in number of free blocks instead of total blocks -- much faster allocates when most of the memory is full
  - ✓ Slightly more complicated allocate and free since needs to move blocks in and out of the list
  - ✓ Some extra space for the links (2 extra words needed for each block)
- Main use of linked lists is in conjunction with segregated free lists
  - ✓ Keep multiple linked lists of different size classes, or possibly for different types of objects

Simple Segregated Storage

- Separate free list for each size class
- No splitting
- Tradeoffs:
  - ✓ Fast, but can fragment badly

Segregated Fits

- Array of free lists, each one for some size class
- To free a block:
  - ✓ Coalesce and place on appropriate list (optional)
  - ✓ Tradeoffs:
  - ✓ Faster search than sequential fits (i.e., log time for power of two size classes)
  - ✓ Controls fragmentation of simple segregated storage
  - ✓ Coalescing can increase search times
    - o Deferred coalescing can help

Known patterns of real programs

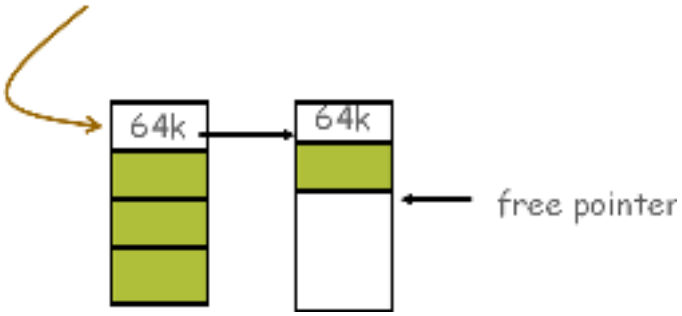
- Ramps: accumulate data monotonically over time

- Peaks: allocate many objects, use briefly, then free all

- Plateaus: allocate many objects, use for a long time

Exploiting peaks

- Peak phases: alloc a lot, then free everything
  - ✓ Advantages: alloc is a pointer increment, free is ¯free‖, & there is no wasted space for tags or list pointers.



Implicit Memory Management: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage -- application never has to free

Garbage Collection

- How does the memory manager know when memory can be freed?
- Need to make certain assumptions about pointers
  - ✓ Memory manager can distinguish pointers from non-pointers
  - ✓ All pointers point to the start of a block
  - ✓ Cannot hide pointers (e.g., by coercing them to an int, and then back again)

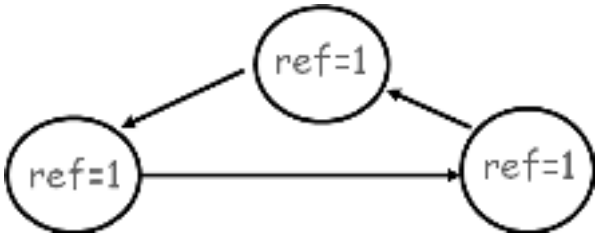
Reference counting

- Algorithm: counter pointers to object
  - ✓ each object has ¯ref count‖ of pointers to it
  - ✓ increment when pointer set to it
  - ✓ decremented when pointer killed

```
void foo(bar c) {
   bar a, b;
   a = c;  ..........................  c->refcnt++;
   b = a;  ..........................  a->refcnt++;
   a = 0;  ..........................  a->refcnt--;
   return; ..........................  b->refcnt--;
}
```

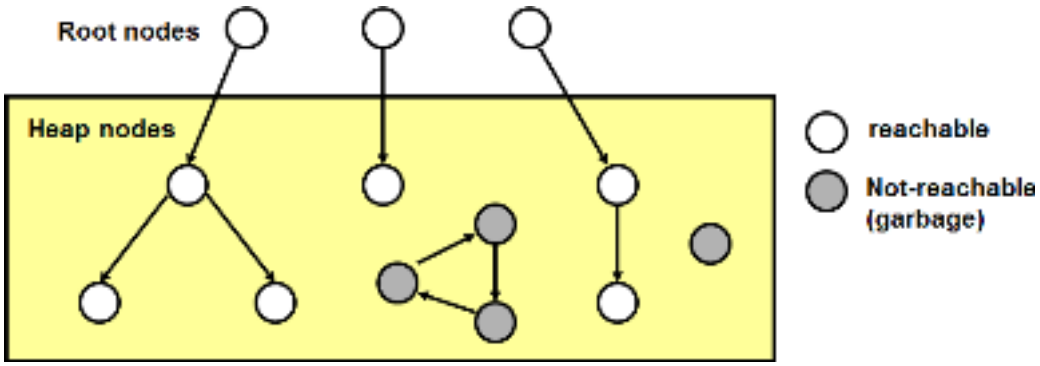
Problems

- Circular data structures always have refcnt > 0



Memory as a Graph

- We view memory as a directed graph
  - ✓ Each block is a node in the graph
  - ✓ Each pointer is an edge in the graph
  - ✓ Locations not in the heap that contain pointers into the heap are called *root* nodes (e.g. registers, locations on the stack, global variables)

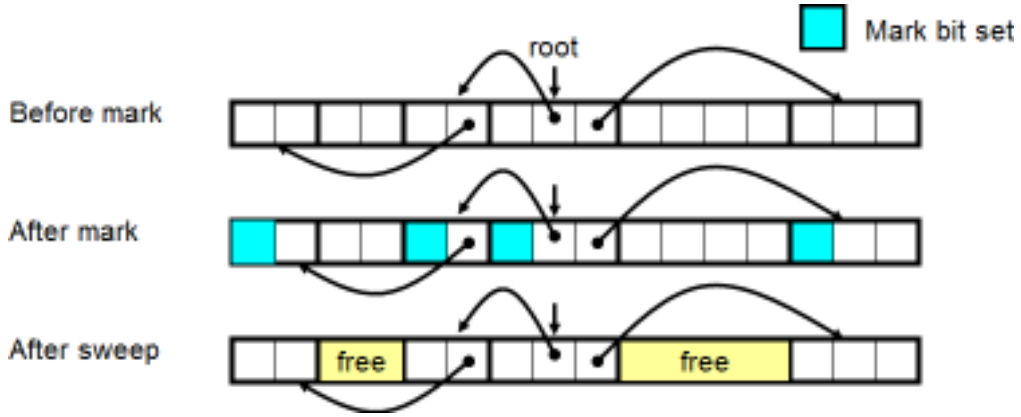


Assumptions

- Instructions used by the Garbage Collector
  - ✓ is_ptr(p): determines whether p is a pointer
  - ✓ length(b): returns the length of block b, not including the header
  - ✓ get_roots(): returns all the roots

Lecture No. 22

Mark and Sweep Collecting



Mark and Sweep

- Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return        // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // mark all children
     mark(p[i]);
   return;
}
```

- Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
   }
}
```

What is an operating system?

- Top-down view
- Bottom-up view
- Time multiplexing
- Space multiplexing

Type of Operating Systems

- Main frame operating systems
- Time-sharing systems
- Multiprocessor operating systems
- PC operating systems
- Real-time operating systems
- Embedded operating system

OS Structure

- Monolithic Design
- Layering
- Micro-kernel

ELF Object File Format

- Elf header
- Program header table
- .text section
- .data section
- .bss section

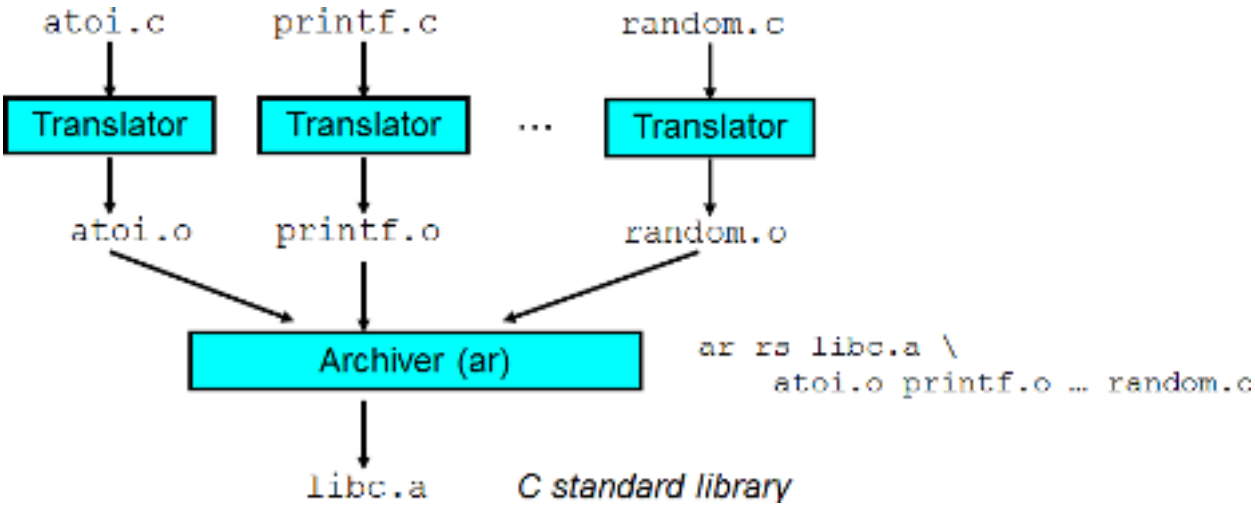
Strong and Weak Symbols

- Program symbols are either strong or weak

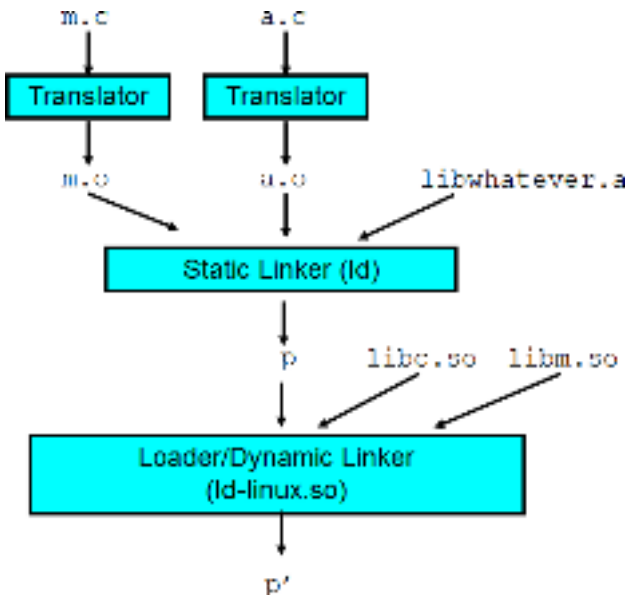
Linker's Symbol Rules

- Rule 1. A strong symbol can only appear once.
- Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
- Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

Creating Static Libraries



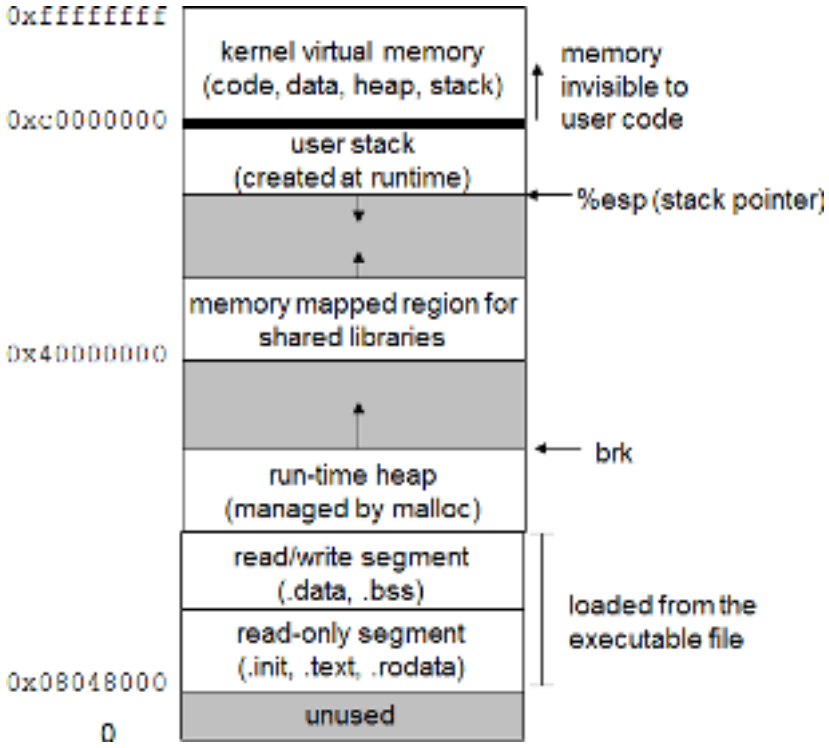
The Complete Picture



Process

- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions

Private Address Spaces



Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor

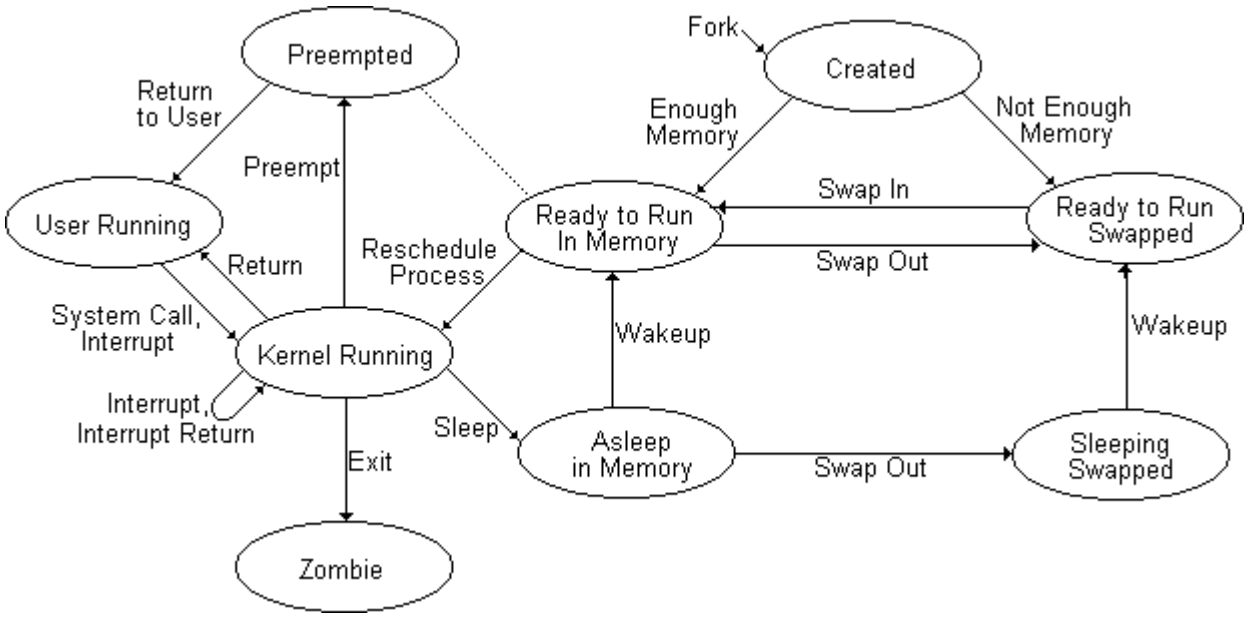
Synchronous Exceptions

- Traps
- Faults
- Aborts

Process Creation

- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block

Unix SVR4 Processes



Process Control Block

- Process state: e.g., running, ready, waiting, halted
- Priority
- Scheduling-related information
- Event

fork: Creating new processes

- int fork(void)

exit: Destroying Process

- void exit(int status)

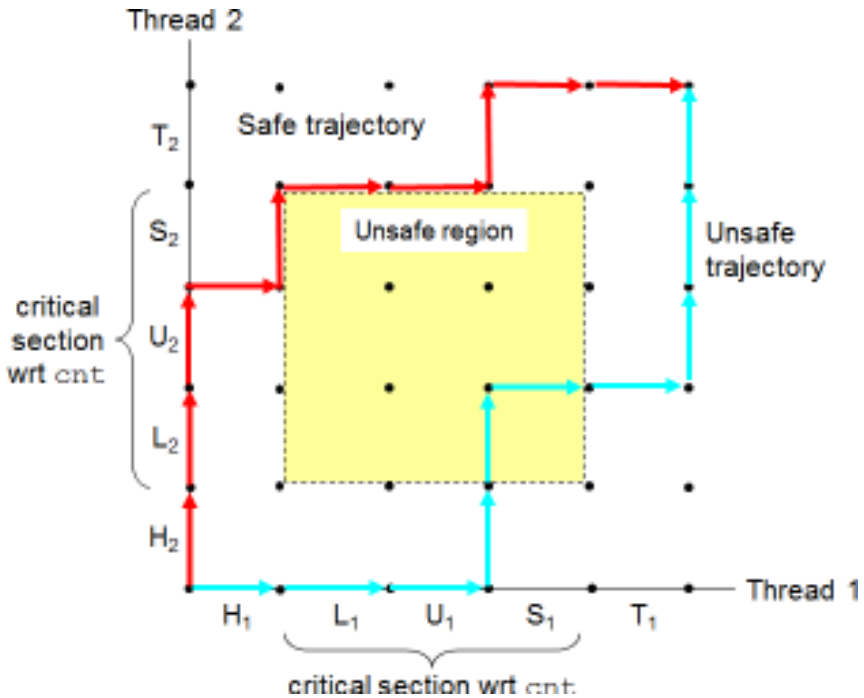
Zombies

- Idea
- Reaping

exec: Running new programs

- int execl(char *path, char *arg0, char *arg1, ...,0)

Safe and Unsafe Trajectories



Semaphores

- semaphore = a synchronization primitive
  - ✓ higher level than locks
  - ✓ invented by Dijkstra in 1968, as part of the THE OS

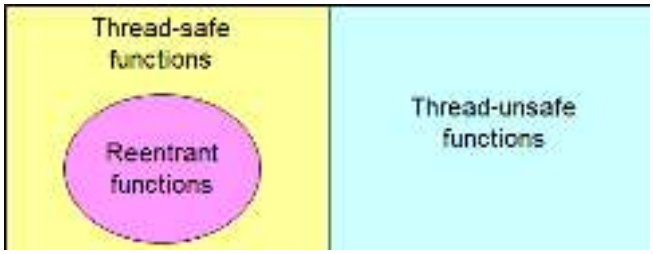
Two uses of semaphores

- Mutual exclusion
- Scheduling constraints

Condition variables

- Condition variable: a queue of threads waiting for something inside a critical section.
- Wait() -- release lock, go to sleep, re-acquire lock
- Signal() -- wake up a waiter, if any
- Broadcast() -- wake up all waiters

Reentrant Functions



Formal definition of a deadlock

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

Conditions for deadlock

- Without all of these, can't have deadlock:
    1. Mutual exclusion
    2. No preemption
    3. Hold and wait
    4. Circular wait

Paradigms of thread usage (from Hauser et al paper)

- Defer work
- General pumps
- Slack processes
- Sleepers
- One-shots
- Deadlock avoidance
- Rejuvenation
- Serializers
- Encapsulated fork
- Exploiting parallelism

Scheduling: First come first served (FCFS or FIFO)

- Simplest scheduling algorithm:
    - ✓ Run jobs in order that they arrive

Lecture No. 23

Overview of today's lecture

- Goals of OS memory management
- Questions regarding memory management
- Multiprogramming
- Virtual addresses
- Fixed partitioning
- Variable partitioning
- Fragmentation

Goals of OS memory management

- Allocate scarce memory resources among competing processes, maximizing memory utilization and system throughput
- Provide isolation between processes

Tools of memory management

- Base and limit registers
- Swapping
- Paging (and page tables and TLBs)
- Segmentation (and segment tables)
- Page fault handling => Virtual memory
- The policies that govern the use of these mechanisms

Our main questions regarding Memory Management

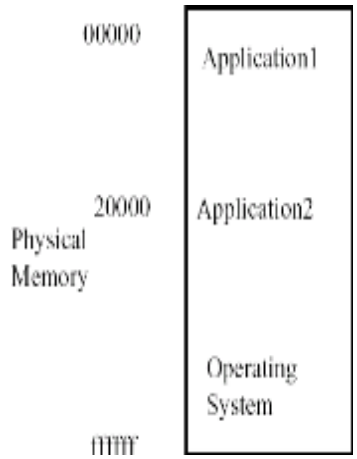
- How is protection enforced?
- How are processes relocated?
- How is memory partitioned?

Today's desktop and server systems

- The basic abstraction that the OS provides for memory management is virtual memory (VM)
  - ✓ VM enables programs to execute without requiring their entire address space to be resident in physical memory
    - o program can also execute on machines with less RAM than it ¯needs‖
  - ✓ many programs don't need all of their code or data at once (or ever)
    - o e.g., branches they never take, or data they never read/write
    - o no need to allocate memory for it, OS should adjust amount allocated based on run-time behavior
  - ✓ virtual memory isolates processes from each other
    - o one process cannot name addresses visible to others; each process has its own isolated address space
- Virtual memory requires hardware and OS support
  - ✓ MMU's, TLB's, page tables, page fault handling, …
- Typically accompanied by swapping, and at least limited segmentation

Multiprogramming: Linker-loader

- Can multiple programs share physical memory, without hardware translation? Yes: when copy program into memory, change its addresses (loads, stores, jumps) to use the addresses of where program lands in memory. This relocation is performed by a linker-loader.
- UNIX **ld** works this way: compiler generates each .o file with code that starts at location 0.
- How do you create an executable from this?



- Scan through each .o, changing addresses to point to where each module goes in larger program (requires help from compiler to say where all the re-locatable addresses are stored).
- With linker-loader, no protection: one program's bugs can cause other programs to crash
- Swapping
  - ✓ save a program‘s entire state (including its memory image) to disk
  - ✓ allows another program to be run
  - ✓ first program can be swapped back in and re-started right where it was
- The first timesharing system, MIT‘s ¯Compatible Time Sharing System‖ (CTSS), was a uni-programmed swapping system
  - ✓ only one memory-resident user
  - ✓ upon request completion or quantum expiration, a swap took place
  - ✓ Amazing even to think about today how bad the performance would be … but it worked!
- Multiple processes/jobs in memory at once
  - ✓ to overlap I/O and computation
- Memory management requirements:
  - ✓ protection: restrict which addresses processes can use, so they can‘t stomp on each other
  - ✓ fast translation: memory lookups must be fast, in spite of the protection scheme
  - ✓ fast context switching: when switching between jobs, updating memory hardware (protection and translation) must be quick

Virtual addresses for multiprogramming

- To make it easier to manage memory of multiple processes, make processes use virtual addresses (which is *not* what we mean by ¯virtual *memory*‖ today!)
  - ✓ virtual addresses are independent of location in physical memory (RAM) where referenced data lives
    - o OS determines location in physical memory

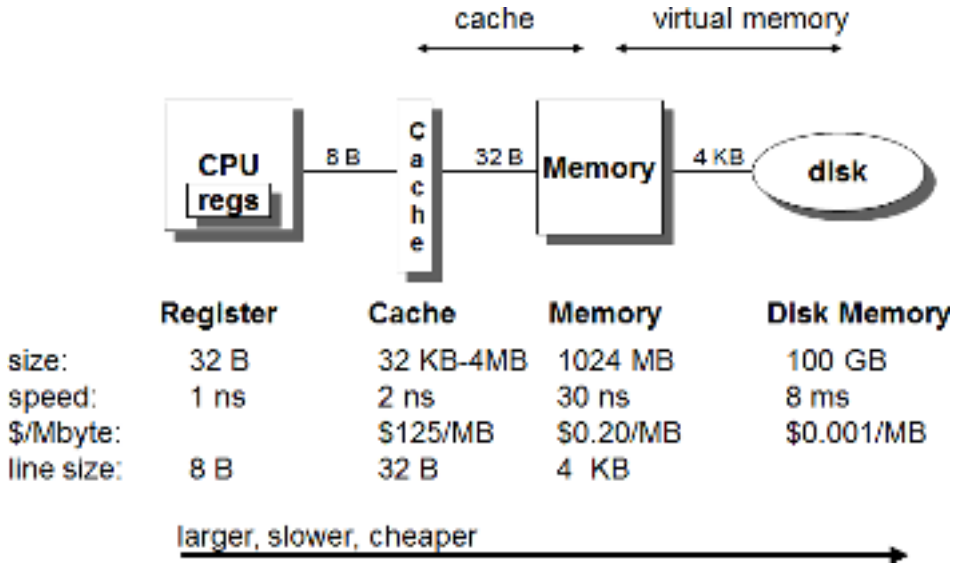
- ✓ instructions issued by CPU reference virtual addresses
  - e.g., pointers, arguments to load/store instructions, PC …
- ✓ virtual addresses are translated by hardware into physical addresses (with some setup from OS)

- The set of virtual addresses a process can reference is its address space
  - ✓ many different possible mechanisms for translating virtual addresses to physical addresses
    - we'll take a historical walk through them, ending up with our current techniques
- Note: We are not yet talking about paging, or virtual memory – only that the program issues addresses in a virtual address space, and these must be ¯adjusted‖ to reference memory (the physical address space)
  - ✓ for now, think of the program as having a contiguous virtual address space that starts at 0, and a contiguous physical address space that starts somewhere else

Memory Hierarchy

- Two principles:
  - ✓ The smaller amount of memory needed, the faster that memory can be accessed.
  - ✓ The larger amount of memory, the cheaper per byte. Thus, put frequently accessed stuff in small, fast, expensive memory; use large, slow, cheap memory for everything else.
- Works because programs aren't random. Exploit **locality**: that computers usually behave in future like they have in the past.
- Temporal locality: will reference same locations as accessed in the recent past
- Spatial locality: will reference locations near those accessed in the recent past

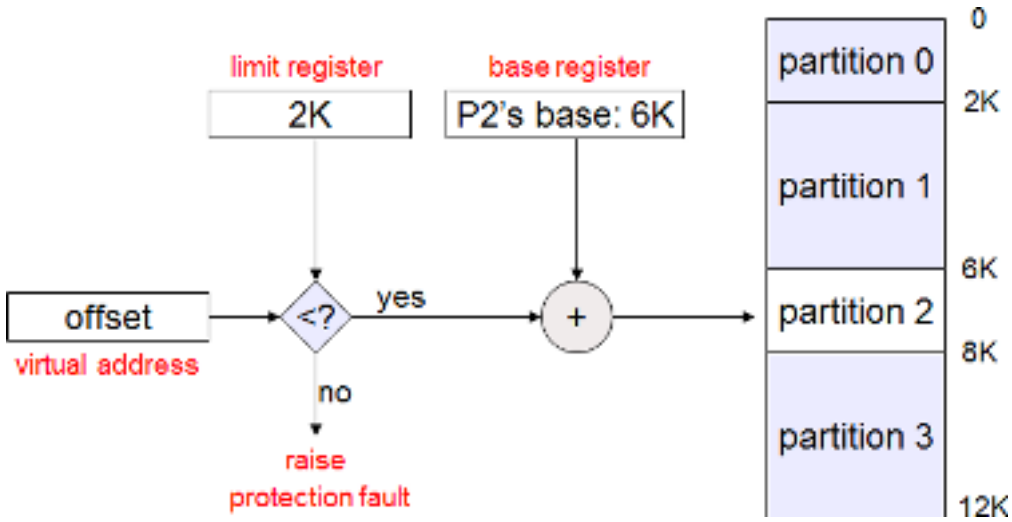
Levels in Memory Hierarchy



Old technique #1: Fixed partitions

- Physical memory is broken up into fixed partitions
  - ✓ partitions may have different sizes, but partitioning never changes
  - ✓ hardware requirement: base register, limit register
    - physical address = virtual address + base register
    - base register loaded by OS when it switches to a process
  - ✓ how do we provide protection?
    - if (physical address > base + limit) then… ?
- Advantages
  - ✓ Simple
- Problems
  - ✓ internal fragmentation: the available partition is larger than what was requested
  - ✓ external fragmentation: two small partitions left, but one big job - what sizes should the partitions be??

