Structure of Processes

CSI 309: Operating System Concepts
United International University

PROCESSES AND PROGRAMS

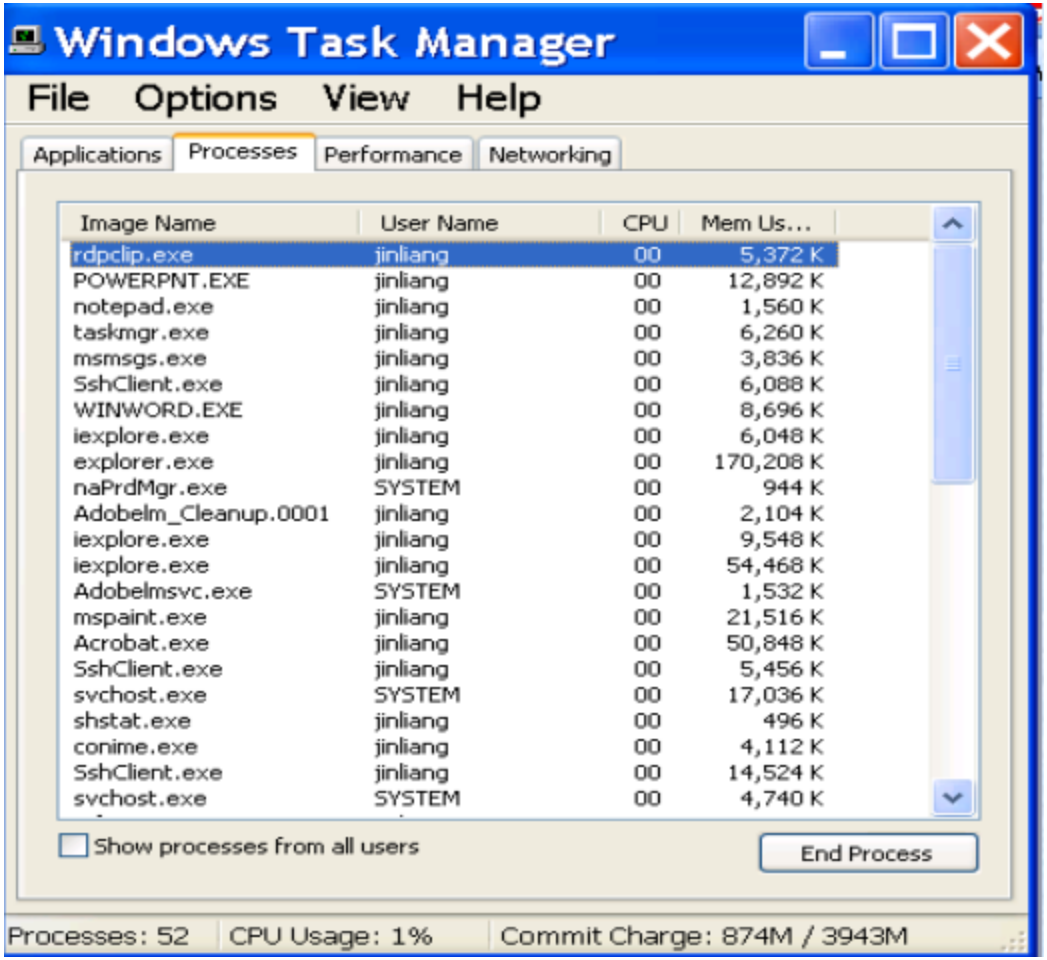
Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
  – Many users may launch same program
  – One user may launch many instances of the same program
- Program: an algorithm expressed in some suitable notation.
- Process: a program in execution.

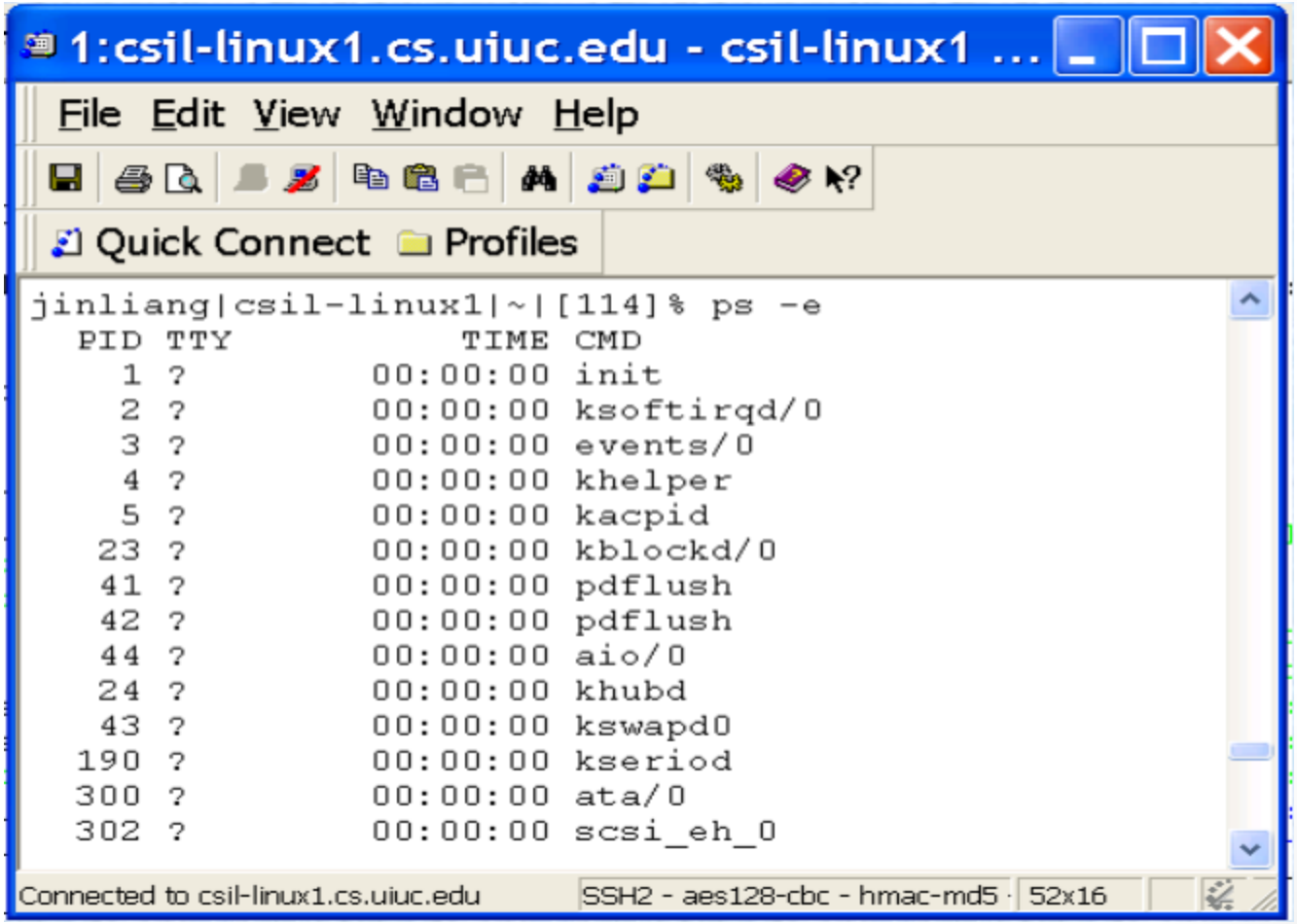
What is a process?

- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- The unit of execution
- Operating system provided ***abstraction*** to represent what is needed to run a single program
- The same as “job” or “task” or “sequential process”.

Example: Windows Task Manager



Example: ps in Unix



What is a program?

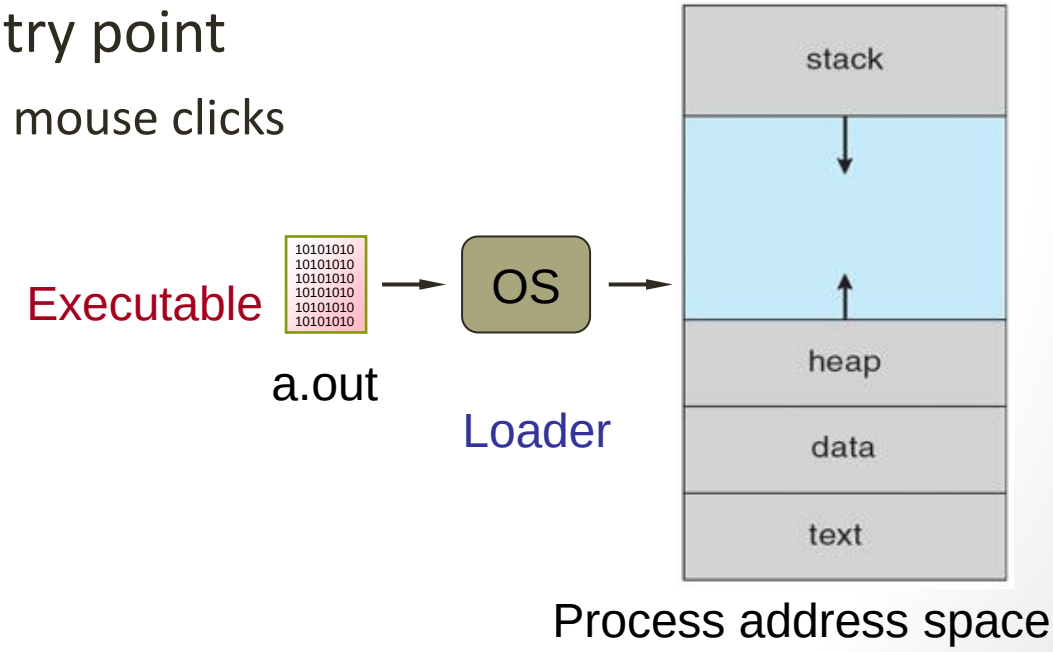
- A Program is an **executable file** that contains:
  - Code: Machine instructions
  - Data: Variables stored and manipulated in memory
    - initialized variables (globals)
    - dynamically allocated variables (malloc, new)
    - stack variables (C automatic variables, function arguments)
  - DLL: libraries that were not compiled or linked with the program
    - containing code & data, possibly shared with other programs

Process != Program

- A process is an executing program.
- Example:
  - We can run 2 instances of Mozilla Firefox:
    - Same program
    - Separate processes
- Program is passive: Code + data
- Process is running program: stack, registers, program counter

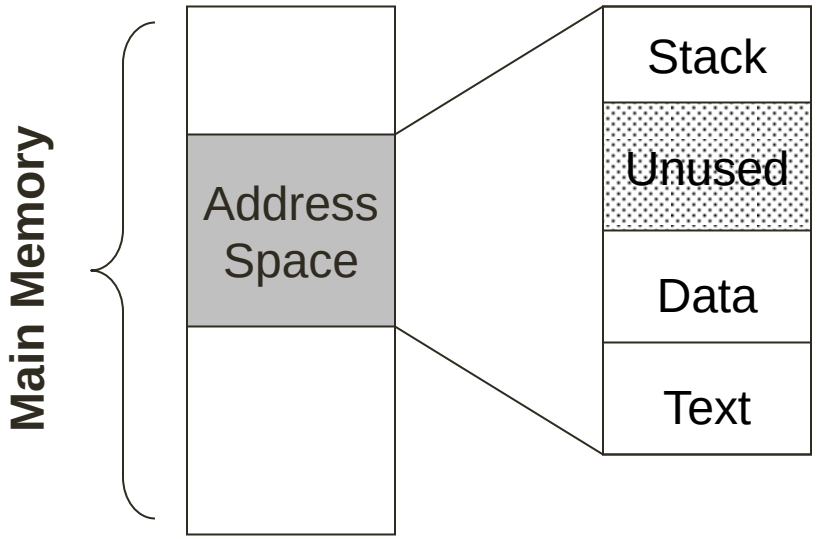
How Program Becomes a Process

- When a program is launched
  - OS loads program into memory
  - Creates kernel data structure for the process
  - Initializes data
  - Starts from an entry point
    - e.g., main(), GUI mouse clicks



Process address space

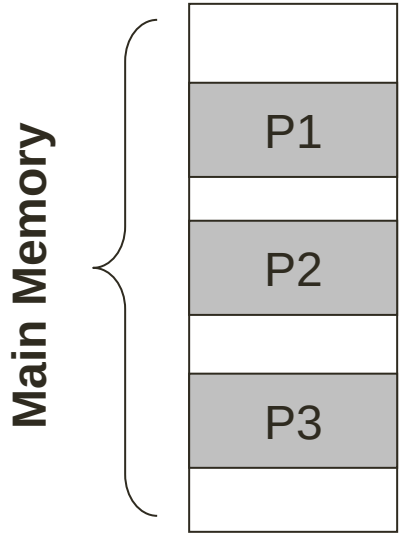
- set of all memory addresses accessible by a process



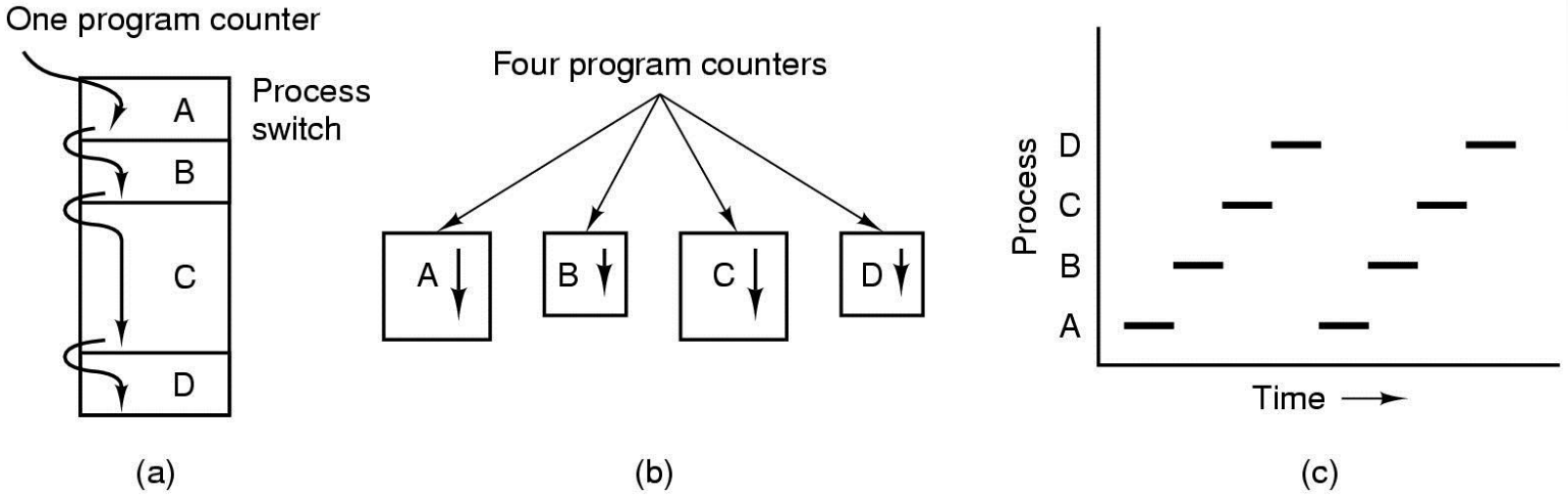
MULTIPROGRAMMING

Multiprogramming

- Each process has its own address space (virtual memory address)
- Even if two processes are running the same program, they have their own address space
- OS schedules processes



Multiprogramming



- Multiprogramming of 4 processes in a single CPU
  CPU switches from one process to other process
- Only 1 physical program counter
  4 logical program counters
- Conceptual model of 4 independent, sequential processes
- Only 1 program active at any instant.
- Real life analogy?
  A daycare teacher trying to feed 4 infants.

PROCESS OPERATIONS

Process Creation

Process Termination

Process State Transitions

Process Creation

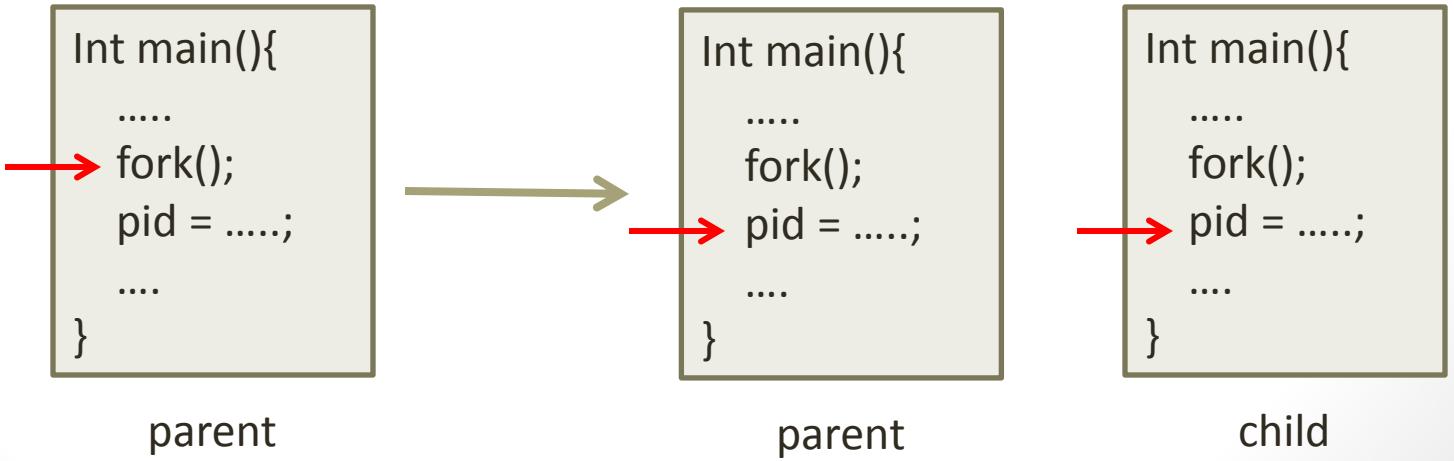
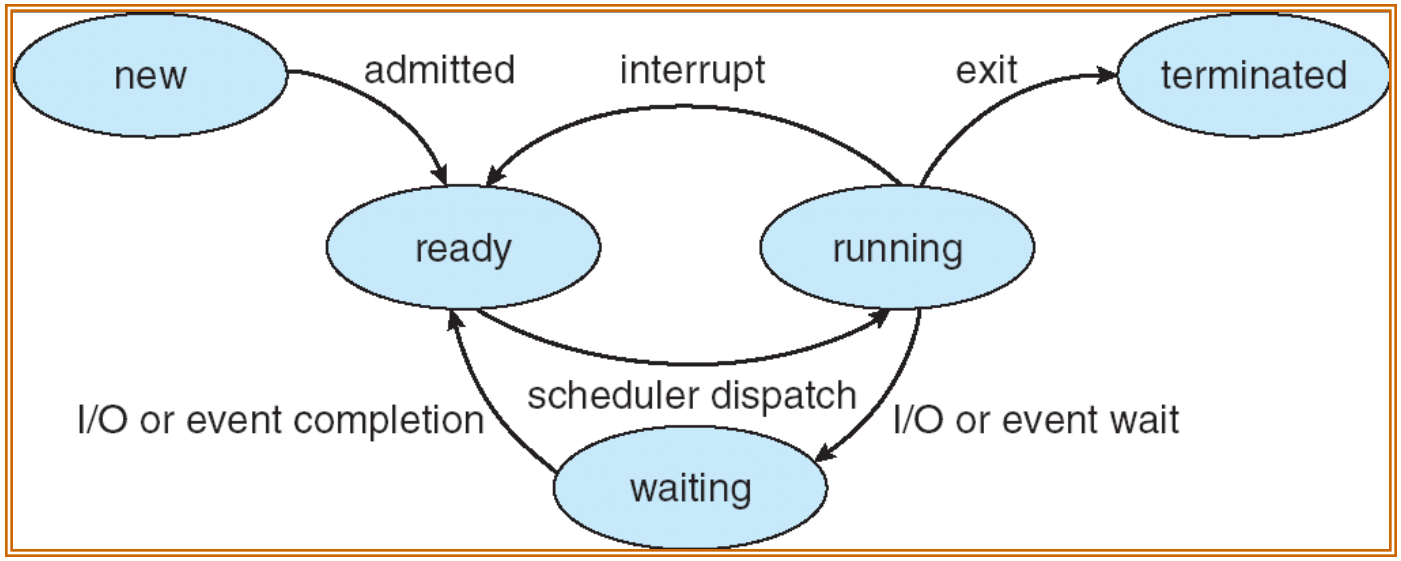
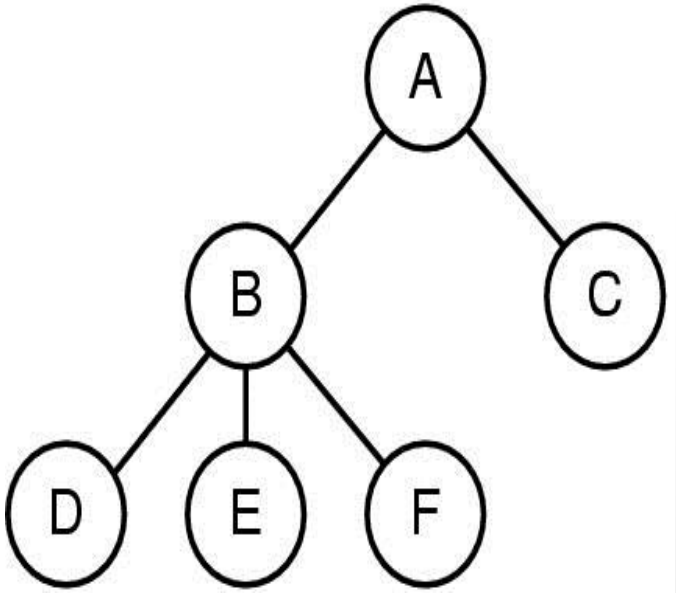
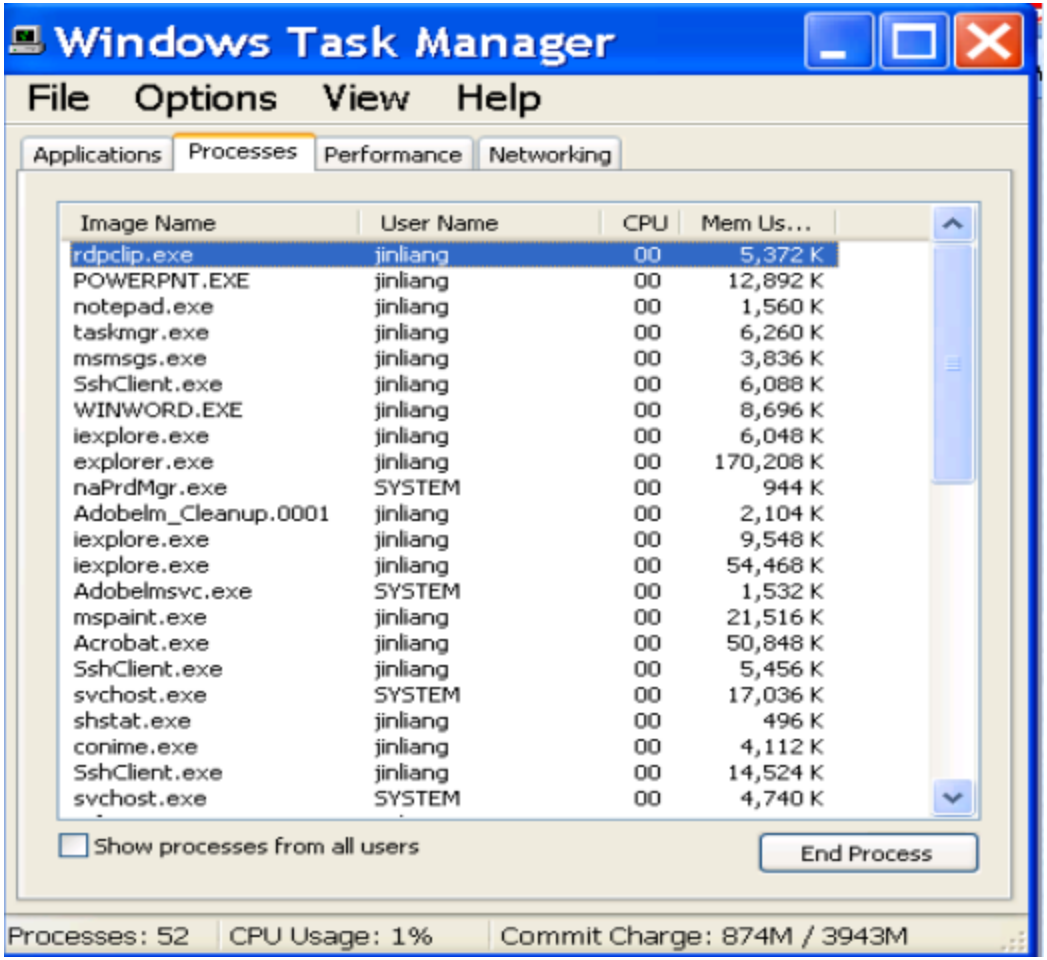
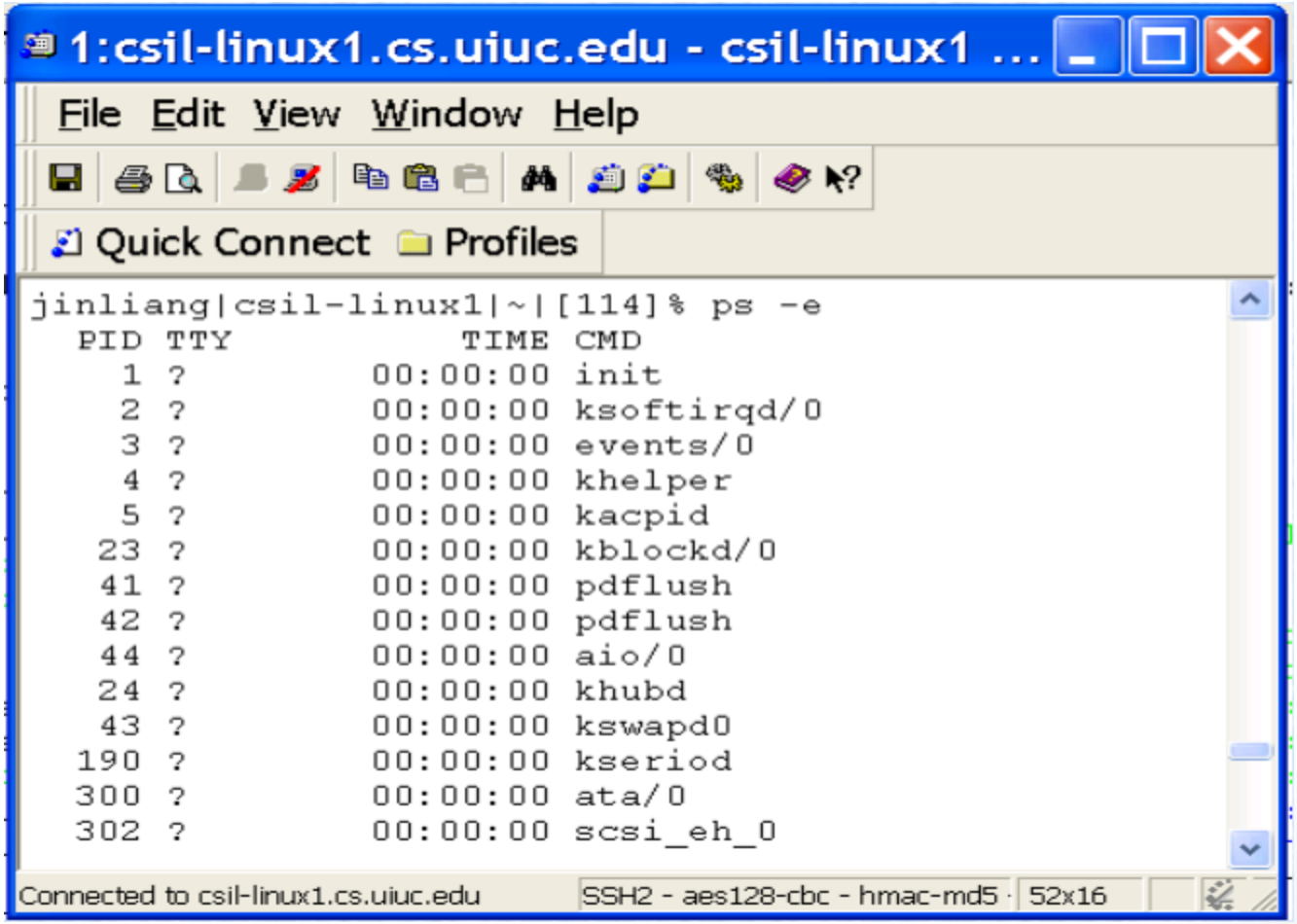
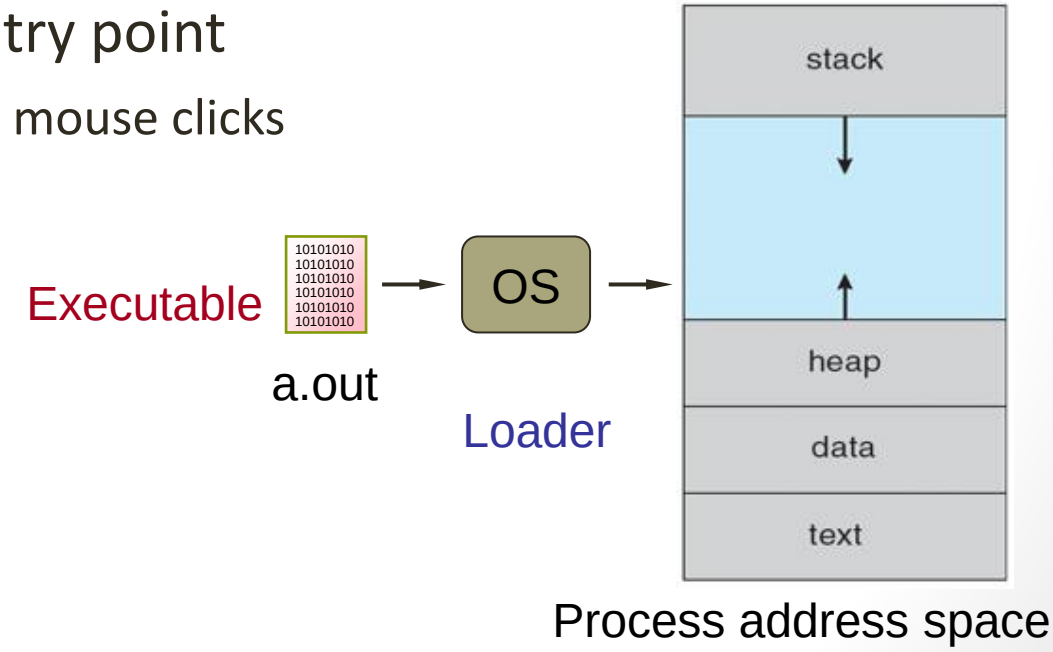
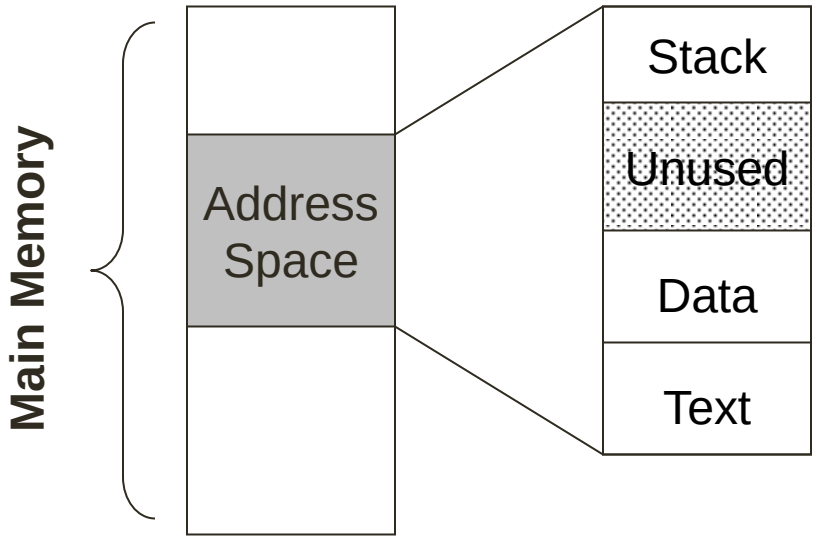
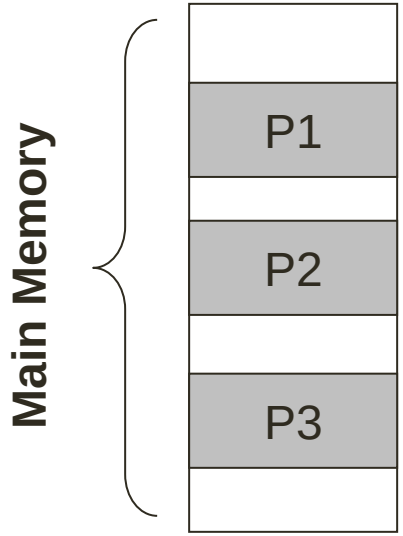
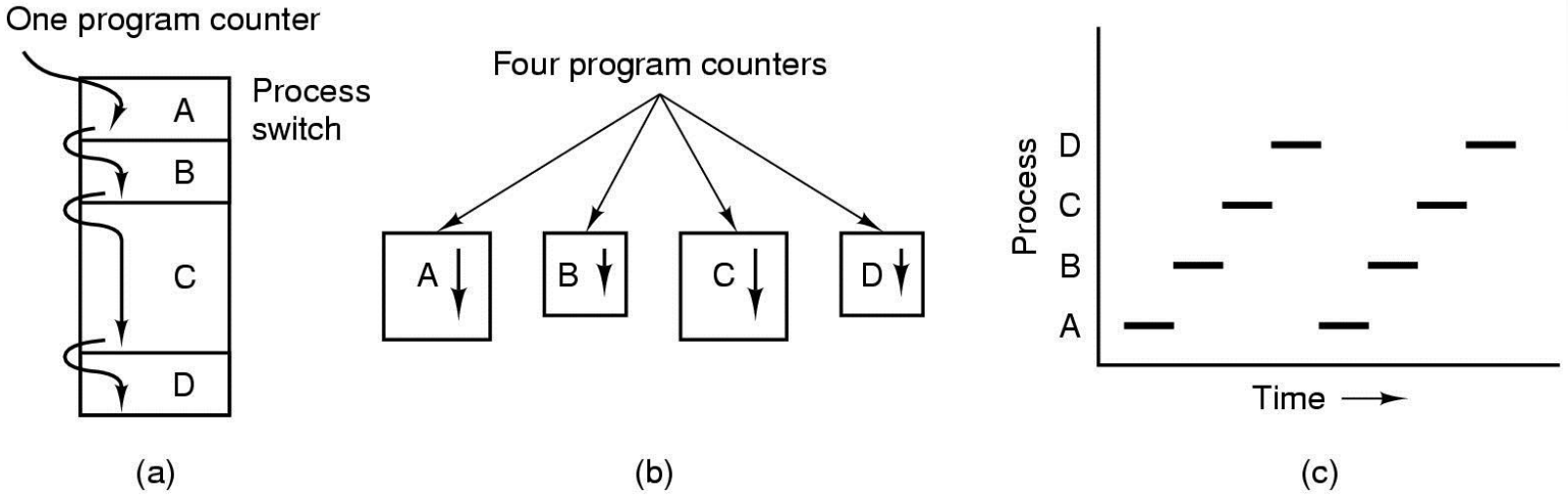
- System initialization
  – Boot, reboot.
- Execution of a process creation system call
  – fork()
- User request to create a new process
  – Command line or click an icon.
- Initiation of a batch job