Mechanics of fixed partitions

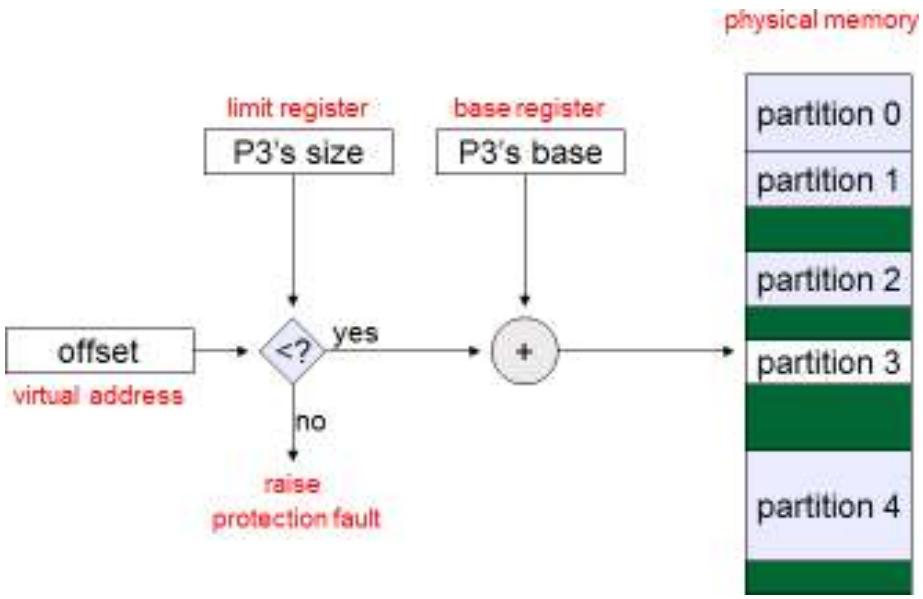


Old technique #2: Variable partitions

- Obvious next step: physical memory is broken up into partitions dynamically - partitions are tailored to programs
  - ✓ hardware requirements: base register, limit register
  - ✓ physical address = virtual address + base register
  - ✓ how do we provide protection?
    - if (physical address > base + limit) then… ?
- Advantages
  - ✓ no internal fragmentation
  - ✓ simply allocate partition size to be just big enough for process (assuming we know what that is!)
- Problems

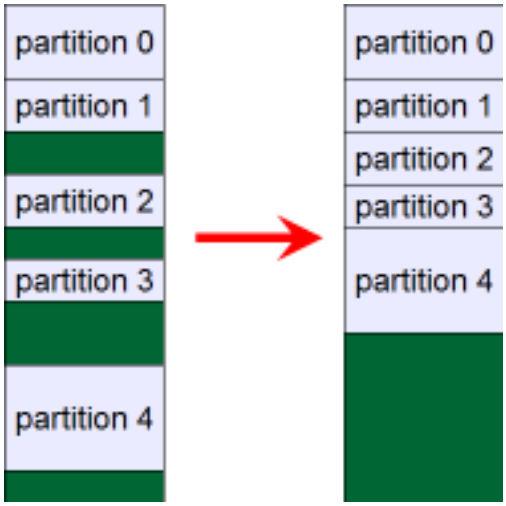
- ✓ external fragmentation
- ✓ as we load and unload jobs, holes are left scattered throughout physical memory
- ✓ slightly different than the external fragmentation for fixed partition systems

Mechanics of variable partitions



Dealing with fragmentation

- Swap a program out
- Re-load it, adjacent to another
- Adjust its base register



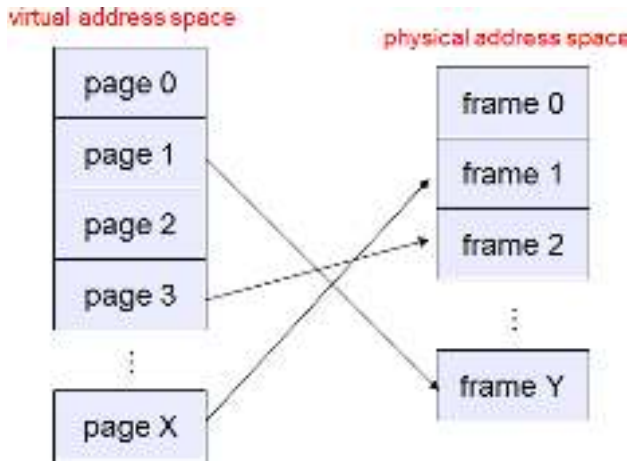
Lecture No. 24

Overview of today's lectures

- Paging
- Address translation
- Page tables and Page table entries
- Multi-level address translation
- Page faults and their handling

Modern technique: Paging

- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory



User's perspective

- Processes view memory as a contiguous address space from bytes 0 through N
  - ✓ virtual address space (VAS)
- In reality, virtual pages are scattered across physical memory frames – not contiguous as earlier
  - ✓ virtual-to-physical mapping
  - ✓ this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - ✓ the virtual address 0x356AF maps to different physical addresses for different processes
- Note: Assume for now that all pages of the address space are resident in memory – no ―page faults‖

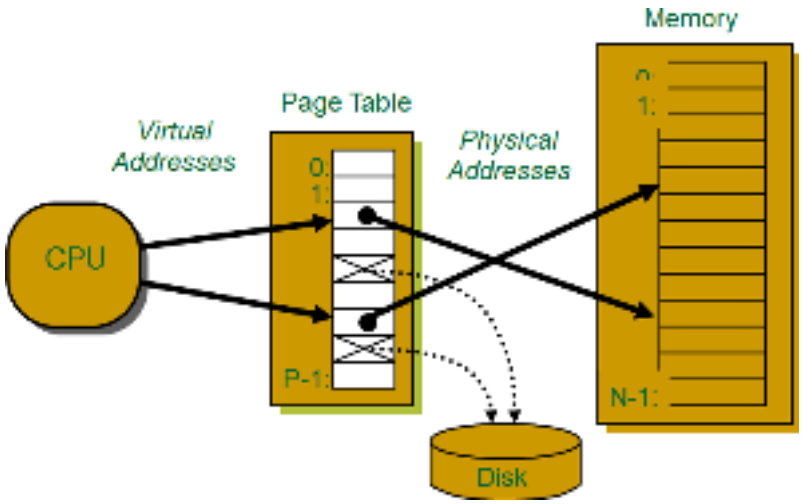
Address translation

- Translating virtual addresses
  - ✓ a virtual address has two parts: virtual page number & offset
  - ✓ virtual page number (VPN) is index into a page table
  - ✓ page table entry contains page frame number (PFN)
  - ✓ physical address is PFN::offset

- Page tables
  - ✓ managed by the OS
  - ✓ map virtual page number (VPN) to page frame number (PFN)
    - VPN is simply an index into the page table
  - ✓ one page table entry (PTE) per page in virtual address space
    - i.e., one PTE per VPN

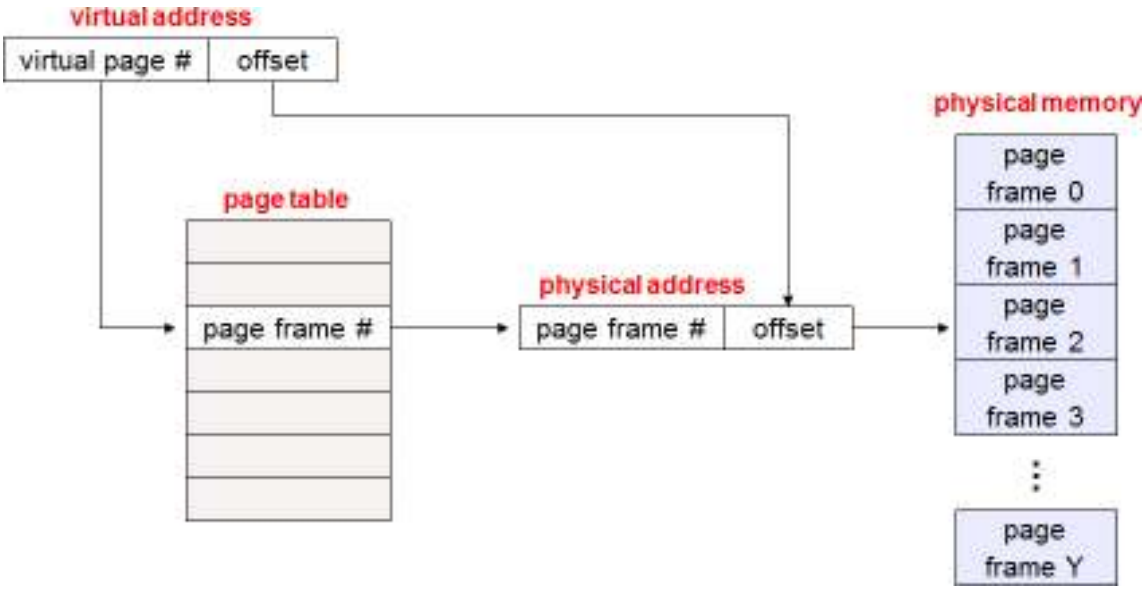
A System with paging based hardware support

- Examples: workstations, servers, modern PCs, etc.



- Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)

Mechanics of address translation



Example of address translation

- Assume 32 bit addresses
  - ✓ assume page size is 4KB (4096 bytes, or 2^{12} bytes)
  - ✓ VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long
- Let's translate virtual address 0x13325328
  - ✓ VPN is 0x13325, and offset is 0x328
  - ✓ assume page table entry 0x13325 contains value 0x03004
    - page frame number is 0x03004
    - VPN 0x13325 maps to PFN 0x03004
  - ✓ physical address = PFN::offset = 0x03004328

Page Table Entries (PTEs)

1	1	1	2	20
V	R	M	prot	page frame number

- PTE's control mapping
  - ✓ the valid bit says whether or not the PTE can be used
    - says whether or not a virtual address is valid
    - it is checked each time a virtual address is used
  - ✓ the referenced bit says whether the page has been accessed
    - it is set when a page has been read or written to
  - ✓ the modified bit says whether or not the page is dirty
    - it is set when a write to the page has occurred
  - ✓ the protection bits control which operations are allowed
    - read, write, execute
  - ✓ the page frame number determines the physical page
    - physical page start address = PFN

Paging advantages

- Easy to allocate physical memory
  - ✓ physical memory is allocated from free list of frames
    - to allocate a frame, just remove it from the free list
  - ✓ external fragmentation is not a problem!
    - managing variable-sized allocations is a huge pain
- Leads naturally to virtual memory
  - ✓ entire program need not be memory resident
  - ✓ take page faults using ¯valid‖ bit
  - ✓ but paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident

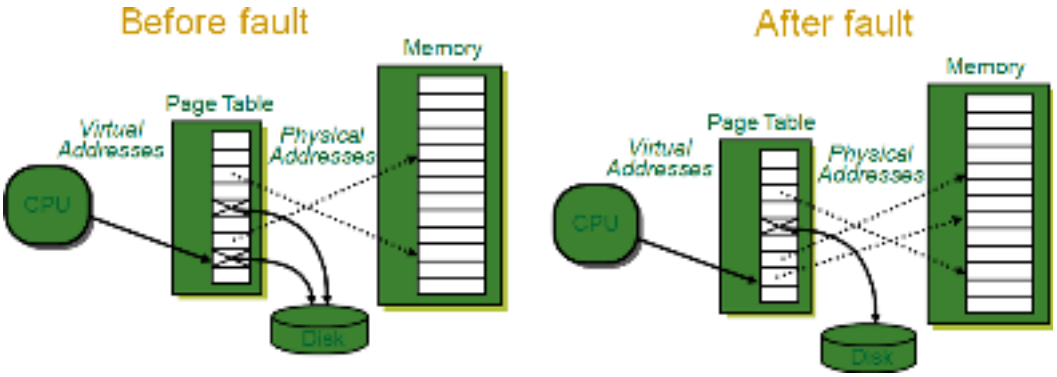
Paging disadvantages

- Can still have internal fragmentation
  - ✓ process may not use memory in exact multiples of pages

- Memory reference overhead
  - ✓ 2 references per address lookup (page table, then memory)
  - ✓ solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB) -
    - Memory required to hold page tables can be large
  - ✓ need one PTE per page in virtual address space
  - ✓ 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - ✓ 4 bytes/PTE = 4MB per page table
    - OS's typically have separate page tables per process
  - ✓ 25 processes = 100MB of page tables
  - ✓ solution: page the page tables (!!!)

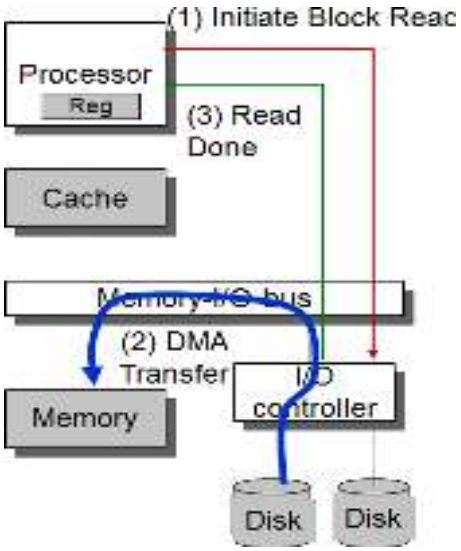
Page Faults (like "Cache Misses")

- What if an object is on disk rather than in memory?
  - ✓ Page table entry indicates virtual address not in memory
  - ✓ OS exception handler invoked to move data from disk into memory
    - current process suspends, others can resume
    - OS has full control over placement, etc.

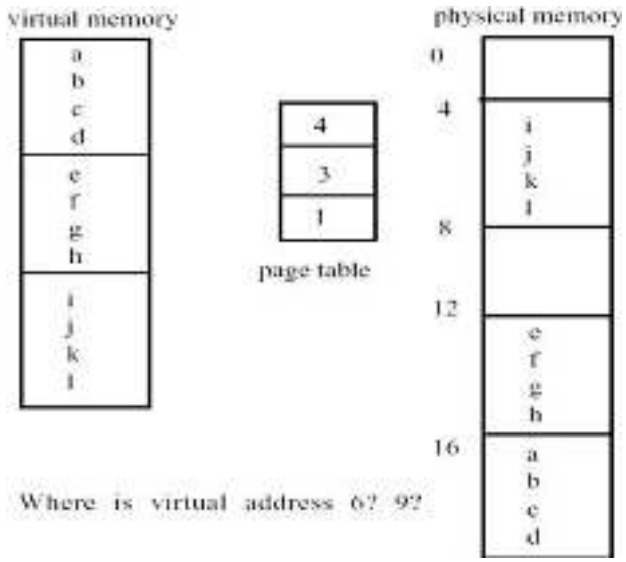


Servicing a Page Fault

- Processor Signals Controller
  - ✓ Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
  - ✓ Direct Memory Access (DMA)
  - ✓ Under control of I/O controller
- I / O Controller Signals Completion
  - ✓ Interrupt processor
  - ✓ OS resumes suspended process



Example: suppose page size is 4 bytes.

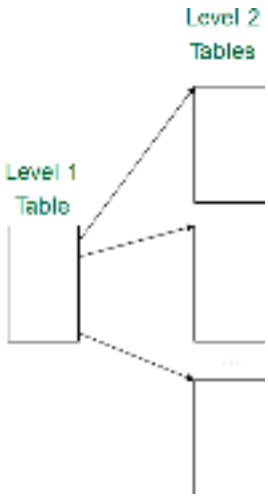


Fragmentation: wasted space

- What if page size is very small? For example, VAX had a
- page size of 512 bytes. Means lots of space taken up with page
- table entries.
- What if page size is really big? Why not have an infinite
- page size? Would waste unused space inside of page. Example
- of internal fragmentation.
- **external** -- free gaps between allocated chunks
- **internal** -- free gaps because don't need all of allocated chunk
- With segmentation need to re-shuffle segments to avoid
- external fragmentation. Paging suffers from internal fragmentation.

Multi-Level Page Tables

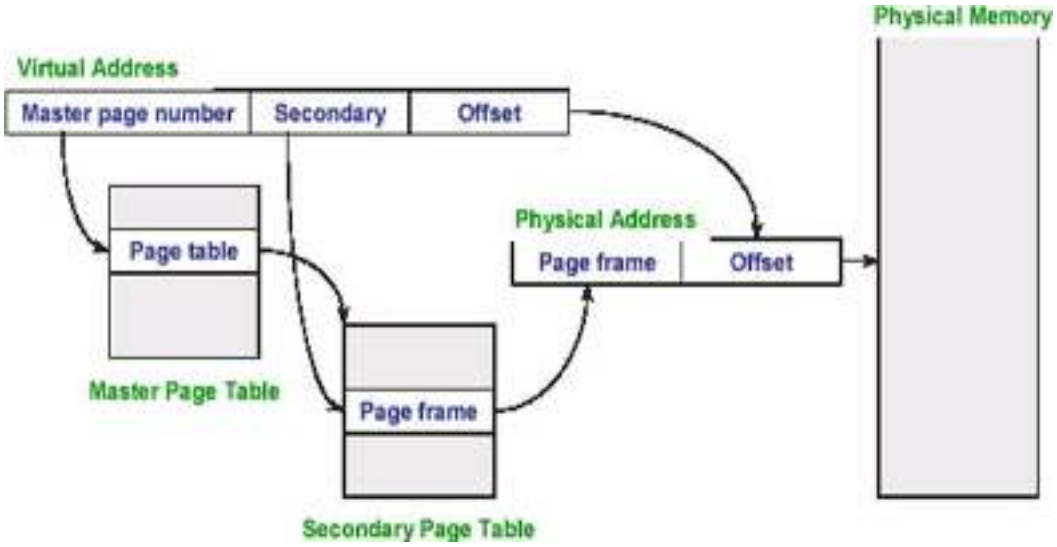
- Given:
  - ✓ 4KB (2^{12}) page size
  - ✓ 32-bit address space
  - ✓ 4-byte PTE
- Problem:
  - ✓ Would need a 4 MB page table!
    - 2^{20} *4 bytes
- Common solution
  - ✓ multi-level page tables
  - ✓ e.g., 2-level table (P6)
    - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
    - Level 2 table: 1024 entries, each of which points to a page



Two Level Page Tables

- Two-level page tables
  - ✓ Virtual address (VAs) have three parts:
    - Master page number
    - Secondary page number
    - Offset
  - ✓ Master page table maps VAs to secondary page table
  - ✓ Secondary page table maps page number to physical page
  - ✓ Offest indicates where in physical page address is located
- Example
- 4K pages, 4 bytes/PTE
- How many bits in offset? 4K = 12bits
- Want master page table in one page: 4K/4 bytes = 1k entries
- Hence, 1k secondary page tables, How many bits?

Two Level Page Tables



Addressing Page Tables

- Where do we store page tables(Which address space)?
- Physical memory
  - ✓ Easy to address, no translation required
  - ✓ But, allocated page tables consume memory for lifetime of VAS
- Virtual memory(OS virtual address space)
  - ✓ Cold(unused) page table pages can be paged out of disk
  - ✓ But, addressing page tables requires translation
  - ✓ How to we stop recursion?
  - ✓ Do not page the outer page table(called wiring(
- If we‘re going to page the page tables, might as wee page the entire OS address space, too
  - ✓ Need to wire special code and data(fault, interrupt handler)

Lecture No. 25

Overview of today's lecture

- Segmentation
- Combined Segmentation and paging
- Efficient translations and caching
- Translation Lookaside Buffer (TLB)

Segmentation

- Paging
  - ✓ mitigates various memory allocation complexities (e.g., fragmentation)
  - ✓ view an address space as a linear array of bytes
  - ✓ divide it into pages of equal size (e.g., 4KB)
  - ✓ use a page table to map virtual pages to physical page frames
    - page (*logical*) => page frame (*physical*)
- Segmentation
  - ✓ partition an address space into *logical* units
    - stack, code, heap, subroutines, …
  - ✓ a virtual address is <segment #, offset>

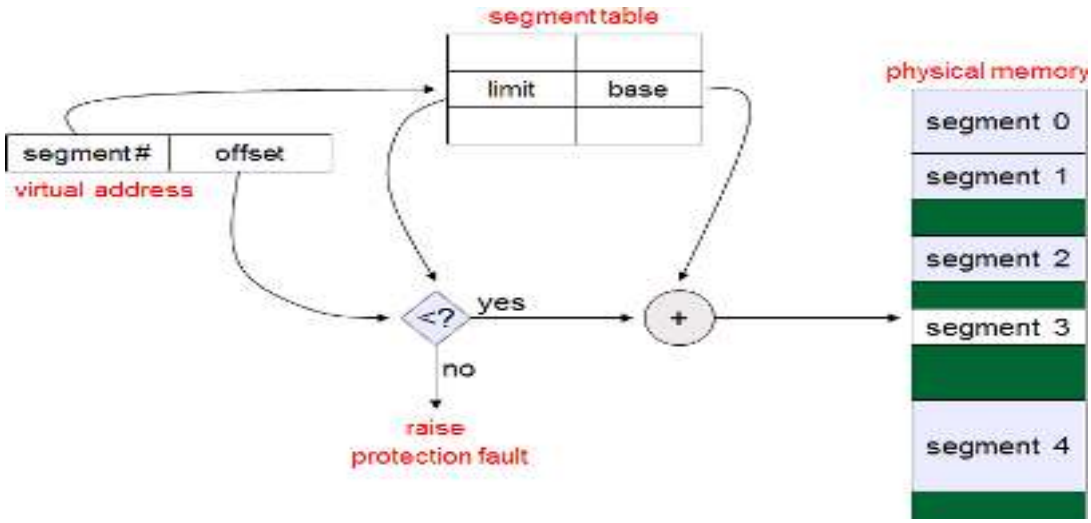
What's the point?

- More ¯logical‖
  - ✓ absent segmentation, a linker takes a bunch of independent modules that call each other and organizes them
  - ✓ they are really independent; segmentation treats them as such
- Facilitates sharing and reuse
  - ✓ a segment is a natural unit of sharing - a subroutine or function
- A natural extension of variable-sized partitions
  - ✓ variable-sized partition = 1 segment/process
  - ✓ segmentation = many segments/process

Hardware support

- Segment table
  - ✓ multiple base/limit pairs, one per segment
  - ✓ segments named by segment #, used as index into table
    - a virtual address is <segment #, offset>
  - ✓ offset of virtual address added to base address of segment to yield physical address

Segment lookups



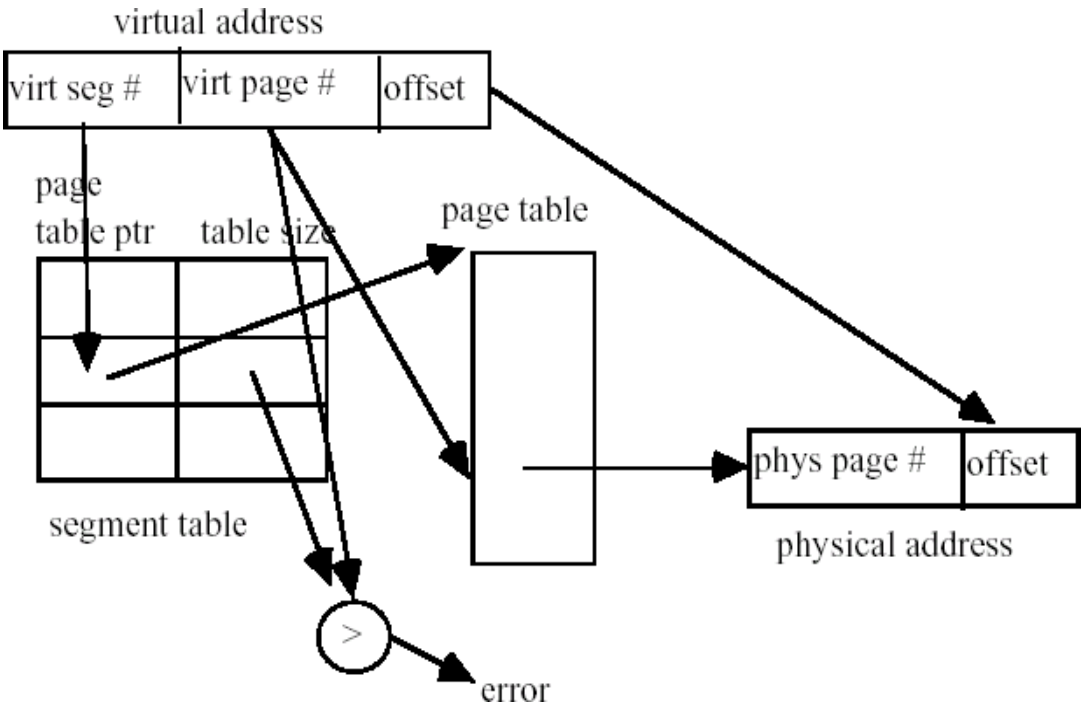
Segmentation pros & cons

- + efficient for sparse address spaces
- + easy to share whole segments (for example, code segment)
- Need to add protection mode in segmentation table. For example, code segment would be read-only (only execution and loads are allowed). Data and stack segment would be read-write (stores allowed).
    1. complex memory allocation
- Still need first fit, best fit, etc., and re-shuffling to coalesce free fragments, if no single free space is big enough for a new segment.
- Linux:
    1. 1 kernel code segment, 1 kernel data segment
    2. 1 user code segment, 1 user data segment
    3. N task state segments (stores registers on context switch)
    4. 1 ¯local descriptor table‖ segment (not really used)
    5. all of these segments are paged

Segmentation and Paging

- Can combine segmentation and paging
    1. The x86 supports segments and paging
- Use segments to manage logically related units
    1. Module, procedure, stack, file, data, etc
    2. Segments vary in size, but usually large(multiple pages)
- Use pages to partition segments into fixed size chunks
    1. Makes segments easier to manage within physical memory
        - become ‖pageable‖ – rather than moving segments into and out of memory, just move page portions of segment
    2. Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex

Segmentation with paging translation



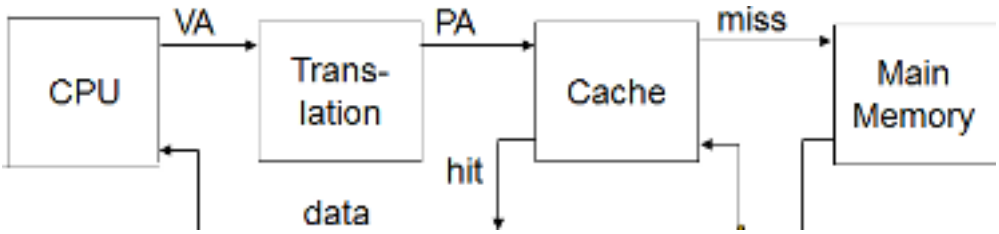
Segmentation with paging translation Pros & Cons

- + only need to allocate as many page table entries as we need. In other words, sparse address spaces are easy.
- + easy memory allocation
- + share at seg or page level
- - pointer per page (typically 4KB - 16KB pages today)
- - page tables need to be contiguous
- - two lookups per memory reference

Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
    1. One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost.
    1. Two lookups into the page tables, a third to fetch the data
    2. And this assume the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
    1. Cache translation in hardware
    2. Translation Lookaside Buffer (TLB)
    3. TLB managed by Memory Management Unit (MMU)

Integrating VM and Cache



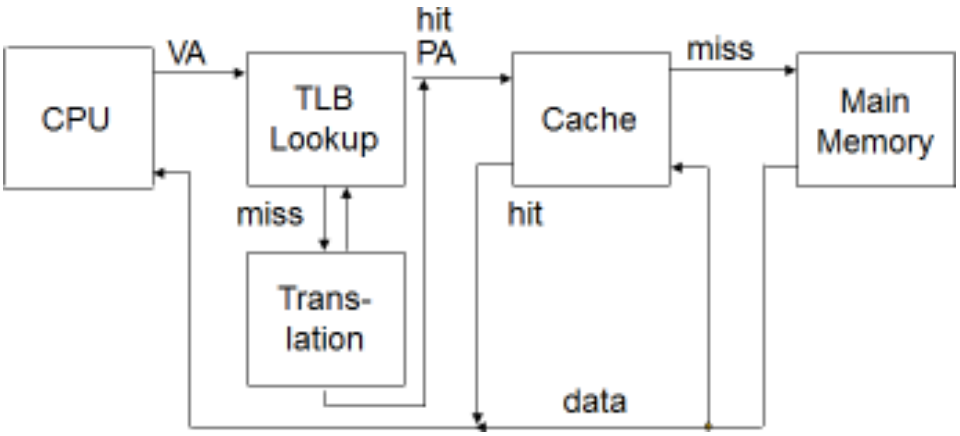
- Most Caches ¯Physically Addressed‖
    1. Accessed by physical addresses
    2. Allows multiple processes to have blocks in cache at same time
    3. Allows multiple processes to share pages
    4. Cache doesn‘t need to be concerned with protection issues
        - Access rights checked as part of address translation
- Perform Address Translation Before Cache Lookup
    1. But this could involve a memory access itself (of the PTE)
    2. Of course, page table entries can also become cached

Caching review

- **Cache**: copy that can be accessed more quickly than original.
- Idea is: make frequent case efficient, infrequent path doesn't matter as much. Caching is a fundamental concept used in lots of places in computer systems. It underlies many of the techniques that are used today to make computers go fast: can cache translations, memory locations, pages, file blocks, file names, network routes, authorizations for security systems, etc.
- **Generic Issues in Caching**
- **Cache hit**: item is in the cache
- **Cache miss**: item is not in the cache, have to do full operation
- **Effective access time** = P (hit) * cost of hit + P (miss) * cost of miss
    1. How do you find whether item is in the cache (whether there is a cache hit)?
    2. If it is not in cache (cache miss), how do you choose what to replace from cache to make room?
    3. Consistency -- how do you keep cache copy consistent with real version?