Process Termination

- Normal exit (voluntary)
  – End of main()
- Error exit (voluntary)
  – exit(2)
- Fatal error (involuntary)
  – Divide by 0, core dump
- Killed by another process (involuntary)
  – kill procID, end task

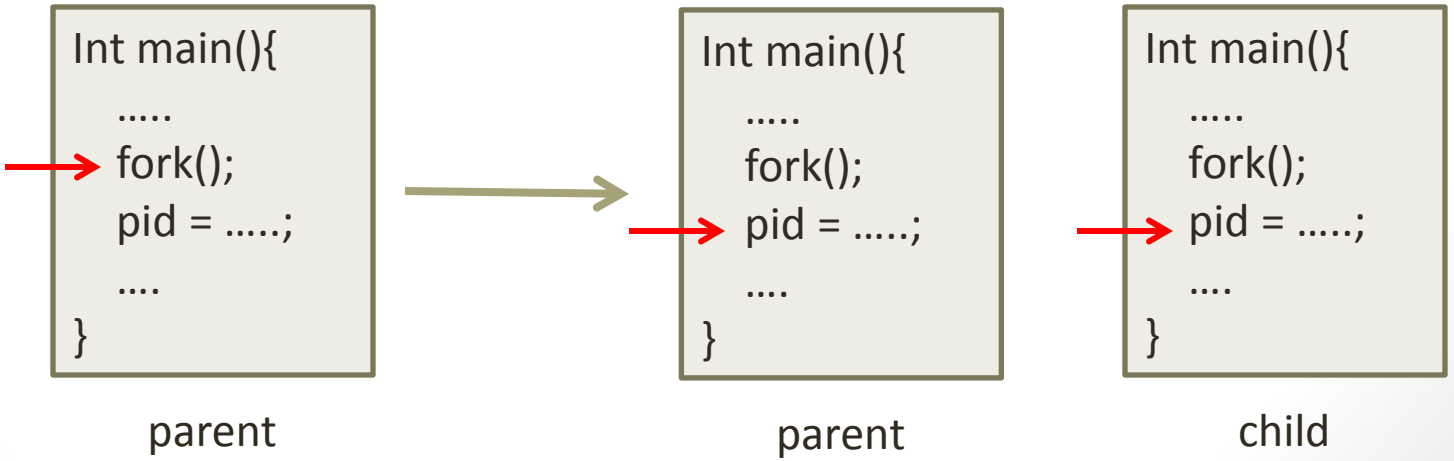
Example: fork() in Unix

- int childpid;

- ```
  if ((childpid = fork()) == -1) {
          perror(can't create a new process);
          exit(1);
  } else if (childpid == 0) {
          // executes child process code
          exit(0);
  } else {
          // executes parent process code
          exit(0);
  }
  ```
  ```

The fork() System Call

- When fork() is called in process A
  - Control is switched to kernel
  - Kernel creates new process B by copying A's
    - Address space
    - Kernel data structure (process descriptor)
  - OS now has two identical processes to run.
    Both resume from after fork().
    However, return value of fork() will be different.



Process States

- Many processes in system, only one on CPU
- "Execution State" of a process:
  – Indicates what it is doing
  – Basically 3 states:
    1. Ready: waiting to be assigned to the CPU
    2. Running: executing instructions on the CPU
    3. Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

Process State Transitions



- As a process executes, it changes *state*
  1. new: The process is being created
  2. ready: The process is waiting to run
  3. running: Instructions are being executed
  4. waiting: Process waiting for some event to occur
  5. terminated: The process has finished execution

Process State Transitions

Processes hop across states as a result of:

- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

Process State Transitions

- Running → Block**:** process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.
- Running → Ready: scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- Ready → Running: all other processes have had their share and it is time for the first process to run again
- Blocked → Ready: external event for which a process was waiting (such as arrival of input) happens.
- New → Ready : process is created.
- Running → Terminated : process has finished execution.

PROCESS HIERARCHIES
AND
PROCESS DATA STRUCTURES

Process Hierarchies

- When process A creates process B, A is called the “parent” process, B is the “child”
- Forms a hierarchy
  – UNIX calls this a "process group“
  – A special process present in boot image is init
- Windows has no process hierarchy
  – Processes are independent
    after creation



Process Data Structures

- OS represents a process using a PCB
  - Process Control Block
  - Has all the details of a process

Process Id	Security Credentials
Process State	Username of owner
General Purpose Registers	Queue Pointers
Stack Pointer	Signal Masks
Program Counter	Memory Management
Accounting Info	…

Process Control Block (PCB)

Fields of a Process Table Entry

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Reference

Modern Operating Systems

Section 2.1

Structure of Processes

CSI 309: Operating System Concepts
United International University

PROCESSES AND PROGRAMS

Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
  – Many users may launch same program
  – One user may launch many instances of the same program
- Program: an algorithm expressed in some suitable notation.
- Process: a program in execution.

What is a process?

- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- The unit of execution
- Operating system provided ***abstraction*** to represent what is needed to run a single program
- The same as “job” or “task” or “sequential process”.

Example: Windows Task Manager



Example: ps in Unix



What is a program?

- A Program is an **executable file** that contains:
  - Code: Machine instructions
  - Data: Variables stored and manipulated in memory
    - initialized variables (globals)
    - dynamically allocated variables (malloc, new)
    - stack variables (C automatic variables, function arguments)
  - DLL: libraries that were not compiled or linked with the program
    - containing code & data, possibly shared with other programs

Process != Program

- A process is an executing program.
- Example:
  - We can run 2 instances of Mozilla Firefox:
    - Same program
    - Separate processes
- Program is passive: Code + data
- Process is running program: stack, registers, program counter

How Program Becomes a Process

- When a program is launched
  - OS loads program into memory
  - Creates kernel data structure for the process
  - Initializes data
  - Starts from an entry point
    - e.g., main(), GUI mouse clicks



Process address space

- set of all memory addresses accessible by a process



MULTIPROGRAMMING

Multiprogramming

- Each process has its own address space (virtual memory address)
- Even if two processes are running the same program, they have their own address space
- OS schedules processes



Multiprogramming



- Multiprogramming of 4 processes in a single CPU
  CPU switches from one process to other process
- Only 1 physical program counter
  4 logical program counters
- Conceptual model of 4 independent, sequential processes
- Only 1 program active at any instant.
- Real life analogy?
  A daycare teacher trying to feed 4 infants.

PROCESS OPERATIONS

Process Creation

Process Termination

Process State Transitions

Process Creation

- System initialization
  – Boot, reboot.
- Execution of a process creation system call
  – fork()
- User request to create a new process
  – Command line or click an icon.
- Initiation of a batch job

Process Termination

- Normal exit (voluntary)
  – End of main()
- Error exit (voluntary)
  – exit(2)
- Fatal error (involuntary)
  – Divide by 0, core dump
- Killed by another process (involuntary)
  – kill procID, end task

Example: fork() in Unix

- int childpid;

- ```
  if ((childpid = fork()) == -1) {
          perror(can't create a new process);
          exit(1);
  } else if (childpid == 0) {
          // executes child process code
          exit(0);
  } else {
          // executes parent process code
          exit(0);
  }
  ```
  ```

The fork() System Call

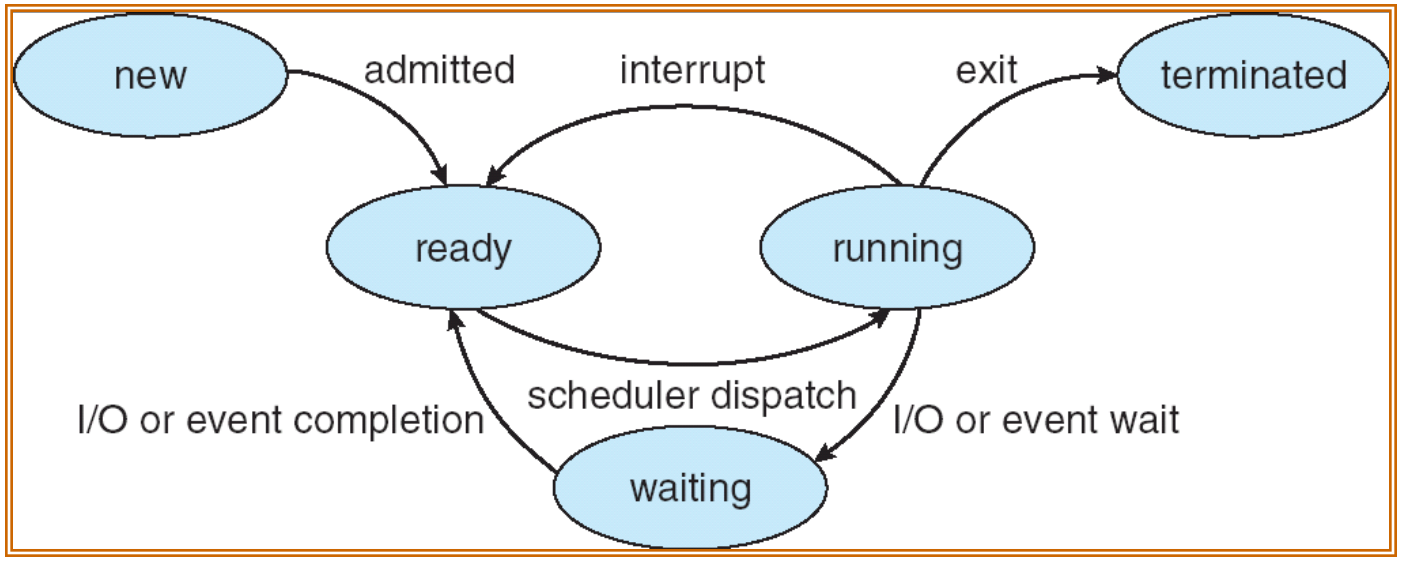
- When fork() is called in process A
  - Control is switched to kernel
  - Kernel creates new process B by copying A's
    - Address space
    - Kernel data structure (process descriptor)
  - OS now has two identical processes to run.
    Both resume from after fork().
    However, return value of fork() will be different.



Process States

- Many processes in system, only one on CPU
- "Execution State" of a process:
  – Indicates what it is doing
  – Basically 3 states:
    1. Ready: waiting to be assigned to the CPU
    2. Running: executing instructions on the CPU
    3. Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

Process State Transitions



- As a process executes, it changes *state*
  1. new: The process is being created
  2. ready: The process is waiting to run
  3. running: Instructions are being executed
  4. waiting: Process waiting for some event to occur
  5. terminated: The process has finished execution

Process State Transitions

Processes hop across states as a result of:

- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

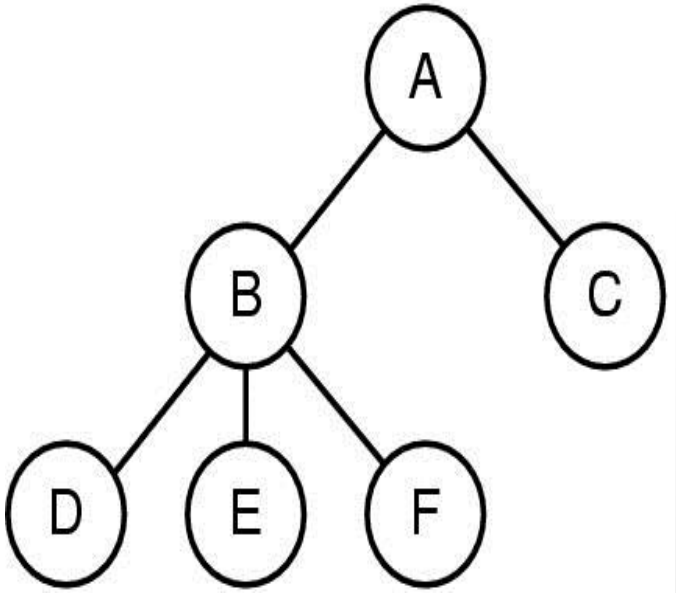
Process State Transitions

- Running → Block**:** process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.
- Running → Ready: scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- Ready → Running: all other processes have had their share and it is time for the first process to run again
- Blocked → Ready: external event for which a process was waiting (such as arrival of input) happens.
- New → Ready : process is created.
- Running → Terminated : process has finished execution.

PROCESS HIERARCHIES
AND
PROCESS DATA STRUCTURES

Process Hierarchies

- When process A creates process B, A is called the “parent” process, B is the “child”
- Forms a hierarchy
  – UNIX calls this a "process group“
  – A special process present in boot image is init
- Windows has no process hierarchy
  – Processes are independent
    after creation



Process Data Structures

- OS represents a process using a PCB
  - Process Control Block
  - Has all the details of a process

Process Id	Security Credentials
Process State	Username of owner
General Purpose Registers	Queue Pointers
Stack Pointer	Signal Masks
Program Counter	Memory Management
Accounting Info	…

Process Control Block (PCB)

Fields of a Process Table Entry

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Reference

Modern Operating Systems

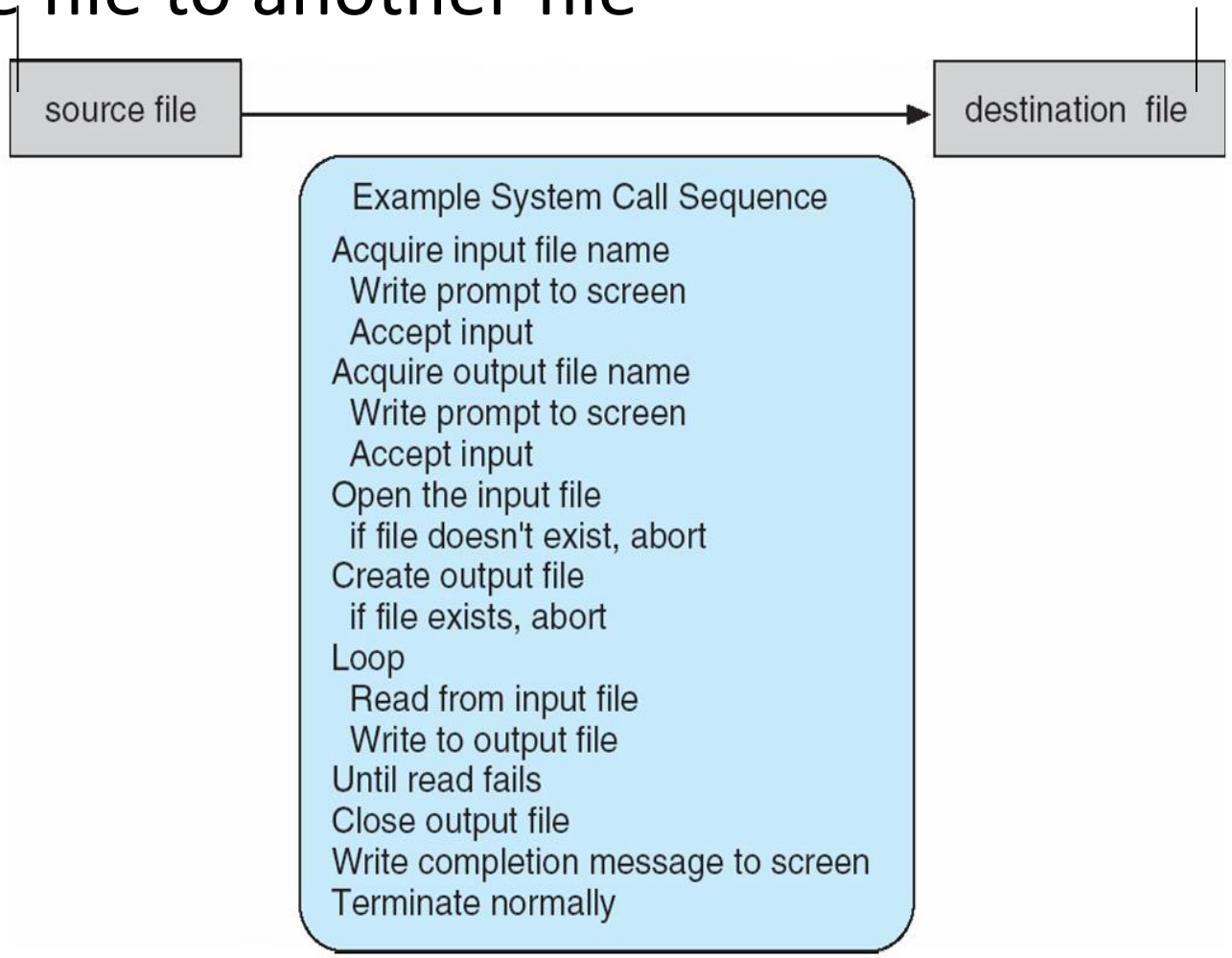
Section 2.1

System Call

- The interface between the operating system and user programs is defined by the set of system calls that the operating system provides
- Programming interface to services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs using Application Programming Interface (API)s
- Three most common APIs:
    - Win32 API for Windows
    - POSIX API for POSIX-based systems (UNIX, Linux, Mac OS X)
    - Java API for the Java virtual machine (JVM)

Example of System Calls

- System call sequence to copy the contents of one file to another file



General Overview

- If a process is
    - running a user program in user mode
    - needs a system service
    - Then it has to execute a trap instruction to transfer control to the operating system
- The operating system then figures out what the calling process wants by inspecting the parameters
    - Then OS carries out the system call
- After that, OS returns control to the instruction following the system call

In a sense, making a system call is like making a special kind of procedure call.

BUT…

Difference with Procedure Call

- The TRAP instruction also differs from the procedure call instruction in two fundamental ways:
  - First, it switches into kernel mode. The procedure call instruction does not change the mode
  - Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address. Depending on the architecture, it either jumps to a single fixed location or equivalent

Example of a SysCall

Example: count = read(fd, buffer, nbytes);

Explanation of the action: The system call returns the number of bytes actually read in count.

Question: why count!=nbytes all the time?

Ways of Passing Parameters

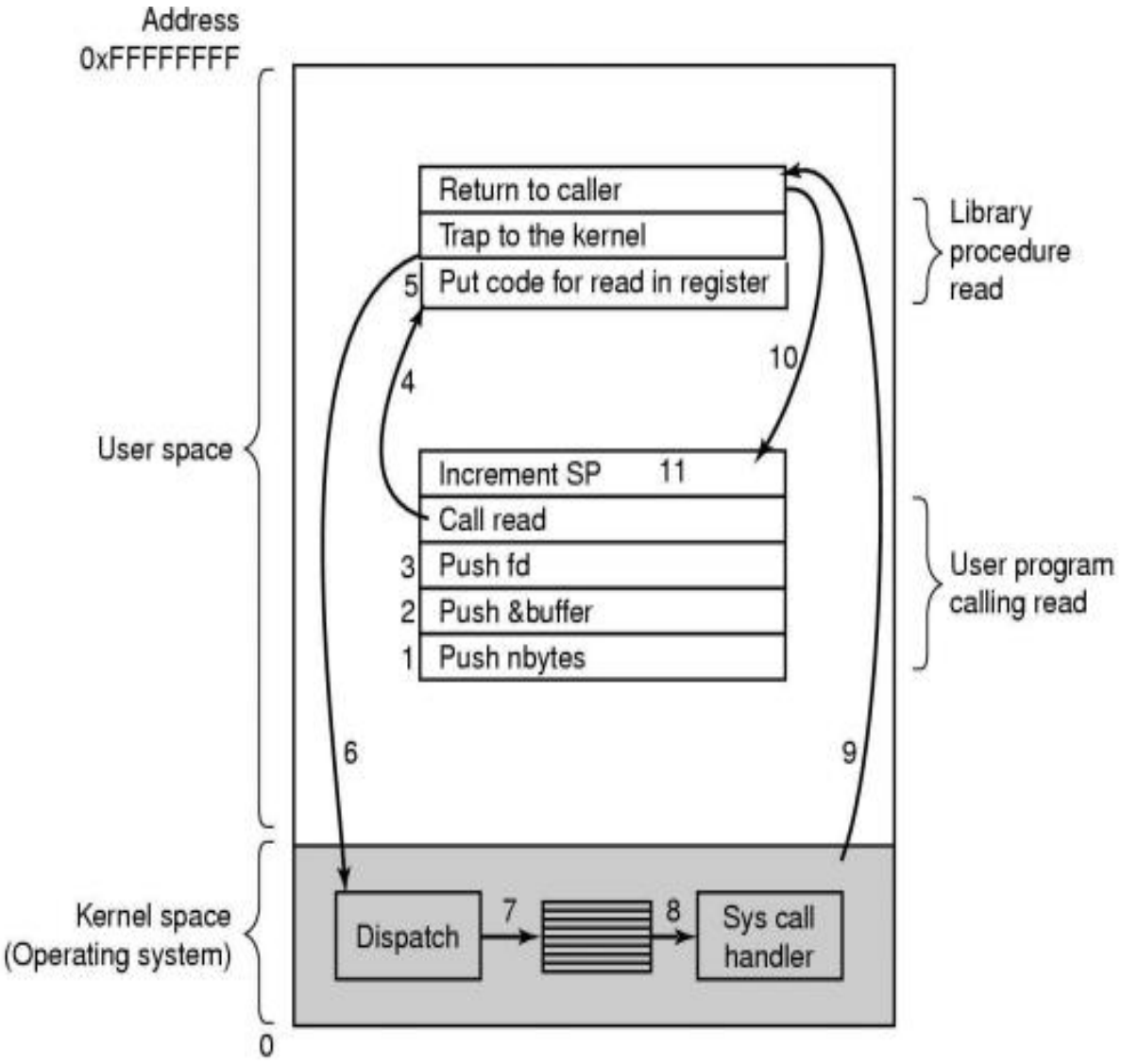
Three general methods for passing parameters:

Method#1: Pass parameters in registers.

Methosd#2: Store the parameters in a table in memory, and the table address is passed as a parameter in a register.

Method#3: Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.

Steps of Making a Sys Call



Steps of Making a SysCall

1. C and C++ compilers pass parameters between a running program and the operating system using parameter passing method#3. (Steps: 1-3)
2. Actual call to library procedure happens. (Step: 4)
3. The library procedure, possibly written in assembly language, passes the system call number using parameter passing method#1. (step 5)
4. A TRAP instruction is performed to switch from user mode to kernel mode. (step 6)
5. The kernel examines the system call number and then dispatches to the correct system call handler, using parameter passing method#2. (step 7)

Steps of Making a SysCall

6. System call handler runs. (step 8)

7. Control may be returned to the user-space library procedure following the TRAP instruction. (step 9)

8. This procedure then returns to the user program in the usual way procedure calls return. (Step 10)

9. To finish the job, the user program has to clean up the stack, as it does after any procedure call. (Step 11)

Types of SysCalls

- Process Control
    - end or abort process; create or terminate process, wait for some time; allocate or de-allocate memory
- File Management
    - open or close file; read, write or seek
- Device Management
    - attach or detach device; read, write or seek
- Information Maintenance
    - get process information; get device attributes, set system time
- Communications
    - create, open, close socket, get transfer status.