Speeding up Translation with a TLB

- ¯Translation Lookaside Buffer‖ (TLB)
    1. Small hardware cache in MMU
    2. Maps virtual page numbers to physical page numbers
    3. Contains complete page table entries for small number of pages

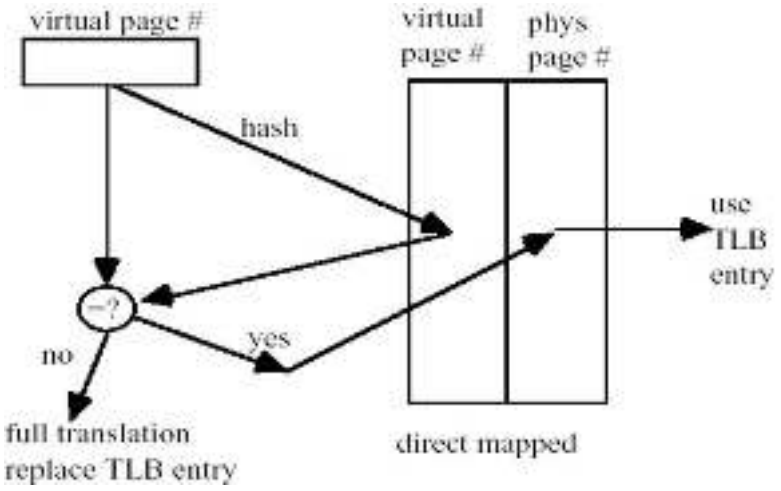


Translation Buffer, Translation Lookaside Buffer

- Hardware table of frequently used translations, to avoid having to go through page table lookup in common case. Typically, on chip, so access time of 2-5ns, instead of 30-100ns for main memory.

- **How do we tell if needed translation is in TLB?**
    1. Search table in sequential order
    2. **Direct mapped:** restrict each virtual page to use specific slot in TLB
        - For example, use upper bits of virtual page number to index TLB. Compare against lower bits of virtual page number to check for match.
        - What if two pages conflict for the same TLB slot? Ex: program counter and stack.
        - One approach: pick hash function to minimize conflicts
        - What if use low order bits as index into TLB?
        - What if use high order bits as index into TLB?
        - Thus, use selection of high order and low order bits as index.
    3. **Set associativity**: arrange TLB (or cache) as N separate banks. Do simultaneous lookup in each bank. In this case, called "N-way set associative cache".
        - More set associativity, less chance of thrashing. Translations can be stored, replaced in either bank.
    4. **Fully associative**: translation can be stored anywhere in TLB, so check all entries in the TLB in parallel.

Direct mapped TLB

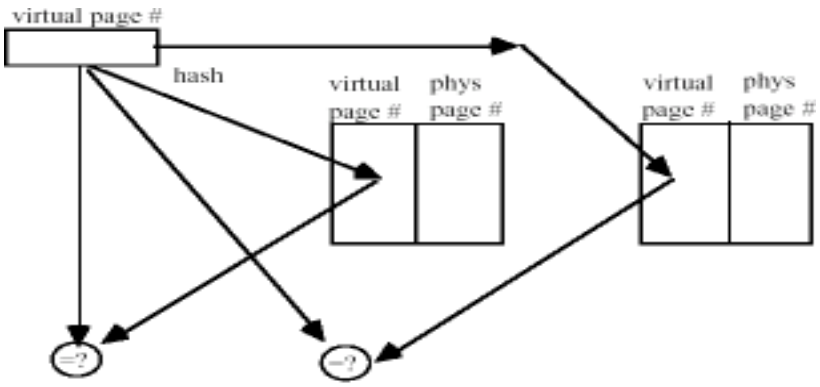


Lecture No. 26

Overview of today's lecture

- Set associative and fully associative caches
- Demand Paging
- Page replacement algorithms

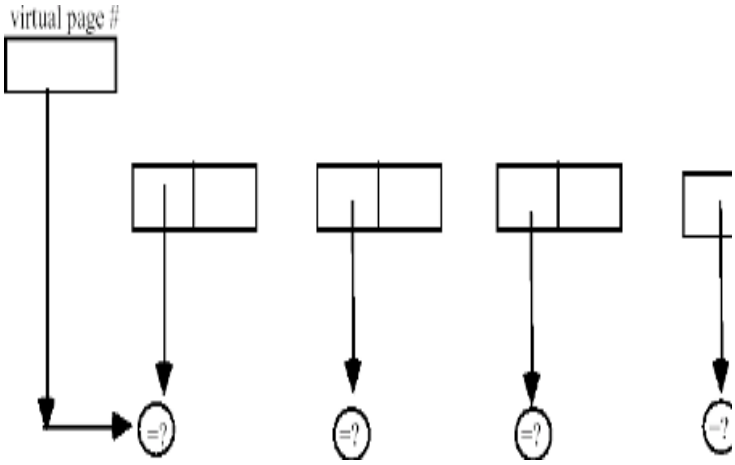
Set associative cache



if either match, use TLB entry, otherwise, translate and replace one of the entries.

Two-way set associative

Fully associative TLB



- Fully associative caches have one element per bank, one comparator per bank.
- Same set of options, whether you are building TLB or any kind of cache. Typically, TLB's are small and fully associative.
- Hardware layer caches are larger, and direct mapped or set associative to a small degree.
- How do we choose which item to replace?
- For direct mapped, never any choice as to which item to replace. But for set associative

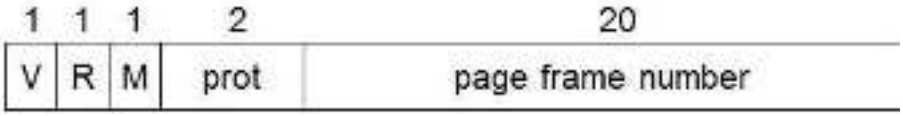
or fully associative cache, have a choice. What should we do?

- Replace least recently used? Random? Most recently used?
- In hardware, often choose item to replace randomly, because it's simple and fast. In software (for example, for page replacement), typically do something more sophisticated. Tradeoff: spend CPU cycles to try to improve cache hit rate.

Consistency between TLB and page tables

- What happens on context switch?
- Have to invalidate entire TLB contents. When new program starts running, will bring in new translations. Alternatively??
- Have to keep TLB consistent with whatever the full translation would give.
- What if translation tables change? For example, to move page from memory to disk, or vice versa. Have to invalidate TLB entry.
- How do we implement this? How can a process run without access to a page table?
- Basic mechanism:
  1. TLB has "present" (valid) bit
     if present, pointer to page frame in memory
     if not present, use page table in memory
  2. Hardware traps to OS on reference not in TLB
  3. OS software:
     a. load page table entry into TLB
     b. continue thread
- All this is transparent -- job doesn't know it happened.

Reminder: Page Table Entries (PTEs)



- PTE's control mapping
  - ✓ the valid bit says whether or not the PTE can be used
    - o says whether or not a virtual address is valid
    - o it is checked each time a virtual address is used
  - ✓ the referenced bit says whether the page has been accessed
    - o it is set when a page has been read or written to
  - ✓ the modified bit says whether or not the page is dirty
    - o it is set when a write to the page has occurred
  - ✓ the protection bits control which operations are allowed
    - o read, write, execute
  - ✓ the page frame number determines the physical page
    - o physical page start address = PFN

Paged virtual memory

- We‘ve hinted that all the pages of an address space do not need to be resident in memory
  - ✓ the full (used) address space exists on secondary storage (disk) in page-sized blocks
  - ✓ the OS uses main memory as a (page) cache
  - ✓ a page that is needed is transferred to a free page frame
  - ✓ if there are no free page frames, a page must be evicted
    - evicted pages go to disk (only need to write if they are dirty)
  - ✓ all of this is transparent to the application (except for performance …)
    - managed by hardware and OS
- Traditionally called paged virtual memory

Page faults

- What happens when a process references a virtual address in a page that has been evicted?
  - ✓ when the page was evicted, the OS set the PTE as invalid and noted the disk location of the page in a data structure (that looks like a page table but holds disk addresses)
  - ✓ when a process tries to access the page, the invalid PTE will cause an exception (page fault) to be thrown
    - OK, it‘s actually an interrupt!
  - ✓ the OS will run the page fault handler in response
    - handler uses the ¯like a page table‖ data structure to locate the page on disk
    - handler reads page into a physical frame, updates PTE to point to it and to be valid
    - OS restarts the faulting process

Demand paging

- Pages are only brought into main memory when they are referenced
  - ✓ only the code/data that is needed (demanded!) by a process needs to be loaded
    - What‘s needed changes over time, of course…
  - ✓ Hence, it‘s called demand paging
- Few systems try to anticipate future needs
  - ✓ OS crystal ball module notoriously ineffective
- But it‘s not uncommon to cluster pages
  - ✓ OS keeps track of pages that should come and go together
  - ✓ bring in all when one is referenced
  - ✓ interface may allow programmer or compiler to identify clusters

How do you “load” a program?

- Create process descriptor (process control block)
- Create page table
- Put address space image on disk in page-sized chunks

- Build page table (pointed to by process descriptor)
  - ✓ all PTE valid bits ‗false‘
  - ✓ an analogous data structure indicates the disk location of the corresponding page
  - ✓ when process starts executing:
    - instructions immediately fault on both code and data pages
    - faults taper off, as the necessary code/data pages enter memory

Page replacement

- When you read in a page, where does it go?
  - ✓ if there are free page frames, grab one
    - what data structure might support this?
  - ✓ if not, must evict something else
  - ✓ this is called page replacement
- Page replacement algorithms
  - ✓ try to pick a page that won‘t be needed in the near future
  - ✓ try to pick a page that hasn‘t been modified (thus saving the disk write)
  - ✓ OS typically tries to keep a pool of free pages around so that allocations don‘t inevitably cause evictions
  - ✓ OS also typically tries to keep some ¯clean‖ pages around, so that even if you have to evict a page, you won‘t have to write it
    - accomplished by pre-writing when there‘s nothing better to do

how does it all work?

- Locality!
  - ✓ temporal locality
    - locations referenced recently tend to be referenced again soon
  - ✓ spatial locality
    - locations near recently referenced locations are likely to be referenced soon (think about why)
- Locality means paging can be infrequent
  - ✓ once you‘ve paged something in, it will be used many times
  - ✓ on average, you use things that are paged in
  - ✓ but, this depends on many things:
    - degree of locality in the application
    - page replacement policy and application reference pattern
    - amount of physical memory vs. application ¯footprint‖ or ¯working set‖

Evicting the best page

- The goal of the page replacement algorithm:
  - ✓ reduce fault rate by selecting best victim page to remove
    - ¯system‖ fault rate or ¯program‖ fault rate??
  - ✓ the best page to evict is one that will never be touched again
    - impossible to predict such page
  - ✓ ¯never‖ is a long time

- Belady's proof: evicting the page that won't be used for the longest period of time minimizes page fault rate

#1: Belady's Algorithm

- Provably optimal: lowest fault rate (remember SJF?)
  - ✓ evict the page that won't be used for the longest time in future
  - ✓ problem: impossible to predict the future
- Why is Belady's algorithm useful?
  - ✓ as a yardstick to compare other algorithms to optimal
    - if Belady's isn't much better than yours, yours is pretty good
      - how could you do this comparison?
- Is there a best practical algorithm?
  - ✓ no; depends on workload
- Is there a worst algorithm?
  - ✓ no, but random replacement does pretty badly
    - there are some other situations where OS's use near-random algorithms quite effectively!

#2: FIFO

- FIFO is obvious, and simple to implement
  - ✓ when you page in something, put it on the tail of a list
  - ✓ evict page at the head of the list
- Why might this be good?
  - ✓ maybe the one brought in longest ago is not being used
- Why might this be bad?
  - ✓ then again, maybe it is being used
  - ✓ have absolutely no information either way
- In fact, FIFO's performance is typically not good
- In addition, FIFO suffers from Belady's Anomaly
  - ✓ there are reference strings for which the fault rate *increases* when the process is given more physical memory

#3: Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - ✓ idea: past experience is a decent predictor of future behavior
  - ✓ on replacement, evict the page that hasn't been used for the longest period of time
    - LRU looks at the past, Belady's wants to look at the future
    - *how is LRU different from FIFO?*
    - in general, it works exceedingly well

Reference string: A B C A B D A D B C B

	FIFO			Belady			LRU		
A B C	A	B	C	A	B	C	A	B	C
A	A	B	C	A	B	C	A	B	C
B	A	B	C	A	B	C	A	B	C
D	D	B	C						
A	D	A	C						
D	D	A	C						
B	D	A	B						
C	C	A	B						
B	C	A	B						

Faults:
FIFO 7
Belady 5
LRU 5

Implementing LRU

- On every memory reference
  - ✓ time stamp each page
- At eviction time:
  - ✓ scan for oldest
- Keep a stack of page numbers
- Course Objectives and Pre-requisites
- Introduction to what an operating system is?
- On every reference, move the page on top of stack
- Problems:
  - ✓ large page lists
  - ✓ no hardware support for time stamps
  - ✓ do something simple & fast that finds an old page
  - ✓ LRU an approximation anyway, a little more won't hurt…

Approximating LRU

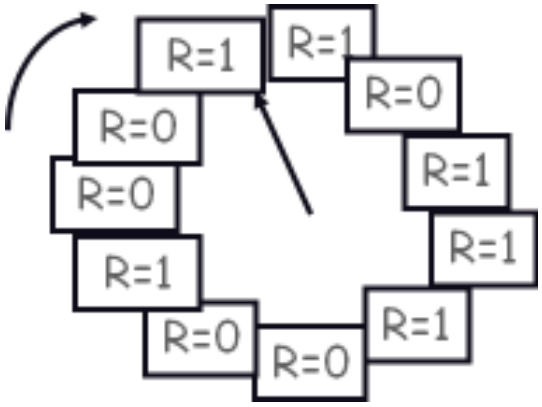
- Many approximations, all use the PTE reference bit
  - ✓ keep a counter for each page
  - ✓ at some regular interval, for each page, do:
    - o if ref bit = 0, increment the counter (hasn't been used)
    - o if ref bit = 1, zero the counter (has been used)
    - o regardless, zero ref bit
  - ✓ the counter will contain the # of intervals since the last reference to the page
    - o page with largest counter is least recently used

#4: LRU Clock

- AKA Not Recently Used (NRU) or Second Chance
  - ✓ replace page that is ¯old enough‖
  - ✓ logically, arrange all physical page frames in a big circle (clock)
    - o just a circular linked list
  - ✓ a ¯clock hand‖ is used to select a good LRU candidate
    - o sweep through the pages in circular order like a clock
    - o if ref bit is off, it hasn't been used recently, we have a victim
      - ▪ so, what is minimum ¯age‖ if ref bit is off?
    - o if the ref bit is on, turn it off and go to next page
  - ✓ arm moves quickly when pages are needed
  - ✓ low overhead if have plenty of memory
  - ✓ if memory is large, ¯accuracy‖ of information degrades
    - o add more hands to fix

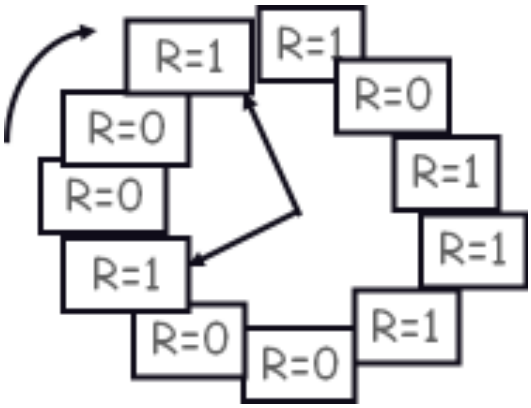
LRU in the real world: the clock algorithm

- Each page has reference bit
  - ✓ hardware sets on use, OS periodically clears
  - ✓ Pages with bit set used more recently than without.
- Algorithm: FIFO + skip referenced pages
  - ✓ keep pages in a circular FIFO list
  - ✓ scan: page's ref bit = 1, set to 0 & skip, otherwise evict.



Problem: what happens as memory gets big?

- Soln: add another clock hand
  - ✓ leading edge clears ref bits
  - ✓ trailing edge is ¯N‖ pages back: evicts pages w/ 0 ref bit
- Implications:
  - ✓ Angle too small?
  - ✓ Angle too large?
- Nth chance algorithm: don't throw page out until hand has
- swept by n times
- OS keeps counter per page -- # of sweeps



- On page fault, OS checks reference bit:
- 1 => clear reference bit and also clear counter, go on
- 0 => increment counter, if < N, go on
- else replace page
- How do we pick N?
- Why pick large N? Better approx to LRU.
- Why pick small N? More efficient; otherwise might have to
- look a long way to find free page.
- Dirty pages have to be written back to disk when replaced.
- Takes extra overhead to replace a dirty page, so give dirty
- pages an extra chance before replacing?
- Common approach:
- clean pages -- use N = 1
- dirty pages -- use N = 2 (and write-back to disk when N=1)

Page replacement : Global or local?

- So far, we‘ve implicitly assumed memory comes from a single global pool (¯Global replacement‖)
  - ✓ when process P faults and needs a page, take oldest page on entire system
  - ✓ Good: adaptable memory sharing. Example if P1 needs 20% of memory and P2 70%, then they will be happy.

  - ✓ Bad: too adaptable. Little protection
    - What happens to P1 if P2 sequentially reads array about the size of memory?

Lecture No. 27

Overview of today's lecture

- Page replacement
- Thrashing
- Working set model
- Page fault frequency
- Copy on write
- Sharing
- Memory mapped files

Page replacement: Global or local?

- So far, we've implicitly assumed memory comes from a single global pool (¯Global replacement‖)
  - ✓ when process P faults and needs a page, take oldest page on entire system
  - ✓ Good: adaptable memory sharing. Example if P1 needs 20% of memory and P2 70%, then they will be happy.

  - ✓ Bad: too adaptable. Little protection
- What happens to P1 if P2 sequentially reads array about the size of memory?

Per-process page replacement

- Per-process
- each process has a separate pool of pages
  - ✓ a page fault in one process can only replace one of this process's frames
  - ✓ isolates process and therefore relieves interference from other processes

  - ✓ but, isolates process and therefore prevents process from using other's (comparatively) idle resources
  - ✓ efficient memory usage requires a mechanism for (slowly) changing the allocations to each pool
  - ✓ Qs: What is ¯slowly‖? How big a pool? When to migrate?

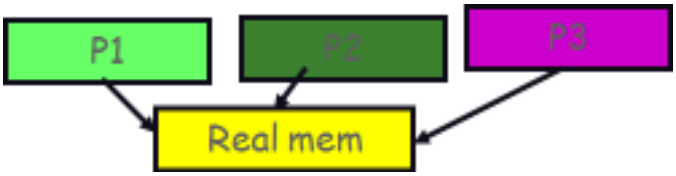
Thrashing

- Thrashing is when the system spends most of its time servicing page faults, little time doing useful work
  - ✓ could be that there is enough memory but a bad replacement algorithm (one

incompatible with program behavior)
- ✓ could be that memory is over-committed
  - too many active processes

Thrashing: exposing the lie of VM

- Thrashing: processes on system require more memory than it has.



  - ✓ Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
  - ✓ Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - ✓ I/O devs at 100% utilization but system not getting much useful work done
- What we wanted: virtual memory the size of disk with access time of of physical memory
- What we have: memory with access time = disk access

Making the best of a bad situation

- Single process thrashing?
  - ✓ If process does not fit or does not reuse memory, OS can do nothing except contain damage.
- System thrashing?
  - ✓ If thrashing arises because of the sum of several processes then adapt:
    - figure out how much memory each process needs
    - change scheduling priorities to run processes in groups whose memory needs can be satisfied (shedding load)
    - if new processes try to start, can refuse (admission control)
- Careful: example of technical vs social.
  - ✓ OS not only way to solve this problem (and others).
  - ✓ solution: go and buy more memory.

The working set model of program behavior

- The working set of a process is used to model the dynamic locality of its memory usage
  - ✓ working set = set of pages process currently ¯needs‖
  - ✓ formally defined by Peter Denning in the 1960‘s
- Definition:
  - ✓ a page is in the working set (WS) only if it was referenced in the last w references
  - ✓ obviously the working set (the particular pages) varies over the life of the program
  - ✓ so does the working set size (the number of pages in the WS)

Working set size

- The working set size changes with program locality
  - ✓ during periods of poor locality, more pages are referenced
  - ✓ within that period of time, the working set size is larger
- Intuitively, the working set must be in memory, otherwise you'll experience heavy faulting (thrashing)
  - ✓ When people ask ¯How much memory does Internet Explorer need?‖, really they're asking ¯what is IE's average (or worst case) working set size?‖

Hypothetical Working Set algorithm

- Estimate for a process
- Allow that process to start only if you can allocate it that many page frames
- Use a local replacement algorithm (e.g. LRU Clock) make sure that ¯the right pages‖ (the working set) are occupying the process's frames
- Track each process's working set size, and re-allocate page frames among processes dynamically
- How do we choose w?

How to implement working set?

- Associate an idle time with each page frame
  - ✓ idle time = amount of CPU time received by process since last access to page
  - ✓ page's idle time > T? page not part of working set
- How to calculate?
  - ✓ Scan all resident pages of a process
    - reference bit on? clear page's idle time, clear use bit
    - reference bit off? add process CPU time (since last scan) to idle time
- Unix:
  - ✓ scan happens every few seconds
  - ✓ T on order of a minute or more

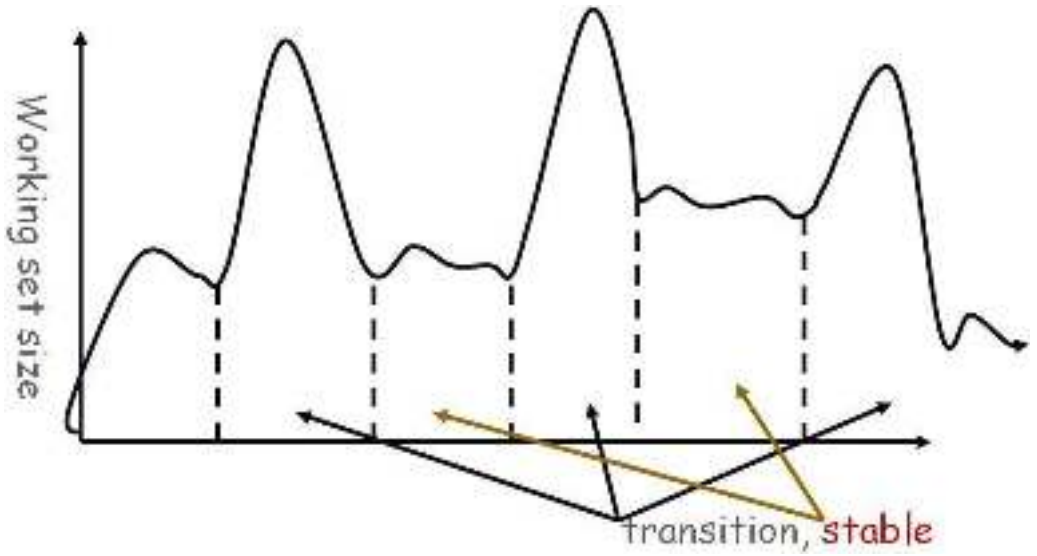
Scheduling details: The balance set

- Sum of working sets of all run-able processes fits in memory? Scheduling same as before.
- If they do not fit, then refuse to run some: divide into two groups
  - ✓ active: working set loaded
  - ✓ inactive: working set intentionally not loaded
  - ✓ balance set: sum of working sets of all active processes
- Long term scheduler:
  - ✓ Keep moving processes from active -> inactive until balance set less than memory size.
  - ✓ Must allow inactive to become active. (if changes too frequently?)
- As working set changes, must update balance set…

Some problems

- T is magic
  - ✓ what if T too small? Too large?
  - ✓ How did we pick it? Usually ¯try and see‖
  - ✓ Fortunately, system‘s aren‘t too sensitive
- What processes should be in the balance set?
  - ✓ Large ones so that they exit faster?
  - ✓ Small ones since more can run at once?
- How do we compute working set for shared pages?

Working sets of real programs



- Typical programs have phases:

Working set less important

- The concept is a good perspective on system behavior.
  - ✓ As optimization trick, it‘s less important: Early systems thrashed a lot, current systems not so much.
- Have OS designers gotten smarter? No. It‘s the hardware (cf. Moore‘s law):
  - ✓ Obvious: Memory much larger (more available for processes)
  - ✓ Less obvious: CPU faster so jobs exit quicker, return memory to free-list faster.
  - ✓ Some app can eat as much as you give, the percentage of them that have ¯enough‖ seems to be increasing.
  - ✓ Very important OS research topic in 80-90s, less so now

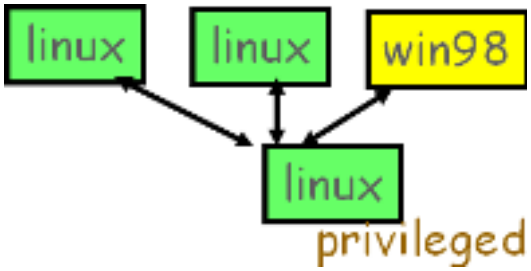
Page Fault Frequency (PFF)

- PFF is a variable-space algorithm that uses a more ad hoc approach
- Attempt to equalize the fault rate among all processes, and to have a ¯tolerable‖ system-wide fault rate

- ✓ monitor the fault rate for each process
- ✓ if fault rate is above a given threshold, give it more memory
  - so that it faults less
- ✓ if the fault rate is below threshold, take away memory
  - should fault more, allowing someone else to fault less

Fault resumption. lets us lie about many things

- Emulate reference bits:
  - ✓ Set page permissions to ¯invalid‖.
  - ✓ On any access will get a fault: Mark as referenced
- Emulate non-existent instructions:
  - ✓ Give inst an illegal opcode. When executed will cause ¯illegal instruction‖ fault. Handler checks opcode: if for fake inst, do, otherwise kill.
- Run OS on top of another OS!
  - ✓ Make OS into normal process
  - ✓ When it does something ¯privileged‖ the real OS will get woken up with a fault.
  - ✓ If op allowed, do it, otherwise kill.
  - ✓ User-mode Linux, vmware.com



Summary

- Virtual memory
- Page faults
- Demand paging
  - ✓ don't try to anticipate
- Page replacement
  - ✓ local, global, hybrid
- Locality
  - ✓ temporal, spatial
- Working set
- Thrashing

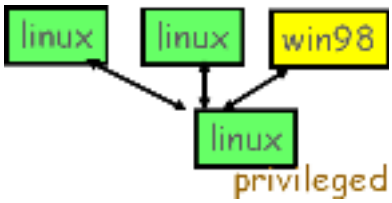
Lecture No. 28

Page Fault Frequency (PFF)

- PFF is a variable-space algorithm that uses a more ad hoc approach
- Attempt to equalize the fault rate among all processes, and to have a ¯tolerable‖ system-wide fault rate
  - ✓ monitor the fault rate for each process
  - ✓ if fault rate is above a given threshold, give it more memory
    - so that it faults less
  - ✓ if the fault rate is below threshold, take away memory
    - should fault more, allowing someone else to fault less

Fault resumption. lets us lie about many things

- Emulate reference bits:
  - ✓ Set page permissions to ¯invalid‖.
  - ✓ On any access will get a fault: Mark as referenced
- Emulate non-existent instructions:
  - ✓ Give inst an illegal opcode. When executed will cause ¯illegal instruction‖ fault. Handler checks opcode: if for fake inst, do, otherwise kill.
- Run OS on top of another OS!
  - ✓ Make OS into normal process
  - ✓ When it does something ¯privileged‖ the real OS will get woken up with a fault.



Sharing

- Private virtual address spaces protect applications from each other
- But this makes it difficult to share data (have to copy)
- We can use shared memory to allow processes to share data using direct memory references

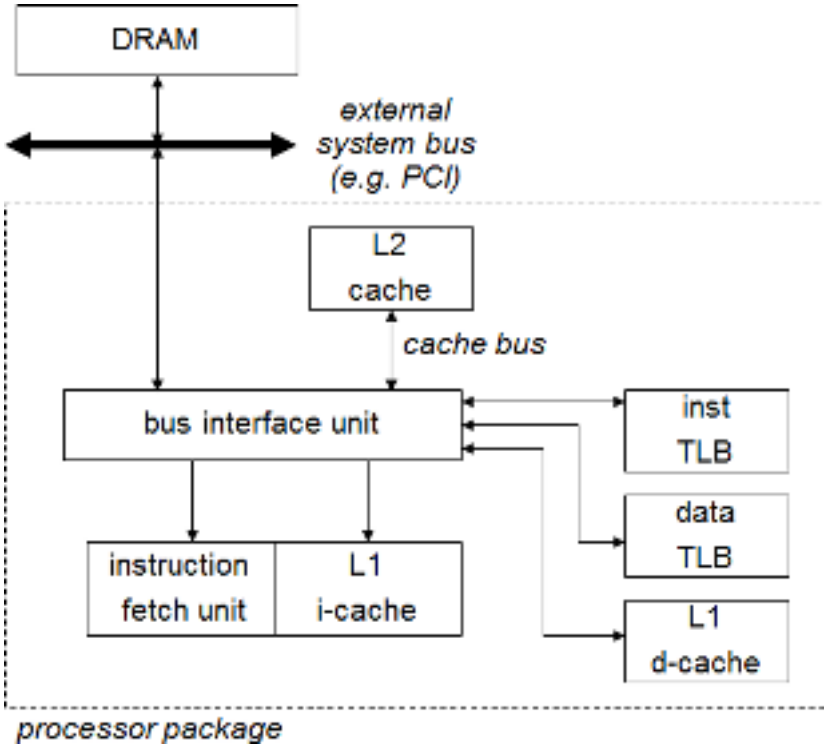
Copy on Write

- OSes spend a lot of time copying data
- Use copy on Write(CoW) to defer large copies as long as possible, hoping to avoid them altogether
- Shared pages are protected as read-only in child
- How does this help fork() (Implemented as Unix vfork())

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of ¯open, read into buffer, operate on buffer, ...‖
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all the pages mapped to file are invalid

P6 Memory System



- 32 bit address space
- 4 KB page size
- L1, L2, and TLBs
  - ✓ 4-way set associative

- inst TLB
  - ✓ 32 entries
  - ✓ 8 sets

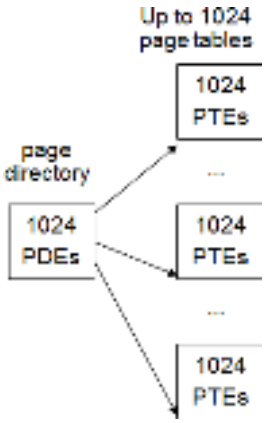
- data TLB
  - ✓ 64 entries
  - ✓ 16 sets

- L1 i-cache and d-cache
  - ✓ 16 KB
  - ✓ 32 B line size
  - ✓ 128 sets

- L2 cache
  - ✓ unified
  - ✓ 128 KB -- 2 MB

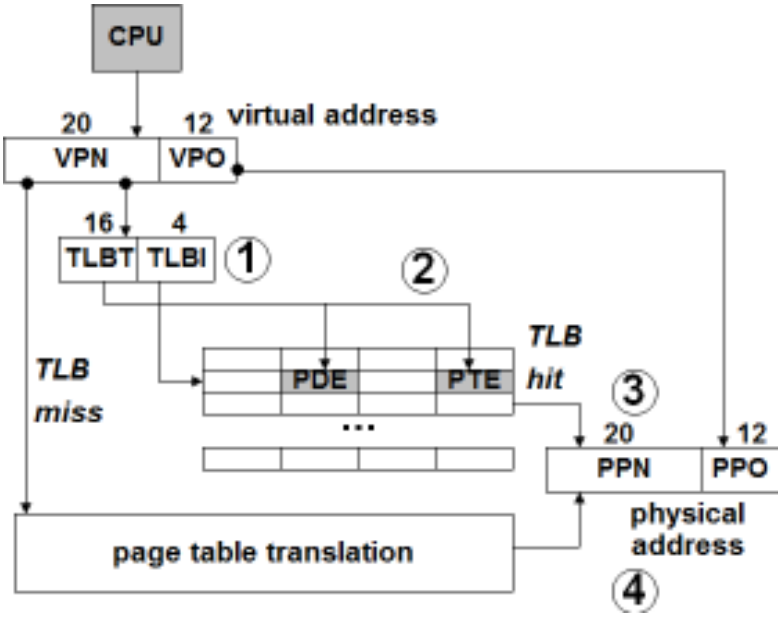
P6 2-level Page Table Structure

- Page directory
  - ✓ 1024 4-byte page directory entries (PDEs) that point to page tables
- Page tables:
  - ✓ 1024 4-byte page table entries (PTEs) that point to pages.



Translating with the P6 TLB

- 1. Partition VPN into TLBT and TLBI.
- 2. Is the PTE for VPN cached in set TLBI?
- 3. Yes: then build physical address.
- 4. No: then read PTE (and PDE if not cached) from memory and build physical address.



Lecture No. 29

File systems

- The implementation of the abstraction for secondary storage
- Logical organization of files into directories
- Sharing of data between processes, people and machines

Files

- A file is a collection of data with some properties
  - ✓ contents, size, owner, last read/write time, protection …

Basic operations

Unix	NT
create(name)	CreateFile(name, CREATE)
open(name, mode)	CreateFile(name, OPEN)
read(fd, buf, len)	ReadFile(handle, …)
write(fd, buf, len)	WriteFile(handle, …)
sync(fd)	FlushFileBuffers(handle, …)
seek(fd, pos)	SetFilePointer(handle, …)
close(fd)	CloseHandle(handle, …)
unlink(name)	DeleteFile(name)
rename(old, new)	CopyFile(name)
	MoveFile(name)

File access methods

- Sequential access
  - ✓ read bytes one at a time, in order
- Direct access
  - ✓ random access given a block/byte number
- Record access
  - ✓ file is array of fixed- or variable-sized records
- Indexed access
  - ✓ FS contains an index to a particular field of each record in a file
  - ✓ apps can find a file based on value in that record (similar to DB)

Directories

- Most file systems support multi-level directories
  - ✓ naming hierarchies (/, /usr, /usr/local, /usr/local/bin, …)
- Most file systems support the notion of current directory
  - ✓ absolute names: fully-qualified starting from root of FS
  - ✓ bash$ cd /usr/local
- relative names: specified with respect to current directory
  - ✓ bash$ cd /usr/local (absolute)
  - ✓ bash$ cd bin (relative, equivalent to cd /usr/local/bin)

Path name translation

- Let‘s say you want to open ¯/one/two/three‖
  fd = open(¯/one/two/three‖, O_RDWR);

File protection

- FS must implement some kind of protection system
  - ✓ to control who can access a file (user)
  - ✓ to control how they can access it (e.g., read, write, or exec)

File System Layout

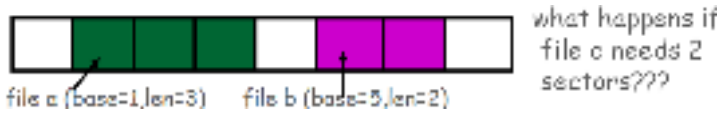
How do file systems use the disk to store files?

- File systems define a block size(e.g., 4kb)
- A ¯Master Block‖ determines location of root directory
- A free map determines which blocks are free, allocated

Disk Layout Strategies

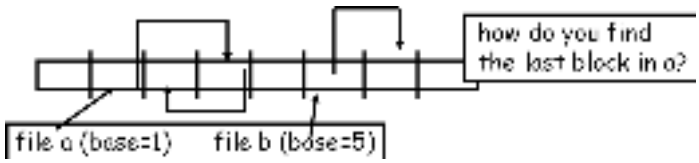
1. Contiguous allocation
   - ✓ Kinke memory
   - ✓ Fast, simplifies directory access
   - ✓ Inflexible, causes fragmentation, needs compacton
2. Linked structure
3. Indexed structure (indirection, hierarchy)

Simple mechanism: contiguous allocation

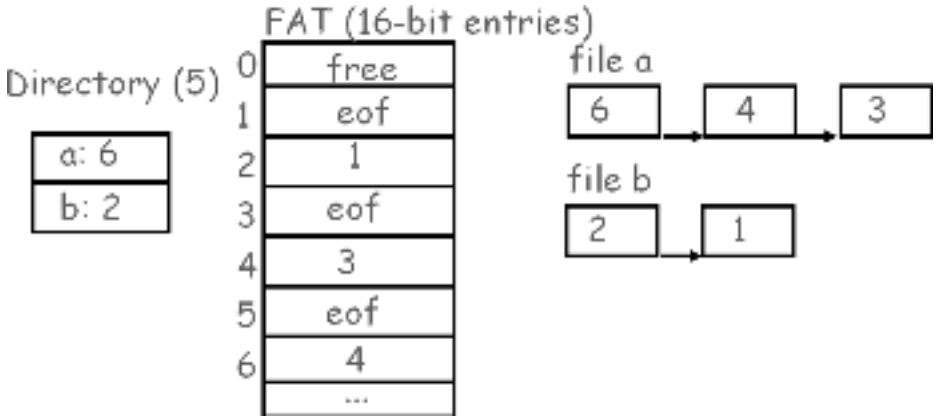


Linked files

- pro: easy dynamic growth & sequential access, no fragmentation
- con?
- Examples (sort-of): Alto, TOPS-10, DOS FAT



Example: DOS FS (simplified)



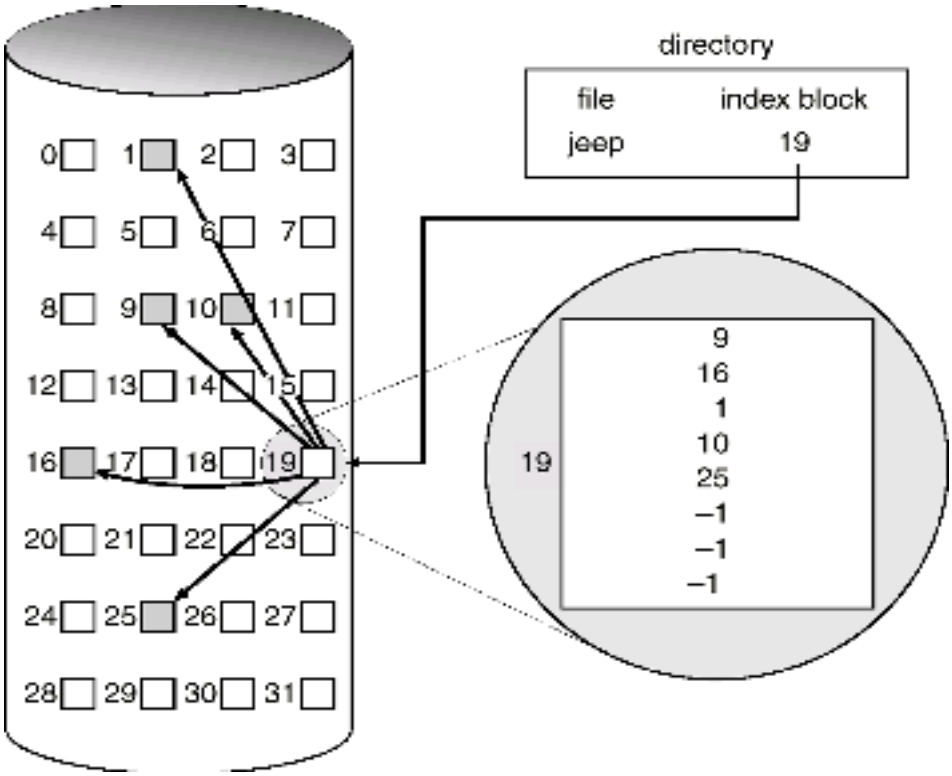
Lecture No. 30

Consistency problem

- The Big File System Promise: persistence
  - ✓ it will hold your data until you explicitly delete it
  - ✓ (and sometimes even beyond that: backup/restore)
- What's hard about this? Crashes
  - ✓ If your data is in main memory, a crash destroys it.
  - ✓ Performance tension: need to cache everything. But if so, then crash = lose everything.
  - ✓ More fundamental: interesting ops = multiple block modifications, but can only atomically modify disk a sector at a time.

Indexed Allocation

- Need index table
- Random access
- Brings all pointers together into the *index block.*



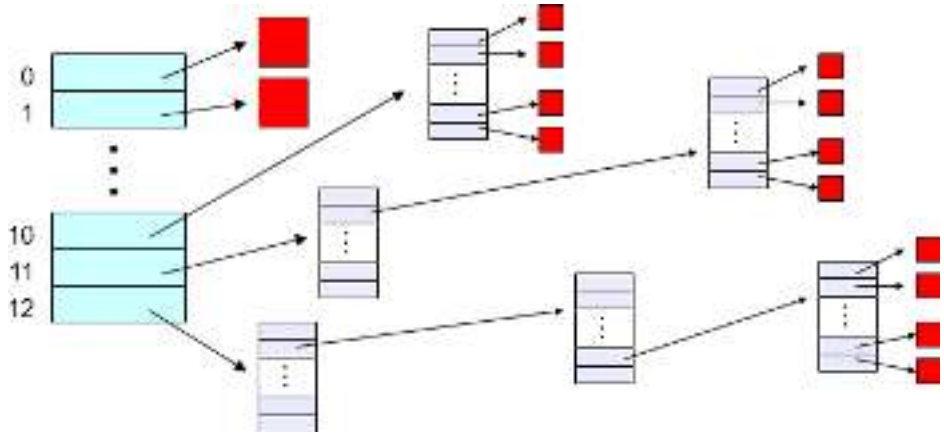
UNIX: All disks are divided into five parts

- Boot block
  - ✓ can boot the system by loading from this block
- Superblock
  - ✓ specifies boundaries of next 3 areas, and contains head of freelists of i-nodes and file blocks

- i-node area
  - ✓ contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area
  - ✓ fixed-size blocks; head of freelist is in the superblock
- Swap area
  - ✓ holds processes that have been swapped out of memory

The "block list" portion of the i-node (Unix Version 7)

- Each i-node contains 13 block pointers
- first 10 are ¯direct pointers‖ (pointers to 512B blocks of file data)
- then, single, double, and triple indirect pointers



- A later version of Bell Labs Unix utilized 12 direct pointers rather than 10
- Berkeley Unix went to 1KB block sizes
  - ✓ What's the effect on the maximum file size?
    - 256x256x256x1K = 17 GB
- Suppose you went 4KB blocks?
  - ✓ 1Kx1Kx1Kx4K = 4TB

i-node Format

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a ¯special file‖ (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
- Link count: number of directories referencing this i-node

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality

Caching Writes

- On a write, some apllications assume that data makes it through the buffer cache and onto the disk
- Several ways to compensate for this
  - ¯write-behind‖
  - Battery backed-up RAM(NVRAM)

Read Ahead

- Many file systems implement ¯read ahead‖
- For sequentially accessed files can be a big win
  - ✓ Unless blocks for the file are scatter across the dis
  - ✓ File systems try to prevent that, though (during allocaiton)

Creating synonyms: Hard and soft links

- More than one dir entry can refer to a given file
- Unix stores count of pointers (¯hard links‖) to inode
- to make: ¯ln foo bar‖ creates a synonym (‗bar') for ‗
- Soft links:
- also point to a file (or dir), but object can be delete underneath it (or never even exist).

Lecture No. 31

Consistency problem

- The Big File System Promise: persistence
  - ✓ it will hold your data until you explicitly delete it
  - ✓ (and sometimes even beyond that: backup/restore)
- What's hard about this? Crashes
  - ✓ If your data is in main memory, a crash destroys it.
  - ✓ Performance tension: need to cache everything. But if so, then crash = lose everything.
  - ✓ More fundamental: interesting ops = multiple block modifications, but can only atomically modify disk a sector at a time.

What to do? Three main approaches

- Sol'n 1: Throw everything away and start over.
  - ✓ Done for most things (e.g., interrupted compiles).
  - ✓ Probably not what you want to happen to your email
- Sol'n 2: Make updates seem indivisible (atomic)
  - ✓ Build arbitrary sized atomic units from smaller atomic ones (e.g., a sector write)
  - ✓ similar to how we built critical sections from locks, and locks from atomic instructions
- Sol'n 3: Reconstruction
  - ✓ try to fix things after crash (many FSes do this: ¯fsck‖)
  - ✓ usually do changes in stylized way so that if crash happens, can look at entire state and figure out where you left off

Arbitrary-sized atomic disk ops

- For disk: construct a pair of operations:
  - ✓ put(blk, address) : writes data in blk on disk at address
  - ✓ get(address) -> blk : returns blk at given disk address
  - ✓ such that ¯put‖ appears to place data on disk in its entirety or not at all and ¯get‖ returns the latest version
  - ✓ what we have to guard against: a system crash during a call to ¯put‖, which results in a partial write.

SABRE atomic disk operations

```
void atomic-put(data)          blk atomic-get()
version++; # unique integer V1 := get(V1)
put(version, V1);              D1data := get(D1);
put(data, D1);                 V2 := get(V2);
put(version, V2);              D2data := get(D2);
put(data, D2);                 if(V1 == V2)
                                  return D1data;
                               else
                                  return D2data;
```

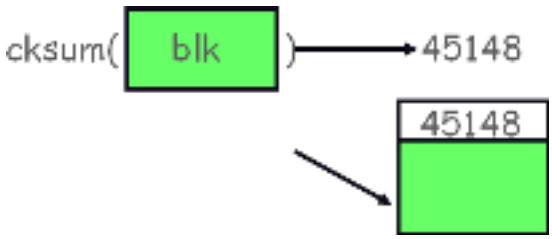
Does it work?

- Assume we have correctly written to disk:
  - ✓ { #2, ¯seat 25‖, #2, ¯seat 25‖ }
- And now we want to change seat 25 to seat 31.
- The system has crashed during the operation atomic-put(¯seat 31‖)
- There are 6 cases, depending on where we failed in atomic-put:

put # fails	possible disk contents	atomic-get returns?
before	{#2, ¯seat 25‖, #2, ¯seat 25‖}	
the first	{#2.5, ¯seat 25‖, #2, ¯seat 25‖ }	
the second	{#3, ¯seat 35‖, #2, ¯seat 25‖}	
the third	{#3, ¯seat 31‖, #2.5, ¯seat 25‖}	
the fourth	{#3, ¯seat 31‖, #3, ¯seat 35‖}	
after	{#3, ¯seat 31‖, #3, ¯seat 31‖}	

Two assumptions

- Once data written, the disk returns it correctly



- Disk is in a correct state when atomic-put starts

Recovery

```
void recover(void) {
   V1data = get(V1); # following 4 ops same as in a-get
   D1data = get(D1);
   V2data = get(V2);
   D2data = get(D2);
   if (V1data == V2data)
       if(D1data != D2data)
          # if we crash & corrupt D2, will get here again.
           put(D1data, D2);
   else
       # if we crash and corrupt D1, will get back here
       put(D2data, D1);
       # if we crash and corrupt V1, will get back here
       put(V2data, V1);
```

The power of state duplication

- Most approaches to tolerating failure have at their core a similar notion of state duplication
  - ✓ Want a reliable tire? Have a spare.
  - ✓ Want a reliable disk? Keep a tape backup. If disk fails, get data from backup. (Make sure not in same building.)
  - ✓ Want a reliable server? Have two, with identical copies of the same information. Primary fails? Switch.

Fighting failure

- In general, coping with failure consists of first defining a failure model composed of
  - ✓ Acceptable failures. E.g., the earth is destroyed by aliens from Mars. The loss of a file viewed as unavoidable.
  - ✓ Unacceptable failures. E.g. power outage: lost file not ok

Unix file system invariants

- File and directory names are unique
- All free objects are on free list
  - ✓ + free list only holds free objects
- Data blocks have exactly one pointer to them
- Inode‘s ref count = the number of pointers to it
- All objects are initialized
  - ✓ a new file should have no data blocks, a just allocated block should contain all zeros.
  - ✓ A crash can violate every one of these!

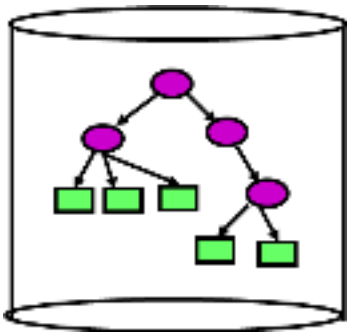
Unused resources marked as “allocated”

- Rule:never persistently record a pointer to any object still on the free list
- Dual of allocation is deallocation. The problem happens there as well.
- Truncate:
  - ✓ 1: set pointer to block to 0.
  - ✓ 2: put block on free list
  - ✓ if the writes for 1 & 2 get reversed, can falsely think something is freed
  - ✓ Dual rule: never reuse a resource before persistently nullifying all pointers to it.

Lecture No. 32

Reactive: reconstruct freelist on crash

- How?
  - ✓ Mark and sweep garbage collection!
  - ✓ Start at root directory
  - ✓ Recursively traverse all objects, removing from free list
  - ✓ Good: is a fixable error. Also fixes case of allocated objects marked as free.
  - ✓ Bad: Expensive. requires traversing all live objects and makes reboot slow.

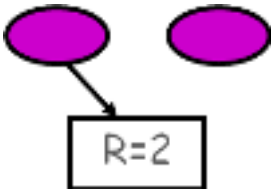


Deleting a file

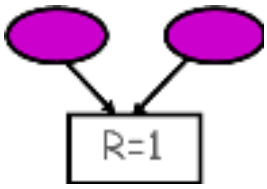
- Unlink(¯foo‖)
  - ✓ traverse current working directory looking for ¯foo‖
    - o if not there, or wrong permissions, return error
  - ✓ clear directory entry
  - ✓ decrement inode‘s reference count
  - ✓ if count is zero, free inode and all blocks it points to

Bogus reference count

- Reference count too high?
  - ✓ inode and its blocks will not be reclaimed
    - o (2 gig file = big trouble)
  - ✓ what if we decrement count before removing pointer?



- Reference count too low
  - ✓ real danger: blocks will be marked free when still in use
  - ✓ major security hole: password file stored in ¯freed‖ blocks.
  - ✓ Proactive: Never decrement reference counter before removing pointer to object. Do synchronous writes
  - ✓ Reactive: fix with mark & sweep



Creating a new file

- Our rule:
  - ✓ never (persistently) point to a resource before it has been initialized
- Implication: file create 2 or 3 synchronous writes!
  - ✓ Write 1: Write out the modified free list to disk. Wait.
  - ✓ Write 2: Write out 0s to initialize inode. Wait.
  - ✓ Write 3: write out directory block containing pointer to inode.