Process Control SysCalls

Process management

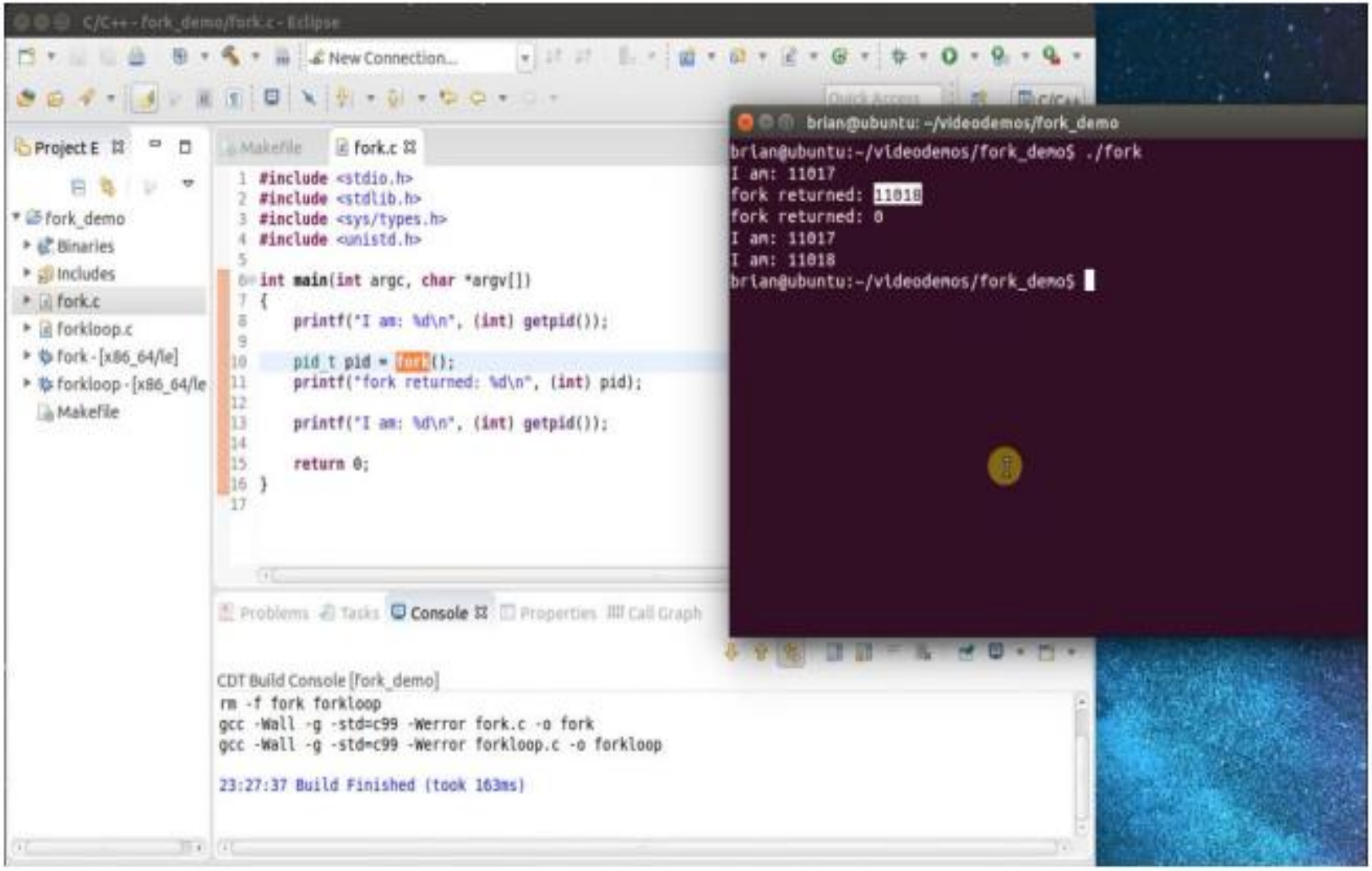
Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
exit(status)	Terminate process execution and return status

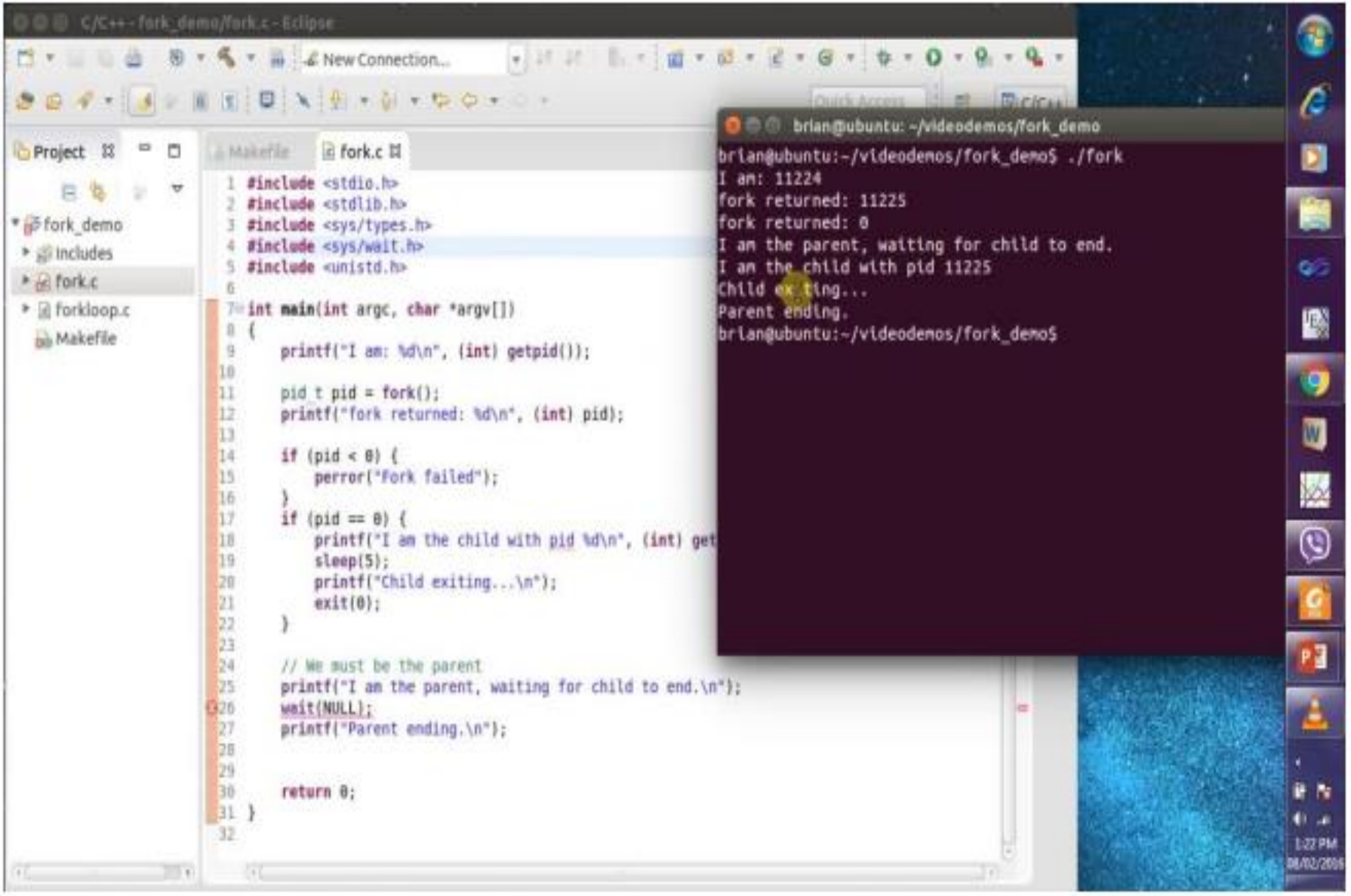
Fork()

- Fork is a way to create a new process

  - Question: Does any change after fork() affect the original and copy processes? Why or why not?

- The fork() call returns a value
  - zero in the child
  - the child's process identifier or PID in the parent


C/C++ - fork_demo/fork.c - Eclipse
New Connection...
Project E
fork_demo
Binaries
Includes
fork.c
forkloop.c
fork - [x86_64/le]
forkloop - [x86_64/le
Makefile
Makefile
fork.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[])
7 {
8 printf("I am: %d\n", (int) getpid());
9
10 pid_t pid = fork();
11 printf("fork returned: %d\n", (int) pid);
12
13 printf("I am: %d\n", (int) getpid());
14
15 return 0;
16 }
17
Problems
Tasks
Console
Properties
Call Graph
CDT Build Console [fork_demo]
rm -f fork forkloop
gcc -Wall -g -std=c99 -Werror fork.c -o fork
gcc -Wall -g -std=c99 -Werror forkloop.c -o forkloop
23:27:37 Build Finished (took 163ms)
brian@ubuntu: ~/videodemos/fork_demo
brian@ubuntu:~/videodemos/fork_demo$ ./fork
I am: 11017
fork returned: 11018
fork returned: 0
I am: 11017
I am: 11018
brian@ubuntu:~/videodemos/fork_demo$


C/C++ - fork_demo/fork.c - Eclipse
New Connection...
Project
fork_demo
Includes
fork.c
forkloop.c
Makefile
Makefile
fork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
printf("I am: %d\n", (int) getpid());
pid_t pid = fork();
printf("fork returned: %d\n", (int) pid);
if (pid < 0) {
perror("Fork failed");
}
if (pid == 0) {
printf("I am the child with pid %d\n", (int) get
sleep(5);
printf("Child exiting...\n");
exit(0);
}
// We must be the parent
printf("I am the parent, waiting for child to end.\n");
wait(NULL);
printf("Parent ending.\n");
return 0;
}
brian@ubuntu: ~/videodemos/fork_demo
brian@ubuntu:~/videodemos/fork_demo$ ./fork
I am: 11224
fork returned: 11225
fork returned: 0
I am the parent, waiting for child to end.
I am the child with pid 11225
Child exiting...
Parent ending.
brian@ubuntu:~/videodemos/fork_demo$
1:22 PM

System Calls for File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Lseek():

Purpose:

For some applications programs need to be able to access any part of a file at random.

Way:

- Associated with each file is a pointer that indicates the current position in the file.
- When reading (writing) sequentially, it normally points to the next byte to be read (written).

Lseek(): The Iseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file. Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.

PROCESSES

PROCESS

- Operating system provided ***abstraction*** to represent what is needed to run a single program
- The unit of execution
- Multiple Parts
  - The Program Code, Also Called **Text Section**
  - Current Activity Including the current values of **PC**, Registers
  - **Stack** Containing Temporary Data
    - Function Parameters, Return Addresses, Local Variables
  - **Data Section** Containing Global Variables
  - **Heap** Containing Memory Dynamically Allocated During Run Time

WHAT IS A PROGRAM?

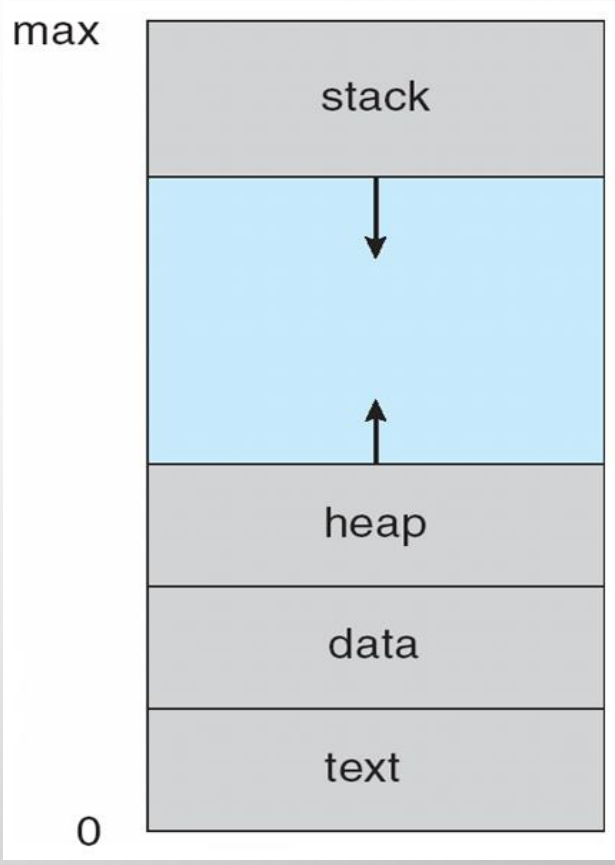
- A Program is an **executable file** that contains:
  - Code: Machine Instructions
  - Data: Variables Stored And Manipulated In Memory
    - initialized variables (globals)
    - dynamically allocated variables (malloc, new)
    - stack variables (C automatic variables, function arguments)
- Process != Program
- Example:
  - We can run 2 instances of *Mozilla Firefox*:
    - Same program
    - Separate processes

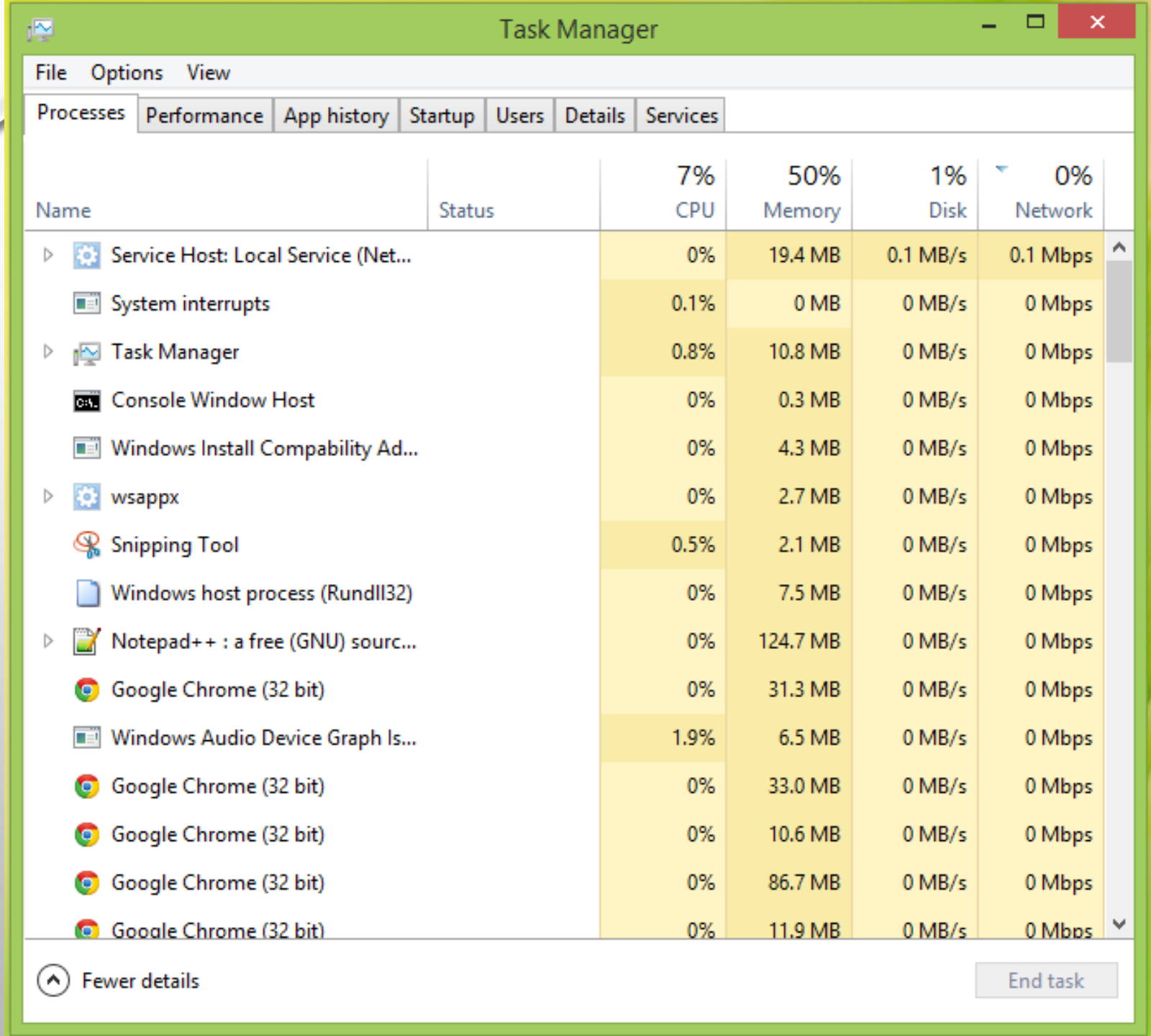
AN ANALOGY
BAKING A CAKE

- Need a cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar ….
- the recipe is the program (i.e., an algorithm expressed in some suitable notation),
- the baker is the processor (CPU),
- and the cake ingredients are the input data.
- The process is the *activity* consisting of the baker reading the recipe, fetching the ingredients, and baking the cake.

PROCESS IN MEMORY

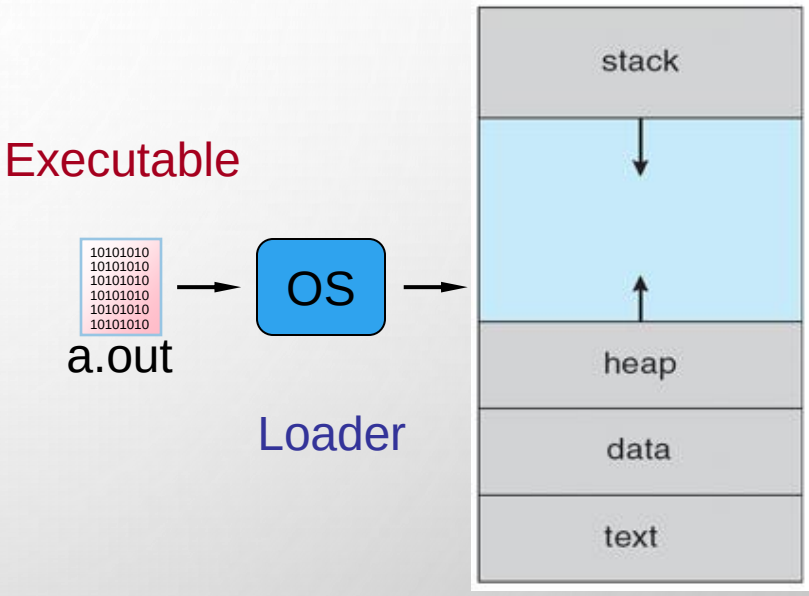
- Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- Process address space
  - set of all memory addresses accessible by a process

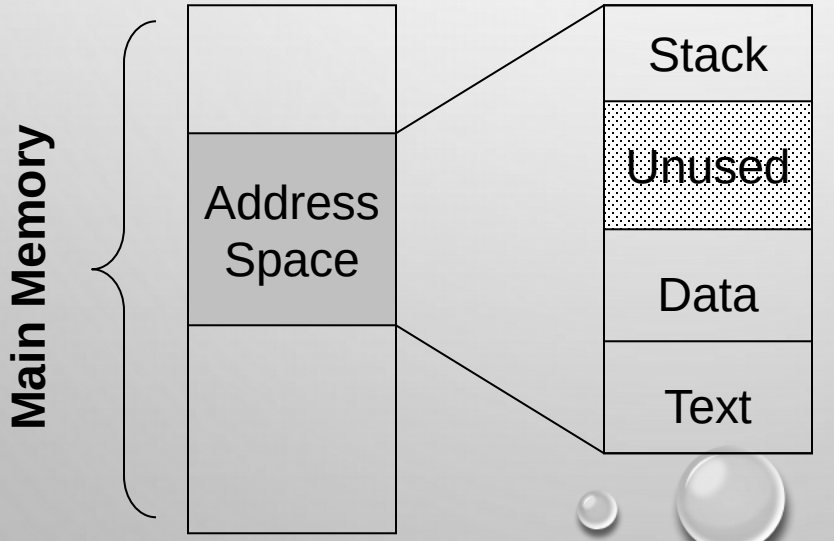



Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Name Status 7% CPU 50% Memory 1% Disk 0% Network
Service Host: Local Service (Net... 0% 19.4 MB 0.1 MB/s 0.1 Mbps
System interrupts 0.1% 0 MB 0 MB/s 0 Mbps
Task Manager 0.8% 10.8 MB 0 MB/s 0 Mbps
Console Window Host 0% 0.3 MB 0 MB/s 0 Mbps
Windows Install Compability Ad... 0% 4.3 MB 0 MB/s 0 Mbps
wsappx 0% 2.7 MB 0 MB/s 0 Mbps
Snipping Tool 0.5% 2.1 MB 0 MB/s 0 Mbps
Windows host process (Rundll32) 0% 7.5 MB 0 MB/s 0 Mbps
Notepad++ : a free (GNU) sourc... 0% 124.7 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 31.3 MB 0 MB/s 0 Mbps
Windows Audio Device Graph Is... 1.9% 6.5 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 33.0 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 10.6 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 86.7 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 11.9 MB 0 MB/s 0 Mbps
Fewer details
End task

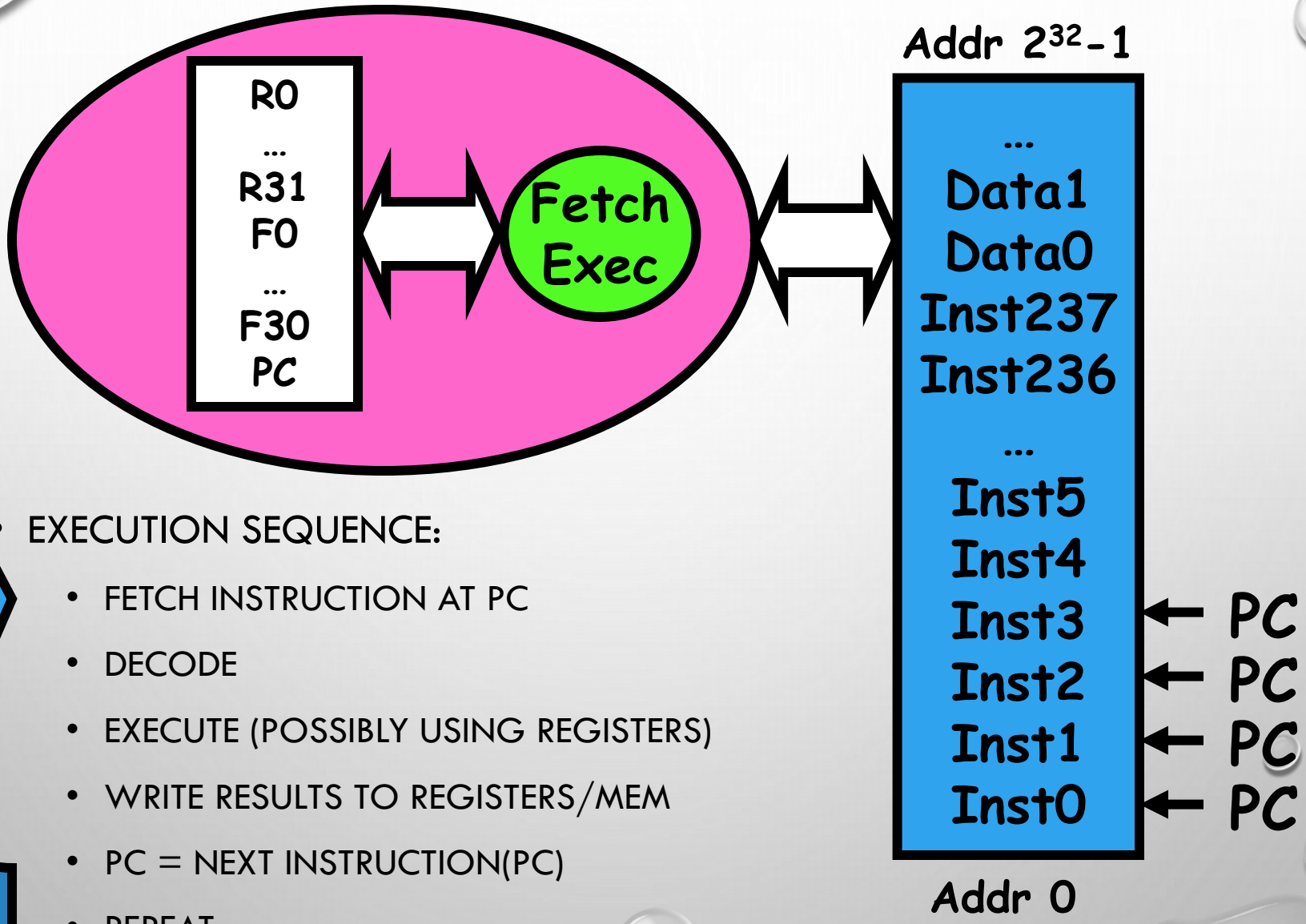
HOW PROGRAM BECOMES PROCESS

- When a program is launched
  - OS loads program into memory
  - Creates kernel data structure for the process
  - Initializes data
  - Starts from an entry point (e.g., main())



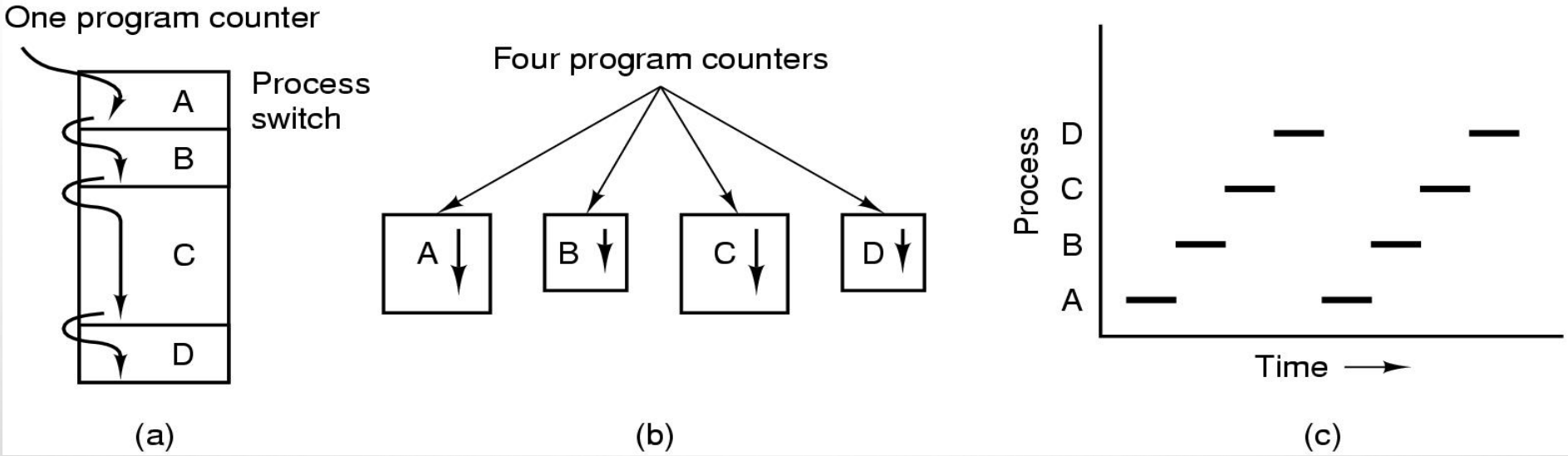


RECALL: WHAT HAPPENS DURING EXECUTION?



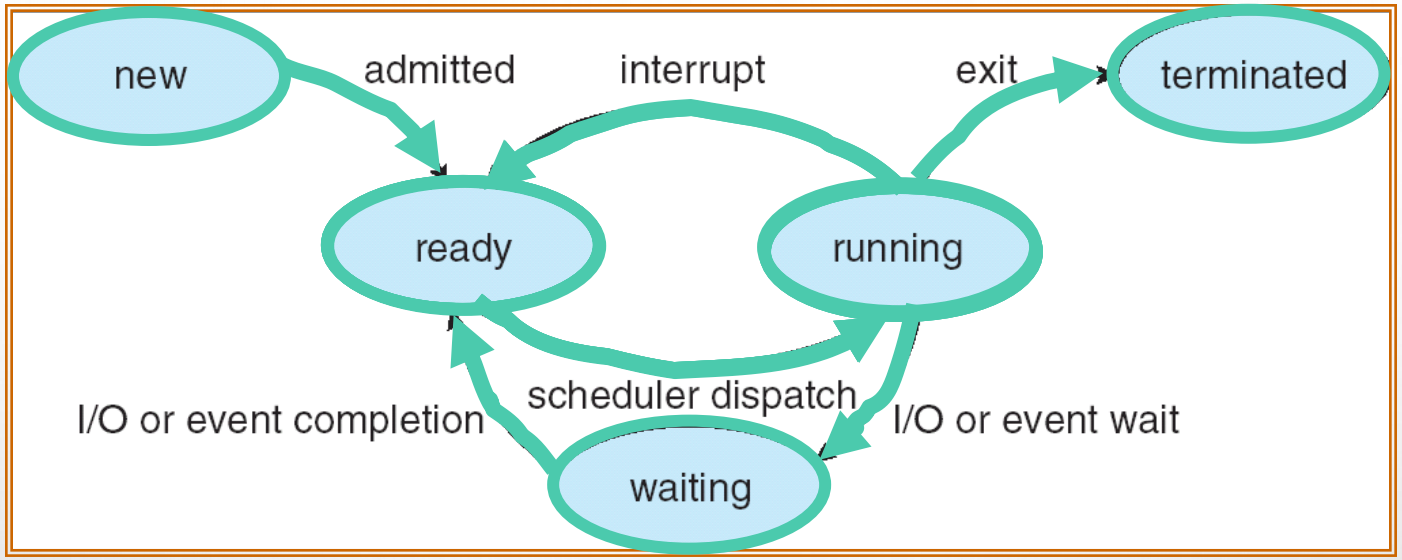
- EXECUTION SEQUENCE:
  - FETCH INSTRUCTION AT PC
  - DECODE
  - EXECUTE (POSSIBLY USING REGISTERS)
  - WRITE RESULTS TO REGISTERS/MEM
  - PC = NEXT INSTRUCTION(PC)
  - REPEAT

MULTIPROGRAMMING



- Multiprogramming Of 4 Processes in a Single CPU
  - CPU switches from one process to other process
- Only One Physical Program Counter, 4 Logical Program Counter.
- Conceptual Model Of 4 Independent, Sequential Processes
- Only One Program Active At Any Instant.
- Real Life Analogy? - A Daycare Teacher Trying To Feed 4 Infants.