Growing a file

- write(fd, &c, 1)
  - ✓ translate current file position (byte offset) into location in inode (or indirect block, double indirect, …)
  - ✓ if inode already points to a block, modify the block and write back
  - ✓ otherwise: (1) allocate a free block, (2) write the modified free list to disk, wait (3) write the newly allocated block to disk, wait (4) write the pointer (inode) to the new block to disk, wait

Conservatively moving a file

- Rule:
  - ✓ never reset old pointer to object before a new pointer has been set
- mv foo bar (assume foo -> inode # 41)
  - ✓ lookup foo in current working directory
    - o if does not exist or wrong permissions, return error
  - ✓ lookup bar in current working directory
    - o if wrong permissions return error
  - ✓ 0: increment inode 41‘s reference count. Write inode to disk, Wait.
  - ✓ 1: insert (¯bar‖, inode 41). , write bar directory block to disk, Wait.
  - ✓ 2: destroy (¯foo‖, inode 41). Write foo directory block to disk, Wait.
  - ✓ 3: decrement inode 41‘s reference count, write inode to disk, wait
- costly: 3 synchronous writes!

Summary: the two fixable cases

- Case 1: Free list holds pointer to allocated block
  - o cause: crash during allocation or deallocation
  - o rule: make free list conservative
  - o free: nullify pointer before putting on free list
  - o allocate: remove from free list before adding pointer
- Case 2: Wrong reference count
  - o too high = lost memory (but safe)
  - o too low = reuse object still in use (very unsafe)
  - o cause: crash while forming or removing a link

- rule: conservatively set reference count to be high
- unlink: nullify pointer before reference count decrement
- link: increment reference count before adding pointer

- Alternative: ignore rules and fix on reboot.

FSCK: Reconstructing File System

```
# mark and sweep + fix reference counts
worklist := { root directory }
  while e := pop(worklist) # sweep down from roots
    foreach pointer p in e
        # if we haven't seen p and p contains pointers, add
        if p.type != dataBlock and !seen{p}
                push(worklist, p);
                refs{p} = p.refcnt; # p's notion of pointers to it
        seen{p} += 1;                    # count references to p
        freelist[p] = ALLOCATED; # mark not free
 foreach e in refs                       # fix reference counts
      if(seen{e} != refs{e})
        e.refcnt = seen{e};
       e.dirty = true;
```

Write ordering

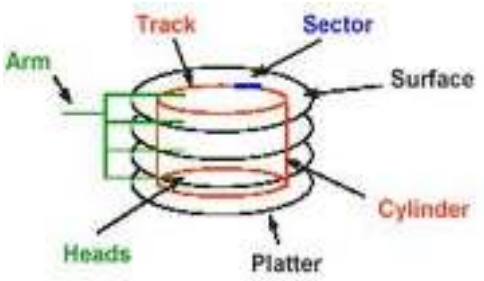
- Synchronous writes expensive
  - sol'n have buffer cache provide ordering support
- Whenever block ¯a‖ must be written before block ¯b‖ insert a dependency
  - Before writing any block, check for dependencies

- (when deadlock?)
- To eliminate dependency, synchronously write out each block in chain until done.
  - Block B & C can be written immediately
  - Block A requires block B be synchronously written first.

Lecture No. 33

Physical Disk Management

- Dis components
  - ✓ Platters
  - ✓ Surfaces
  - ✓ Tracks
  - ✓ Sectors
  - ✓ Cylinders
  - ✓ Arm
  - ✓ Heads



Disk Interaction

- Specifying disk requests requires a lot of info:
  - ✓ Cylinder #, surface #, track #, sector #, transfer size, . . .
- Current disks provide a higher-level interface (SCSI)
  - ✓ The disk exports its data as a logical array of blocks [0 … N]
    - o Disk maps logical blocks to cylinder/ surface/ track/ sector.

Some useful facts

- Disk reads/writes in terms of sectors, not bytes
  - ✓ read/write single sector or adjacent groups
- How to write a single byte? ¯Read-modify-write‖
  - ✓ read in sector containing the byte
  - ✓ modify that byte
  - ✓ write entire sector back to disk
  - ✓ key: if cached, don‘t need to read in
- Sector = unit of atomicity.
  - ✓ sector write done completely, even if crash in middle
    - o (disk saves up enough momentum to complete)
  - ✓ larger atomic units have to be synthesized by OS

Disk Scheduling

- Because seeks are so expensive (milliseconds!), the OS tries to schedule disk requests that are queued waiting for the disk
  - ✓ FCFS (do nothing)
    - o Reasonable when load is low
    - o Long waiting times for long request queues
  - ✓ SSTF (shortest seek time first)
    - o Minimize arm movement (seek time), maximize request rate
    - o Favors middle blocks
  - ✓ SCAN (elevator)
    - o Service requests in one direction until done, then reverse
  - ✓ C-SCAN
    - o Like SCAN, but only go in one direction (typewriter)

Some useful trends

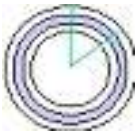
- Disk bandwidth and cost/bit improving exponentially
  - ✓ similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving *very* slowly
  - ✓ why? require moving physical object (disk arm)
- Some implications:
  - ✓ disk accesses a huge system bottleneck & getting worse
  - ✓ bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
    - o Result? Can improve performance if you can read lots of related stuff.
    - o How to get related stuff? Cluster together on disk
- Memory size increasing faster than typical workload size
  - ✓ More and more of workload fits in file cache
  - ✓ disk traffic changes: mostly writes and new data

BSD 4.4 Fast File system (FFS)

- Used a minimum of 4096 size disk block
- Records the block size in superblock
- Multiple file systems with different block sizes can co-reside
- Improves performance in several ways
- Superblock is replicated to provide fault tolerance

FFS Cylinder Groups

- FFS defines cylinder groups as the unit of disk locality, and it factors locality into allocation choices.
  - ✓ Typical: thousands of cylinder, dozens of groups
  - ✓ Strategy: Place ¯related‖ data books in the same cylinder group whenever possible
    - o Seek latency is proportional to seek distance
  - ✓ Smear large files across groups
    - o Place a run of contiguous blocks in each group
  - ✓ Reserve inode blocks in each cylinder group
    - o This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files)



FFS Allocation Policies

1. Allocate file inodes close to their containing directories.
   - ✓ For mkdir, select a cylinder group with a more-than-average number of free inodes.
   - ✓ For creat, place inode in the same group as the parent.

2. Concentrate related file data blocks in cylinder groups.
    - ✓ Most files are read and written sequentially.
    - ✓ Place initial blocks of a file in the same group as its inode.
        - How should we handle directory blocks?
    - ✓ Place adjacent logical blocks in the same cylinder group.
        - Logical block n+1 goes in the same group as block n.
        - Switch to a different group for each indirect block.

Representing Small Files

- Internal fragmentation in the file system blocks can waste significant space for small files.
- FFS solution: optimize small files for space efficiency.
    - ✓ Subdivide blocks into 2/ 4/ 8 fragments (or just frags).

Clustering in FFS

- Clustering improves bandwidth utilization for large files read and written sequentially.
- FFS can allocate contiguous runs of blocks ¯most of the time‖ on disks with sufficient free space.

FFS consistency and recovery

- Reconstructs free list and reference counts on reboot
- Enforces two invariants:
    - directory names always reference valid inodes
    - no block claimed by more than one inode
- Does this with three ordering rules:
    - write newly allocated inode to disk before name entered in directory
    - remove directory name before inode deallocated
    - write deallocated inode to disk before its blocks are placed on free list
- File creation and deletion take 2 synchronous writes
- Why does FFS need third rule? Inode recovery

FFS: inode recovery

- Files can be lost if directory destroyed or crash happens before link can be set
    - New twist: FFS can find lost inodes
- Facts:
    - FFS pre-allocates inodes in known locations on disk
    - Free inodes are initialized to all 0s.
- So?
    - Fact 1 lets FFS find all inodes (whether or not there are any pointers to them)
    - Fact 2 tells FFS that any inode with non-zero contents is (probably) still in use.
    - fsck places unreferenced inodes with non-zero contents in the lost+found directory

Lecture No. 34

Log Structured File Systems

- Log structured (or journaling) file systems record each update to the file system as a transaction.
- All transactions are written to a log. A transaction is considered committed once it is written to the log. However, the file system may not yet be updated.

Logging

- Idea: lets keep track of what operations are in progress and use this for recovery. It's keep a ¯log‖ of all operations, upon a crash we can scan through the log and find problem areas that need fixing.

Implementation

- Add log area to disk.

- Always write changes to log first - called *write-ahead logging* or *journaling*.
- Then write the changes to the file system.
- All reads go to the file system.
- Crash recovery - read log and correct any inconsistencies in the file system.

Issue - Log management

- Observation: Log only needed for crash recovery
- Checkpoint operation – make in-memory copy of file system (file cache) consistent with disk.
- After a checkpoint, can truncate log and start again.
- Most logging file systems only log metadata (file descriptors and directories) and not file data to keep log size down.

Issue – Performance

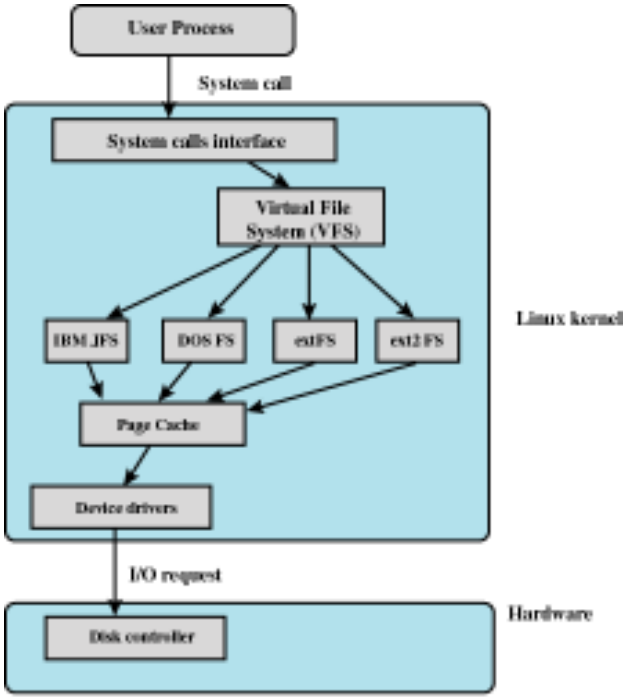
- Two disk writes (on different parts of the disk) for every change?
- Synchronous writes are on every file system change?
- Observation: Log writes are sequential on disk so even synchronous writes can be fast.
- Best performance if log on separate disk.

Current trend is towards logging FS

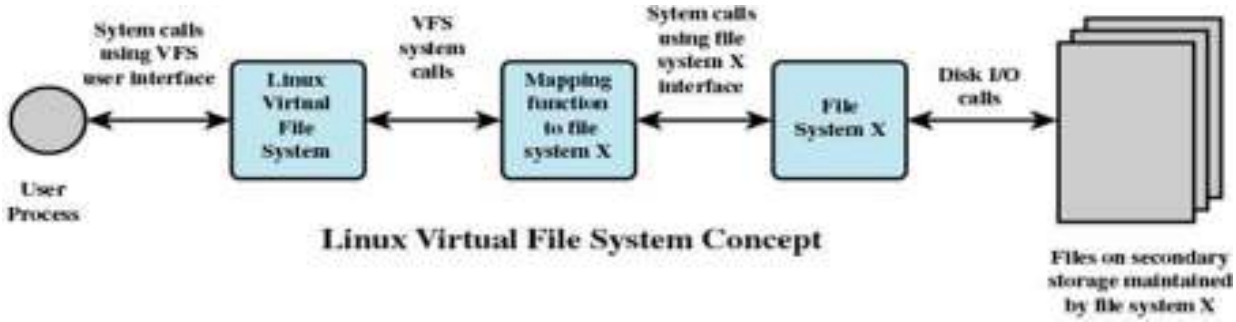
- Fast recovery: recovery time O(active operations) and not O(disk size)
- Better performance if changes need to be reliable
- If you need to do synchronous writes, sequential synchronous writes are much faster than non-sequential ones.
- Examples: Windows NTFS, Veritas on Sun
- Many competing logging file system for Linux

Linux Virtual File System

- Uniform file system interface to user processes
- Represents any conceivable file system's general feature and behavior
- Assumes files are objects that share basic properties regardless of the target file system



Linux Virtual File System Context



Linux Virtual File System Concept

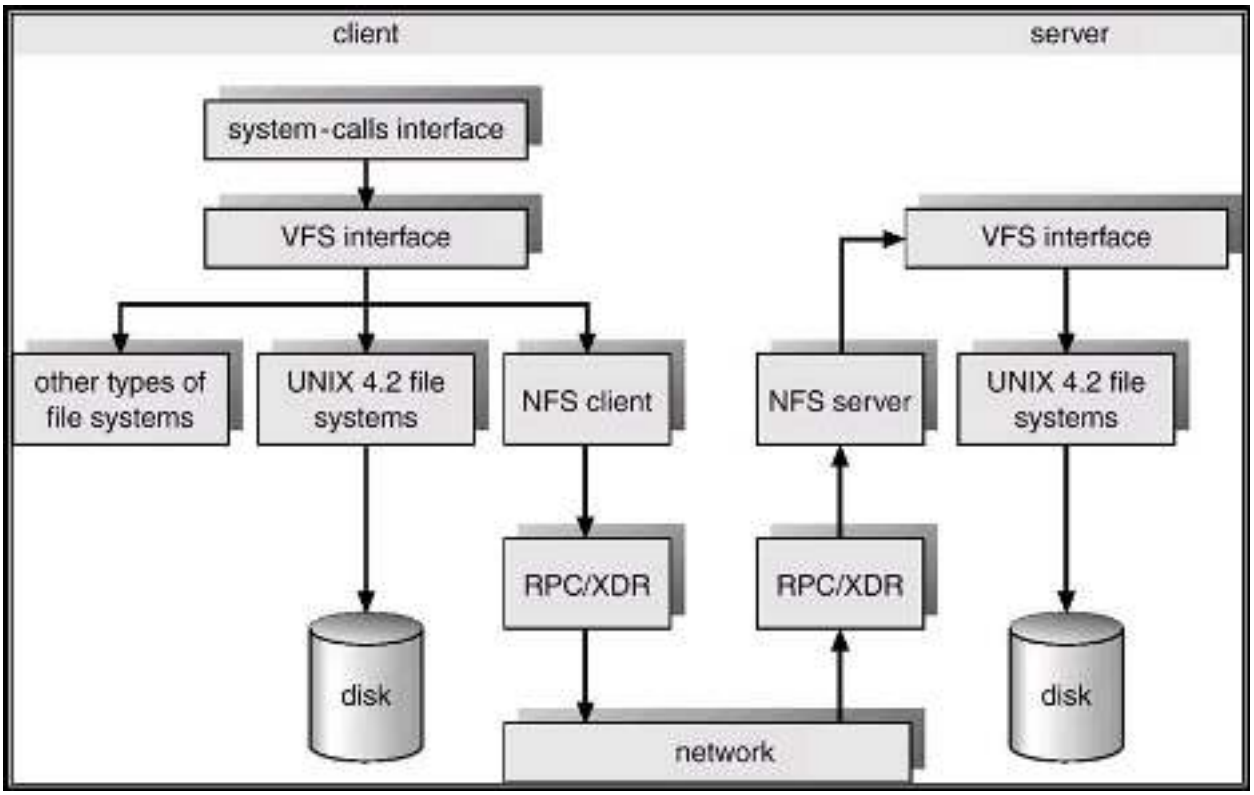
Primary Objects in VFS

- Superblock object
  - ✓ Represents a specific mounted file system
- Inode object
  - ✓ Represents a specific file
- Dentry object
  - ✓ Represents a specific directory entry
- File object
  - ✓ Represents an open file associated with a process

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet.

Schematic View of NFS Architecture



Lecture No. 35

Overview of today's lecture

- Goals of I/O software
- Layers of I/O software
- Direct Vs memory mapped I/O
- Interrupt driven I/O
- Polled I/O
- Direct Memory Access (DMA)

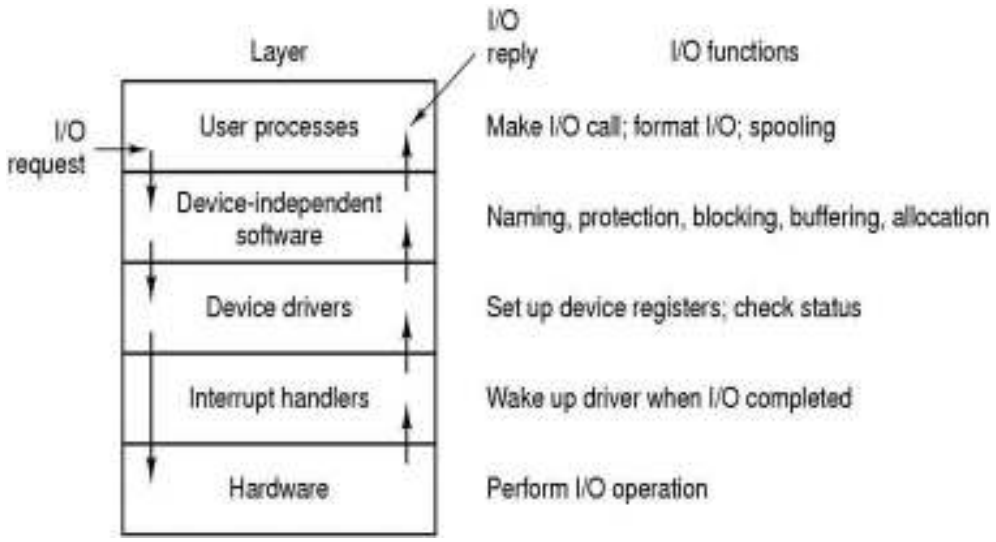
Principles of I/O Software

- Goals of I/O Software
- Device independence
  - programs can access any I/O device without specifying device in advance (floppy, hard drive, or CD-ROM)
- Uniform naming
  - name of a file or device a string or an integer not depending on which machine
- Error handling
  - handle as close to the hardware as possible
- Synchronous vs. asynchronous transfers
  - blocked transfers vs. interrupt-driven
- Buffering
  - data coming off a device cannot be stored in final destination immediately

Device Controllers

- I/O devices have components:
  - mechanical component
  - electronic component
- The electronic component is the device controller
  - may be able to handle multiple devices
- Controller's tasks
  - convert serial bit stream to block of bytes
  - perform error correction as necessary
  - make available to main memory

Layers of the I/O system and the main functions of each layer



Memory mapped I/O Vs Direct I/O

- Memory mapped I/O
- Device controller‘s registers and internal memory are directly mapped into the processor‘s address space
- Direct I/O
- Device controller‘s registers and internal memory is accessible via special instructions in the assembly language instruction set. The arguments to these instructions are usually called ports.

Programmed I/O

```
copy_from_user(buffer, p, count);          /* p is the kernel bufer */
for (i = 0; i < count; i++) {              /* loop on every character */
    while (*printer_status_reg != READY) ; /* loop until ready */
    *printer_data_register = p[i];         /* output one character */
}
return_to_user( );
```

- Writing a string to the printer using programmed I/O

Interrupt-Driven I/O

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

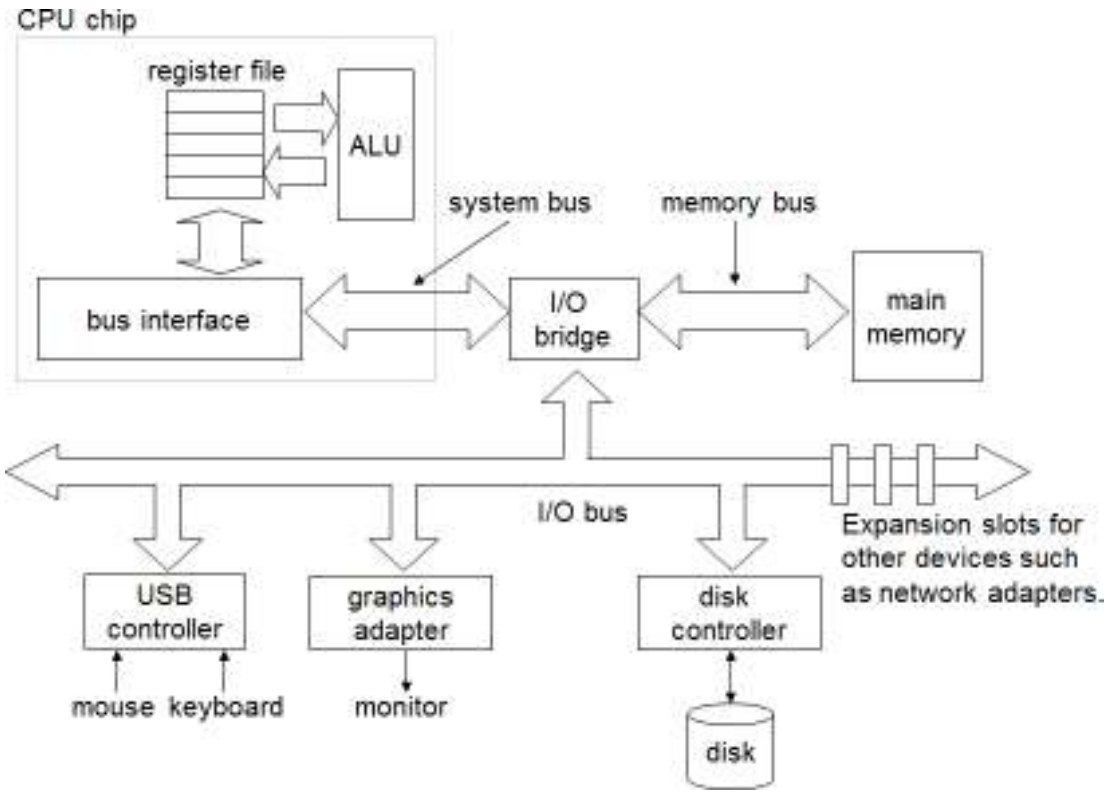
(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

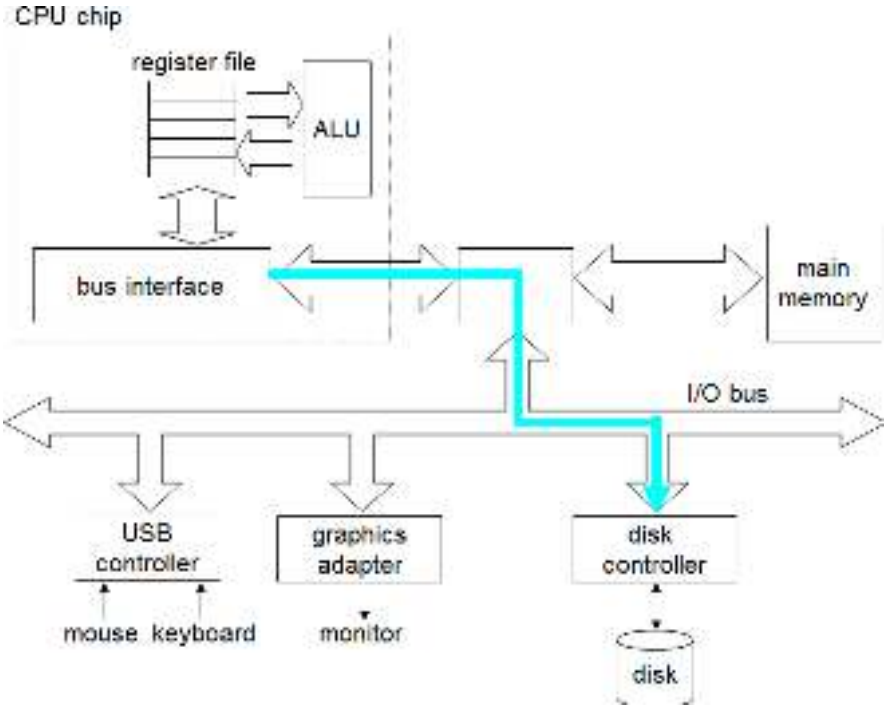
- Writing a string to the printer using interrupt-driven I/O
    a. Code executed when print system call is made
    b. Interrupt service procedure

A Typical Hardware System



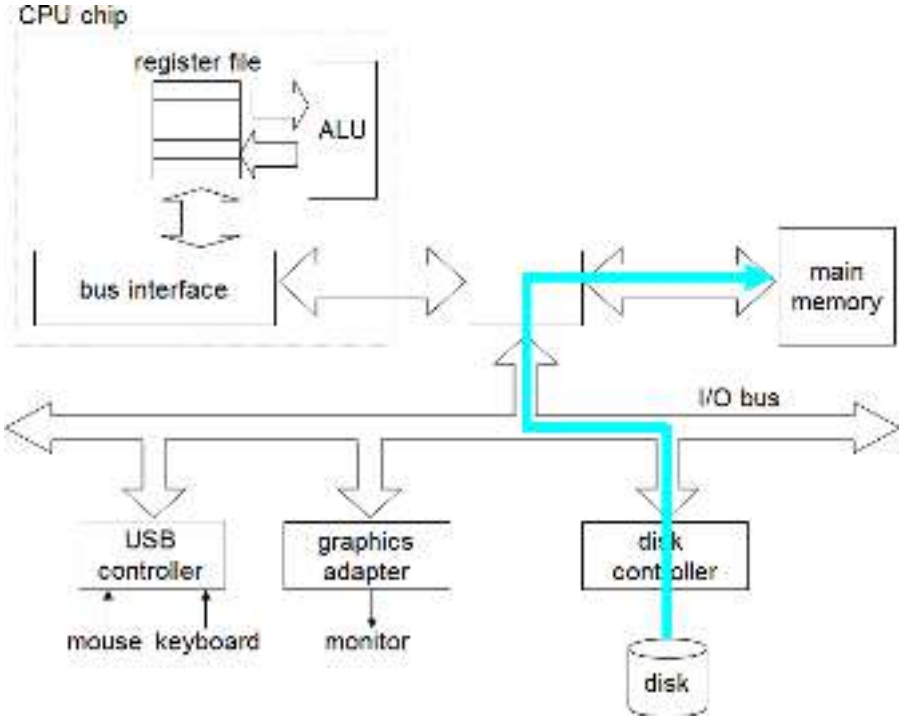
Reading a Disk Sector: Step 1

- CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.