LIFECYCLE OF A PROCESS



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

PROCESS DATA STRUCTURES

- OS represents a process using a Process Control Block (*PCB)*
  - Has all the details of a process
  - Context of the process

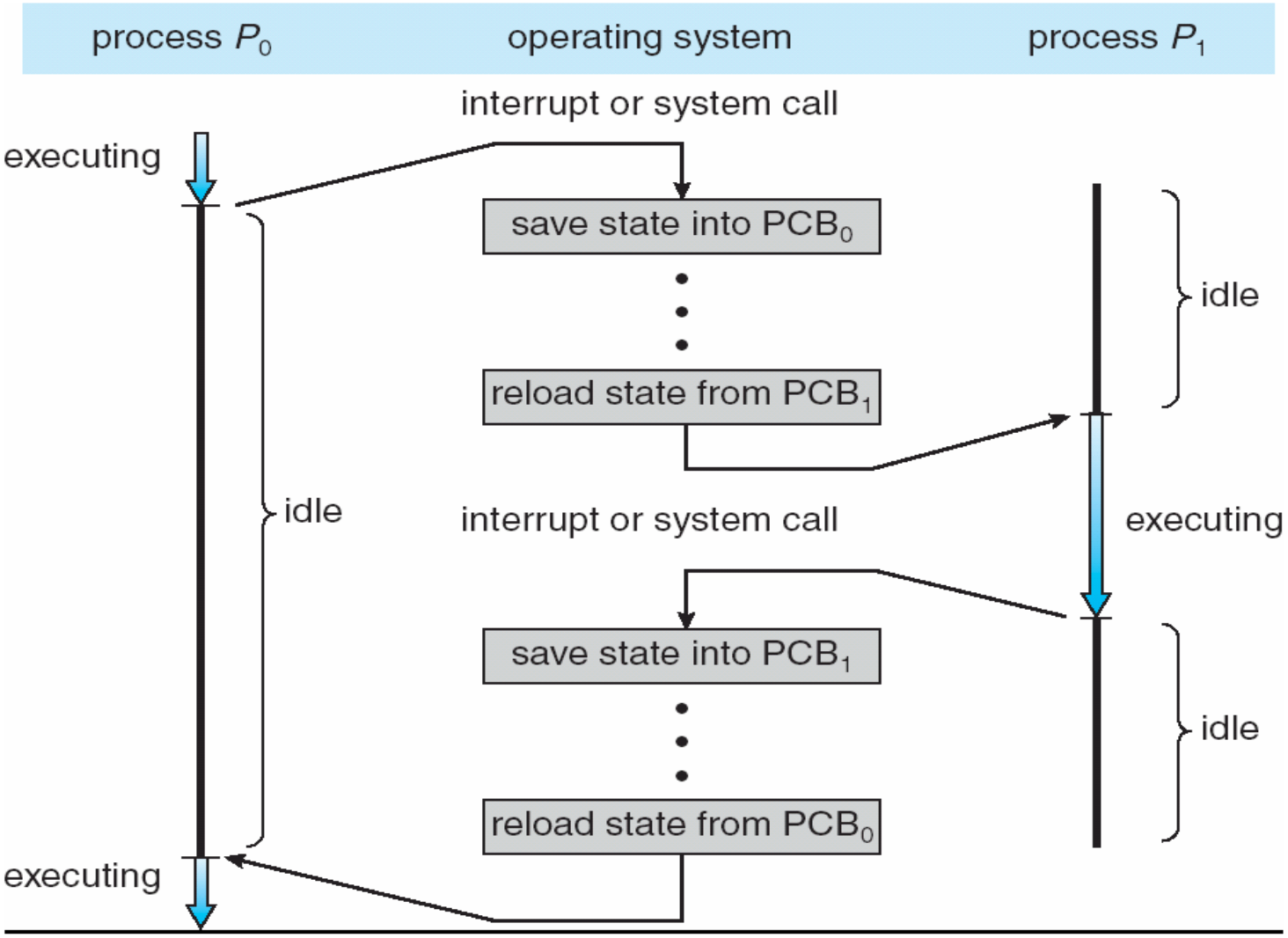
Process Id	**Security Credentials**
Process State	**Username of owner**
General Purpose Registers	**Queue Pointers**
Stack Pointer	**Signal Masks**
Program Counter	**Memory Management**
Accounting Info	**…**

PROCESS CONTROL BLOCK (PCB)

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure: Fields of a PCB

CPU SWITCH FROM PROCESS TO PROCESS



CONTEXT SWITCH

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process

⇒Context Switch

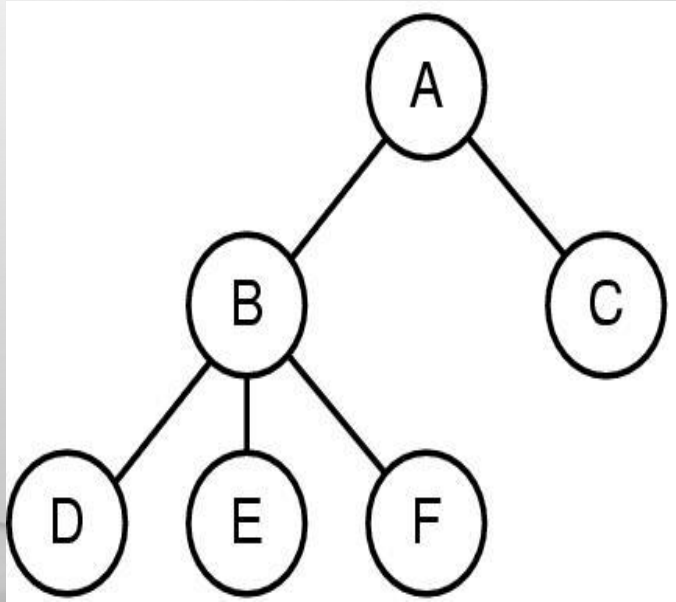
- Process of switching CPU from one process to another
- Very machine dependent for types of registers

CONTEXT SWITCH

- **Overheads**: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

PROCESS HIERARCHIES

- When process A creates process B, A is called the “parent” process, B is the “child”
- Forms a hierarchy
  - UNIX calls this a "process group“
- Windows has NO process hierarchy
  - Processes are independent after creation



WHAT DOES IT TAKE TO CREATE A PROCESS?

- Must construct new PCB
  - Inexpensive
- Must set up new address space
  - More expensive

PROCESS OVERHEADS

- A full process includes numerous things:
  - an address space (defining all the code and data)
  - OS resources and accounting information
  - a “thread of control”,
    - defines where the process is currently executing
    - That is the stack and registers

- Creating a new process is costly
  - all of the structures that must be allocated

- Context switching is costly
  - Implicit and explicit costs as we talked about

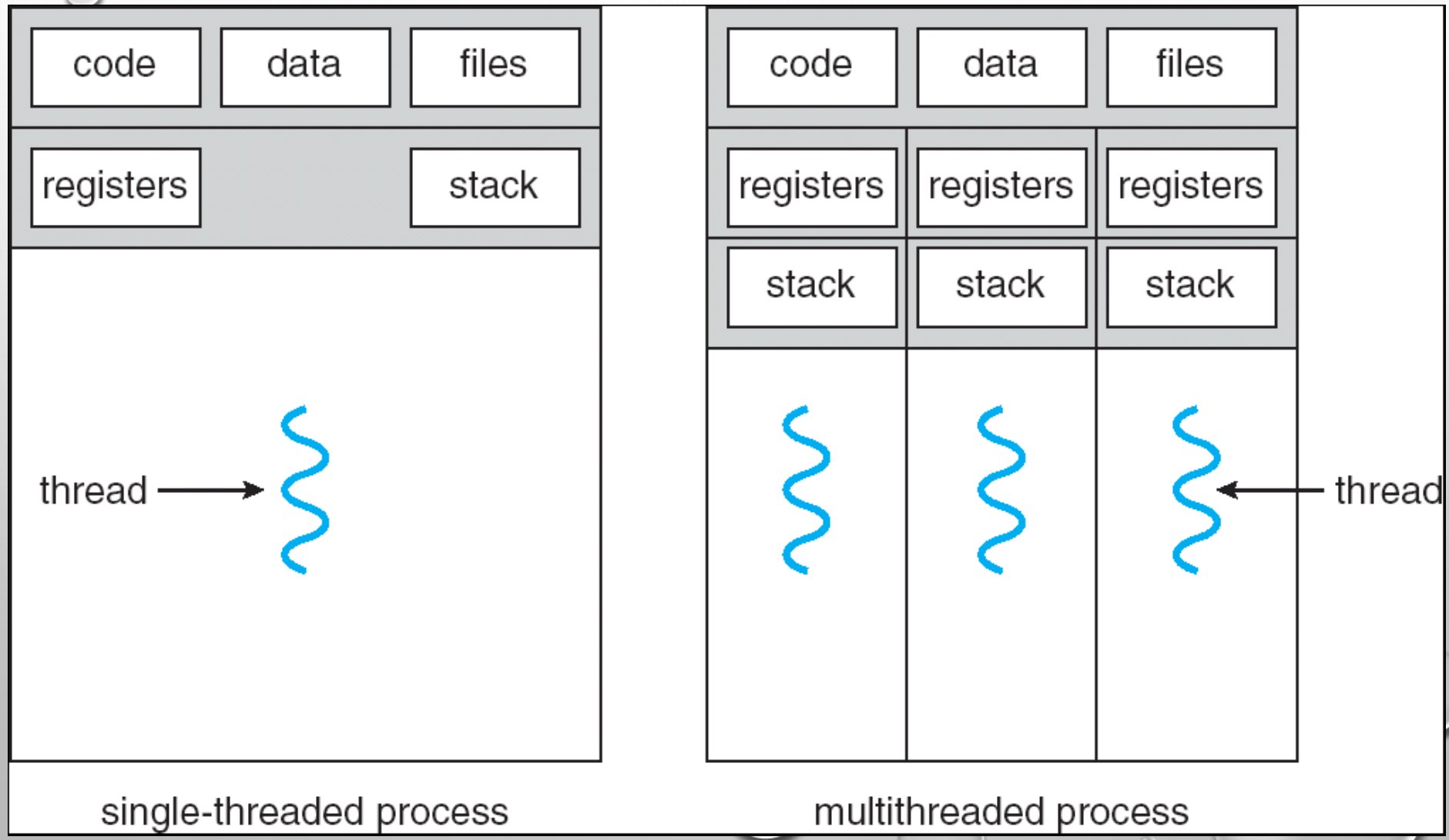
NEED SOMETHING MORE LIGHTWEIGHT

- What's similar in these processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They share almost everything in the process
- What don't they share?
  - Each has its own PC, registers, and stack
- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, stack, registers)?

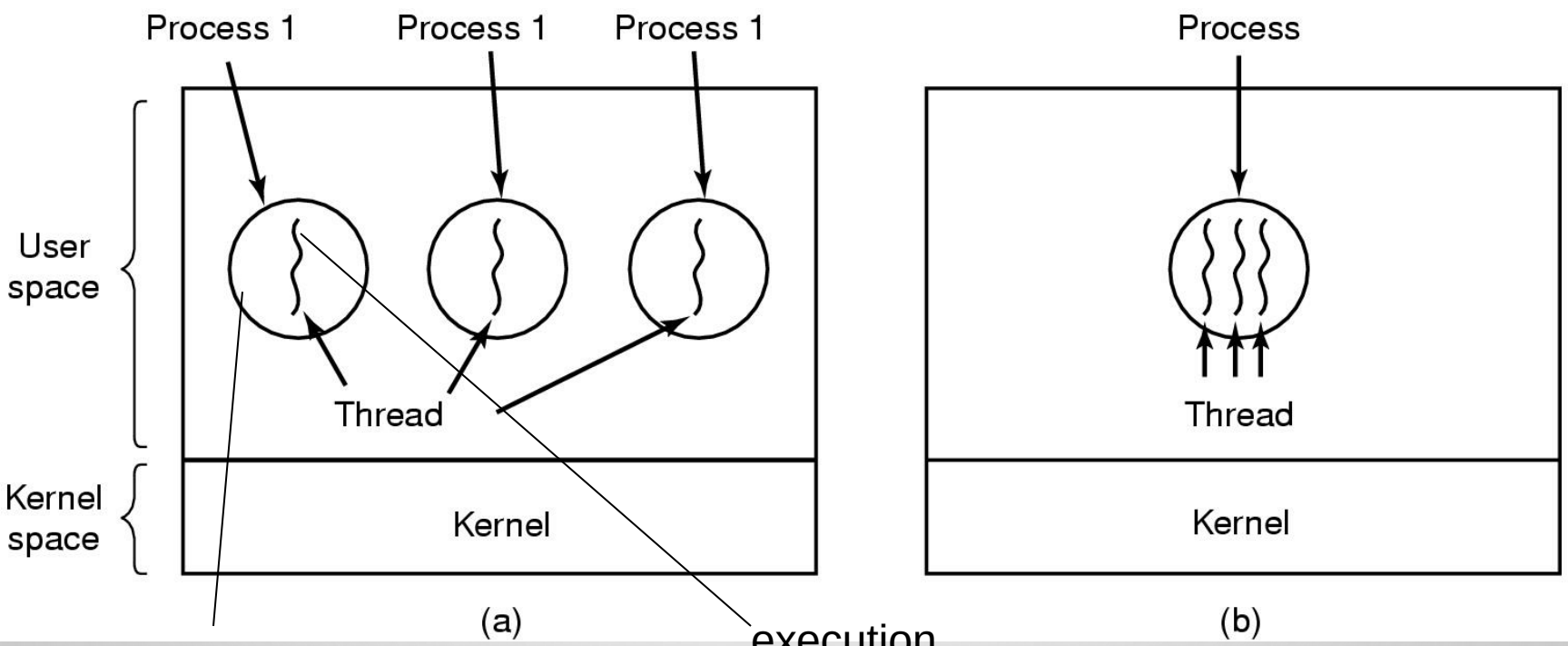
THREADS AND PROCESSES

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a **sequential** execution stream within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

MULTITHREADED PROCESSES



THREADS



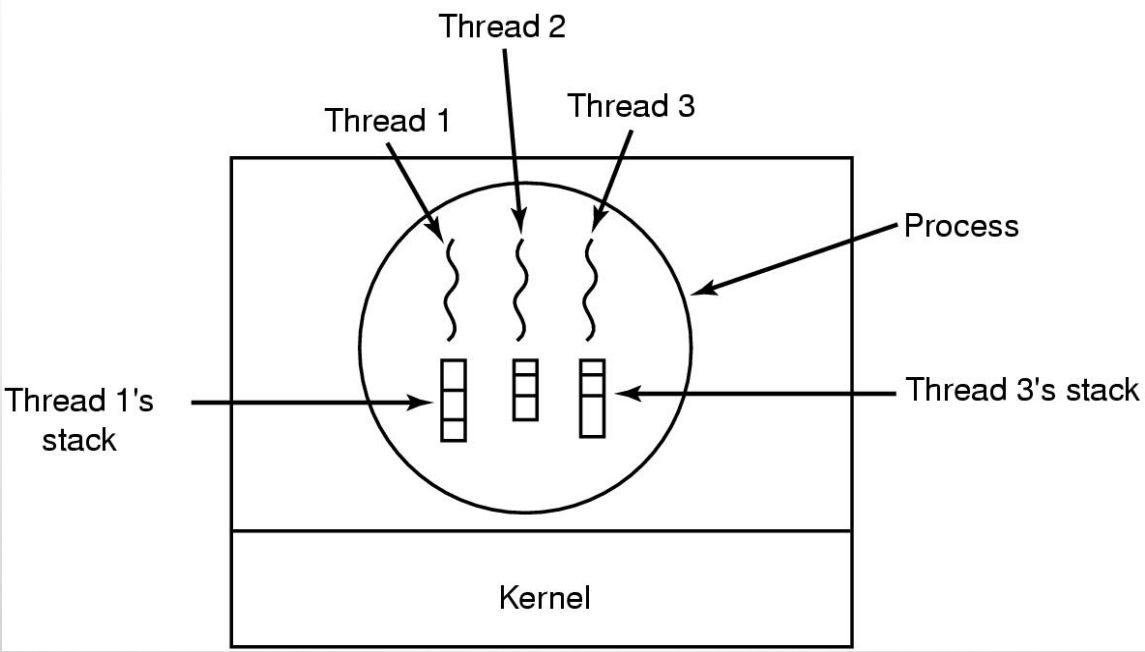
(A) THREE PROCESSES EACH WITH ONE THREAD

(B) ONE PROCESS WITH THREE THREADS

THE THREAD MODEL

- Shared information
  - Address space: text, data structures, etc
  - I/O and file: comm. ports, directories and file descriptors, etc
  - Global variables and child processes.
  - Accounting info: stats
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack
- Each thread execute separately

WHY EACH THREAD HAS ITS OWN STACK?

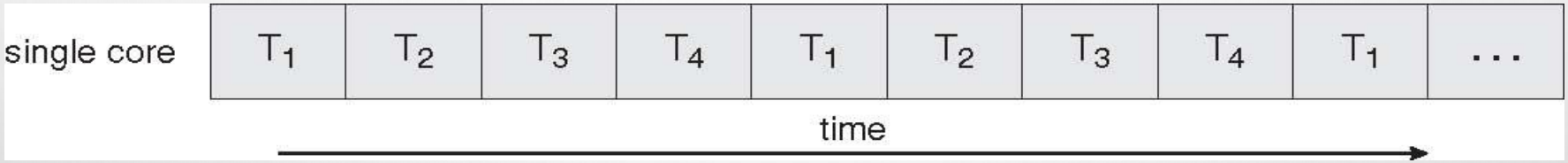


- What will happen if they share one stack?
  - Each thread call different procedures and each has a different execution history.

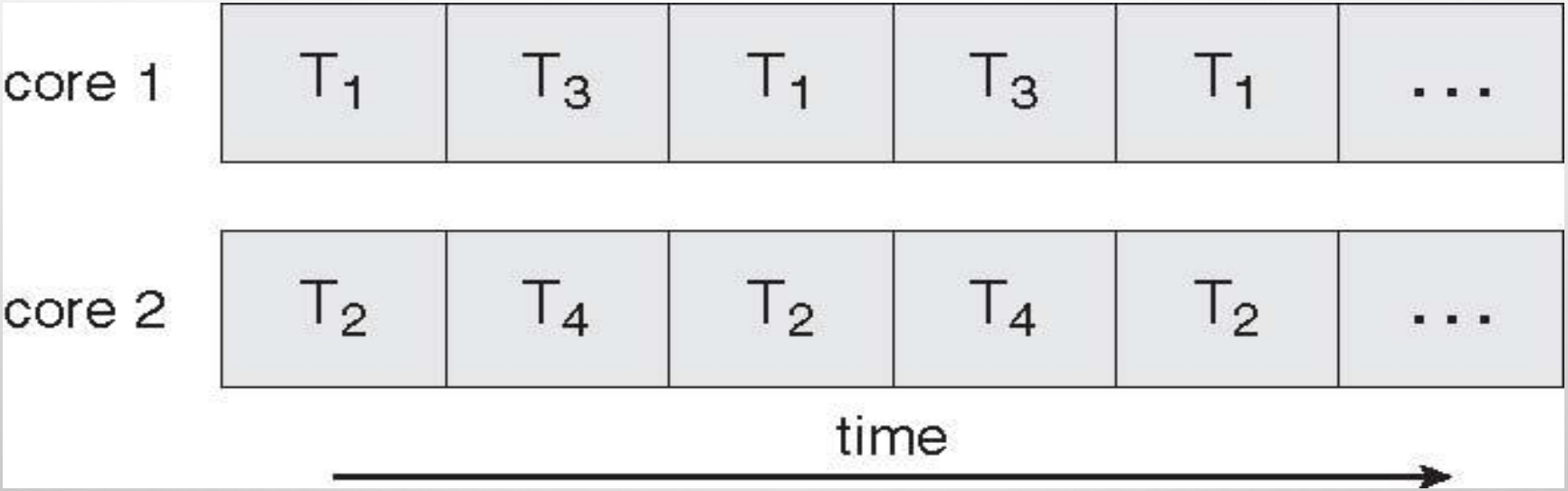
THREAD CONTEXT SWITCH

- Multiplex multiple threads on single CPU
- Similar to process context switch, but less expensive
  - Still needs to switch register set
  - But no memory management related work!!!

CONCURRENT EXECUTION ON A SINGLE-CORE SYSTEM



PARALLEL EXECUTION ON A MULTICORE SYSTEM



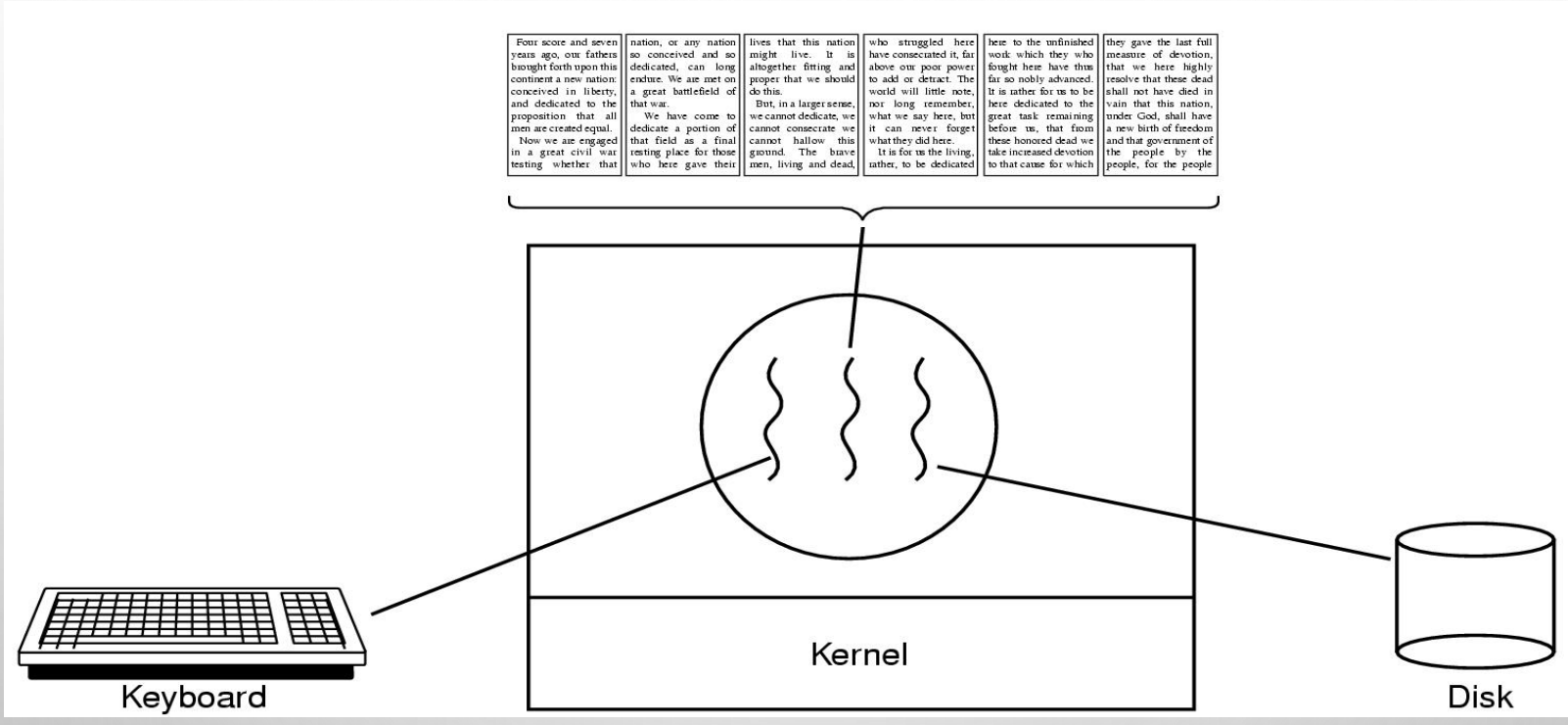
THREAD DYNAMICS

- Threads are dynamically created/terminated
- Multiple threads need to be scheduled
  - Ready
  - Blocked
  - Running
  - Terminated
- Threads share CPU and on single processor machine only one thread can run at a time

THREAD USAGE

- Why need threads?
  - Simplify coding
  - Concurrent activities within a process
    - Better CPU utilization
    - Better responsiveness
  - Less costly to create & switch
  - Utilizing parallelism of multi-processor systems

THREAD USAGE: WORD PROCESSOR



- A thread can wait for I/O, while the others can still be running.
- What if it is single-threaded?

THREAD USAGE: WEB SERVER

- How does one server serves multiple requests
  - Requests come asynchronously
  - Each request takes variable amount of time.

THREAD USAGE: WEB SERVER

- Single process, single thread

```
for(;;){
    wait_for_request();
    handle_request();
}
```

- Problem?

THREAD USAGE: WEB SERVER

- Multi-process

```
for(;;){
    wait_for_request();
    child_pid = fork();
    if(child_pid==0) {
       handle_request(); exit(0);
    }
}
```

- Problem?

THREAD USAGE: WEB SERVER

- Multi-thread

```
for(;;){
    wait_for_request();
    //create a thread to handle request
}
```

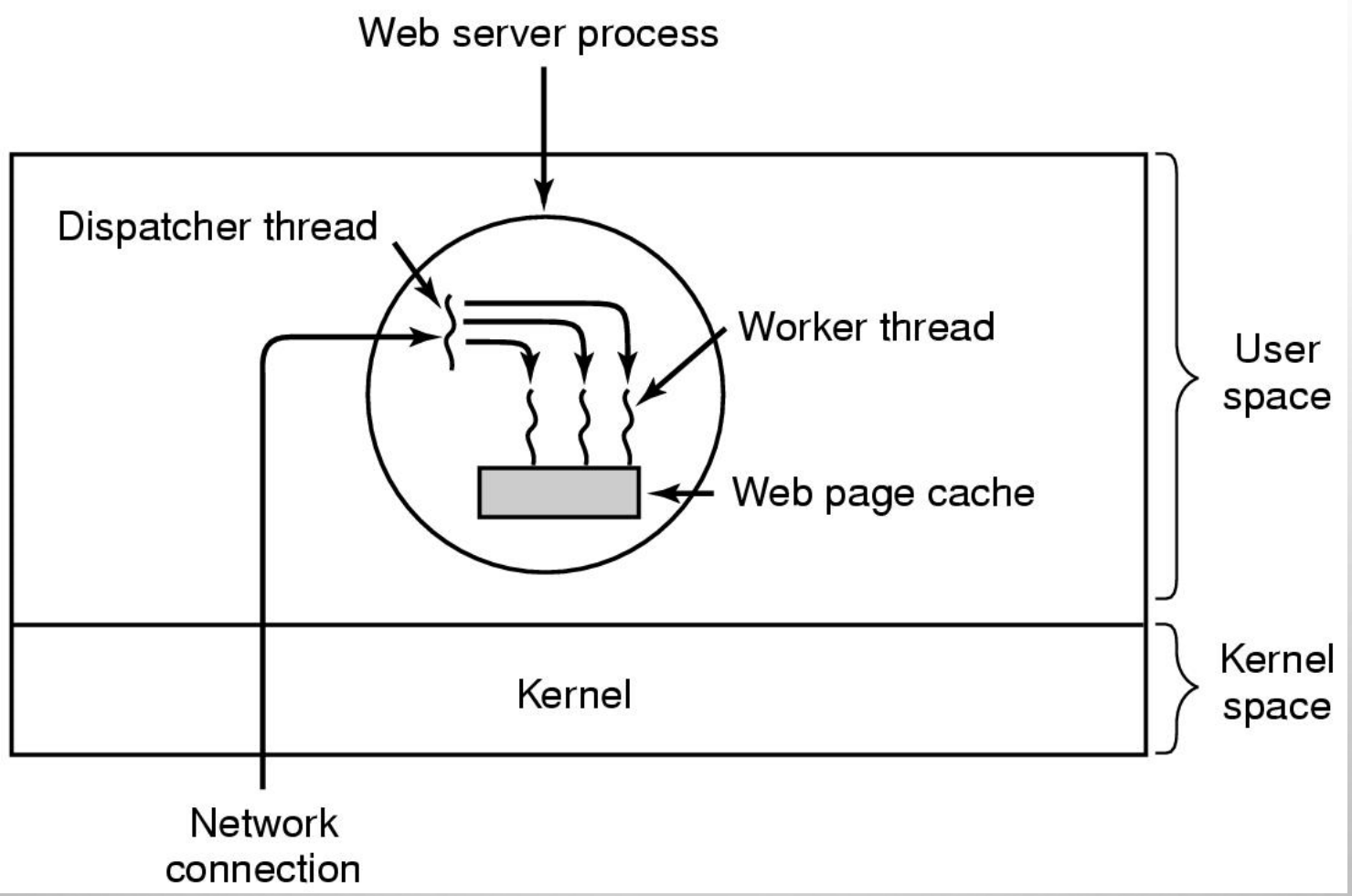
- Problem?