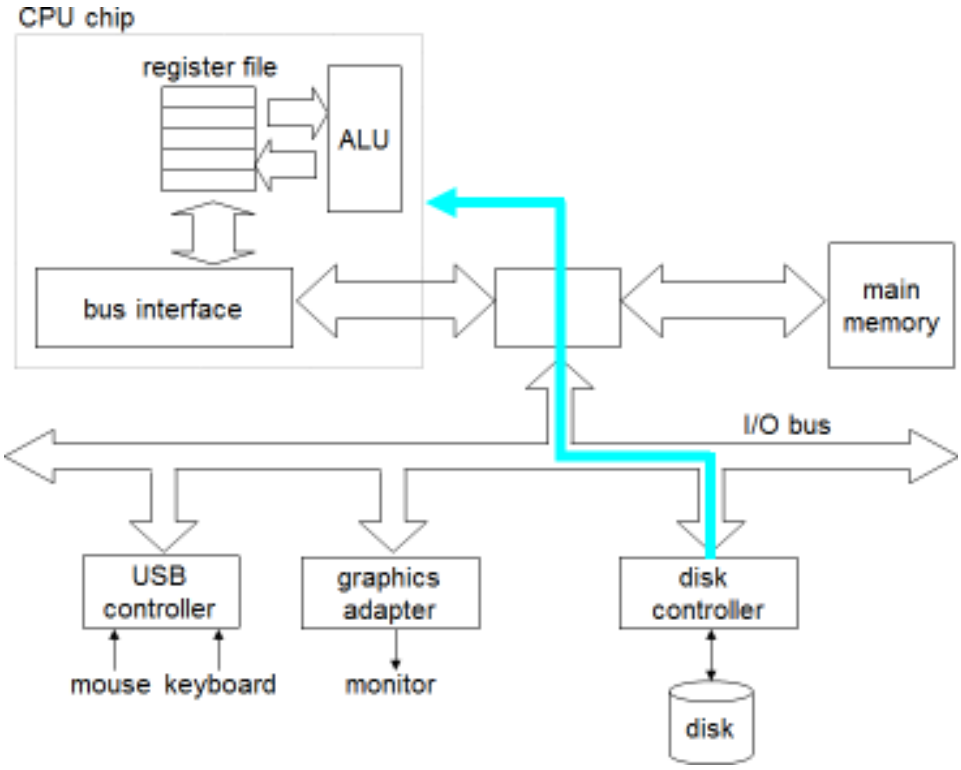
Reading a Disk Sector: Step 2

- Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

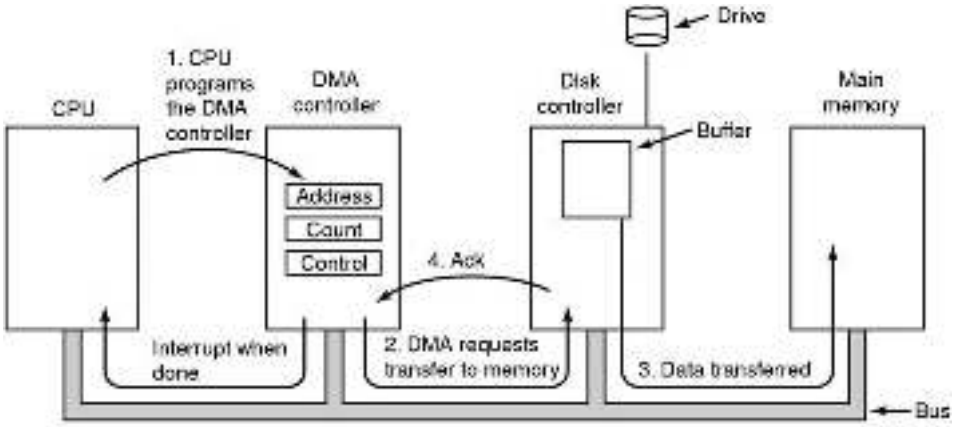


Reading a Disk Sector: Step 3

- When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special ¯interrupt‖ pin on the CPU)



Direct Memory Access (DMA)



Operation of a DMA transfer

I/O Using DMA

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

- Printing a string using DMA
    a. code executed when the print system call is made
    b. interrupt service procedure

Lecture No. 36

Overview of today's lecture

- Device independent I/O software layer
- Buffered and un-buffered I/O
- Block and character devices
- Network devices
- Kernel I/O subsystem and data structures
- Life cycle of a typical I/O request
- Life cycle of a typical network I/O request

Device-Independent I/O Software

Uniform interfacing for device drivers Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Functions of the device-independent I/O software

- Usually interfaces to device drivers through a standard interface
- Buffering options
    1. Unbuffered input
    2. Buffering in user space
    3. Buffering in the kernel followed by copying to user space
    4. Double buffering in the kernel

Block and Character Devices

- Block devices include disk drives
    - ✓ Commands include read, write, seek
    - ✓ Raw I/O or file-system access
    - ✓ Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
    - ✓ Commands include get, put
    - ✓ Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9*i*/2000/XP include socket interface
    - ✓ Separates network protocol from network operation
    - ✓ Includes select functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers

- Provide current time, elapsed time, timer
- Programmable interval timer used for timings, periodic interrupts
- ioctl (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O

- Blocking - process suspended until I/O completed
  - ✓ Easy to use and understand
  - ✓ Insufficient for some needs

- Nonblocking - I/O call returns as much as available
  - ✓ User interface, data copy (buffered I/O)
  - ✓ Implemented via multi-threading inside the kernel
  - ✓ Returns quickly with count of bytes read or written

- Asynchronous - process runs while I/O executes
  - ✓ Difficult to use
  - ✓ I/O subsystem signals process when I/O completed

Kernel I/O Subsystem

- Scheduling
  - ✓ Some I/O request ordering via per-device queue
  - ✓ Some OSs try fairness

- Buffering - store data in memory while transferring between devices
  - ✓ To cope with device speed mismatch
  - ✓ To cope with device transfer size mismatch
  - ✓ To maintain ¯copy semantics‖

- Caching - fast memory holding copy of data
  - ✓ Always just a copy
  - ✓ Key to performance

- Spooling - hold output for a device from multiple sources
  - ✓ If device can serve only one request at a time
  - ✓ i.e., Printing

- Device reservation - provides exclusive access to a device
  - ✓ System calls for allocation and deallocation
  - ✓ Watch out for deadlock

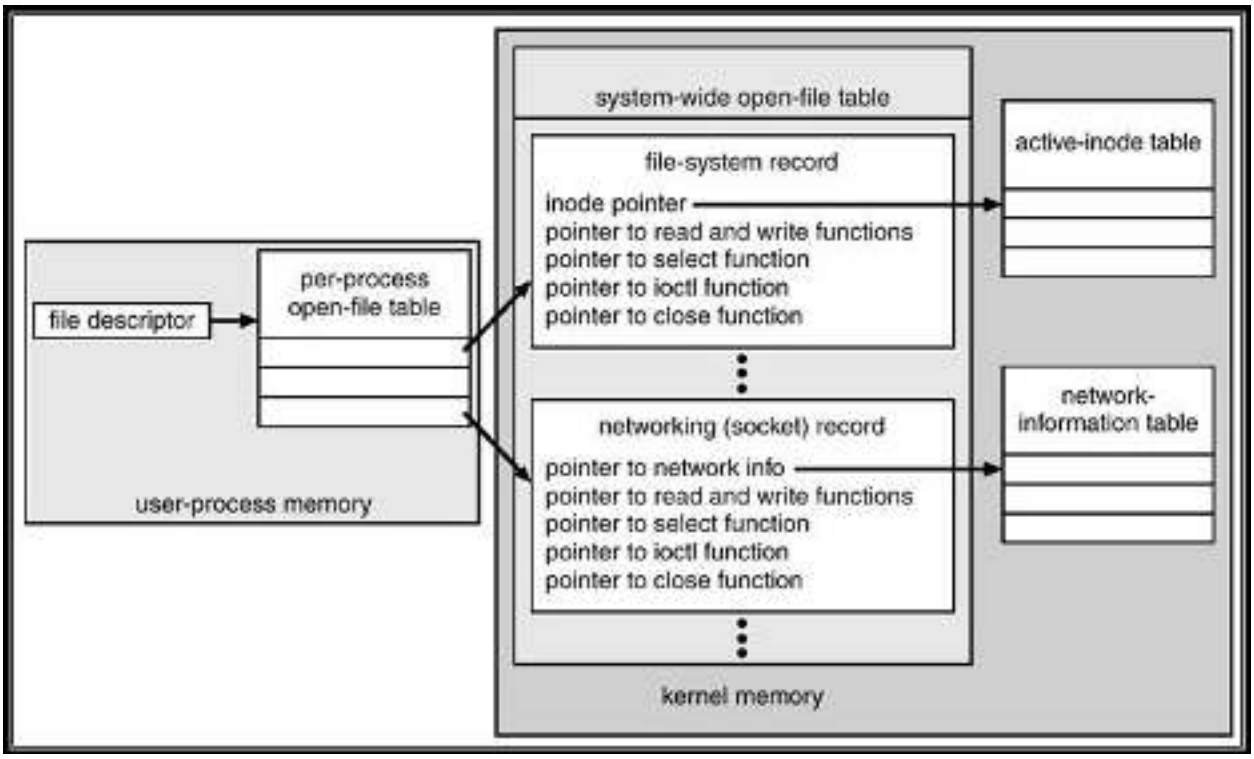
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, ¯dirty‖ blocks
- Some use object-oriented methods and message passing to implement I/O

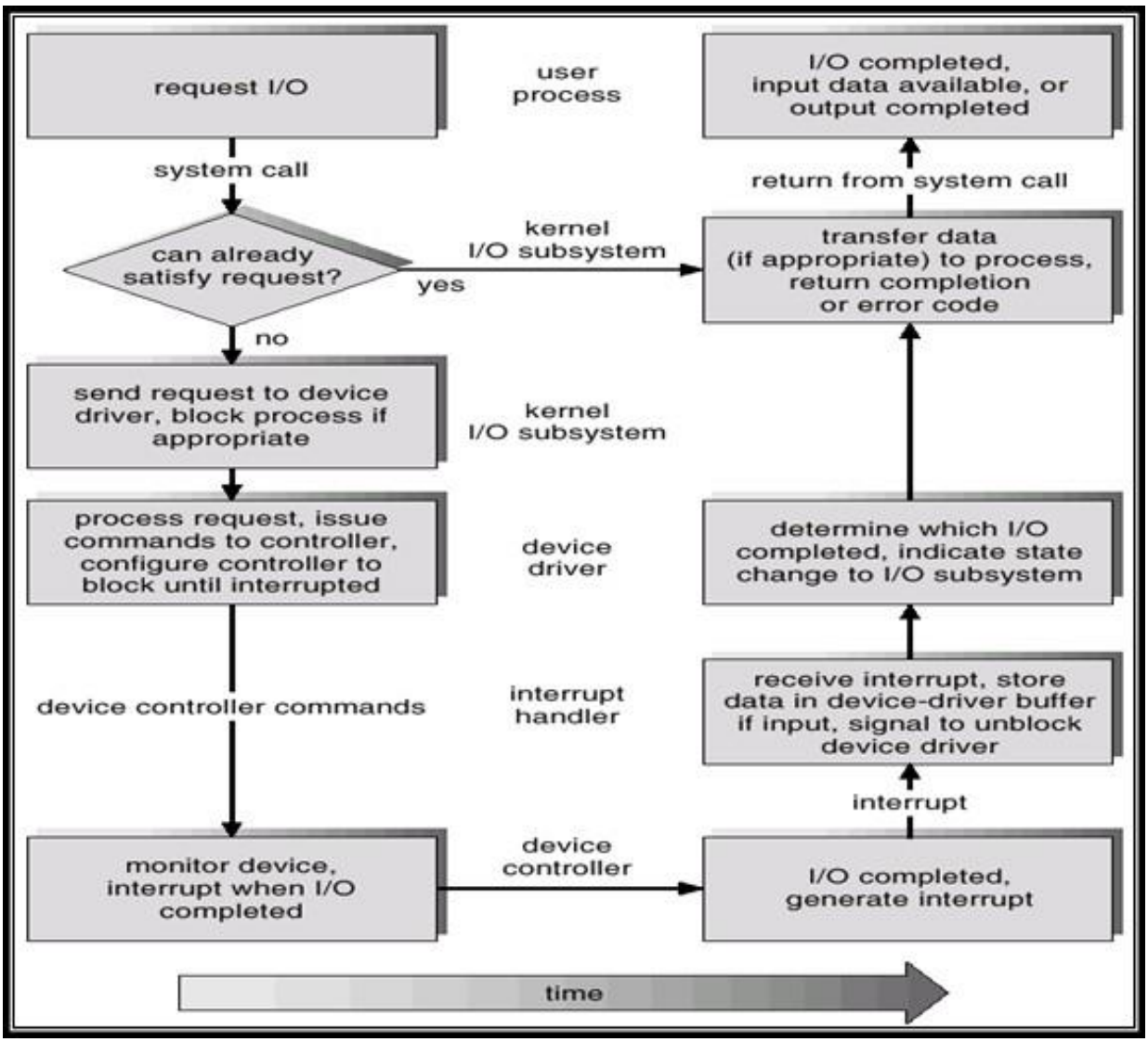
Unix I/O Kernel Structure



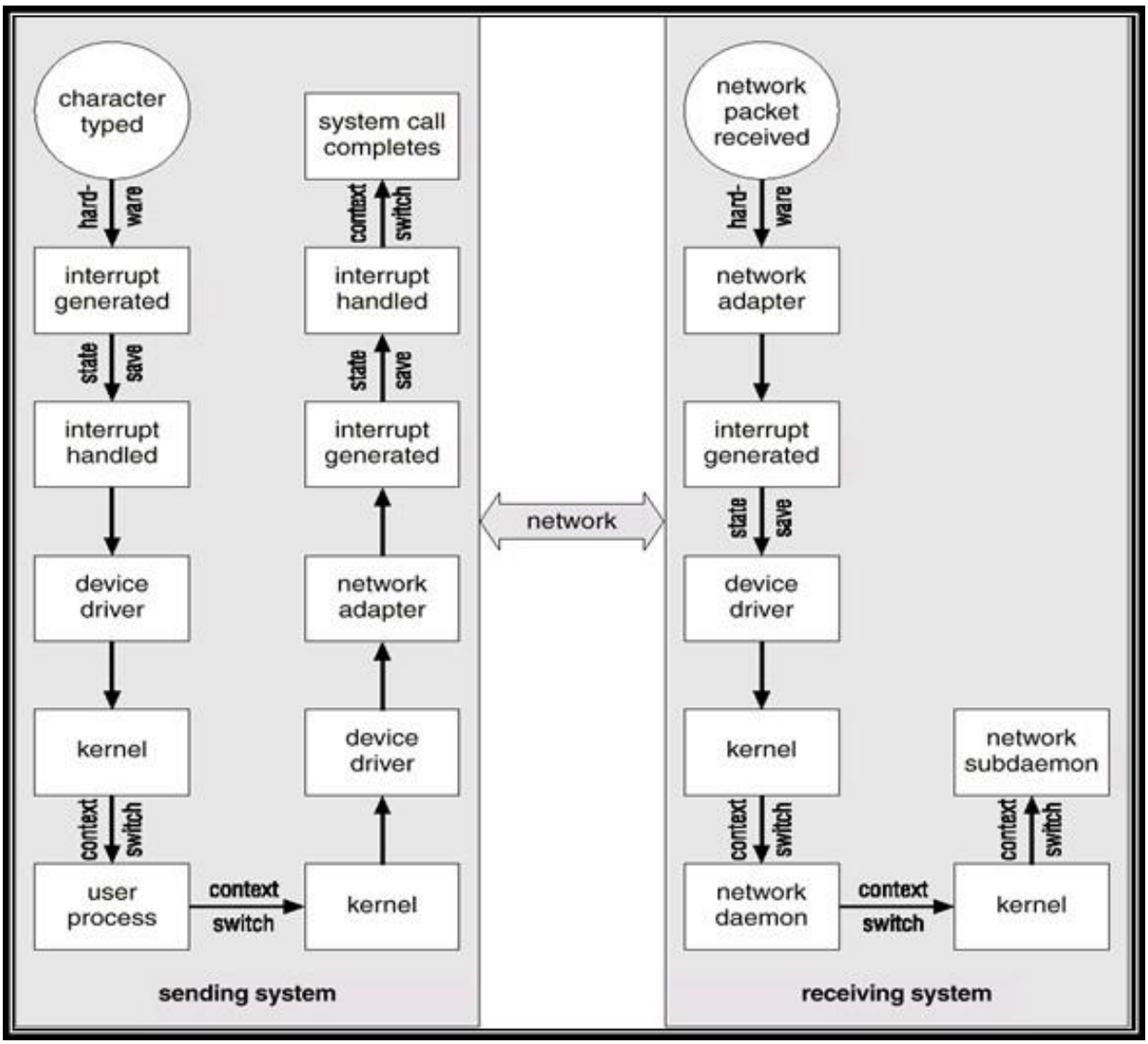
Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

Life Cycle of An I/O Request



Inter-computer Communications



Lecture No. 37

Overview of today's lecture

- Interrupt handlers
- Interrupts and exceptions
- Linux interrupt handling
- Top halfs, bottom halfs and tasklets
- Timings and timer devices
- Linux kernel timers and interval timers

Interrupt Handlers

- Interrupt handlers are best hidden
- Interrupt procedure does its task

Interrupts vs Exceptions

- Varying terminology but for Intel:
  - ✓ **Interrupt** (device generated)
    - **Maskable:** device-generated, associated with IRQs (interrupt request lines); may be temporarily disabled (still pending)
    - **Nonmaskable:** some critical hardware failures
  - ✓ **Exceptions** (Processor-detected)
    - **Faults** - correctable (restartable); e.g. page fault
    - **Traps** - no reexecution needed; e.g. breakpoint
    - **Aborts** - severe error; process usually terminated (by signal)
  - ✓ **Programmed exceptions** (**software interrupts**)
    - int (system call), int3 (breakpoint)
    - into (overflow), bounds (address check)

Interrupts and Exceptions

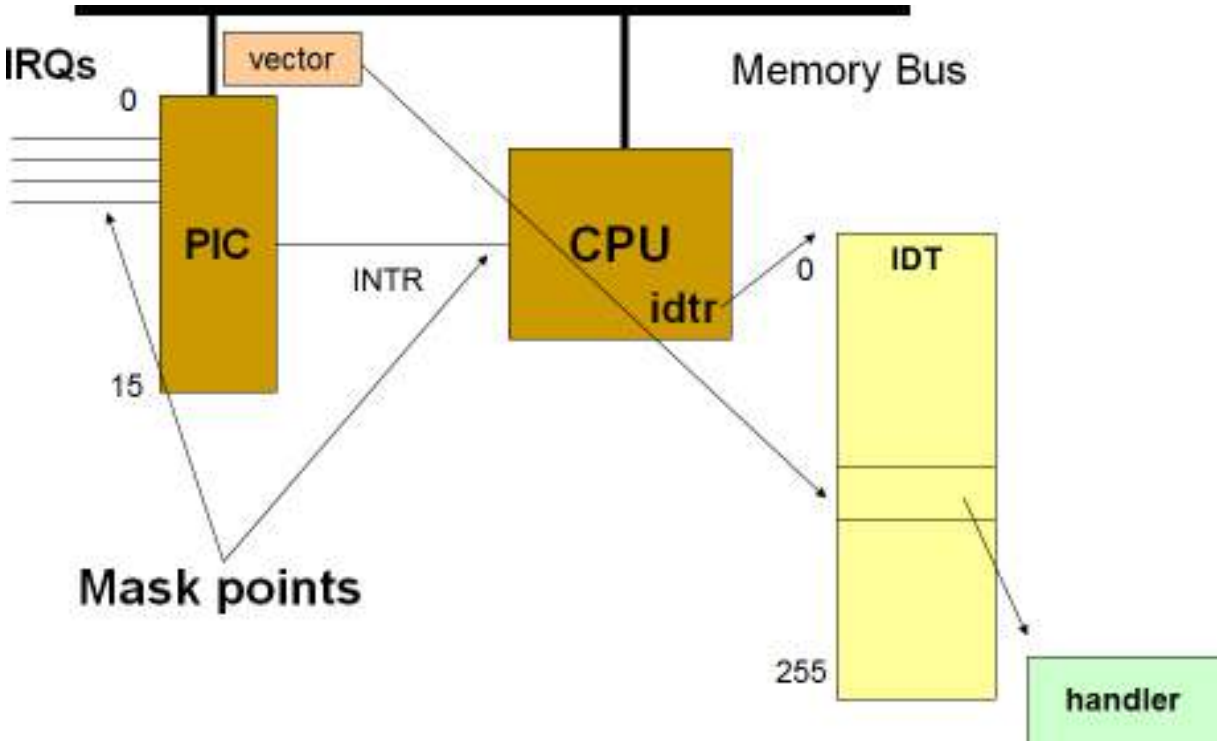
- Hardware support for getting CPUs attention
  - ✓ Often transfers from user to kernel mode
  - ✓ Asynchronous: device or timer generated
  - ✓ Synchronous: immediate result of last instruction
- Intel terminology and hardware
  - ✓ Irqs, vectors, IDT, gates, PIC, APIC

Interrupt Handling

- More complex than exceptions
  - ✓ Requires registry, deferred processing, etc.
- Some issues:
  - ✓ IRQs are often shared; all handlers (ISRs) are executed so they must query device

- Three types of actions:
  - ✓ Critical: Top-half (interrupts disabled - briefly!)
  - ✓ Non-critical: Top-half (interrupts enabled)
  - ✓ Non-critical deferrable: Do it ¯later‖ (interrupts enabled)

Interrupt Handling Components



Timing and Timers

- Accurate timing crucial for many aspects of OS
  - ✓ Device-related timeouts
  - ✓ File timestamps (created, accessed, written)
  - ✓ Time-of-day (gettimeofday()
  - ✓ High-precision timers (code profiling, etc.)
  - ✓ Scheduling, cpu usage, etc.

- Intel timer hardware
  - ✓ RTC: Real Time Clock
  - ✓ PIT: Programmable Interrupt Timer
  - ✓ TSC: TimeStamp Counter (cycle counter)
  - ✓ Local APIC Timer: per-cpu alarms

- Timer implementations
  - ✓ Kernel timers (dynamic timers)
  - ✓ User ¯interval‖ timers (alarm(), setitimer())

- *Dynamic timers* may be dynamically created and destroyed. No limit is placed on the number of currently active dynamic timers.
- A dynamic timer is stored in the following timer_list structure:
- struct timer_list

```
{
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

Software timers in Linux

- Each timer contains a field that indicates how far in the future the timer should expire. This field is initially calculated by adding the right number of ticks to the current value of jiffies.
- The field does not change. Every time the kernel checks a timer, it compares the expiration field to the value of jiffies at the current moment, and the timer expires when jiffies is greater or equal to the stored value.

User Mode Interval Timers

- setitimer(): 3 distinct user mode interval timers
  - ✓ can set for one-time or periodic alarm
  - ✓ sends signals on timer expiry

- three timers:
  - ✓ ITIMER_REAL: elapsed time (SIGALRM)
  - ✓ ITIMER_VIRT: user (SIGVTALRM)
  - ✓ ITIMER_PROF: user + kernel (SIGPROF)

- implementations:
  - ✓ VIRT, PROF – updates by PIT or APIC
  - ✓ REAL - requires kernel timer
    - o may need to deliver to blocked process
    - o current->real_timer
    - o shared by alarm() API so can't use both

Lecture No. 38

Overview of today's lecture

- Loadable Kernel modules and device drivers
- Linux module management
- Linux module conflict resolution
- Linux module registration
- Signals and asynchronous event notification

Loadable Kernel Modules (Linux & Solaris)

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Three components to Linux module support:
  - ✓ module management
  - ✓ driver registration
  - ✓ conflict resolution

Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel.
- Module loading is split into two separate sections:
  - ✓ Managing sections of module code in kernel memory
  - ✓ Handling symbols that modules are allowed to reference
- The module requestor manages loading of the requested, but currently unloaded modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available.
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.
- Registration tables include the following items:
  - ✓ Device drivers
  - ✓ File systems
  - ✓ Network protocols
  - ✓ Binary format

Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver
- The conflict resolution module aims to:
  - ✓ Prevent modules from clashing over access to hardware resources
  - ✓ Prevent *autoprobes* from interfering with existing device drivers
  - ✓ Resolve conflicts with multiple drivers trying to access the same hardware

Signals

- Early minimal IPC (no info) mechanism
  - ✓ **asynchronous** event notification system
  - ✓ software analog of hardware interrupts

- Three distinct APIs
  - ✓ original (buggy, unreliable) signals
    - o slightly differing semantics between Sys V, BSD
  - ✓ reliable (Posix) signals
  - ✓ real-time (Posix) signals

- Things you can do with signals
  - ✓ **generate** (send, raise): kill()
  - ✓ **deliver** (receive, handle): during kernel to user transition
  - ✓ **block**, **mask**: temporarily disable delivery (but not generation)
  - ✓ **ignore**: throw away on delivery
  - ✓ **catch** (handle): execute a user-supplied handler on delivery

Signals: Basics

- Signals have **names** (macros) and **numbers**
  - ✓ examples: SIGINT (2), SIGKILL (9), SIGPWR (30)
  - ✓ kill -l lists platform assignments
  - ✓ some are architecture and processor dependent
    - o SIGSTKFLT – coprocessor stack error (Intel)

- Signals can be **generated** by
  - ✓ **users**
    - o via special shell character (control-c)
    - o via user-level commands (kill -9 1234)
  - ✓ **programs** via system calls (kill(pid, sig))
  - ✓ **the kernel** (e.g. in response to exceptions)

- Signals in Linux
  - ✓ **regular**: 1-31 (assigned specific functions)
  - ✓ **realtime**: 32-64 (user assignable)

- **Pending signals**
  - ✓ generated but not delivered
  - ✓ may be blocked or not-blocked

- **Regular signals** ¯can't count‖
  - ✓ generation of an already pending signal not recorded
  - ✓ think of it as a single bit that is ¯set‖ on generation

- **Realtime signals** ¯queue‖
  - ✓ linked list of generated signals (up to some maximum)

- Basic system calls
  - ✓ **generate**: kill(), rt_sigqueueinfo()
  - ✓ **block, unblock**: sigprocmask(), rt_sigprocmask()
  - ✓ **check pending**: sigpending(), rt_sigpending()
  - ✓ **establish handler**: sigaction(), signal(), rt_sigaction()
  - ✓ **wait for signal**: sigsuspend(), rt_sigsuspend()

- Calls often operation on **signal sets**
  - ✓ two element arrays of ints (64 bit bitmask)

- Blocking, pending, delivery
  - ✓ blocked: delivery delayed until unblocked
    - possible for signal to be blocked with no signal pending
  - ✓ generated: pending for a short while even if unblocked
  - ✓ unblocked pending signals: delivered on kernel to user transition
    - delivery opportunities every timer interrupt (but only for current)

- Masking signals
  - ✓ current signal delivery **masked** during handler execution
    - like interrupt masking
    - handlers need not be re-entrant
  - ✓ old, buggy semantics: current signal **not** masked

- Default actions
  - ✓ all signals have a **default action**
    - **terminate**
    - **dump** - terminate and dump core
    - **ignore** - throw away on delivery
    - **stop** - control-z
    - **continue** -
  - ✓ possible to **catch** most signals
    - establish user-specified handler
  - ✓ SIGKILL, SIGSTOP can't be caught, blocked, or ignored

Signals: Old Unreliable Signals

- Old call (unreliable, buggy): signal()
  - ✓ further signal delivery not masked during handler
  - ✓ default action restored on delivery

- Common programming idiom
  - ✓ myhandler() {