THREAD USAGE: WEB SERVER

- Thread pool

```
//create multiple worker threads first
for(;;){
    wait_for_request();
    //pass request to one worker thread
}
```

THREAD USAGE: WEB SERVER



BLOCKING SYSTEM CALLS

- Usually I/O related: read(), fread(), getc(), write()
- Doesn't return until the call completes
- The process/thread is switched to blocked state
- When the I/O completes, the process/thread becomes ready
- Simple to implement

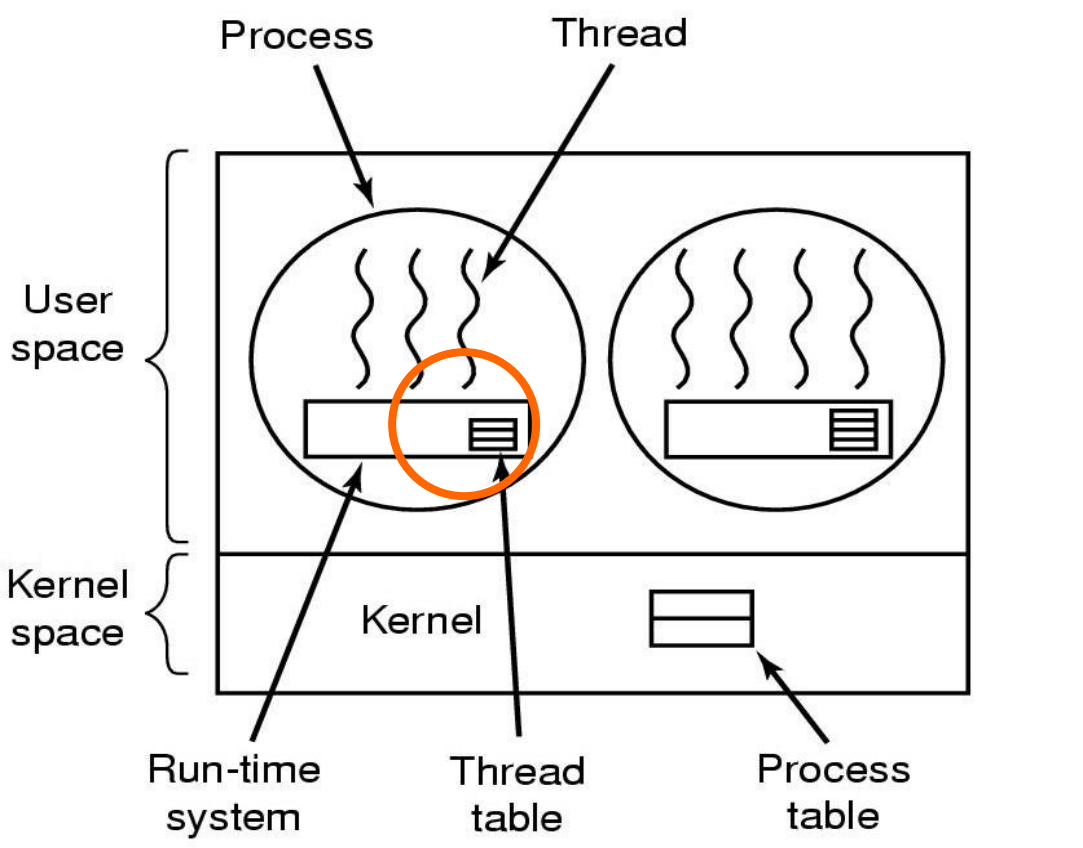
THREAD IMPLEMENTATION

- In user space
  - Kernel unaware of multiple threads
  - User level runtime system does scheduling
- In kernel space
  - Kernel supports threads (lightweight process)

USER-LEVEL THREADS

- The thread scheduler is part of a *user-level library*
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

IMPLEMENTING THREADS IN USER SPACE



A USER-LEVEL THREADS PACKAGE

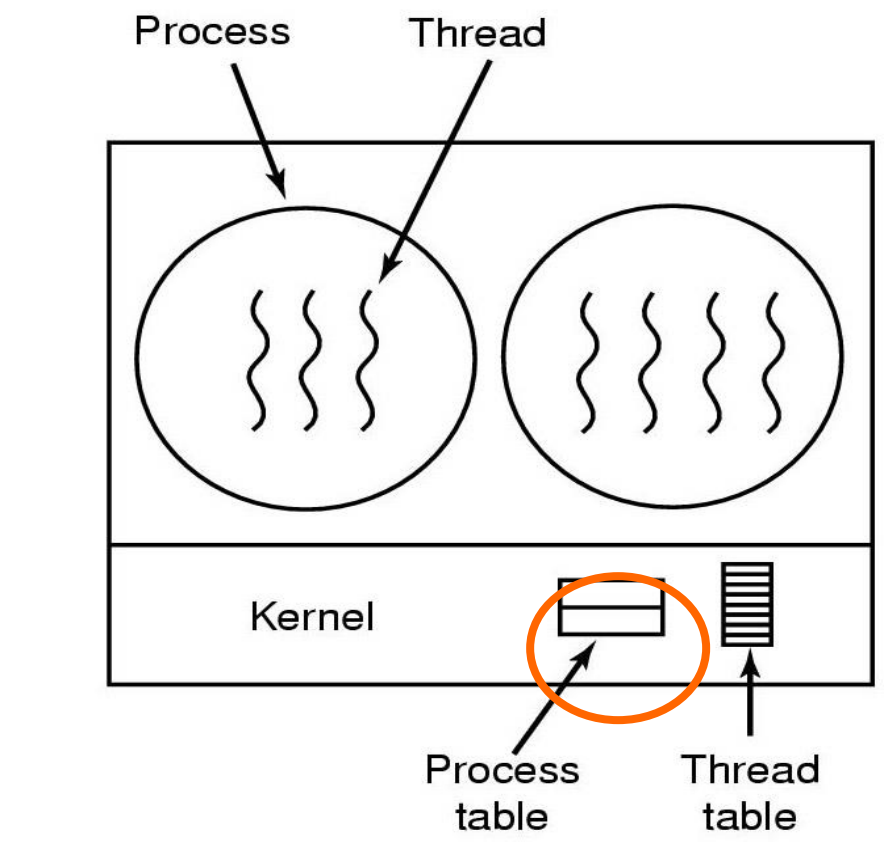
USER-LEVEL THREADS

- Advantages
  - Fast Context Switching:
    - Switching entirely in user mode – local procedures.
    - No need to trap to kernel, no memory flush;
  - Customized Scheduling
- Disadvantages
  - Blocking
    - Any user-level thread can block the entire task executing a single system call (page fault is similar case).
  - No protection, threads are expected to be polite to share CPU.
    - Uncooperative/buggy threads may monopolize CPU.

KERNEL THREADS

- Kernel threads may not be as heavy weight as processes, but they still suffer from performance problems:
  - Any thread operation still requires a system call.
  - The kernel doesn’t trust the user
    - there must be lots of checking on kernel calls

IMPLEMENTING THREADS IN THE KERNEL



A thread package managed by the kernel

KERNEL-LEVEL THREADS

- Advantages:
  - Kernel aware of threads, if one thread blocks, can schedule another thread in the process.
- Disadvantages:
  - Context switch is more expensive.

USER-LEVEL VS. KERNEL THREADS

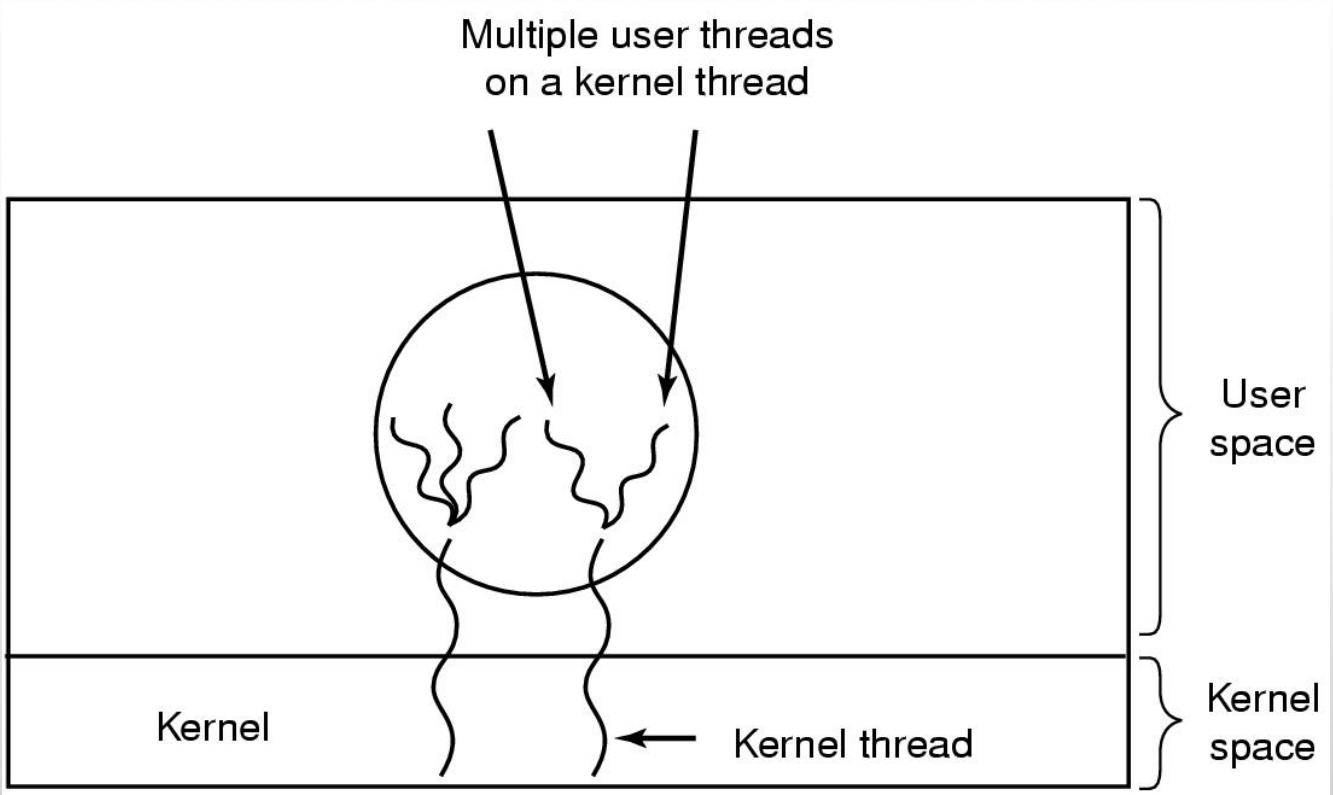
USER-LEVEL

- MANAGED BY APPLICATION
- KERNEL NOT AWARE OF THREAD
- CONTEXT SWITCHING CHEAP
- CREATE AS MANY AS NEEDED
- MUST BE USED WITH CARE

KERNEL-LEVEL

- MANAGED BY KERNEL
- CONSUMES KERNEL RESOURCES
- CONTEXT SWITCHING EXPENSIVE
- NUMBER LIMITED BY KERNEL RESOURCES
- SIMPLER TO USE

HYBRID IMPLEMENTATIONS



Multiplexing user-level threads onto kernel- level threads

HYBRID IMPLEMENTATIONS

- Combining the advantages of the 2 methods
- the kernel is aware of only the kernel-level threads and schedules those.
- ,each kernel-level thread has some set of user-level threads that take turns using it.
- These user-level threads are created, destroyed, and scheduled just like user-level threads in a process

THANKS FOR YOUR PATIENCE

INTERPROCESS COMMUNICATION

INTERPROCESS COMMUNICATION

- Consider shell pipeline
  - `cat chapter1 chapter2 chapter3 | grep tree`
  - 2 processes
  - Information sharing
  - Order of execution

INTERPROCESS COMMUNICATION

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes require a mechanism to exchange data and information

IPC ISSUES

1. How one process passes information to another ?
2. How to make sure that two or more processes do not get into each other's way when engaging in critical activities?
3. How to do proper sequencing when dependencies are present?

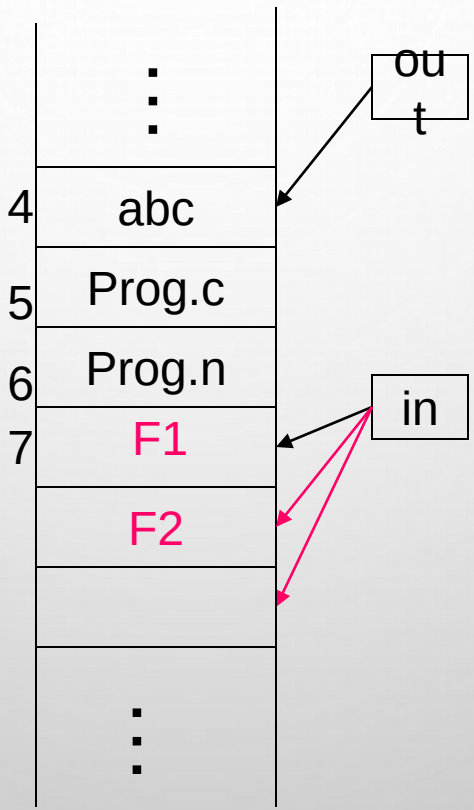
- 1: easy for threads, for processes different approaches (e.g., message passing, shared memory)
- 2 and 3: same problems and same solutions apply for threads and processes
  - Mutual exclusion & Synchronization

SPOOLING EXAMPLE: CORRECT

Process 1

int next_free;

1. next_free = in;
2. Stores F1 into next_free;
3. in=next_free+1



Process 2

int next_free;

4. next_free = in
5. Stores F2 into next_free;
6. in=next_free+1

SPOOLING EXAMPLE: RACES

Process 1

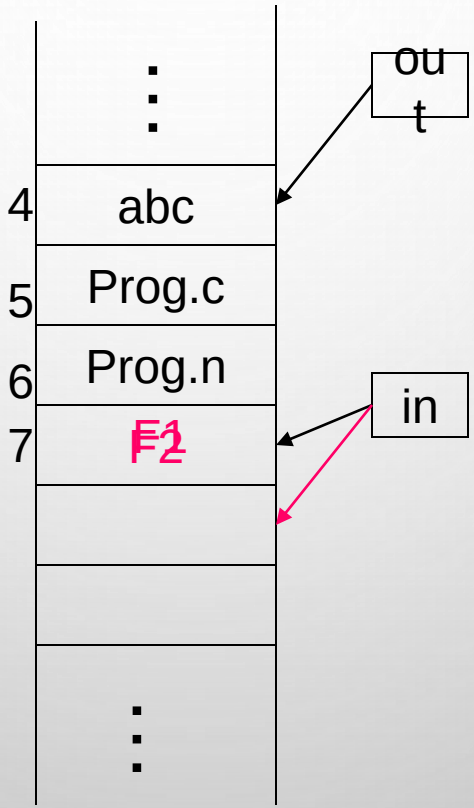
int next_free;

1. next_free = in;

3. Stores F1 into next_free;

4. in=next_free+1

Shared memory



Process 2

int next_free;

2. next_free = in
/* value: 7 */

5. Stores F2 into next_free;

6. in=next_free+1

BETTER CODING?

- In previous code

```
for(;;){
   int next_free = in;
   slot[next_free] = file;
   in = next_free+1;
}
```

- What if we use one line of code?

```
for(;;){
   slot[in++] = file
}
```

WHEN CAN PROCESS BE SWITCHED?

- After each machine instruction!
- `in++` is a C/C++ statement, translated into three machine instructions:
  - load mem, R
  - inc R
  - store R, mem
- Interrupt (and hence process swichting) can happen in between.

RACE CONDITION

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- Very hard to Debug

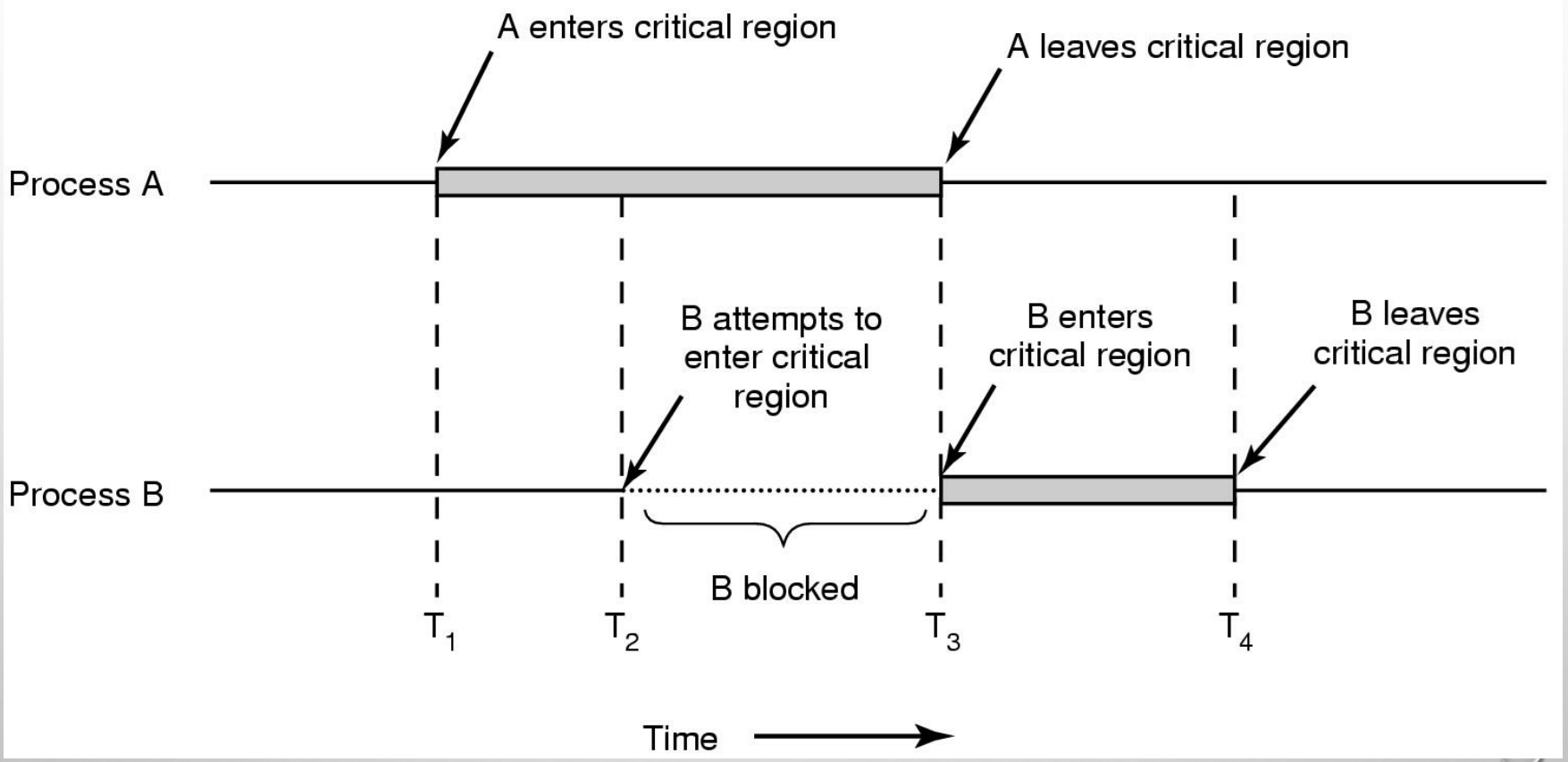
CRITICAL REGION

- That part of the program that do critical things such as accessing shared memory
- Can lead to race condition

SOLUTION REQUIREMENT

1) No two processes simultaneously in critical region
2) No assumptions made about speeds or numbers of CPUs
3) No process running outside its critical region may block another process
4) No process must wait forever to enter its critical region

SOLUTION REQUIREMENT



12

MUTUAL EXCLUSION WITH BUSY WAITING

- Possible Solutions
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's solution
  - TSL

DISABLING INTERRUPTS

- How does it work?
  - Disable all interrupts just after entering a critical section
  - Re-enable them just before leaving it.
- Why does it work?
  - With interrupts disabled, no clock interrupts can occur
  - No switching can occur
- Problems:
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? (disabling interrupts only affects one CPU)
- Only used inside OS

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

LOCK VARIABLES

```
int lock = 0;
while (lock);
lock = 1;
//EnterCriticalSection;
   access shared variable;
//LeaveCriticalSection;
lock = 0;
```

Does the above code work?

STRICT ALTERNATION

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(a) Process 0 **(b) Process 1**

Proposed solution to critical region problem

16

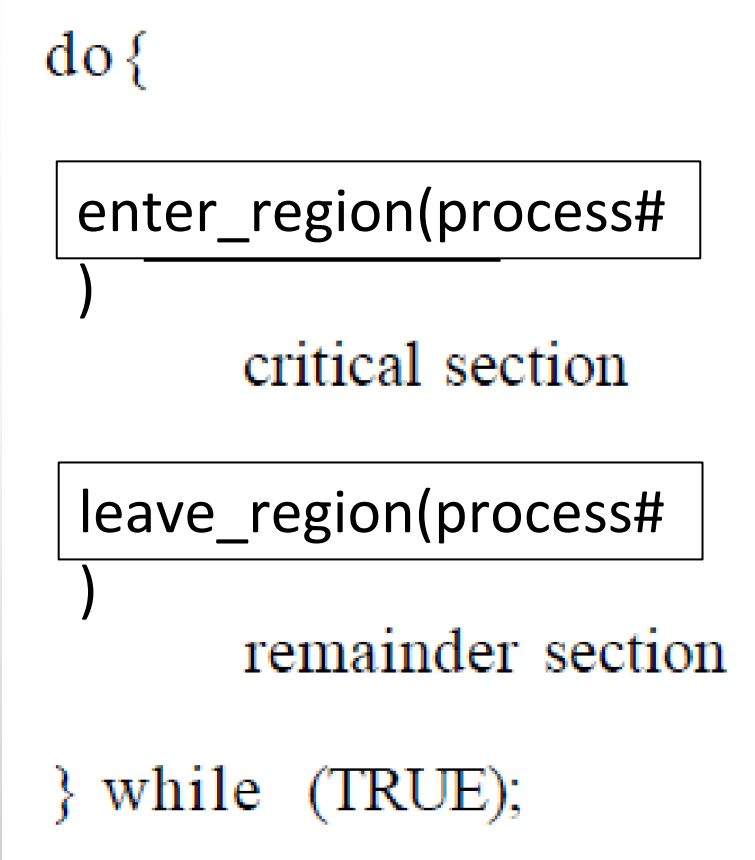
PROBLEM

- Busy waiting: Continuously testing a variable until some value appear
  - Wastes CPU time
- Violates condition 3
  - When one process is much slower than the other

17

PETERSON'S SOLUTION

- Consists of 2 procedures
- Each process has to call
  - `enter_region` with its own process # before entering its C.R.
  - And `Leave_region` after leaving C.R.



18

PETERSON'S SOLUTION (FOR 2 PROCESSES)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```


19

PETERSON'S SOLUTION: ANALYSIS(1)

- Let Process 1 is not interested and Process 0 calls enter_region with 0
- So, turn = 0 and interested[0] = true and Process 0 is in CR
- Now if Process 1 calls enter_region, it will hang there until interested[0] is false. Which only happens when Process 0 calls leave_region i.e. leaves the C.R.

PETERSON'S SOLUTION: ANALYSIS(2)

- Let both processes call enter_region simultaneously
- Say turn = 1. (i.e. Process 1 stores last)
- Process 0 enters critical region: while (turn = = 0 && …) returns false since turn = 1.
- Process 1 loops until process 0 exits: while (turn = = 1 && interested[0] = = true) returns true.
- Done!!

TSL

- Requires hardware support
- TSL instruction: test and set lock
  - Reads content of *lock* into a Register
  - Stores a nonzero value at *lock*.
- CPU executing TSL locks the memory bus prohibiting other CPUs from accessing memory

TSL REGISTER,LOCK

Indivisible/Atomic

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

BUSY WAITING: PROBLEMS

- Waste CPU time since it sits on a tight loop
- May have unexpected effects:
  - **Priority Inversion Problem**

Example:

- 2 Cooperating Processes: H (high priority) and L (low priority)
- *Scheduling rule:* H is run whenever it is ready
- Let L in C. R. and H is ready and wants to enter C.R.
- Since H is ready it is given the CPU and it starts busy waiting
- L will never gets the chance to leave its C.R.
- H loops forever
- http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

23

SLEEP & WAKEUP

- When a process has to wait, change its state to **BLOCKED/WAITING**
- Switched to **READY** state, when it is OK to retry entering the critical section
- Sleep is a system call that causes the caller to block
  - be suspended until another process wakes it up
- Wakeup system call has one parameter, the process to be awakened.