```
        // window of vulnerability here!
        signal(SIGWHATEVER, myhandler) // reestablish
        // do something to handle signal
    }
```

- Consequence
  - ✓ possible for new delivery to occur during window
  - ✓ not possible to reliably catch all signals!

- New call (reliable): sigaction()
  - ✓ avoids problem: signals masked, no reset
  - ✓ parameterized to make old semantics still available
  - ✓ signal() just calls sigaction() with appropriate parameters

Signals: System Calls

- kill(pid, sig)
- sigaction(sig, act, oact)
  - ✓ signal()
- sigpending()
- sigprocmask()
- sigsuspend()

Lecture No. 39

Overview of today's lecture

- Introduction to security and protection
- Security issues
- Policy vs Mechanism
- Design principles for security
- Security requirements
- Security related terminology
- Introduction to user authentication

Security & Protection

- The purpose of a protection system is to prevent accidental or intentional misuse of a system
- Accidents
  - ✓ A program mistakenly deletes the root directory. No one can login.
  - ✓ This sort of problem (relatively) easy to solve: just make the likelihood small.
- Malicious abuse:
  - ✓ A high school hacker breaks the password for user B of accounting system A and transfers $3 million to his account.
  - ✓ This kind of problem very hard to completely eliminate (no loopholes, can't play on probabilities)

Security issues

- Isolation
  - ✓ Separate processes execute in separate memory space
  - ✓ Process can only manipulate allocated pages
- Authentication
  - ✓ Who can access the system. Involves proving identities to the system
- Access control
  - ✓ When can process create or access a file?
  - ✓ Create or read/write to socket?
  - ✓ Make a specific system call?
- Protection problem
  - ✓ Ensure that each object is accessed correctly and only by those processes that are allowed to do so
- Comparison between different operating systems
  - ✓ Compare protection models: which model supports least privilege most effectively?
  - ✓ Which system best enforces its protection model?

Policy versus mechanism

- A good way to look at the problem is to separate policy (what) from mechanism (how)
- A protection system is the mechanism to enforce a security policy
  - ✓ roughly the same set of choices, no matter what policy

- A security policy delineates what acceptable behavior and unacceptable behavior.
- Example security policies:
  - ✓ that each user can only allocate 40MB of disk
  - ✓ that no one but root can write to the password file
  - ✓ that you can't read my mail.

There is no perfect protection system

- Very simple point, very easy to miss:
  - ✓ Protection can only increase the effort (¯work factor‖) needed to do something bad. It cannot prevent it.
- Even assuming a technically perfect system, there are always the four Bs:
  - ✓ Burglary: if you can't break into my system, you can always steal it (called ¯physical security‖)
  - ✓ Bribery: find whoever has access to what you want and bribe them.
  - ✓ Blackmail.
  - ✓ Bludgeoning. Or just beat them until they tell you.

Design Principles for Security

1. System design should be public
2. Default should be no access
3. Check for current authority
4. Give each process least privilege possible
5. Protection mechanism should be
   - simple
   - uniform
   - in lowest layers of system
6. Scheme should be psychologically acceptable

First: What are Your Security Requirements?

- What is your security environment?
  - ✓ What threats, and how severe?
  - ✓ Who's not trusted?
  - ✓ What assumption?
  - ✓ What environment (Platforms, network)
  - ✓ What organizational policies?
  - ✓ What assets?
- What are your product's security objectives?
  - ✓ Confidentiality (¯can't read‖)
  - ✓ Integrity (¯can't change‖)
  - ✓ Availability (¯works continuously‖)
  - ✓ Other: Privacy (¯doesn't reveal‖), Audit, ...
- What functions and assurance measures are needed?
- Common Criteria useful checklist of requirements

Terminology I: the entities

- Principals – who is acting?
  - ✓ User / Process Creator
  - ✓ Code Author
- Objects – what is that principal acting on?
  - ✓ File
  - ✓ Network connection
- Rights – what actions might you take?
  - ✓ Read
  - ✓ Write
- Familiar UNIX file system example:
  - ✓ owner / group / world
  - ✓ read / write / execute.

Terminology II: the activities

- Authentication – who are you?
  - ✓ identifying principals (users / programs)
- Authorization – what are you allowed to do?
  - ✓ determining what access users and programs have to specific objects
- Auditing – what happened
  - ✓ record what users and programs are doing for later analysis / prosecution

User Authentication

- Basic Principles. Authentication must identify:
  1. Something the user knows
  2. Something the user has
  3. Something the user is
- This is done before user can use the system

Lecture No. 40

Overview of today's lecture

- User authentication
- Password based authentication
- UNIX password scheme
- One-time password schemes
- Challenge response authentication
- Biometrics and other authentication schemes
- Access control and authorization
- Access control matrix

Authentication

- Usually done with passwords.
  - ✓ This is usually a relatively weak form of authentication, since it's something that people have to remember
  - ✓ Empirically is typically based on wife's/husband's or kid's name, favorite movie name etc.
- Passwords should not be stored in a directly-readable form
  - ✓ Use some sort of one-way-transformation (a ¯secure hash‖) and store that
  - ✓ if you look in /etc/passwords will see a bunch of gibberish associated with each name. That is the password
- Problem: to prevent guessing (¯dictionary attacks‖) passwords should be long and obscure
  - ✓ unfortunately easily forgotten and usually written down.
- Unix password security
  - ✓ Encrypt passwords
- One time passwords
  - ✓ Lamport's clever scheme (Read Tanenbaum for details)
- Challenge-Response based authentication
  - ✓ Used in PPP and many other applications

Authentication alternatives

- Badge or key
  - ✓ Does not have to be kept secret. usually some sort of picture ID worn on jacket (e.g., at military bases)
- Should not be forgeable or copy-able
- Can be stolen, but the owner should know if it is
  - ✓ (but what to do? If you issue another, how to invalidate old?)
- This is similar to the notion of a ¯capability‖ that we'll see later

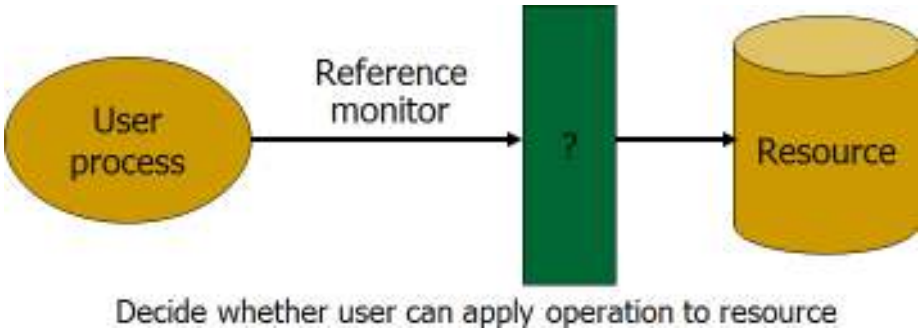
Biometrics

- Biometrics
  - ✓ Authentication of a person based on a physiological or behavioral characteristic.

- Example features:
  - ✓ Face, Fingerprints, Hand geometry, Handwriting,
  - ✓ Iris, Retinal, Vein, Voice.
- Strong authentication but still need a ¯Trusted Path‖.

Access control

- Context
  - ✓ System knows who the user is
    - User has entered a name and password, or other info
  - ✓ Access requests pass through gatekeeper
    - OS must be designed so monitor cannot be bypassed



Access control matrix [Lampson]

Objects

Subjects

	File 1	File 2	File 3	...	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
...					
User m	read	write	read	write	read

Two implementation concepts

- Access control list (ACL)
  - ✓ Store column of matrix with the resource
- Capability
  - ✓ User holds a ¯ticket‖ for each resource

	File 1	File 2	…
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
…			
User m	read	write	write

- Access control lists are widely used, often with groups
- Some aspects of capability concept are used in Kerberos, …

Lecture No. 41

Overview of today's lecture

- ACL Vs capabilities
- Delegation and revocation
- Operations on capabilities
- Capabilities and roles
- Capabilities and groups
- Confidentiality model
- Integrity model
- Other security models

ACL vs Capabilities

- Access control list
  - ✓ Associate list with each object
  - ✓ Check user/group against list
  - ✓ Relies on authentication: need to know user

- Capabilities
  - ✓ Capability is unforgeable ticket
    - o Random bit sequence, or managed by OS
    - o Can be passed from one process to another
  - ✓ Reference monitor checks ticket
    - o Does not need to know identity of user/process

- Delegation
  - ✓ Cap: Process can pass capability at run time
  - ✓ ACL: Try to get owner to add permission to list

- Revocation
  - ✓ ACL: Remove user or group from list
  - ✓ Cap: Try to get capability back from process?
    - o Possible in some systems if appropriate bookeeping
      - OS knows what data is capability
      - If capability is used for multiple resources, have to revoke all or none ...

Operations on Capabilities

- Copy: create a new capability for the same object
- Copy object: create a duplicate object with a new capability
- Remove capability: Delete an entry from the capability list; object remains unaffected
- Destroy object: Permanently remove an object and a capability

Sandboxing mobile code

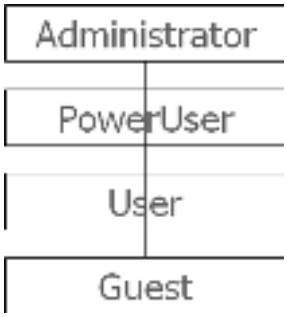
- Foreign program started in a process
- Process given a set of capabilities:
  - ✓ Read and write on the monitor
  - ✓ Read and write a scratch directory
- Principle of least privilege

Capabilities

- Operating system concept
  - ✓ ¯... of the future (and always will be?) ...||
- Examples
  - ✓ Dennis and van Horn, MIT PDP-1 Timesharing
  - ✓ Hydra, StarOS, Intel iAPX 432, Eros, …
  - ✓ Amoeba: distributed, unforgeable tickets
- References
  - ✓ Henry Levy, Capability-based Computer Systems http://www.cs.washington.edu/homes/levy/capabook/
  - ✓ Tanenbaum, Amoeba papers

Roles (also called Groups)

- Role = set of users
  - ✓ Administrator, PowerUser, User, Guest
  - ✓ Assign permissions to roles; each user gets permission
- Role hierarchy
  - ✓ Partial order of roles
  - ✓ Each role gets
- permissions of roles below
  - ✓ List only new permissions given to each role



Groups for resources, rights

- Permission = ⟨right, resource⟩
- Permission hierarchies
  - ✓ If user has right r, and r>s, then user has right s
  - ✓ If user has read access to directory, user has read access to every file in directory

- Big problem in access control
  - ✓ Complex mechanisms require complex input
  - ✓ Difficult to configure and maintain
  - ✓ Roles, other organizing ideas try to simplify problem

Multi-Level Security (MLS) Concepts

- Military security policy
  - ✓ Classification involves sensitivity levels, compartments
  - ✓ Do not let classified information leak to unclassified files
- Group individuals and resources
  - ✓ Use some form of hierarchy to organize policy
- Other policy concepts
  - ✓ Separation of duty
  - ✓ ¯Chinese Wall‖ Policy

Confidentiality Model

- When is it OK to release information?
- Two Properties
  - ✓ Simple security property
    - o A subject S may read object O only if C(O) ≤ C(S)
  - ✓ *-Property
    - o subject S with read access to object O may write object P if C(O) ≤ C(P)
- In words,
  - ✓ You may only *read below* your classification and
- When is it OK to release information?
- Two Properties
  - ✓ Simple security property
    - o A subject S may read object O only if C(O) ≤ C(S)
  - ✓ *-Property
    - o subject S with read access to object O may write object P if C(O) ≤ C(P)
- In words,
  - ✓ You may only *read below* your classification and only *write above* your classification

Integrity Model

- Rules that preserve integrity of information
- Two Properties
- Simple integrity property
  - ✓ A subject S may write object O only if C(S) ≥ C(O) (Only trust S to modify O if S has higher rank …)

- ✓ *-Property
  - A subject S with read access to O may write object P only if C(O) ≥ C(P) (Only move info from O to P if O is more trusted than P)
- ✓ In words,
  - You may only *write below* your classification and only *read above* your classification

Problem: Models appear contradictory

- Confidentiality
  - ✓ Read down, write up
- Integrity
  - ✓ Read up, write down
- Want both confidentiality and integrity
  - ✓ Contradiction is partly an illusion
  - ✓ May use confidentiality for some classification of personnel and data, integrity for another
    - Otherwise, only way to satisfy both models is only allow read and write at same classification
- In reality: confidentiality used more than integrity model, e.g., Common Criteria

Other policy concepts

- Separation of duty
  - ✓ If amount is over $10,000, check is only valid if signed by two *authorized* people
  - ✓ Two people must be *different*
  - ✓ Policy involves role membership and ≠
- Chinese Wall Policy
  - ✓ Lawyers L1, L2 in Firm F are experts in banking
  - ✓ If bank B1 sues bank B2,
    - L1 and L2 can each work for either B1 or B2
    - No lawyer can work for opposite sides in any case
  - ✓ Permission depends on use of other permissions

Lecture No. 42

Overview of today's lecture

- Trojan Horses
- Login spoofing attacks
- Logic bombs
- Trap doors
- Buffer and stack overflow attacks
- Unsafe C library functions

Trojan Horses (from Tanenbaum's book)

- Free program made available to unsuspecting user
  - ✓ Actually contains code to do harm
- Place altered version of utility program on victim's computer
  - ✓ Trick user into running that program

Login Spoofing

a) Correct login screen

b) Phony login screen

Logic Bombs

- Company programmer writes program
  - ✓ potential to do harm
  - ✓ OK as long as he/she enters password daily
  - ✓ ff programmer fired, no password and bomb explodes

Trap Doors

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

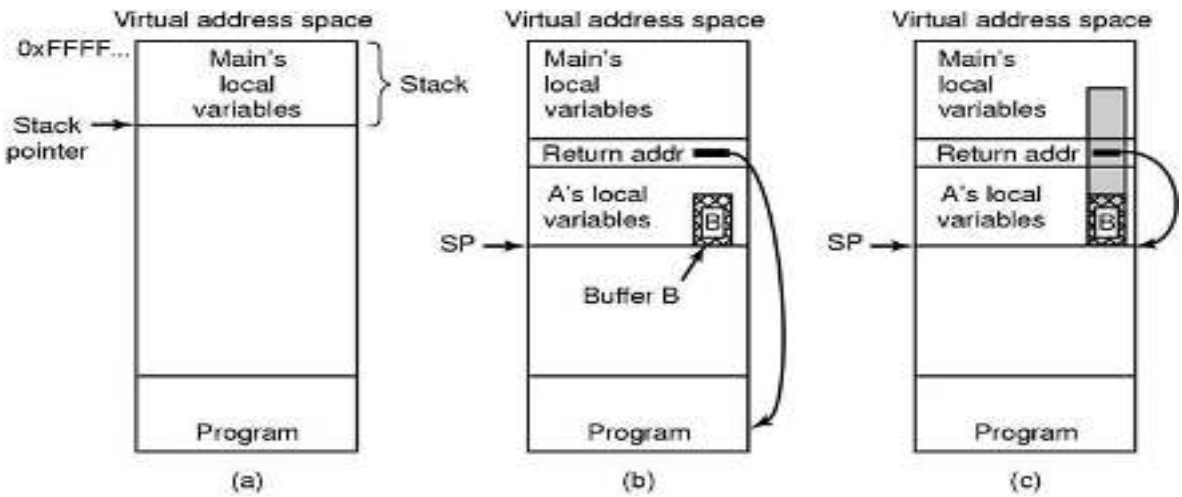
(b)

a) Normal code.

b) Code with a trapdoor inserted

Buffer overflows

- Extremely common bug.
  - ✓ First major exploit: 1988 Internet Worm. fingerd.
- 10 years later: over 50% of all CERT advisories:
  - ✓ 1997: 16 out of 28 CERT advisories.
  - ✓ 1998: 9 out of 13 -||-
  - ✓ 1999: 6 out of 12 -||-
- Often leads to total compromise of host.
  - ✓ Fortunately: exploit requires expertise and patience.
  - ✓ Two steps:
    - Locate buffer overflow within an application.
    - Design an exploit.



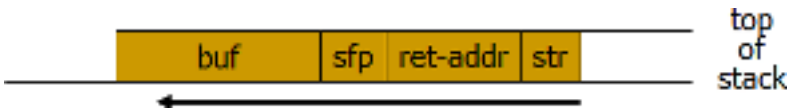
a) Situation when main program is running
b) After program *A* called
c) Buffer overflow shown in gray

What are buffer overflows?

- Suppose a web server contains a function:

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
    do-something(buf);
}
```

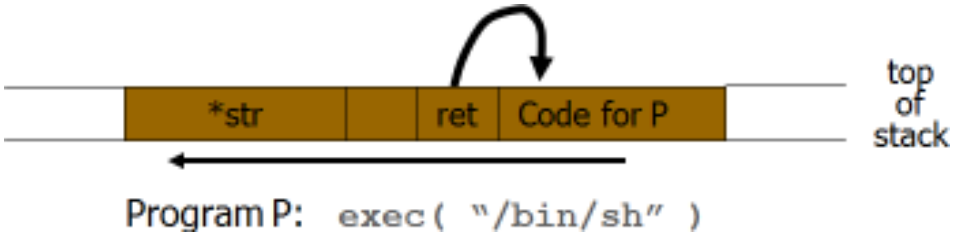
- When the function is invoked the stack looks like:



- What if *str is 136 bytes long? After strcpy:

Basic stack exploit

- Main problem: no range checking in strcpy().
- Suppose *str is such that after strcpy stack looks like:



- When func() exits, the user will be given a shell !!
- Note: attack code runs *in stack*.
- To determine ret guess position of stack when func() is called.
- Use a stream of NOPs

Some unsafe C lib functions

strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, …)
printf (conts char *format, …)

How does an attacker actually launches this attack.

- Inspection of source code
- Help of Debuggers
- Cramming a lot of data into a program

Exploiting buffer overflows

- Suppose web server calls func() with given URL.
- Attacker can create a 200 byte URL to obtain shell on web server.
- Some complications:
  - ✓ Program P should not contain the '\0' character.
  - ✓ Overflow should not crash program before func() exists.
- Sample buffer overflows of this type:
  - ✓ Overflow in MIME type field in MS Outlook.
  - ✓ Overflow in ISAPI in IIS.

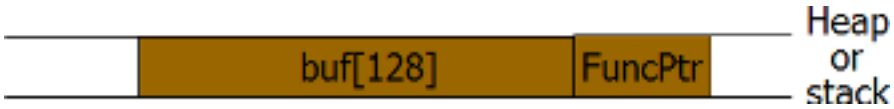
Lecture No. 43

Overview of today's lecture

- Types of buffer overflow attacks
- Methods for fighting buffer overflows
- StackGuard and PointGuard
- Libsafe
- Address obfuscation
- Viruses and worms
- Mobile code security & sandboxing
- Java security

Causing program to exec attack code

- Stack smashing attack:
  - ✓ Override return address in stack activation record by overflowing a local buffer variable.
- Function pointers: (used in attack on Linux superprobe)
  - ✓ Overflowing buf will override function pointer.



- Longjmp buffers: longjmp(pos) (used in attack on Perl 5.003)
  - ✓ Overflowing buf next to pos overrides value of pos.

Finding buffer overflows

- Hackers find buffer overflows as follows:
  - ✓ Run web server on local machine.
  - ✓ Issue requests with long tags. All long tags end with ¯$$$$$‖.
  - ✓ If web server crashes, search core dump for ¯$$$$$‖ to find overflow location.
- Some automated tools exist. (eEye Retina, ISIC).

Preventing buf overflow attacks

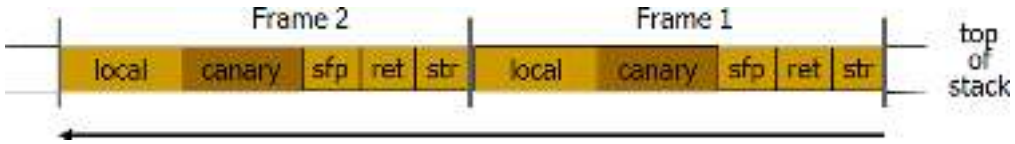
- Main problem:
  - ✓ strcpy(), strcat(), sprintf() have no range checking.
  - ✓ ¯Safe‖ versions strncpy(), strncat() are misleading
    - strncpy() may leave buffer unterminated.
- Defenses:
  - ✓ Type safe languages (Java, ML). Legacy code?
  - ✓ Mark stack as non-execute.
  - ✓ Static source code analysis.
  - ✓ Run time checking: StackGuard, Libsafe, SafeC, (Purify).
  - ✓ Black box testing (e.g. eEye Retina, ISIC).

Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executable or randomizing stack location.
  - ✓ Code patches exist for Linux and Solaris.
- Problems:
  - ✓ Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to func pointer.
  - ✓ Some apps need executable stack (e.g. LISP interpreters).
- Patch not shipped by default for Linux and Solaris.

Run time checking: StackGuard

- Many many run-time checking techniques …
- Solutions 1: StackGuard (WireX)
  - ✓ Run time tests for stack integrity.
  - ✓ Embed ¯canaries‖ in stack frames and verify their integrity prior to function return.



Canary Types

- <u>Random canary:</u>
  - ✓ Choose random string at program startup.
  - ✓ Insert canary string into every stack frame.
  - ✓ Verify canary before returning from function.
  - ✓ To corrupt random canary, attacker must learn current random string.

- <u>Terminator canary:</u>
  - ✓ Canary = 0, newline, linefeed, EOF
  - ✓ String functions will not copy beyond terminator.
  - ✓ Hence, attacker cannot use string functions to corrupt stack.

StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - ✓ Program must be recompiled.
- Minimal performance effects: 8% for Apache.

- Newer version: PointGuard.
  - ✓ Protects function pointers and setjmp buffers by placing canaries next to them.
  - ✓ More noticeable performance effects.

- Note: Canaries don‘t offer full protection.
  - ✓ Some stack smashing attacks can leave canaries untouched.

Run time checking: Libsafe

- Solutions 2: Libsafe (Avaya Labs)
  - ✓ Dynamically loaded library.
  - ✓ Intercepts calls to strcpy (dest, src)
    - o Validates sufficient space in current stack frame:
      |frame-pointer - dest| > strlen(src)
    - o If so, does strcpy.
      Otherwise, terminates application.

More methods

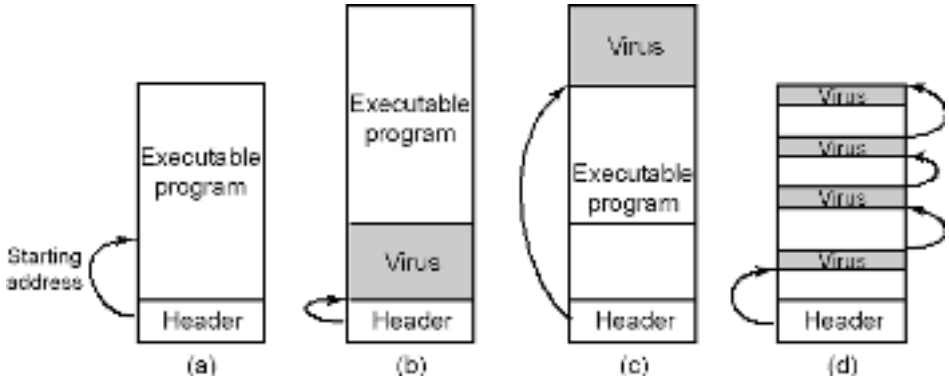
- Address obfuscation. (Stony Brook '03)
  - ✓ Encrypt return address on stack by XORing with random string. Decrypt just before returning from function.
  - ✓ Attacker needs decryption key to set return address to desired value.
- Randomize location of functions in libc.
  - ✓ Attacker cannot jump directly to exec function.