PRODUCER CONSUMER PROBLEM

- Also called bounded-buffer problem
- Two (*m*+*n*) processes share a common buffer
- One (*m*) of them is (are) **producer**(s): put(s) information in the buffer
- One (*n*) of them is (are) **consumer**(s): take(s) information out of the buffer
- Trouble and solution
  - Producer wants to put but buffer full- Go to **sleep** and **wake up** when consumer takes one or more
  - Consumer wants to take but buffer empty- go to sleep and wake up when producer puts one or more

SLEEP AND WAKEUP

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

PRODUCER-CONSUMER PROBLEM

SLEEP AND WAKEUP

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */
```

```
void producer(void)
{
     int item;

     while (TRUE) {
          item = produce_item( );
          if (count == N) sleep( );
          insert_item(item);
          count = count + 1;
          if (count == 1) wakeup(consumer);
     }
}
```

```
void consumer(void)
{
     int item;

     while (TRUE) {
          if (count == 0) sleep( );
          item = remove_item( );
          count = count – 1;
          if (count == N – 1) wakeup(producer
          consume_item(item);
     }
}
```

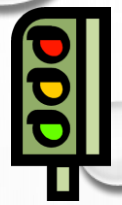

PRODUCER-CONSUMER PROBLEM

SLEEP AND WAKEUP: RACE CONDITION

- **Race condition**
- Unconstrained access to *count*
  - CPU is given to P just after C has read count to be 0 but not yet gone to sleep.
  - P calls wakeup
  - Result is lost wake-up signal
  - Both will sleep forever

28

SEMAPHORES



- A new variable type
- A kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are *up* and *down* – can't read or write value, except to set it initially

SEMAPHORES

- Operation “down”:
  - if value > 0; value-- and then continue.
  - if value = 0; process is put to sleep without completing the down for the moment
    - Checking the value, changing it, and possibly going to sleep, is all done as an ***atomic*** *action*.
- Operation “up”:
  - increments the value of the semaphore addressed.
  - If one or more process were sleeping on that semaphore, one of them is chosen by the system (e.g. at random) and is allowed to complete its *down*
    - The operation of incrementing the semaphore and waking up one process is also indivisible
  - No process ever blocks doing an *up.*

30

SEMAPHORES

- Operations must be atomic
  - Two *down*'s together can't decrement value below zero
  - Similarly, process going to sleep in *down* won't miss wakeup from *up* – even if they both happen at same time

31

SEMAPHORES

- **Counting semaphore**.
  - The value can range over an unrestricted domain
- **Binary semaphore**
  - The value can range only between 0 and 1.
  - On some systems, binary semaphores are known as mutex locks as they provide mutual exclusion

SEMAPHORES USAGE

1. Mutual exclusion
2. Controlling access to limited resource
3. Synchronization

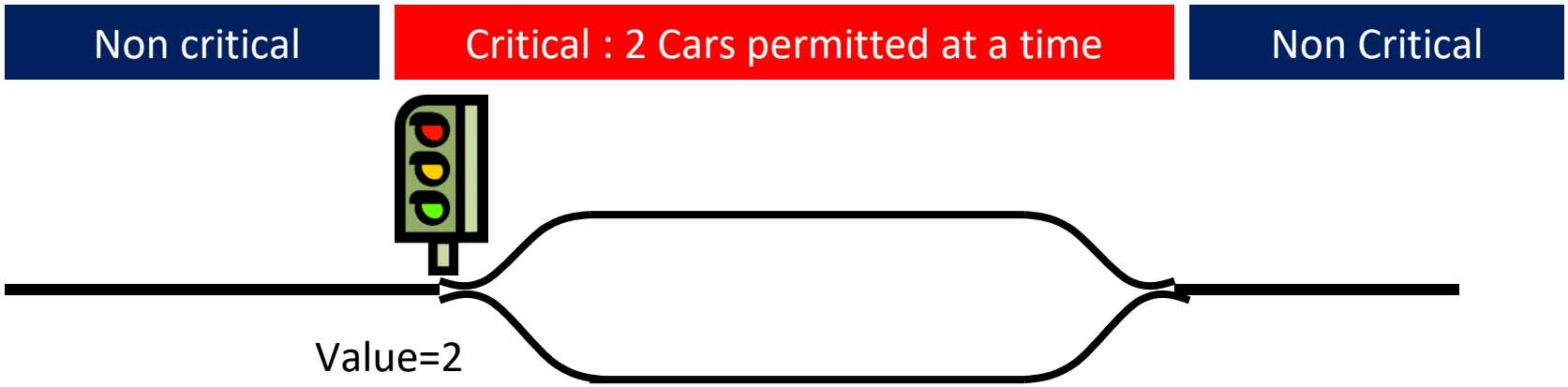
Mutual exclusion

- How to ensure that only one process can enter its C.R.?
- Binary semaphore initialized to 1
- Shared by all collaborating processes
- If each process does a *down* just before entering CR and an *up* just after leaving then mutual exclusion is guarrented

```
do {
        down(mutex);
        // critical section
        up(mutex);
        // remainder section
} while (TRUE);
```

Controlling access to a resource

- What if we want maximum **m** process/thread can use a resource simultaneously ?
- Counting semaphore initialized to the number of available resources
- Semaphore from railway analogy
  - Here is a semaphore **initialized to 2** for resource control:



Synchronization

- How to resolve dependency among processes
- Binary semaphore initialized to 0
- consider 2 concurrently running processes:
  - P1 with a statement S1 and
  - P2 with a statement S2.
  - Suppose we require that S2 be executed only after S1 has completed.

P1

```
S1;
up(synch);
```

P2

```
down(synch);
S2;
```

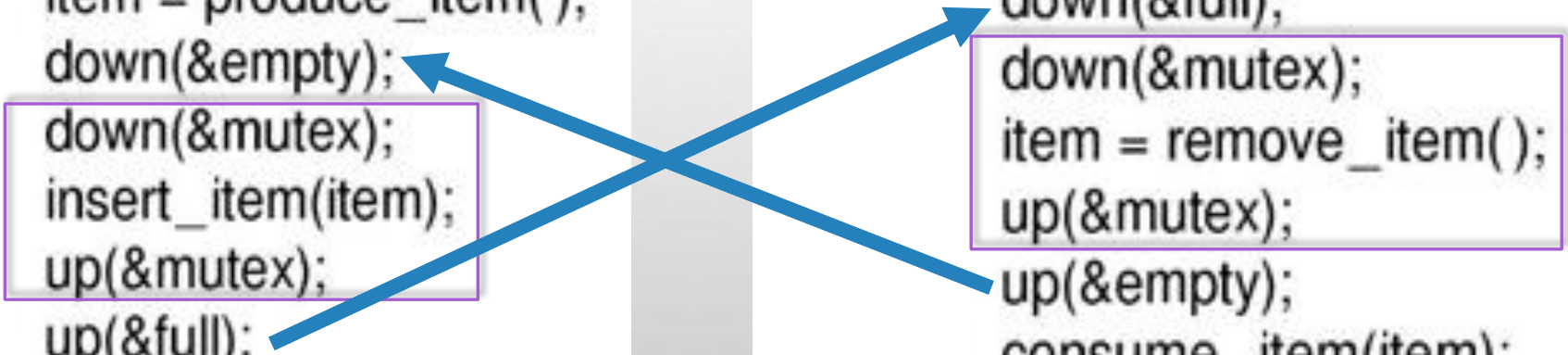
PRODUCER & CONSUMER

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */
```

```
void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```



SEMAPHORES IN PRODUCER CONSUMER PROBLEM: ANALYSIS

- 3 semaphores are used
  - *full* (initially 0) for counting occupied slots
  - *Empty* (initially *N*) for counting empty slots
  - *mutex* (initially 1) to make sure that Producer and Consumer do not access the buffer at the same time
- Here 2 uses of semaphores
  - Mutual exclusion (mutex)
  - Synchronization (full and empty)
    - To guarantee that certain event sequences do or do not occur

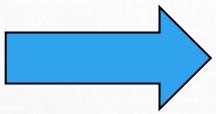
Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

38

SEMAPHORES: "BE CAREFUL"

- Suppose the following is done in Producer's code

```
…
down(&empty)
down(&mutex)
…
```



```
…
down(&mutex)
down(&empty)
…
```

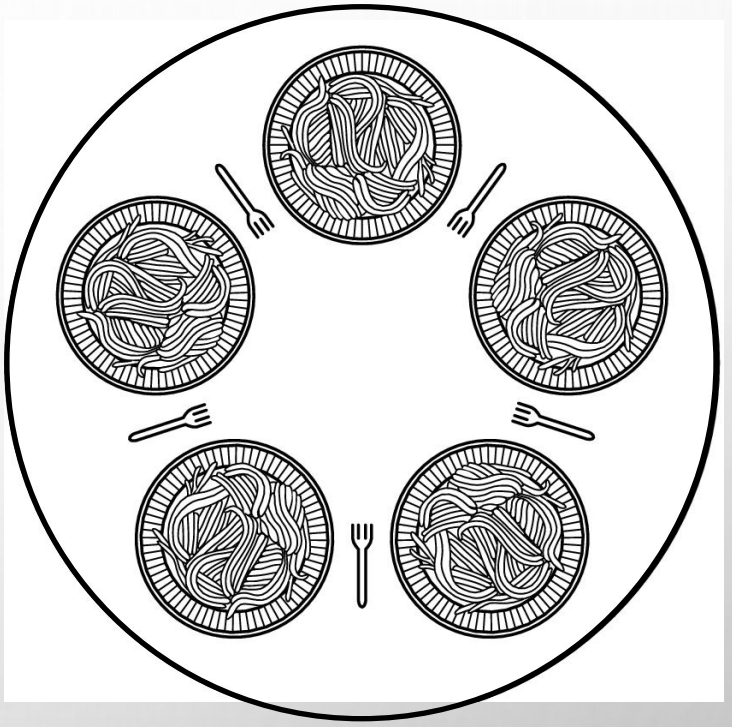
Just the order is reversed

- If buffer full Producer would block due to down(&empty) with mutex = 0.
- So now if Consumer tries to access the buffer, it would block too due to its down(&mutex).
- Both processes would stay blocked forever: DEADLOCK

39

DINING PHILOSOPHERS

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors
- When get hungry try to pick up 2 chopsticks (one at a time in either order) to eat
- Need both to eat, then release both when done
- How to program the scenario avoiding all concurrency problems?



40

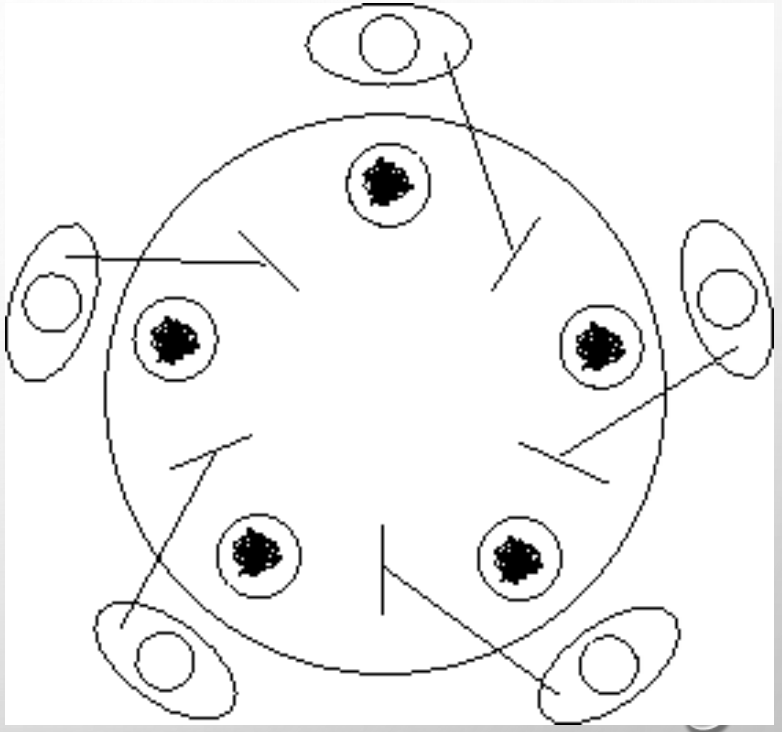
DINING PHILOSOPHERS: A SOLUTION

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```


41

DINING PHILOSOPHERS: PROBLEMS WITH PREVIOUS SOLUTION

- Deadlock may happen
- Does this solution prevents any such thing from happening ?
  - Everyone takes the left fork simultaneously



DINING PHILOSOPHERS: PROBLEMS WITH PREVIOUS SOLUTION

Tentative Solution:

- After taking left fork, check whether right fork is available.
- If not, then return left one, wait for some time and repeat again.

Problem:

- All of them start and do the algorithm synchronously and simultaneously:
- STARVATION (A situation in which all the programs run indefinitely but fail to make any progress)
- Solution: Random wait; but what if the most unlikely of same random number happens?

43

ANOTHER ATTEMPT, SUCCESSFUL!

```
void philosopher(int i)
{
   while (true)
   {
       think();
       down(&mutex);
       take_fork(i);
       take_fork((i+1)%N);
       eat();
       put_fork(i);
       put_fork((i+1)%N);
       up(&mutex);
   }
}
```

- Theoretically solution is OK- no deadlock, no starvation.
- Practically with a performance bug:
  - Only one philosopher can be eating at any instant: absence of parallelism

FINAL SOLUTION PART 1

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```


45

FINAL SOLUTION PART 2

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```


46

THE READERS AND WRITERS PROBLEM

- Dining Philosopher Problem: Models processes that are competing for exclusive access to a limited resource
- Readers Writers Problem: Models access to a database
- Example: An airline reservation system- many competing process wishing to read and write-
  - Multiple readers simultaneously- acceptable
  - Multiple writers simultaneously- not acceptable
  - Reading, while write is writing- not acceptable

47

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                    /* repeat forever */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc + 1;                /* one reader more now */
            if (rc == 1) down(&db);     /* if this is the first reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            read_data_base( );          /* access the data */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc – 1;                /* one reader fewer now */
            if (rc == 0) up(&db);       /* if this is the last reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            use_data_read( );           /* noncritical region */
      }
}


void writer(void)
{
      while (TRUE) {                    /* repeat forever */
            think_up_data( );           /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base( );         /* update the data */
            up(&db);                    /* release exclusive access */
      }
}
```


48

ISSUE REGARDING THE SOLUTION

- Inherent priority to the readers
- Say a new reader arrives every 2 seconds and each reader takes 5 seconds to do its work. What will happen to a writer?
- Issue regarding second variation
  - Writer don't have to wait for readers that came along after it
  - Less concurrency, lower performance

49

THANKS FOR YOUR PATIENCE

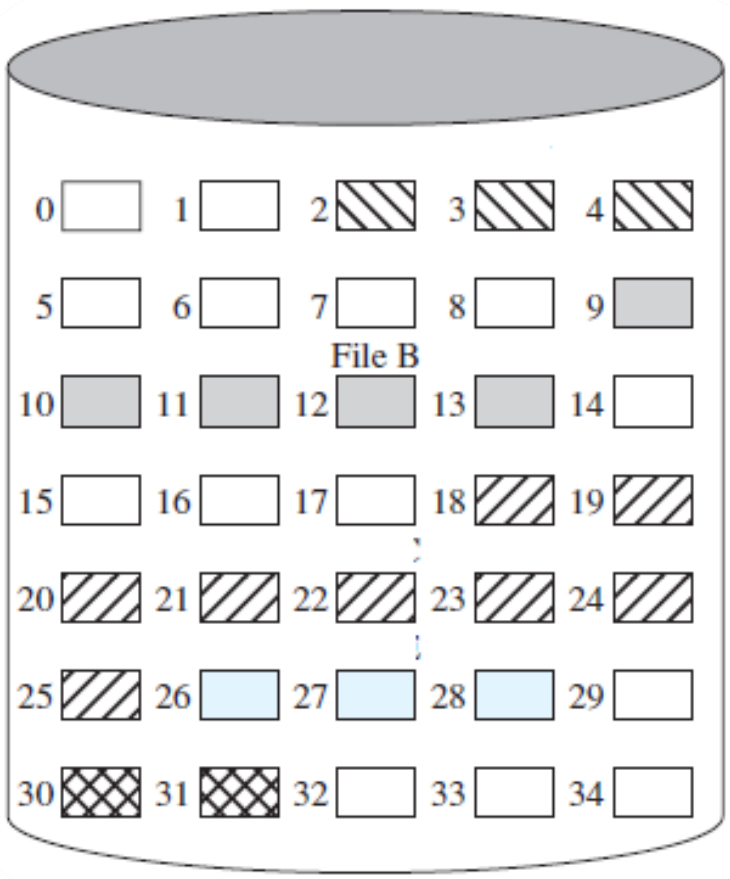
FILE SYSTEM

LONG TERM STORAGE MANAGEMENT

- We often need to store
  - large amount of information
  - permanently
- Usually we use Hard disk & newly solid-state drives for such long term storage

LONG TERM STORAGE MANAGEMENT

- We can view this disks as a
  - linear sequence of fixed-size blocks
  - supports two operations:
    - Read K^{th} block.
    - Write K^{th} block
- Now we have to answer
  - How to find desired information?
  - How to know which blocks are free?
  - …
- Is it user friendly??



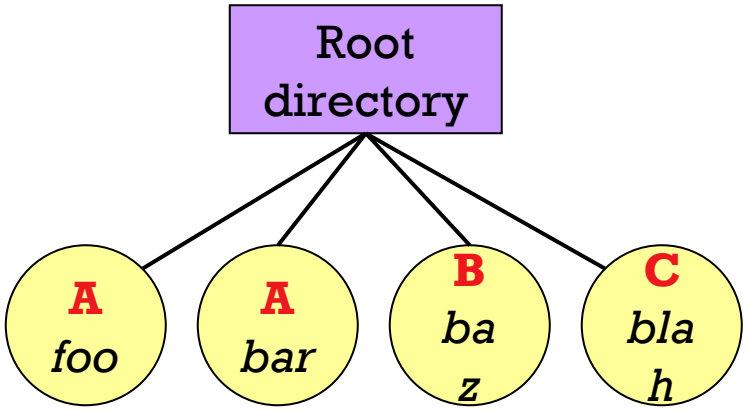
FILE SYSTEM

- file system consists of two distinct parts:
  - A collection *of files*
    - each storing related data
  - A Directory structure
    - organize and provide information about all the files in the system

DIRECTORIES/FOLDERS

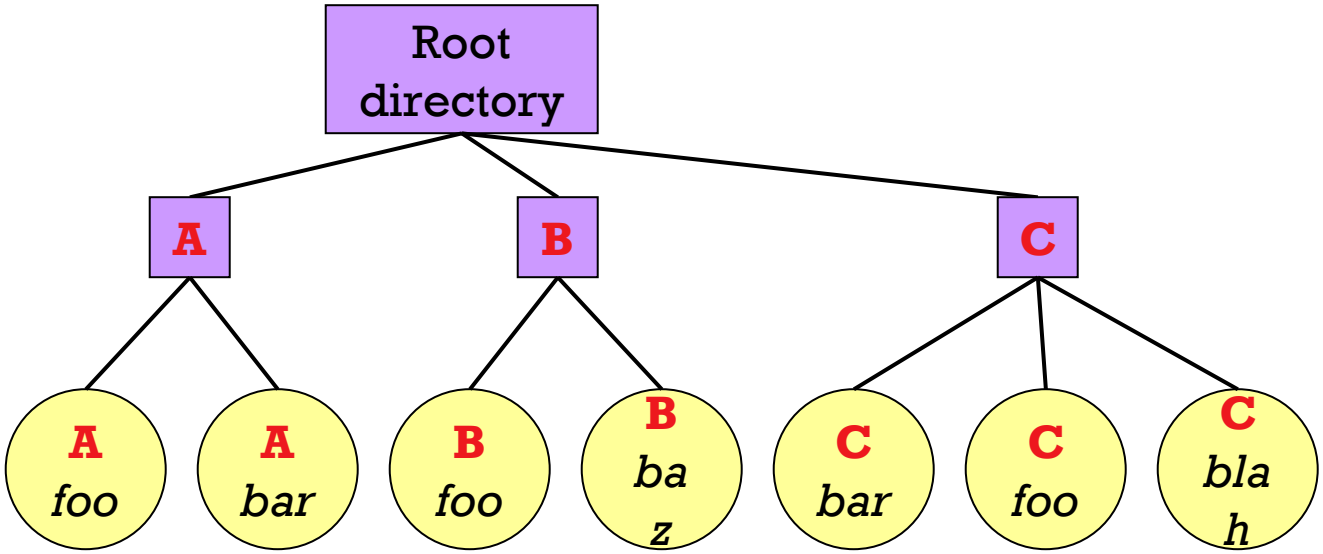
- Naming is nice, but limited
- Humans like to group things together for convenience
- File systems allow this to be done with *directories* (sometimes called *folders*)
- Grouping makes it easier to
  - Find files in the first place: remember the enclosing directories for the file
  - Locate related files (or just determine which files are related)

SINGLE-LEVEL DIRECTORY SYSTEMS



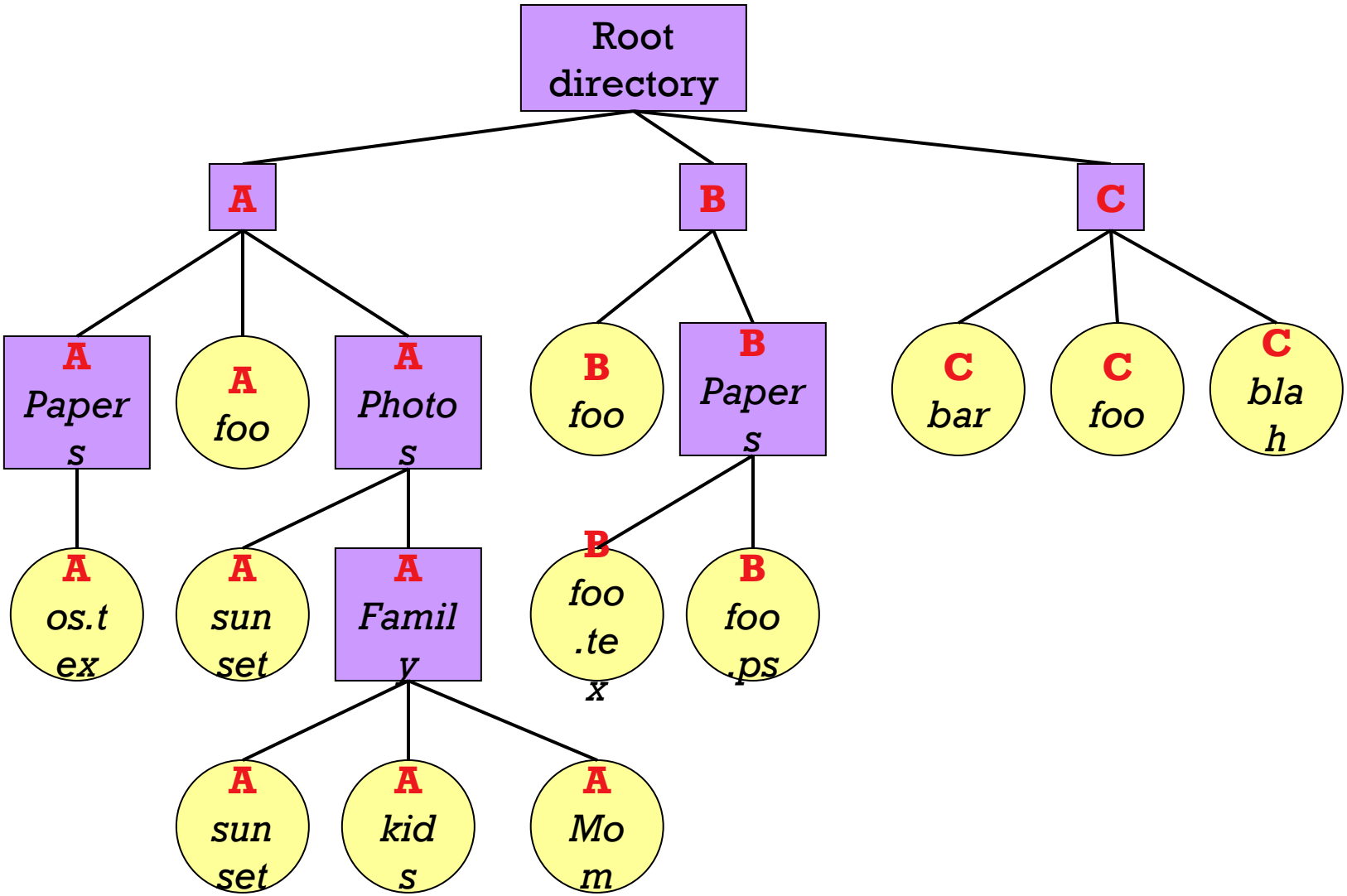
- One directory in the file system
- Example directory
  - Contains 4 files (*foo*, *bar*, *baz*, *blah*)
  - owned by 3 different people: A, B, and C (owners shown in red)
- Problem: what if user B wants to create a file called *foo*?

TWO-LEVEL DIRECTORY SYSTEM

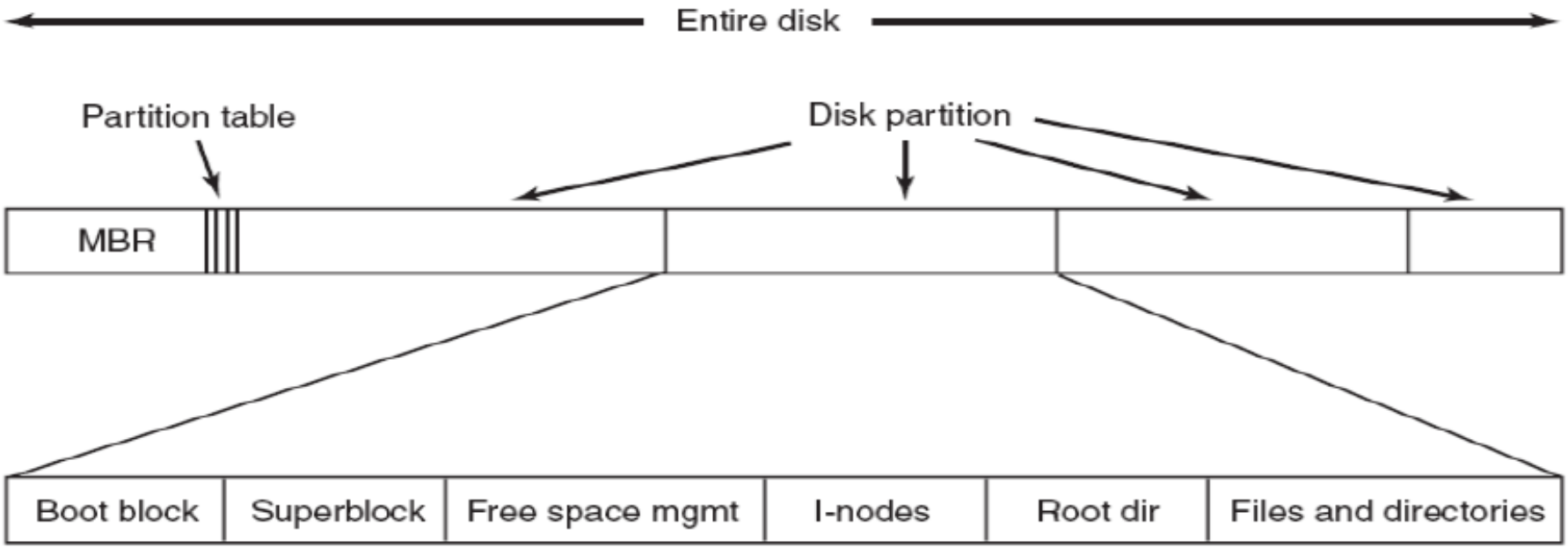


- Solves naming problem: each user has her own directory
- Multiple users can use the same file name
- By default, users access files in their own directories
- Extension: allow users to access files in others' directories

HIERARCHICAL DIRECTORY SYSTEM



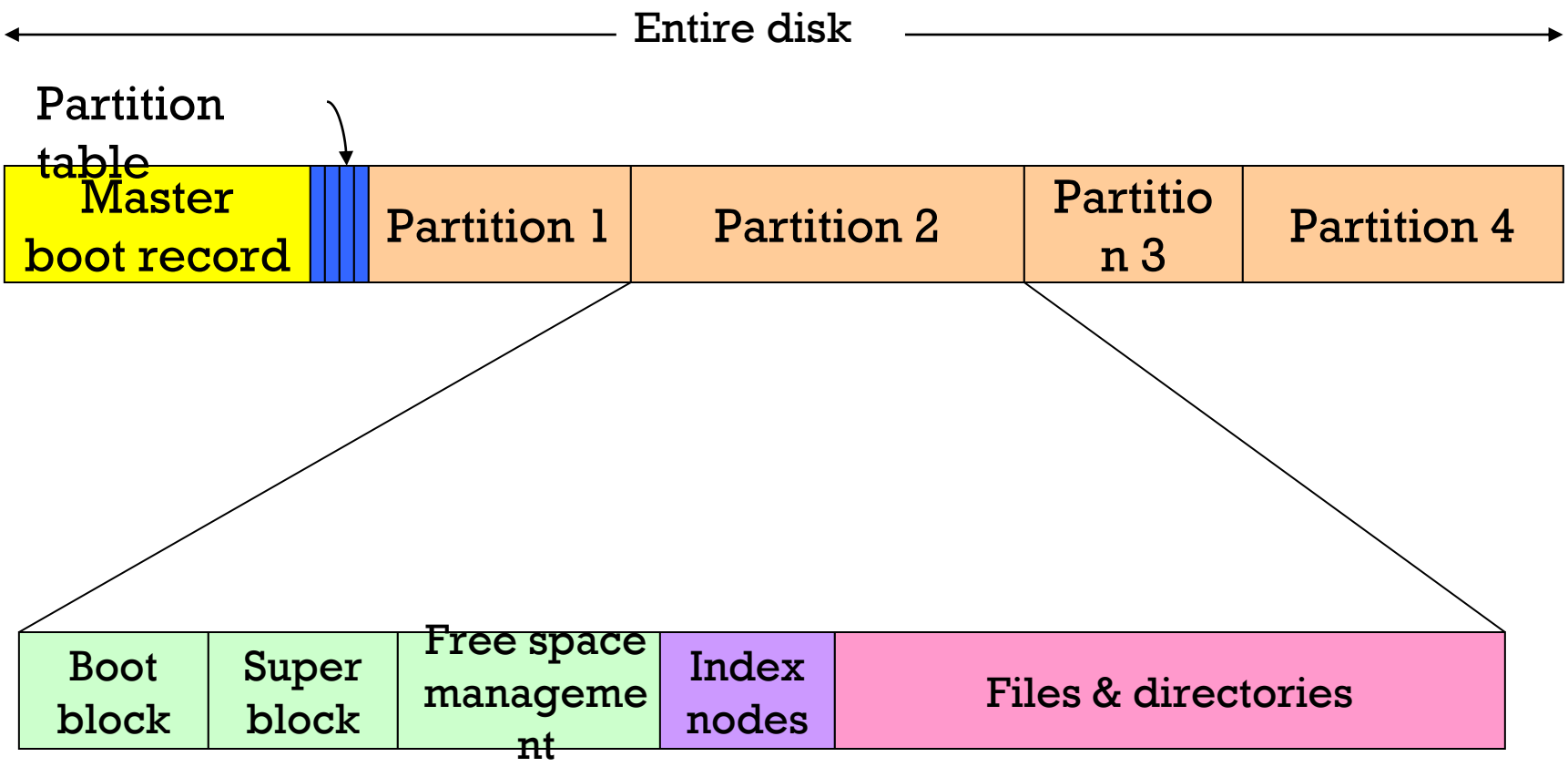
FILE SYSTEM LAYOUT



FILE SYSTEM LAYOUT

- File systems are stored on hard disk
- Hard disk can be divided up into one or more partitions, with independent file systems on each partition
- Sector 0 of the disk is called the MBR(Master Boot Record) and is used to boot the computer
- The end of the MBR contains the partition table

FILE SYSTEM LAYOUT



FILE SYSTEM LAYOUT

- Partition table gives the starting and ending addresses of each partition
- One of the partitions in the table is marked as active
- When the computer is booted, the BIOS reads in and executes the MBR
- The first thing the MBR program does is locate the active partition, read in its first block, called the boot block , and execute it.
- The program in the boot block loads the operating system contained in that partition.

FILE SYSTEM LAYOUT

- For uniformity, every partition starts with a boot block,
  - even if it does not contain a bootable operating system.
  - Besides, it might contain one in the future
- Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system
- Often the file system contains a superblock
- It contains all the key parameters about the file system
  - read into memory when the computer is booted or the file system is first touched.

FILE SYSTEM LAYOUT

- Typical information in the superblock includes
  - ✔ a magic number to identify the file-system type
  - ✔ the number of blocks in the file system
  - ✔ other key administrative information.

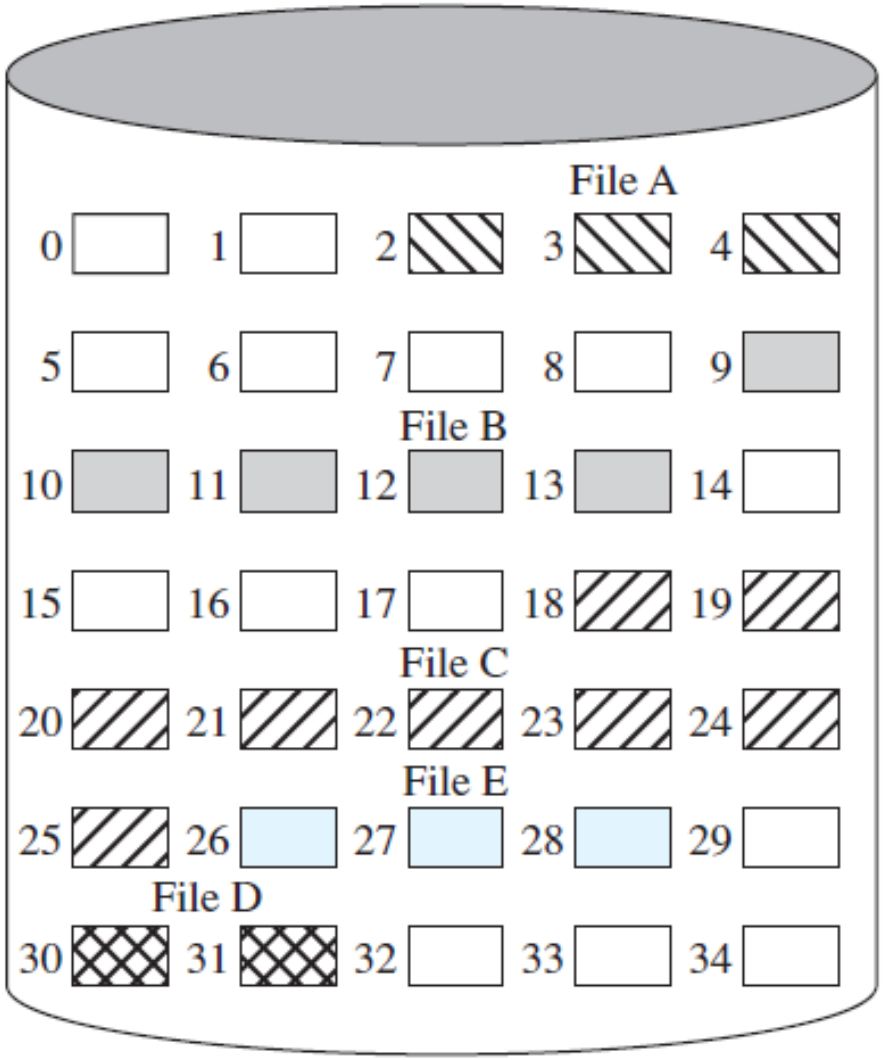
I-NODES

- The structure that describes
  - where the file is on the disk and
  - the attributes of the file
- Associated with each file
- I-nodes have to be stored on disks

DISK BLOCK ALLOCATION

- Keeping track of which disk blocks go with which file
- The most important issue in implementing file system
- Several options
  - Contiguous Allocation
  - Linked list allocation
  - Linked list allocation using a table in memory
  - I-nodes

CONTIGUOUS ALLOCATION



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

CONTIGUOUS ALLOCATION

- OS maintains an ordered list of free disk blocks
- OS allocates a contiguous chunk of free blocks when it creates a file.
- Need to store only the start location and size in the file descriptor

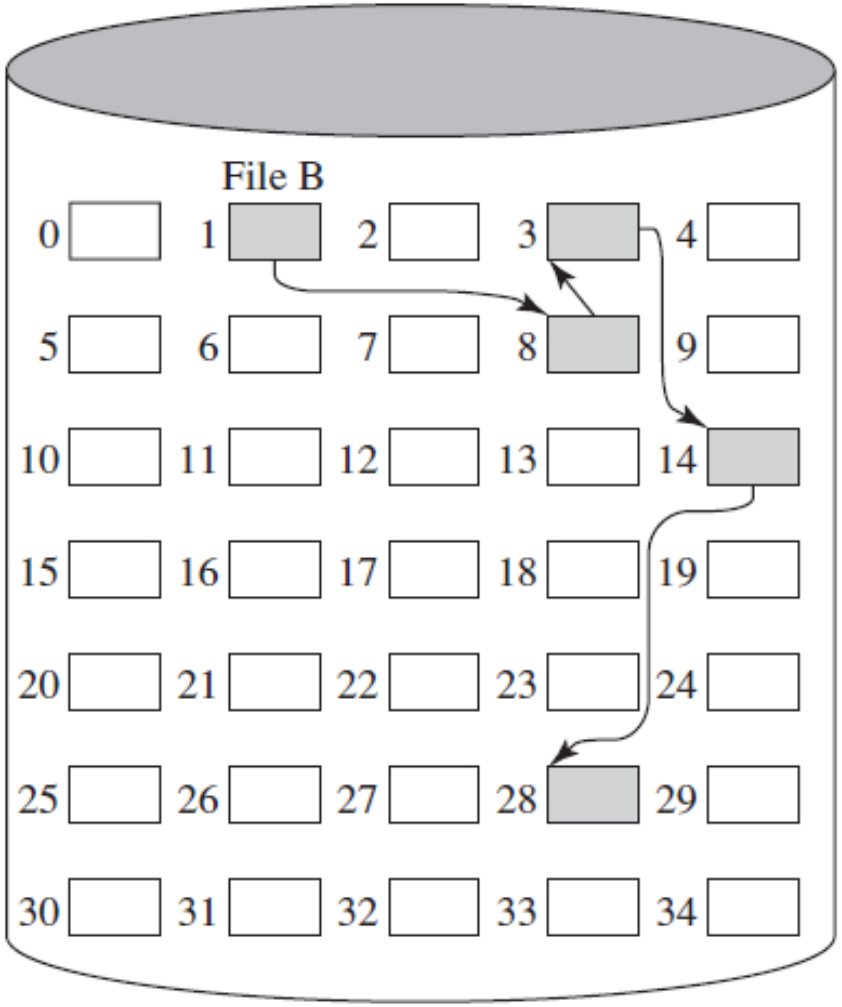
CONTIGUOUS ALLOCATION ADVANTAGE

- Simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file.
- the read performance is excellent because the entire file can be read from the disk in a single operation.
- Only one seek is needed (to the first block).
- After that, no more seeks or rotational delays are needed, so data come in at the full bandwidth of the disk.
- Thus contiguous allocation is simple to implement and has high performance.
- Usage: CD-ROM, DVD-ROM

CONTIGUOUS ALLOCATION DISADVANTAGE

- Each file begins at the start of a new block, so that if file A was really 3½ blocks, some space is wasted at the end of the last block.
- over the course of time, the disk becomes fragmented.
- Leads to unusable data hole.

LINKED LIST ALLOCATION



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

LINKED LIST ALLOCATION

- Keep a list of all the free blocks.
- In the file descriptor, keep a pointer to the first block.
- In each block, keep a pointer to the next block

LINKED LIST ALLOCATION ADVANTAGE

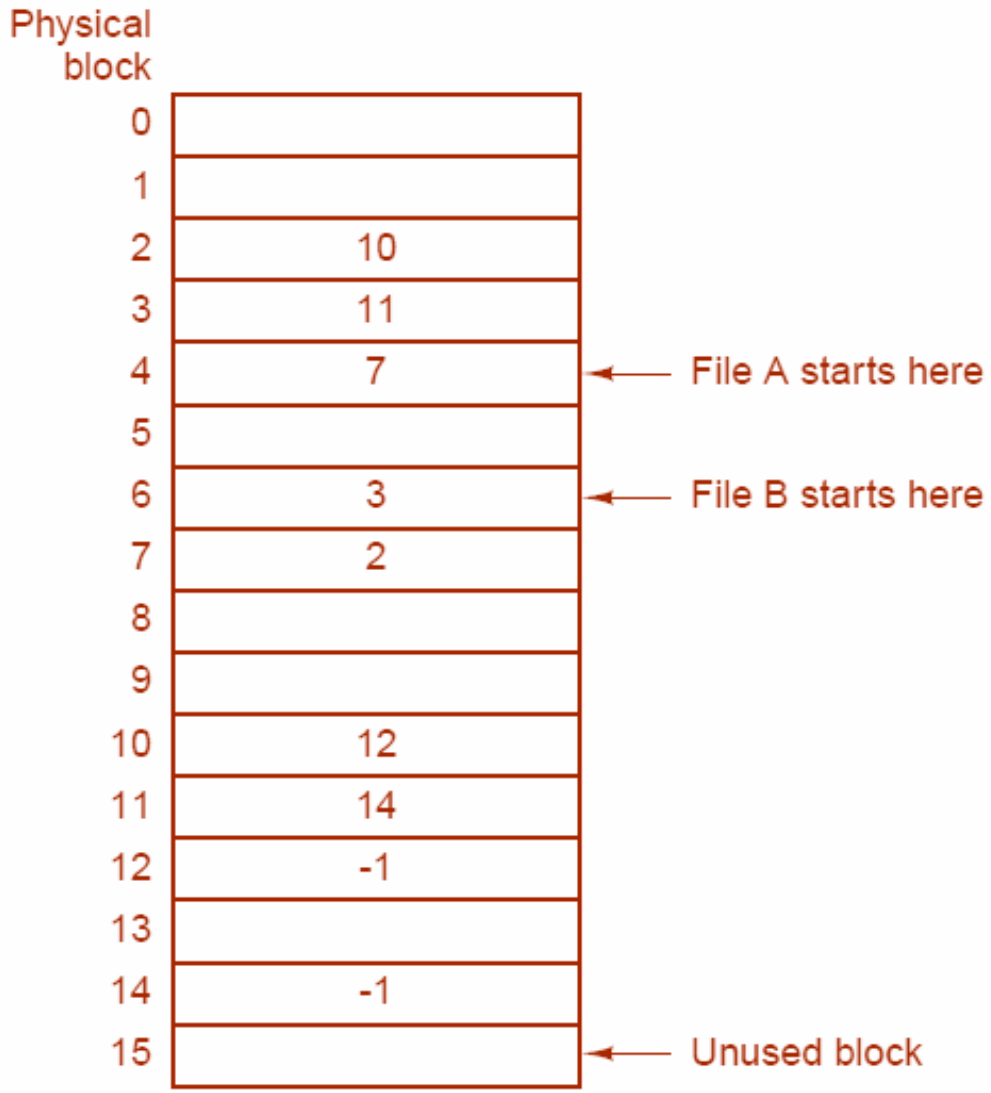
- every disk block can be used in this method.
- No space is lost to disk fragmentation (except for internal fragmentation in the last block).
- It is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

LINKED LIST ALLOCATION DISADVANTAGE

- although reading a file sequentially is straightforward, random access is extremely slow.
- The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes.
- While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two.
- With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

LINKED LIST ALLOCATION USING A TABLE IN MEMORY

- Taking the pointer out of each disk block, and putting it into a table in memory
- Fast random access (chain is in RAM)



LINKED LIST ALLOCATION USING A TABLE IN MEMORY

ADVANTAGE

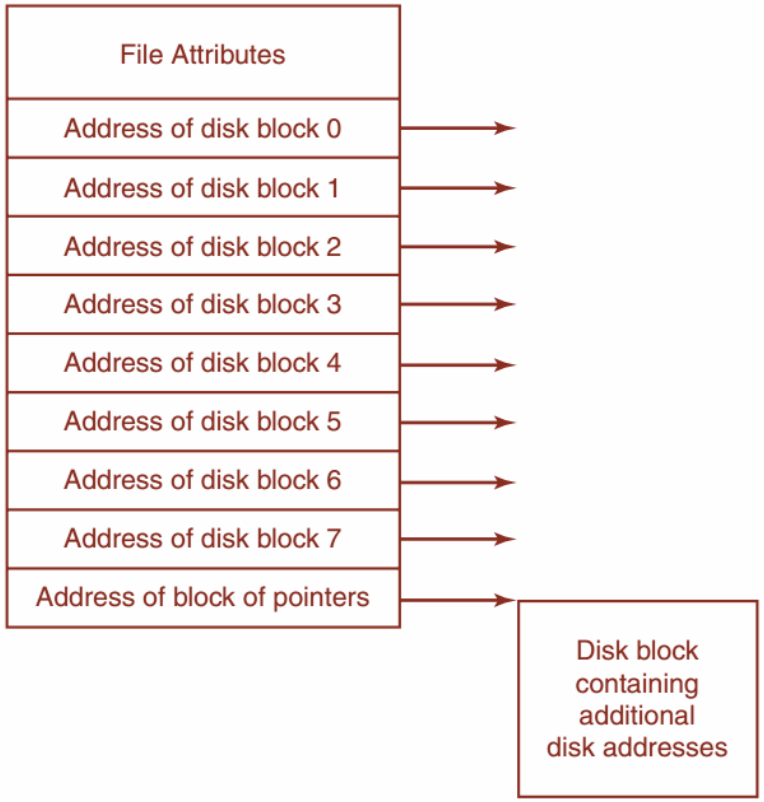
- Using this organization, the entire block is available for data.
- Furthermore, random access is much easier.
- Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory,
- so it can be followed without making any disk references.
- Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is

LINKED LIST ALLOCATION USING A TABLE IN MEMORY DISADVANTAGE

- The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.
- With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks.
- Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes.
- Thus the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical.
- Clearly the FAT idea does not scale well to large disks. It was the original MS-DOS file system and is still fully supported by all versions of Windows though.

I-NODES

- A data structure associated with each file
  - keeps track of which blocks belong to which file
  - used in UNIX file system
- One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit?
- One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk-block addresses

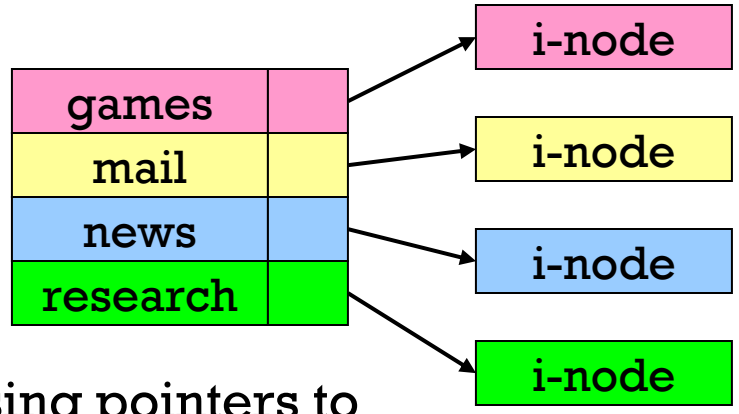


WHAT'S IN A DIRECTORY?

- Two types of information
  - File names
  - File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - Store all information in directory
    - Fixed size entries, one per file
    - Disk addresses and attributes in directory entry
  - Store names & pointers to index nodes (i-nodes)

games	attributes
mail	attributes
news	attributes
research	attributes

Storing all information in the directory



Using pointers to index nodes

FIXED LENGTHY FILE NAMING

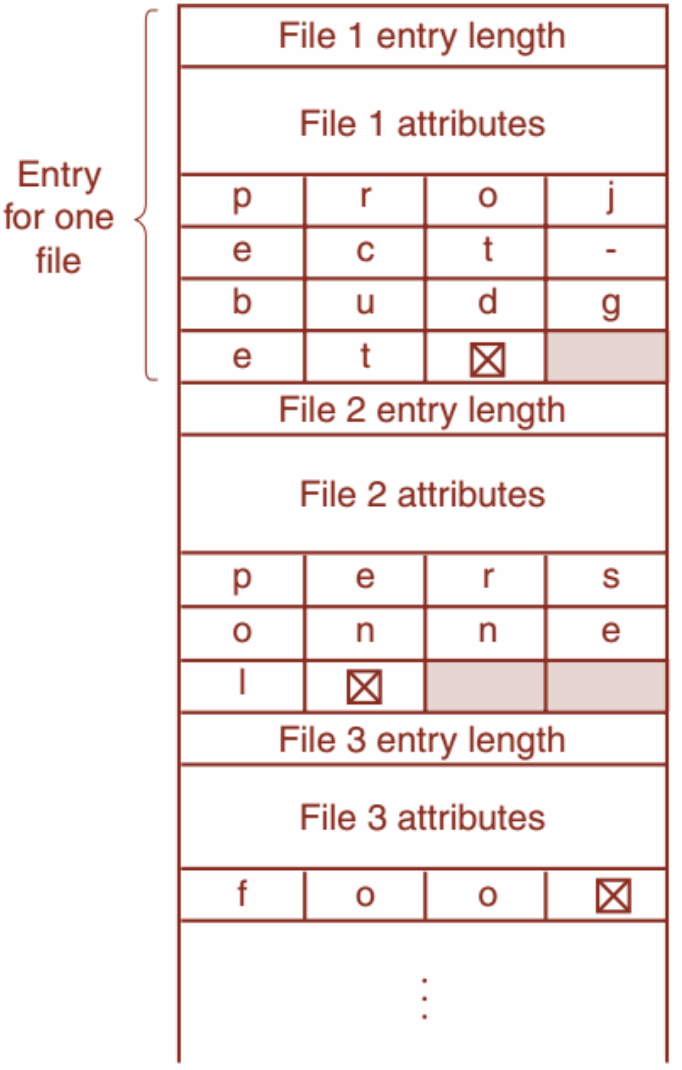
- The simplest approach is to set a limit on file-name length, typically 255 characters,

- Advantages
  - Simple, easy to implement

- Disadvantages
  - wastes a great deal of directory space, since few files have such long names.

IN-LINE FILE NAMING

- all directory entries are of the same size.
- Each directory entry contains a fixed portion containing
  - ✔ The length of the entry
  - ✔ File attributes
  - ✔ Actual File Name
- Each file name is terminated by a special character (usually 0)
- To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words

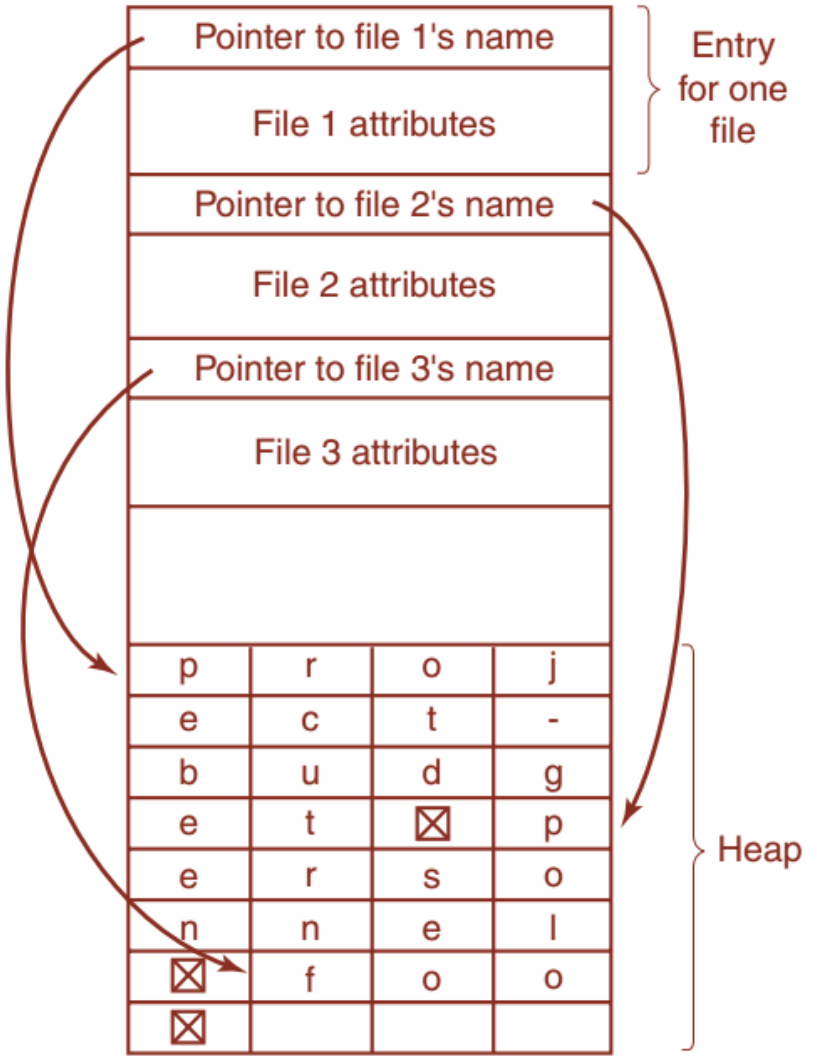


IN-LINE FILE NAMING

- Disadvantages
  - When a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit.
  - A single directory entry may span multiple pages, so a page fault may occur while reading a file name

IN-HEAP FILE NAMING

- Another way to handle variable-length names is to make the directory entries themselves all fixed length
- keep the file names together in a heap at the end of the directory

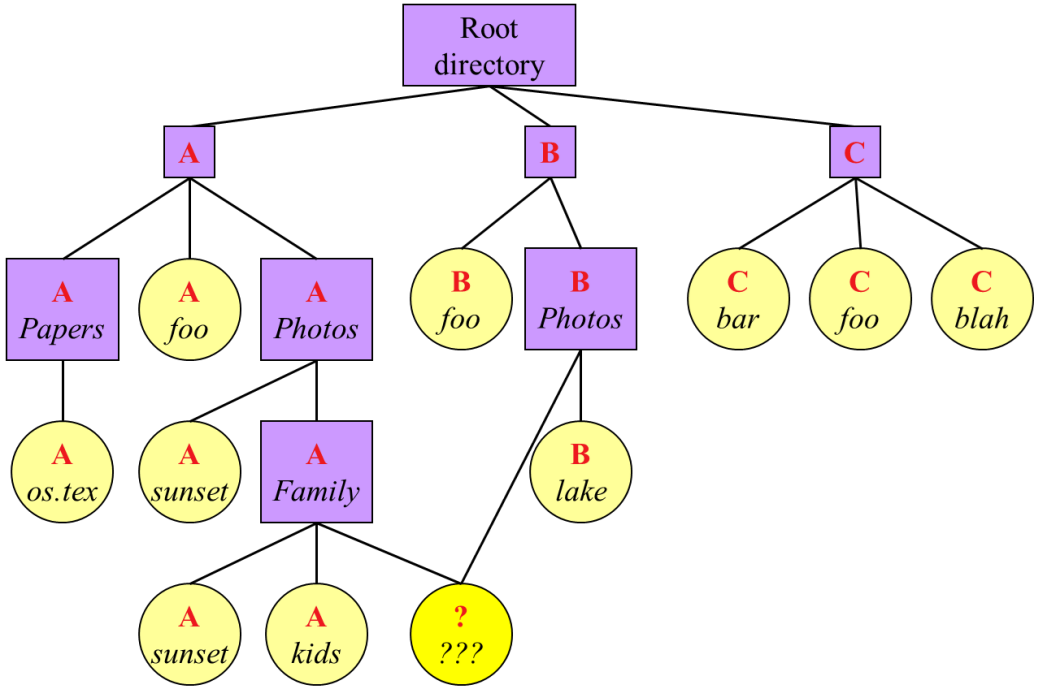


IN-HEAP FILE NAMING

- Advantages
  - Solves the fragmentation problem
- Disadvantages
  - Heap management
  - Page fault still may occur

SHARED FILES

- When several users are working together on a project, they often need to share files.
- As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users.



SHARED FILE PROBLEMS

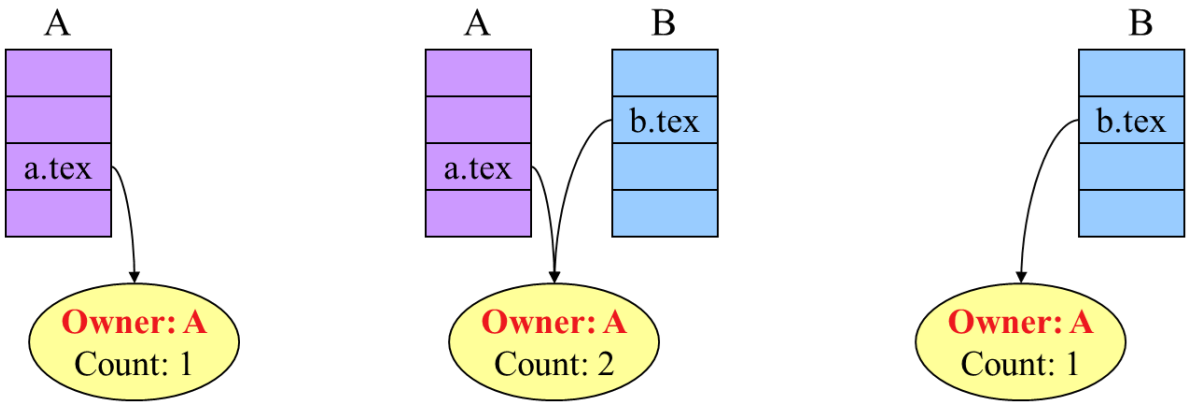
- If directories really do contain disk addresses, then a copy of the disk addresses will have to be made in B's directory when the file is linked.
- If either A or B subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append.
- The changes will not be visible to the other user
- **Thus defeating the purpose of sharing.**

SHARED FILES PROBLEMS SOLUTION 1

- Disk blocks are not listed in directories, but in a i-node.
- The directories would then point just to the relevant i-nodes.

SHARED FILES PROBLEMS SOLUTION 1

- Drawbacks
  - B links to the shared file, the i-node records the file's owner as A.
  - Creating a link does not change the ownership
  - But it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.



SHARED FILES PROBLEMS SOLUTION 1

- Drawbacks
  - If A later removes the file, the only thing to do is remove A's directory entry, but leave the i-node intact, with count set to 1,
  - We now have a situation in which B is the only user having a directory entry for a file owned by A.
  - **If the system does accounting or has quotas**, A will continue to be billed for the file until B decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

SHARED FILES PROBLEMS SOLUTION 2

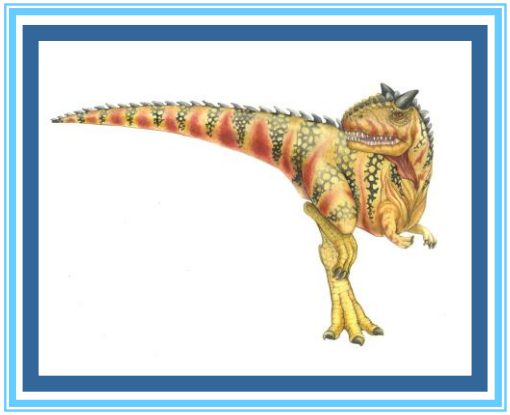
- B links to one of A's files by having the system create a new file, of type LINK, and entering that file in B's directory.
- The new file contains just the path name of the file to which it is linked.
- When B reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file.
- This approach is called symbolic linking, to contrast it with traditional (hard) linking.

SHARED FILES PROBLEMS SOLUTION 2

- Advantages
  - With symbolic links previously described problem does not arise because only the true owner has a pointer to the i-node.
  - Users who have linked to the file just have path names, not i-node pointers.
  - When the owner removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file.
  - Removing a symbolic link does not affect the file at all.
  - They can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

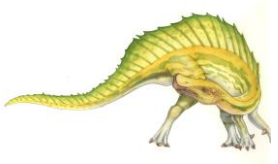
THANK YOU

Chapter 14: Protection



Chapter 14: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection



Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

Goals of Protection

- In one protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so

Principles of Protection

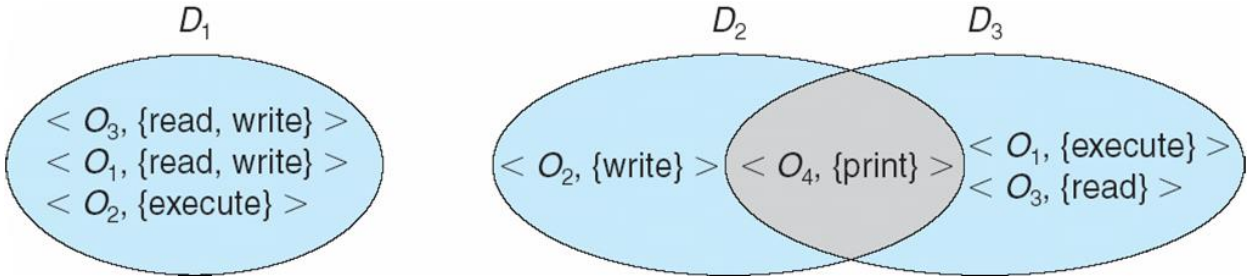
- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
  - Limits damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – **domain switching**, **privilege escalation**
  - "Need to know" a similar concept regarding access to data

Principles of Protection (Cont.)