Viruses and worms

- External threat
  - ✓ code transmitted to target machine
  - ✓ code executed there, doing damage
- Goals of virus writer
  - ✓ quickly spreading virus
  - ✓ difficult to detect
  - ✓ hard to get rid of
- Virus = program can reproduce itself
  - ✓ attach its code to another program
  - ✓ additionally, do harm

How Viruses Work (1)

- Virus written in assembly language
- Inserted into another program
  - ✓ use tool called a ¯dropper‖
- Virus dormant until program executed
  - ✓ then infects other programs
  - ✓ eventually executes its ¯payload‖



- An executable program
- With a virus at the front
- With the virus at the end
- With a virus spread over free space within program

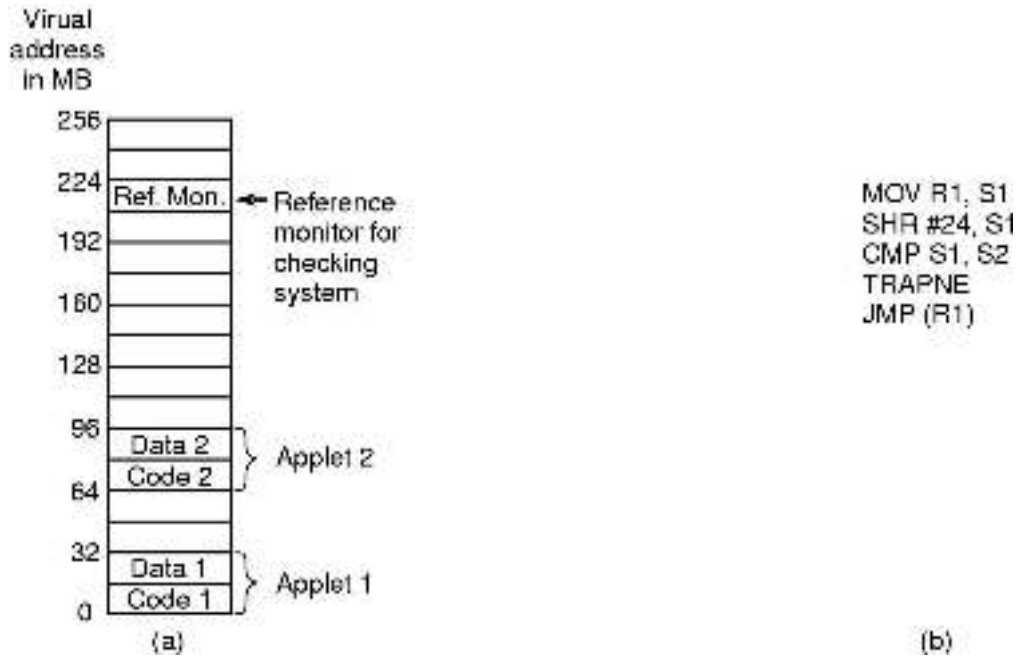
How Viruses Spread

- Virus placed where likely to be copied
- When copied
  - ✓ infects programs on hard drive, floppy
  - ✓ may try to spread over LAN
- Attach to innocent looking email
  - ✓ when it runs, use mailing list to replicate

Antivirus and Anti-Antivirus Techniques

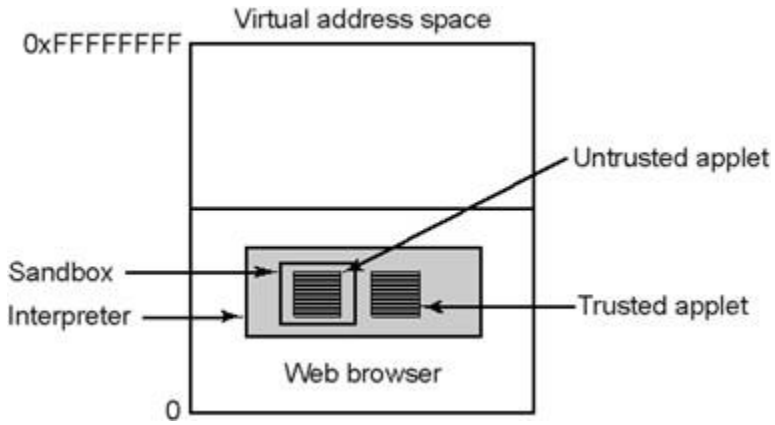
- Signature based
- Integrity checkers
- Behavioral checkers
- Virus avoidance
  - ✓ good OS
  - ✓ install only shrink-wrapped software
  - ✓ use antivirus software
  - ✓ do not click on attachments to email
  - ✓ frequent backups
- Recovery from virus attack
  - ✓ halt computer, reboot from safe disk, run antivirus

Mobile Code (1) Sandboxing



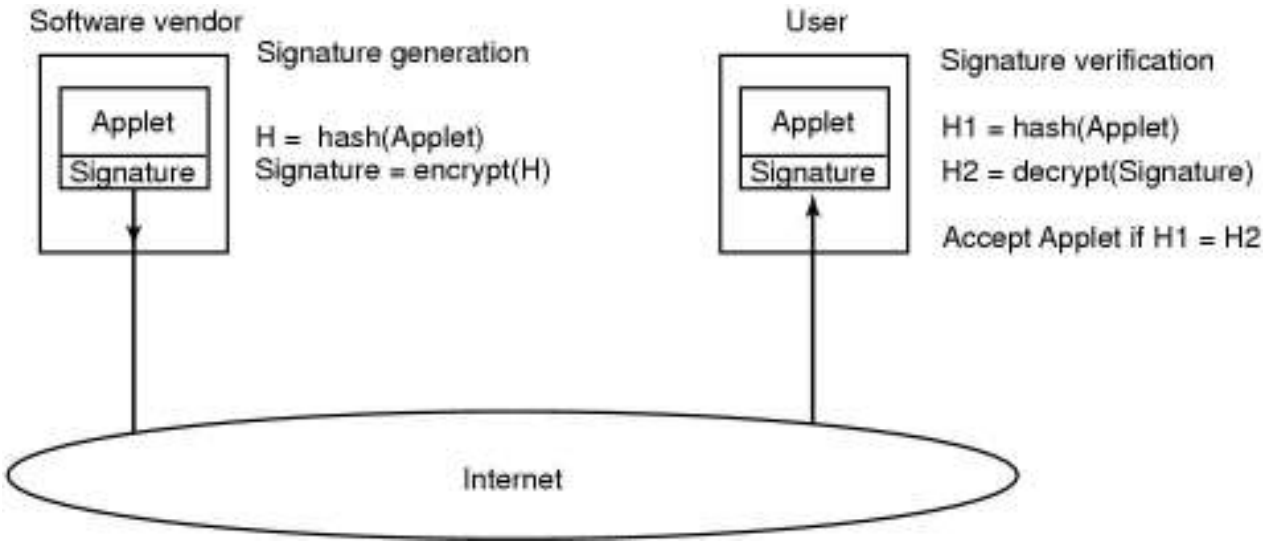
a. Memory divided into 1-MB sandboxes
b. One way of checking an instruction for validity

Mobile Code (2)



Applets can be interpreted by a Web browser

Mobile Code (3)



How code signing works

Java Security (1)

- A type safe language
  - ✓ compiler rejects attempts to misuse variable
- Checks include …
  - ✓ Attempts to forge pointers
  - ✓ Violation of access restrictions on private class members
  - ✓ Misuse of variables by type
  - ✓ Generation of stack over/underflows
  - ✓ Illegal conversion of variables to another type

Lecture No. 44

Overview of today's lecture

- Java Security
- UNIX file security
- Setuid programs
- Windows security
  - ✓ Tokens, security descriptors and reference monitor
- SE Linux
- Features of a secure OS summarized
  - ✓ DAC Vs MAC
  - ✓ Orange book criteria etc.

Java Security (2)

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

Examples of specified protection with JDK 1.2

Unix file security

- Each file has owner and group
- Permissions set by owner
  - ✓ Read, write, execute
  - ✓ Owner, group, other
- Represented by vector of four octal values
- Only owner, root can change permissions
  - ✓ This privilege cannot be delegated or shared
- Setid bits - Discussed later

Question

- Owner can have fewer privileges than other
  - ✓ What happens?
    - Owner gets access?
    - Owner does not?
- Prioritized resolution of differences

  if user = owner then *owner* permission
  else if user in group then *group* permission
  else *other* permission

Setid bits on executable Unix file

- Three setid bits
  - ✓ Setuid – set EUID of process to ID of file owner
  - ✓ Setgid – set EGID of process to GID of file
  - ✓ Sticky
    - Off: if user has write permission on directory, can rename or remove files, even if not owner
    - On: only file owner, directory owner, and root can rename or remove file in the directory

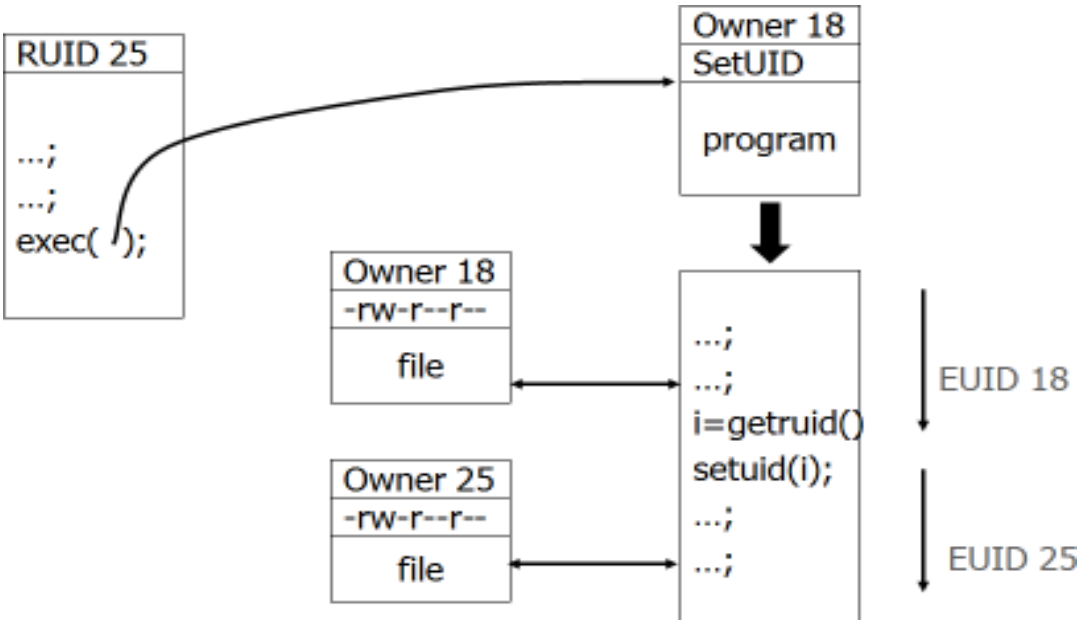
Effective user id (EUID)

- Each process has three Ids (+ more under Linux)
  - ✓ Real user ID (RUID)
    - same as the user ID of parent (unless changed)
    - used to determine which user started the process
  - ✓ Effective user ID (EUID)
    - from set user ID bit on the file being executed, or sys call
    - determines the permissions for process
      - file access and port binding
  - ✓ Saved user ID (SUID)
    - So previous EUID can be restored
- Real group ID, effective group ID, used similarly

Process Operations and IDs

- Root
  - ✓ ID=0 for superuser root; can access any file
- Fork and Exec
  - ✓ Inherit three IDs, except exec of file with setuid bit
- Setuid system calls
  - ✓ seteuid(newid) can set EUID to
    - Real ID or saved ID, regardless of current EUID
    - Any ID, if EUID=0
- Details are actually more complicated
  - ✓ Several different calls: setuid, seteuid, setreuid

Example



- Careful with Setuid !
  - ✓ Can do anything that owner of file is allowed to do
  - ✓ Be sure not to
    - Take action for untrusted user
    - Return secret data to untrusted user
- Note: anything possible if root; no middle ground between user and root

Unix summary

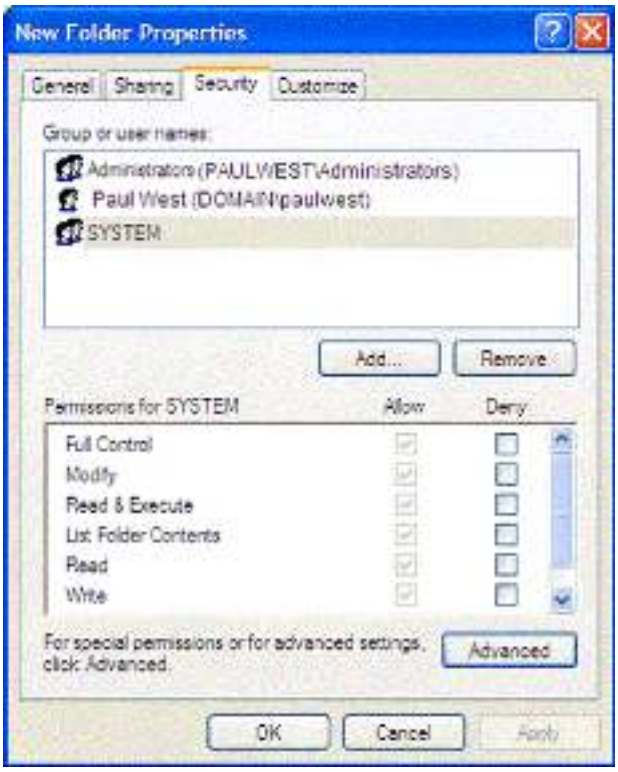
- Good things
  - ✓ Some protection from most users
  - ✓ Flexible enough to make things possible
- Main bad thing
  - ✓ Too tempting to use root privileges
  - ✓ No way to assume some root privileges without all root privileges

Access control in Windows (NTFS)

- Some basic functionality similar to Unix
  - ✓ Specify access for groups and users
    - Read, modify, change owner, delete
- Some additional concepts
  - ✓ Tokens
  - ✓ Security attributes
- Generally
  - ✓ More flexibility than Unix
    - Can define new permissions
    - Can give some but not all administrator privileges

Sample permission options

- Security ID (SID)
  - ✓ Identity (replaces UID)
    - SID revision number
    - 48-bit authority value
    - variable number of Relative Identifiers (RIDs), for uniqueness
  - ✓ Users, groups, computers, domains, domain members all have SIDs



Permission Inheritance

- Static permission inheritance (Win NT)
  - ✓ Initially, subfolders inherit permissions of folder
  - ✓ Folder, subfolder changed independently
  - ✓ *Replace Permissions on Subdirectories* command
    - Eliminates any differences in permissions

- Dynamic permission inheritance (Win 2000)
  - ✓ Child inherits parent permission, remains linked
  - ✓ Parent changes are inherited, except explicit settings
  - ✓ Inherited and explicitly-set permissions may conflict
    - Resolution rules
      - Positive permissions are additive
      - Negative permission (deny access) takes priority

Tokens

- Security Reference Monitor
  - ✓ uses tokens to identify the security context of a process or thread
- Security context
  - ✓ Privileges and groups associated with the process or thread
- Impersonation token
  - ✓ thread uses temporarily to adopt a different security context, usually of another user

Impersonation Tokens

- Process uses security attributes of another
  - ✓ Client passes impersonation token to server

- Client specifies impersonation level of server
  - ✓ Anonymous
    - o Token has no information about the client

  - ✓ Identification
    - o server obtains the SIDs of client and client's privileges, but server cannot impersonate the client

  - ✓ Impersonation
    - o server identifies and impersonates the client

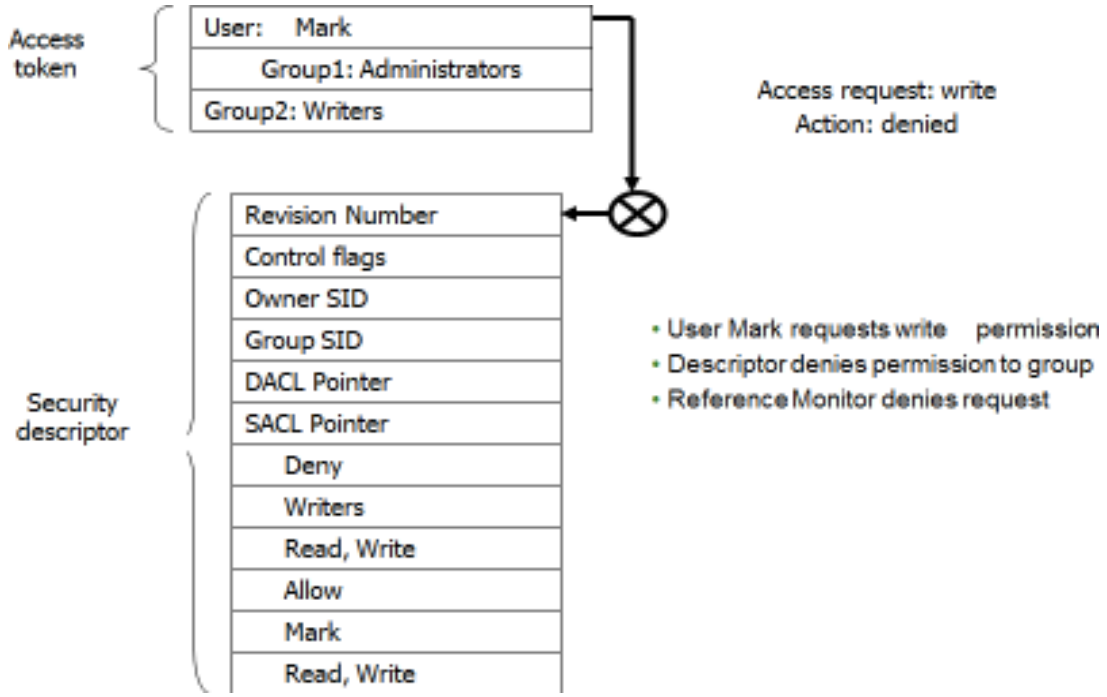
  - ✓ Delegation
    - o lets server impersonate client on local, remote systems

Security Descriptor

- Information associated with an object
  - ✓ who can perform what actions on the object

- Several fields
  - ✓ Header
    - o Descriptor revision number
    - o Control flags, attributes of the descriptor
      E.g., memory layout of the descriptor
  - ✓ SID of the object's owner
  - ✓ SID of the primary group of the object
  - ✓ Two attached optional lists:
    - o Discretionary Access Control List (DACL) – users, groups, …
    - o System Access Control List (SACL) – system logs, ..

Example access request



SELinux

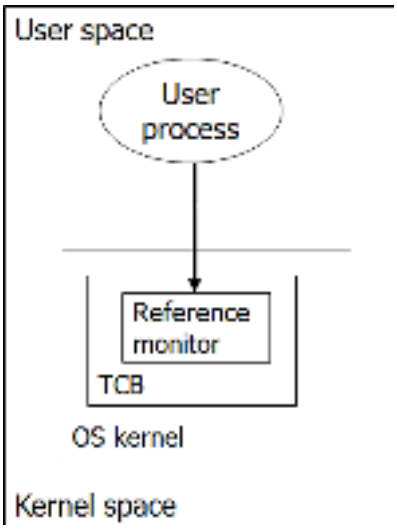
- Security-enhanced Linux system (NSA)
  - ✓ Enforce separation of information based on confidentiality and integrity requirements
  - ✓ Mandatory access control incorporated into the major subsystems of the kernel
    - Limit tampering and bypassing of application security mechanisms
    - Confine damage caused by malicious applications
- Why Linux? Open source
  - ✓ Already subject to public review
  - ✓ NSA can review source, modify and extend

SELinux Security Policy Abstractions

- Type enforcement
  - ✓ Each process has an associated domain
  - ✓ Each object has an associated type
  - ✓ Configuration files specify
    - How domains are allowed to access types
    - Allowable interactions and transitions between domains
- Role-based access control
  - ✓ Each process has an associated role
    - Separate system and user processes
  - ✓ configuration files specify
    - Set of domains that may be entered by each role

Kernelized Design

- Trusted Computing Base
  - ✓ Hardware and software for enforcing security rules
- Reference monitor
  - ✓ Part of TCB
  - ✓ All system calls go through reference monitor for security checking
  - ✓ Most OS not designed this way



What makes a "secure" OS?

- Extra security features
- Stronger authentication mechanisms, Example: require token + password
  - ✓ More security policy options
    - Example: only let users read file f for purpose p
  - ✓ Logging and other features
- More secure implementation
  - ✓ Apply secure design and coding principles
  - ✓ Assurance and certification
    - Code audit or formal verification
  - ✓ Maintenance procedures
    - Apply patches, etc.

Sample Features of "Trusted OS"

- Mandatory access control
  - ✓ MAC not under user control, precedence over DAC
- Object reuse protection
  - ✓ Write over old data when file space is allocated
- Complete mediation
  - ✓ Prevent any access that circumvents monitor

- Audit
  - ✓ Log security-related events and check logs
- Intrusion detection
  - ✓ Anomaly detection
    - Learn normal activity, Report abnormal actions
  - ✓ Attack detection
    - Recognize patterns associated with known attacks

DAC and MAC

- Discretionary Access Control
  - ✓ Restrict a subject's access to an object
    - Generally: limit a user's access to a file
    - Owner of file controls other users' accesses
- Mandatory Access Control
  - ✓ Needed when security policy dictates that:
    - protection decisions must not be left to object owner
    - system enforces a security policy over the wishes or intentions of the object owner

DAC vs MAC

- DAC
  - ✓ Object owner has full power
  - ✓ Complete trust in users
  - ✓ Decisions are based only on user id and object ownerships
  - ✓ Impossible to control information flow
- MAC
  - ✓ Object owner CAN have some power
  - ✓ Only trust in administrators
  - ✓ Objects and tasks themselves can have ids
  - ✓ Makes information flow control possible

Audit

- Log security-related events
- Protect audit log
  - ✓ Write to write-once non-volatile medium
- Audit logs can become huge
  - ✓ Manage size by following policy
    - Storage becomes more feasible
    - Analysis more feasible since entries more meaningful
- Example policies
  - ✓ Audit only first, last access by process to a file
  - ✓ Do not record routine, expected events
    - E.g., starting one process always loads …

Assurance methods

- Testing
  - ✓ Can demonstrate existence of flaw, not absence
- Formal verification
  - ✓ Time-consuming, painstaking process
- ¯Validation‖
  - ✓ Requirements checking
  - ✓ Design and code reviews
  - ✓ Module and system testing

Orange Book Criteria

- Level D
  - ✓ No security requirements
- Level C For environments with cooperating users
  - ✓ C1 – protected mode OS, authenticated login, DAC, security testing and documentation (Unix)
  - ✓ C2 – DAC to level of individual user, object initialization, auditing (Windows NT 4.0)
- Level B, A
  - ✓ All users and objects must be assigned a security label (classified, unclassified, etc.)
  - ✓ System must enforce Bell-LaPadula confidentiality model

Levels B, A

- Level B
  - ✓ B1 - classification and Bell-LaPadula
  - ✓ B2 – system designed in top-down modular way, must be possible to verify security of modules
  - ✓ B3 – ACLs with users and groups, formal TCB must be presented, adequate security auditing, secure crash recovery
- Level A1
  - ✓ Formal proof of protection system, formal proof that model is correct, demonstration that implementation conforms to model.

Lecture No. 45

Overview of today's lectures

- OS research directions
- Reliability of commodity OSes
- Mobile phone risks and security issues
- Embedded operating systems
- Symbian OS for mobile devices
- Virtual Machine Monitors
- Asynchronous I/O interfaces in Linux kernel
- Quick review of memory management and I/O topics

Reliability in commodity OSes (e.g. Nooks)

- Drivers run in protection domains defined by hardware and software just like processes
- Requires kernel modification
- Solution good for drivers as well as other kernel extensions e.g. in-kernel file systems

Mobile phone risks

- Toll fraud:
  - ✓ Auto dialers.
  - ✓ High cost SMS/MMS.
  - ✓ Phone Proxy
- Loss or theft:
  - ✓ Data loss.
  - ✓ Data compromise.
  - ✓ Loss of Identity (caller ID)
- Availability:
  - ✓ SPAM.
  - ✓ Destruction of the device (flash)
  - ✓ Destruction of data.
- Risks induced by usage:
  - ✓ Mobile banking.
  - ✓ Confidential e-mail, documents.
  - ✓ Device present at confidential meetings: snooping
- Attack vectors
  - ✓ Executables
  - ✓ Bluetooth
  - ✓ GPRS / GSM
  - ✓ OTA
  - ✓ IrDa
  - ✓ Browser
  - ✓ SMS / MMS
  - ✓ SD card
  - ✓ WAP
  - ✓ E-mail
  - ✓ Too many entry points to list all

Symbian OS for mobile devices

- Symbian Ltd. formed in 1998
  - ✓ Ericsson, Nokia, Motorola and Psion
  - ✓ EPOC renamed to Symbian OS
  - ✓ Currently ~30 phones with Symbian,15 licensees
- Current ownership

Nokia 47.5%	Panasonic 10.5%
Ericsson 15.6%	Siemens 8.4%
SonyEricsson 13.1%	Samsung 4.5%

Architecture

- Multitasking, preemptive kernel
- MMU protection of kernel and process spaces
- Strong Client-Server architecture
- Plug-in patterns
- Filesystem in ROM, Flash, RAM and on SD-card

Symbian security features

- Crypto:
  - ✓ Algorithms
  - ✓ Certificate framework
  - ✓ Protocols: HTTPS, WTLS, …

- Symbian signed:
  - ✓ Public key signatures on applications
  - ✓ Root CA's in ROM

- Separation
  - ✓ Kernel vs. user space;
  - ✓ process space
  - ✓ Secured ‗wallet' storage

- Access controls
  - ✓ SIM PIN, device security code
  - ✓ Bluetooth pairing

- Artificial Limitations / patches
  - ✓ Prevent loading device drivers in the kernel (Nokia).
  - ✓ Disallow overriding of ROM based plug-ins

- Limitations
  - ✓ No concept of roles or users.
  - ✓ No access controls in the file system.
  - ✓ No user confirmation needed for access by applications.
  - ✓ User view on device is limited: partial filesystem, selected processes.
  - ✓ Majority of interesting applications is unsigned.

- Are attacks prevented?
  - ✓ Fraud: user should not accept unsigned apps
  - ✓ Loss/theft: In practice, little protection
  - ✓ Availability: any application can render phone unusable (skulls trojan).

Virtual Machine Monitors

- Export a virtual machine to user programs that resembles hardware.
- A virtual machine consists of all hardware features e.g. user/kernel modes, I/O, interrupts and pretty much everything a real machine has.
- A virtual machine may run any OS.

- Examples:
  JVM, VmWare, User-Mode Linux (UML).
  Advantage: portability
  Disadvantage: slow speed

What Is It?

- *Virtual machine monitor* (VMM) virtualizes system resources
  - ✓ Runs directly on hardware
  - ✓ Provides interface to give each program running on it the illusion that it is the only process on the system and is running directly on hardware
  - ✓ Provides illusion of contiguous memory beginning at address 0, a CPU, and secondary storage to *each* program

Privileged Instructions

1. VMM running operating system *o*, which is running process *p*
   - ✓ *p* tries to read–privileged operation traps to hardware
2. VMM invoked, determines trap occurred in *o*
   - ✓ VMM updates state of *o* to make it look like hardware invoked *o* directly, so *o* tries to read, causing trap
3. VMM does read
   - ✓ Updates *o* to make it seem like *o* did read
   - ✓ Transfers control to *o*
4. *o* tries to switch context to *p*, causing trap
5. VMM updates virtual machine of *o* to make it appear *o* did context switch successfully
   - ✓ Transfers control to *o*, which (as *o* apparently did a context switch to *p*) has the effect of returning control to *p*

When Is VM Possible?

- Can virtualize an architecture when:
  1. All sensitive instructions cause traps when executed by processes at lower levels of privilege
  2. All references to sensitive data structures cause traps when executed by processes at lower levels of privilege

- Asynchronous kernel interfaces
- Their implementation in the Linux kernel
- May require major changes to several parts and sub-systems of the kernel
- May result in enhanced kernel and application performance

- Goals of OS memory management
- Questions regarding memory management
- Multiprogramming
- Virtual addresses
- Fixed partitioning
- Variable partitioning
- Fragmentation

- Paging
- Address translation
- Page tables and Page table entries
- Multi-level address translation
- Page faults and their handling

- Segmentation
- Combined Segmentation and paging
- Efficient translations and caching
- Translation Lookaside Buffer (TLB)

- Set associative and fully associative caches
- Demand Paging
- Page replacement algorithms

- Page replacement
- Thrashing
- Working set model
- Page fault frequency
- Copy on write
- Sharing
- Memory mapped files

- Allocation
  - ✓ Linked allocation
  - ✓ FAT
  - ✓ Indexed allocation
  - ✓ i-nodes
- File buffer cache
- Read ahead
- Consistency problem and its solutions

- SABRE airline example
- UNIX file system invariants
- Consistency ensuring techniques and rules
  - ✓ Write ordering etc.

- Disks structure and internals
  - ✓ Platters, Cylinders, heads, tracks, sectors etc.
- Fast File system
  - ✓ Cylinder groups
  - ✓ Fragments for small files

- Log structured (or journaling) file systems record each update to the file system as a transaction.
- All transactions are written to a log. A transaction is considered committed once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.
- Uniform file system interface to user processes
- Represents any conceivable file system‘s general feature and behavior
- Assumes files are objects that share basic properties regardless of the target file system

- Goals of I/O software
- Layers of I/O software
- Direct Vs memory mapped I/O
- Interrupt driven I/O
- Polled I/O
- Direct Memory Access (DMA)

- Device independent I/O software layer
- Buffered and un-buffered I/O
- Block and character devices
- Network devices
- Kernel I/O subsystem and data structures
- Life cycle of a typical I/O request
- Life cycle of a typical network I/O request

- Interrupt handlers
- Interrupts and exceptions
- Linux interrupt handling
- Top halfs, bottom halfs and tasklets
- Timings and timer devices
- Linux kernel timers and interval timers

- Loadable Kernel modules and device drivers
- Linux module management
- Linux module conflict resolution
- Linux module registration
- Signals and asynchronous event notification