- Must consider “grain” aspect
  - Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have abilities of the associated user, or of root
  - Fine-grained management more complex, more overhead, but more protective
    - File ACL lists, RBAC
- Domain can be user, process, procedure

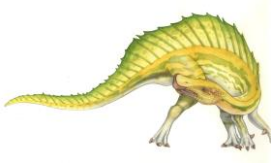
Domain Structure

- Access-right = <*object-name*, *rights-set*>
  where *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights



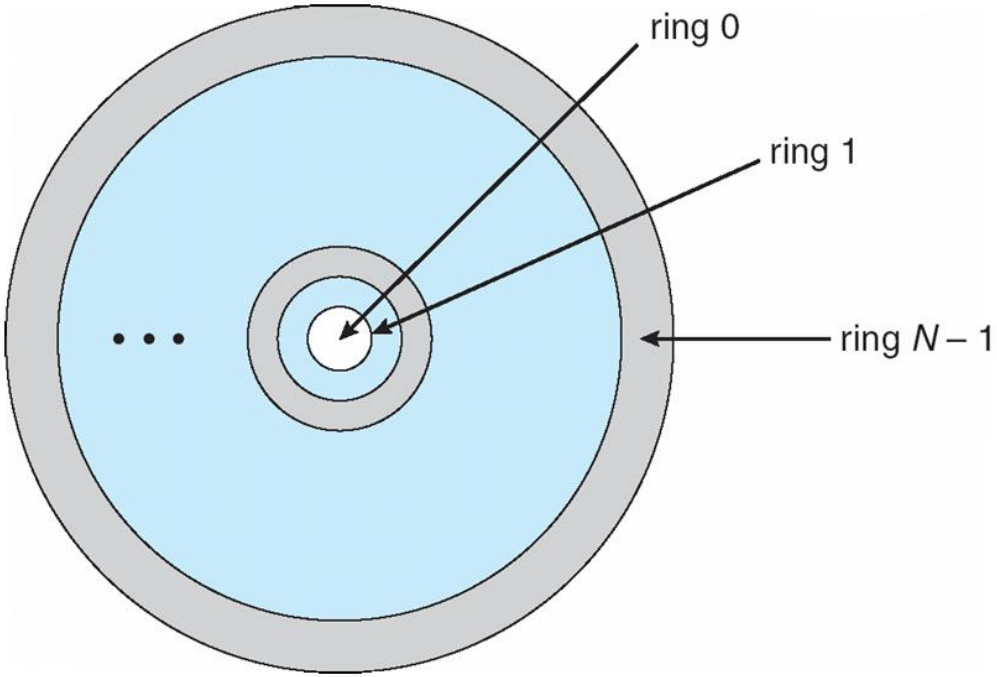
Domain Implementation (UNIX)

- Domain = user-id
- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (setuid bit)
  - When file is executed and setuid = on, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset
- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user's domain when other domain's password provided
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)



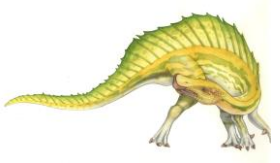
Domain Implementation (MULTICS)

- Let D_i and D_j be any two domain rings
- If $j < I \Rightarrow D_i \subseteq D_j$



Multics Benefits and Limits

- Ring / hierarchical structure provided more than the basic kernel / user or root / normal user design
- Fairly complex -> more overhead
- But does not allow strict need-to-know
  - Object accessible in D_j but not in D_i, then ***j*** must be < ***i***
  - But then every segment accessible in D_i also accessible in D_j



Access Matrix

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **`Access(i, j)`** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i
    - *copy op from* O_i *to* O_j *(denoted by "*")*
    - *control –* D_i *can modify* D_j *access rights*
    - *transfer – switch from domain* D_i *to* D_j
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object

Use of Access Matrix (Cont.)

- **Access matrix** design separates mechanism from policy
  - Mechanism
    - Operating system provides access-matrix + rules
    - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - Policy
    - User dictates policy
    - Who can access what object and in what mode
- But doesn't solve the general confinement problem

Access Matrix of Figure A with Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access Matrix with *Copy* Rights

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Access Matrix With *Owner* Rights

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

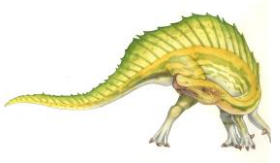
(b)

Modified Access Matrix of Figure B

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Implementation of Access Matrix

- Generally, a sparse matrix
- Option 1 – Global table
  - Store ordered triples **<domain, object, rights-set>** in table
  - A requested operation M on object O_j within domain D_i -> search table for < D_i, O_j, R_k >
    - with M ∈ R_k
  - But table could be large -> won't fit in main memory
  - Difficult to group objects (consider an object that all domains can read)



Implementation of Access Matrix (Cont.)

- Option 2 – Access lists for objects
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs **<domain, rights-set>** defining all domains with non-empty set of access rights for the object
  - Easily extended to contain default set -> If M ∈ default set, also allow access

Implementation of Access Matrix (Cont.)

- Each column = Access-control list for one object
  Defines who can perform what operation

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

- Each Row = Capability List (like a key)
  For each domain, what operations allowed on what objects

  Object F1 – Read

  Object F4 – Read, Write, Execute

  Object F5 – Read, Write, Delete, Copy

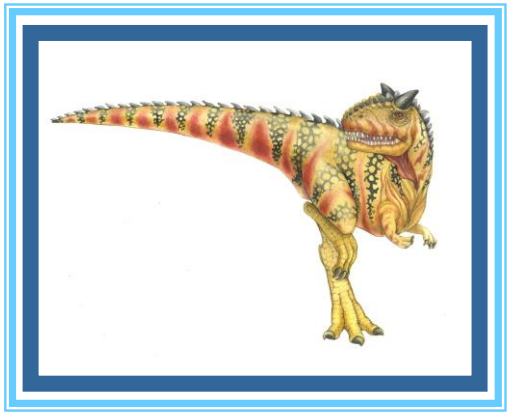
Implementation of Access Matrix (Cont.)

- Option 3 – Capability list for domains
  - Instead of object-based, list is domain based
  - **Capability list** for domain is list of objects together with operations allows on them
  - Object represented by its name or address, called a **capability**
  - Execute operation M on object O_j, process requests operation and specifies capability as parameter
    - Possession of capability means access is allowed
  - Capability list associated with domain but never directly accessible by domain
    - Rather, protected object, maintained by OS and accessed indirectly
    - Like a "secure pointer"
    - Idea can be extended up to applications

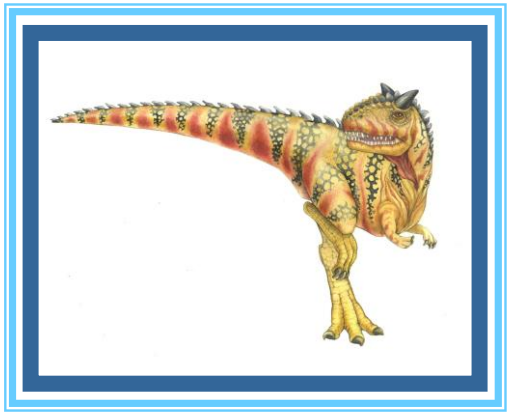
Implementation of Access Matrix (Cont.)

- Option 4 – Lock-key
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, called **locks**
  - Each domain as list of unique bit patterns called **keys**
  - Process in a domain can only access object if domain has key that matches one of the locks

End of Chapter 14



Chapter 15: Security



Chapter 15: Security

- The Security Problem
- Program Threats
- System and Network Threats
- Cryptography as a Security Tool
- User Authentication
- Implementing Security Defenses
- Firewalling to Protect Systems and Networks
- Computer-Security Classifications
- An Example: Windows 7

Objectives

- To discuss security threats and attacks
- To explain the fundamentals of encryption, authentication, and hashing
- To examine the uses of cryptography in computing
- To describe the various countermeasures to security attacks

The Security Problem

- System **secure** if resources used and accessed as intended under all circumstances
  - Unachievable
- **Intruders** (**crackers**) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse

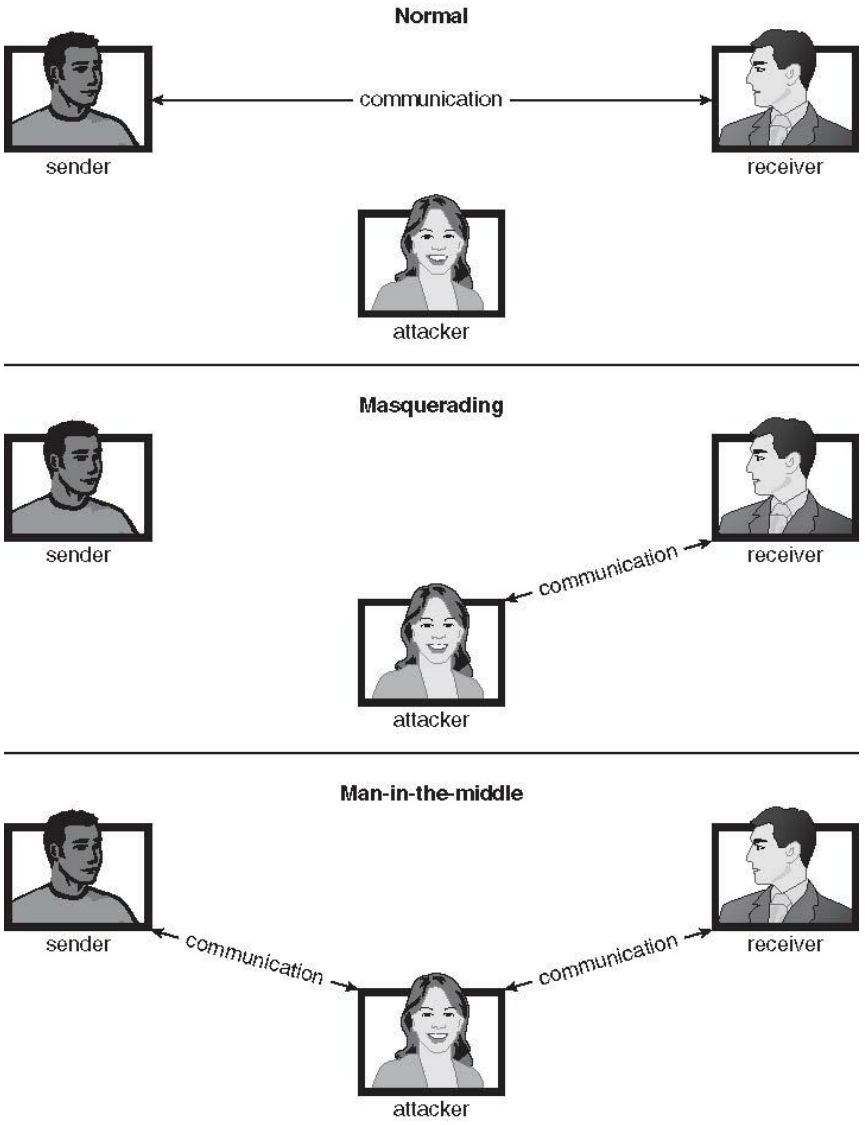
Security Violation Categories

- **Breach of confidentiality**
  - Unauthorized reading of data
- **Breach of integrity**
  - Unauthorized modification of data
- **Breach of availability**
  - Unauthorized destruction of data
- **Theft of service**
  - Unauthorized use of resources
- **Denial of service (DOS)**
  - Prevention of legitimate use

Security Violation Methods

- **Masquerading** (breach **authentication**)
  - Pretending to be an authorized user to escalate privileges
- **Replay attack**
  - As is or with **message modification**
- **Man-in-the-middle attack**
  - Intruder sits in data flow, masquerading as sender to receiver and vice versa
- **Session hijacking**
  - Intercept an already-established session to bypass authentication

Standard Security Attacks



Security Measure Levels

- Impossible to have absolute security, but make cost to perpetrator sufficiently high to deter most intruders
- Security must occur at four levels to be effective:
  - **Physical**
    - Data centers, servers, connected terminals
  - **Human**
    - Avoid **social engineering**, **phishing**, **dumpster diving**
  - **Operating System**
    - Protection mechanisms, debugging
  - **Network**
    - Intercepted communications, interruption, DOS
- Security is as weak as the weakest link in the chain
- But can too much security be a problem?

Program Threats

- Many variations, many names
- **Trojan Horse**
  - Code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - **Spyware**, **pop-up browser windows**, **covert channels**
  - Up to 80% of spam delivered by spyware-infected systems
- **Trap Door**
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
  - How to detect them?

Program Threats (Cont.)

- **Logic Bomb**
  - Program that initiates a security incident under certain circumstances
- **Stack** and **Buffer Overflow**
  - Exploits a bug in a program (overflow either the stack or memory buffers)
  - Failure to check bounds on inputs, arguments
  - Write past arguments on the stack into the return address on stack
  - When routine returns from call, returns to hacked address
    4. Pointed to code loaded onto stack that executes malicious code
  - Unauthorized user or privilege escalation

Program Threats (Cont.)

- **Viruses**
  - Code fragment embedded in legitimate program
  - Self-replicating, designed to infect other computers
  - Very specific to CPU architecture, operating system, applications
  - Usually borne via email or as a macro
  - Visual Basic Macro to reformat hard drive

```
Sub AutoOpen()
Dim oFS
  Set oFS = CreateObject(''Scripting.FileSystemObject'')
  vs = Shell(''c:command.com /k format c:'',vbHide)
End Sub
```

Program Threats (Cont.)

- **Virus dropper** inserts virus onto the system
- Many categories of viruses, literally many thousands of viruses
  - File / parasitic
  - Boot / memory
  - Macro
  - Source code
  - Polymorphic to avoid having a **virus signature**
  - Encrypted
  - Stealth
  - Tunneling
  - Multipartite
  - Armored

System and Network Threats (Cont.)

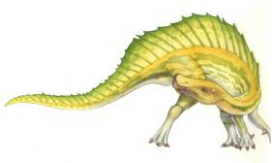
- **Worms** – use **spawn** mechanism; standalone program
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
  - Exploited trust-relationship mechanism used by *rsh* to access friendly systems without use of password
  - **Grappling hook** program uploaded main worm program
    - 99 lines of C code
  - Hooked system then uploaded main code, tried to attack connected systems
  - Also tried to break into other users accounts on local system via password guessing
  - If target system already infected, abort, except for every 7th time

System and Network Threats (Cont.)

- **Port scanning**
  - Automated attempt to connect to a range of ports on one or a range of IP addresses
  - Detection of answering service protocol
  - Detection of OS and version running on system
  - `nmap` scans all ports in a given IP range for a response
  - `nessus` has a database of protocols and bugs (and exploits) to apply against a system
  - Frequently launched from **zombie systems**
    - To decrease trace-ability

System and Network Threats (Cont.)

- **Denial of Service**
  - Overload the targeted computer preventing it from doing any useful work
  - **Distributed denial-of-service** (**DDOS**) come from multiple sites at once
  - Consider the start of the IP-connection handshake (SYN)
    - How many started-connections can the OS handle?
  - Consider traffic to a web site
    - How can you tell the difference between being a target and being really popular?
  - Accidental – CS students writing bad `fork()` code
  - Purposeful – extortion, punishment



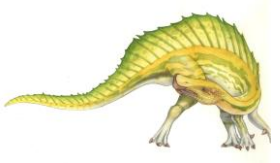
Sobig.F Worm

- More modern example
- Disguised as a photo uploaded to adult newsgroup via account created with stolen credit card
- Targeted Windows systems
- Had own SMTP engine to mail itself as attachment to everyone in infect system's address book
- Disguised with innocuous subject lines, looking like it came from someone known
- Attachment was executable program that created **`WINPPR23.EXE`** in default Windows system directory
  Plus the Windows Registry

```
[HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
   "TrayX" = %windir%\winppr32.exe /sinc
[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
   "TrayX" = %windir%\winppr32.exe /sinc
```

Cryptography as a Security Tool

- Broadest security tool available
  - Internal to a given computer, source and destination of messages can be known and protected
    - OS creates, manages, protects process IDs, communication ports
  - Source and destination of messages on network cannot be trusted without cryptography
    - Local network – IP address?
      - Consider unauthorized host added
    - WAN / Internet – how to establish authenticity
      - Not via IP address

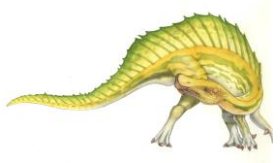


Cryptography

- Means to constrain potential senders (*sources*) and / or receivers (*destinations*) of *messages*
  - Based on secrets (**keys**)
  - Enables
    - Confirmation of source
    - Receipt only by certain destination
    - Trust relationship between sender and receiver

Encryption

- Constrains the set of possible receivers of a message
- **Encryption** algorithm consists of
  - Set *K* of keys
  - Set *M* of Messages
  - Set *C* of ciphertexts (encrypted messages)
  - A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, E_k is a function for generating ciphertexts from messages
    - Both *E* and E_k for any *k* should be efficiently computable functions
  - A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, D_k is a function for generating messages from ciphertexts
    - Both *D* and D_k for any *k* should be efficiently computable functions

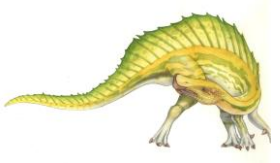


Encryption (Cont.)

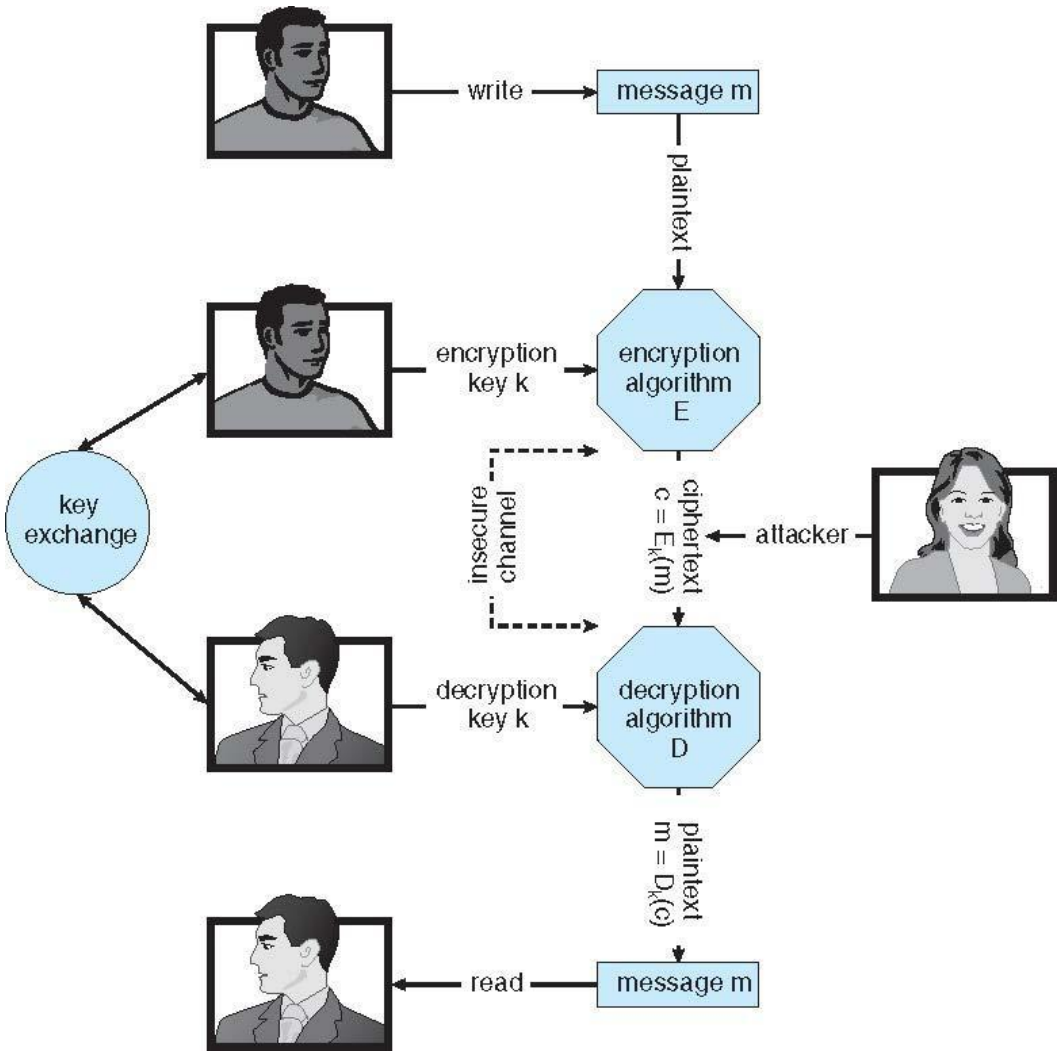
- An encryption algorithm must provide this essential property: Given a ciphertext $c \in C$, a computer can compute m such that $E_k(m) = c$ only if it possesses *k*
  - Thus, a computer holding *k* can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding *k* cannot decrypt ciphertexts
  - Since ciphertexts are generally exposed (for example, sent on the network), it is important that it be infeasible to derive *k* from the ciphertexts

Symmetric Encryption

- Same key used to encrypt and decrypt
  - Therefore *k* must be kept secret
- DES was most commonly used symmetric block-encryption algorithm (created by US Govt)
  - Encrypts a block of data at a time
  - Keys too short so now considered insecure
- Triple-DES considered more secure
  - Algorithm used 3 times using 2 or 3 keys
  - For example $c = E_{k3}(D_{k2}(E_{k1}(m)))$
- 2001 NIST adopted new block cipher - Advanced Encryption Standard (**AES**)
  - Keys of 128, 192, or 256 bits, works on 128 bit blocks
- RC4 is most common symmetric stream cipher, but known to have vulnerabilities
  - Encrypts/decrypts a stream of bytes (i.e., wireless transmission)
  - Key is a input to pseudo-random-bit generator
    - Generates an infinite **keystream**



Secure Communication over Insecure Medium



Asymmetric Encryption

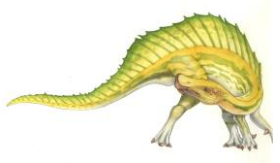
- **Public-key encryption** based on each user having two keys:
  - **public key** – published key used to encrypt data
  - **private key** – key known only to individual user used to decrypt data
- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme
  - Most common is **RSA** block cipher
  - Efficient algorithm for testing whether or not a number is prime
  - No efficient algorithm is know for finding the prime factors of a number

Asymmetric Encryption (Cont.)

- Formally, it is computationally infeasible to derive $k_{d,N}$ from $k_{e,N}$, and so k_e need not be kept secret and can be widely disseminated
  - k_e is the **public key**
  - k_d is the **private key**
  - N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each)
  - Encryption algorithm is $E_{ke,N}(m) = m^{ke} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$
  - The decryption algorithm is then $D_{kd,N}(c) = c^{kd} \bmod N$

Asymmetric Encryption Example

- For example. make $p = 7$and $q = 13$
- We then calculate $N = 7*13 = 91$ and $(p-1)(q-1) = 72$
- We next select k_e relatively prime to 72 and< 72, yielding 5
- Finally, we calculate k_d such that $k_e k_d \bmod 72 = 1$, yielding 29
- We how have our keys
  - Public key, $k_{e,N} = 5, 91$
  - Private key, $k_{d,N} = 29, 91$
- Encrypting the message 69 with the public key results in the cyphertext 62
- Cyphertext can be decoded with the private key
  - Public key can be distributed in cleartext to anyone who wants to communicate with holder of public key

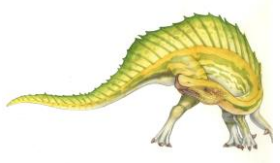


Cryptography (Cont.)

- Note symmetric cryptography based on transformations, asymmetric based on mathematical functions
  - Asymmetric much more compute intensive
  - Typically not used for bulk data encryption

Authentication

- Constraining set of potential senders of a message
  - Complementary to encryption
  - Also can prove message unmodified
- Algorithm components
  - A set K of keys
  - A set M of messages
  - A set A of authenticators
  - A function $S : K \rightarrow (M \rightarrow A)$
    - That is, for each $k \in K$, S_k is a function for generating authenticators from messages
    - Both S and S_k for any k should be efficiently computable functions
  - A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \in K$, V_k is a function for verifying authenticators on messages
    - Both V and V_k for any k should be efficiently computable functions

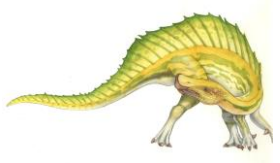


Authentication (Cont.)

- For a message *m*, a computer can generate an authenticator $a \in A$ such that $V_k(m, a) =$ `true` only if it possesses *k*
- Thus, computer holding *k* can generate authenticators on messages so that any other computer possessing *k* can verify them
- Computer not holding *k* cannot generate authenticators on messages that can be verified using V_k
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive *k* from the authenticators
- Practically, if $V_k(m,a) =$ **`true`** then we know *m* has not been modified and that send of message has *k*
  - If we share *k* with only one entity, know where the message originated

Authentication – Hash Functions

- Basis of authentication
- Creates small, fixed-size block of data **message digest** (**hash value)** from *m*
- Hash Function *H* must be collision resistant on *m*
  - Must be infeasible to find an *m'* ≠ *m* such that *H*(*m*) = *H*(*m'*)
- If *H*(*m*) = *H*(*m'*), then *m* = *m*'
  - The message has not been modified
- Common message-digest functions include **MD5**, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash
- Not useful as authenticators
  - For example *H(m*) can be sent with a message
    - But if *H* is known someone could modify *m* to *m'* and recompute *H(m')* and modification not detected
    - So must authenticate *H(m)*

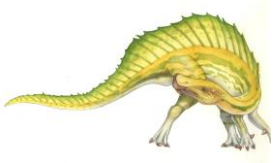


Authentication - MAC

- Symmetric encryption used in **message-authentication code** (**MAC**) authentication algorithm
- Cryptographic checksum generated from message using secret key
  - Can securely authenticate short values
- If used to authenticate *H(m)* for an *H* that is collision resistant, then obtain a way to securely authenticate long message by hashing them first
- Note that *k* is needed to compute both S_k and V_k, so anyone able to compute one can compute the other

Authentication – Digital Signature

- Based on asymmetric keys and digital signature algorithm
- Authenticators produced are **digital signatures**
- Very useful – ***anyone*** can verify authenticity of a message
- In a digital-signature algorithm, computationally infeasible to derive k_s from k_v
  - V is a one-way function
  - Thus, k_v is the public key and k_s is the private key
- Consider the RSA digital-signature algorithm
  - Similar to the RSA encryption algorithm, but the key use is reversed
  - Digital signature of message $S_{ks}(m) = H(m)^{ks} \bmod N$
  - The key k_s again is a pair (d, N), where N is the product of two large, randomly chosen prime numbers p and q
  - Verification algorithm is $V_{kv}(m, a) \stackrel{?}{=} (a^{kv} \bmod N = H(m))$
    - Where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$



Authentication (Cont.)

- Why authentication if a subset of encryption?
  - Fewer computations (except for RSA digital signatures)
  - Authenticator usually shorter than message
  - Sometimes want authentication but not confidentiality
    - Signed patches et al
  - Can be basis for **non-repudiation**

Key Distribution

- Delivery of symmetric key is huge challenge
  - Sometimes done **out-of-band**
- Asymmetric keys can proliferate – stored on **key ring**
  - Even asymmetric key distribution needs care – man-in-the-middle attack

Digital Certificates

- Proof of who or what owns a public key
- Public key digitally signed a trusted party
- Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- **Certificate authority** are trusted party – their public keys included with web browser distributions
  - They vouch for other authorities via digitally signing their keys, and so on

Implementation of Cryptography

- Can be done at various **layers** of ISO Reference Model
  - SSL at the Transport layer
  - Network layer is typically **IPSec**
    - **IKE** for key exchange
    - Basis of **Virtual Private Networks (VPNs)**
- Why not just at lowest level?
  - Sometimes need more knowledge than available at low levels
    - i.e. User authentication
    - i.e. e-mail delivery

OSI model

7. Application Layer

NNTP · SIP · SSI · DNS · FTP · Gopher · HTTP · NFS · NTP · SMPP · SMTP · SNMP · Telnet · Netconf · (more)

6. Presentation Layer

MIME · XDR · TLS · SSL

5. Session Layer

Named Pipes · NetBIOS · SAP · L2TP · PPTP · SPDY

4. Transport Layer

TCP · UDP · SCTP · DCCP · SPX

3. Network Layer

IP (IPv4, IPv6) · ICMP · IPsec · IGMP · IPX · AppleTalk

2. Data Link Layer

ATM · SDLC · HDLC · ARP · CSLIP · SLIP · GFP · PLIP · IEEE 802.3 · Frame Relay · ITU-T G.hn DLL · PPP · X.25 · Network Switch · DHCP

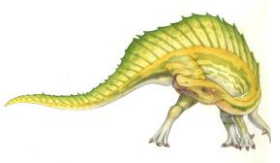
1. Physical Layer

EIA/TIA-232 · EIA/TIA-449 · ITU-T V-Series · I.430 · I.431 · POTS · PDH · SONET/SDH · PON · OTN · DSL · IEEE 802.3 · IEEE 802.11 · IEEE 802.15 · IEEE 802.16 · IEEE 1394 · ITU-T G.hn PHY · USB · Bluetooth · Hubs

This box: view · talk · edit

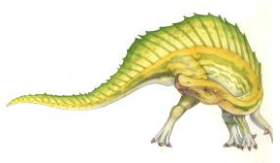
OSI Model			
	Data unit	Layer	Function
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication
	Segments	4. Transport	End-to-end connections and reliability, flow control
Media layers	Packet/Datagram	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Source: http://en.wikipedia.org/wiki/OSI_model



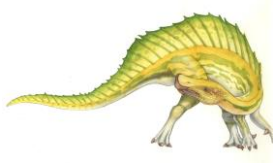
User Authentication

- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through **passwords**, can be considered a special case of either keys or capabilities
- Passwords must be kept secret
  - Frequent change of passwords
  - History to avoid repeats
  - Use of “non-guessable” passwords
  - Log all invalid access attempts (but not the passwords themselves)
  - Unauthorized transfer
- Passwords may also either be encrypted or allowed to be used only once
  - Does encrypting passwords solve the exposure problem?
    - Might solve **sniffing**
    - Consider **shoulder surfing**
    - Consider Trojan horse keystroke logger
    - How are passwords stored at authenticating site?



Passwords

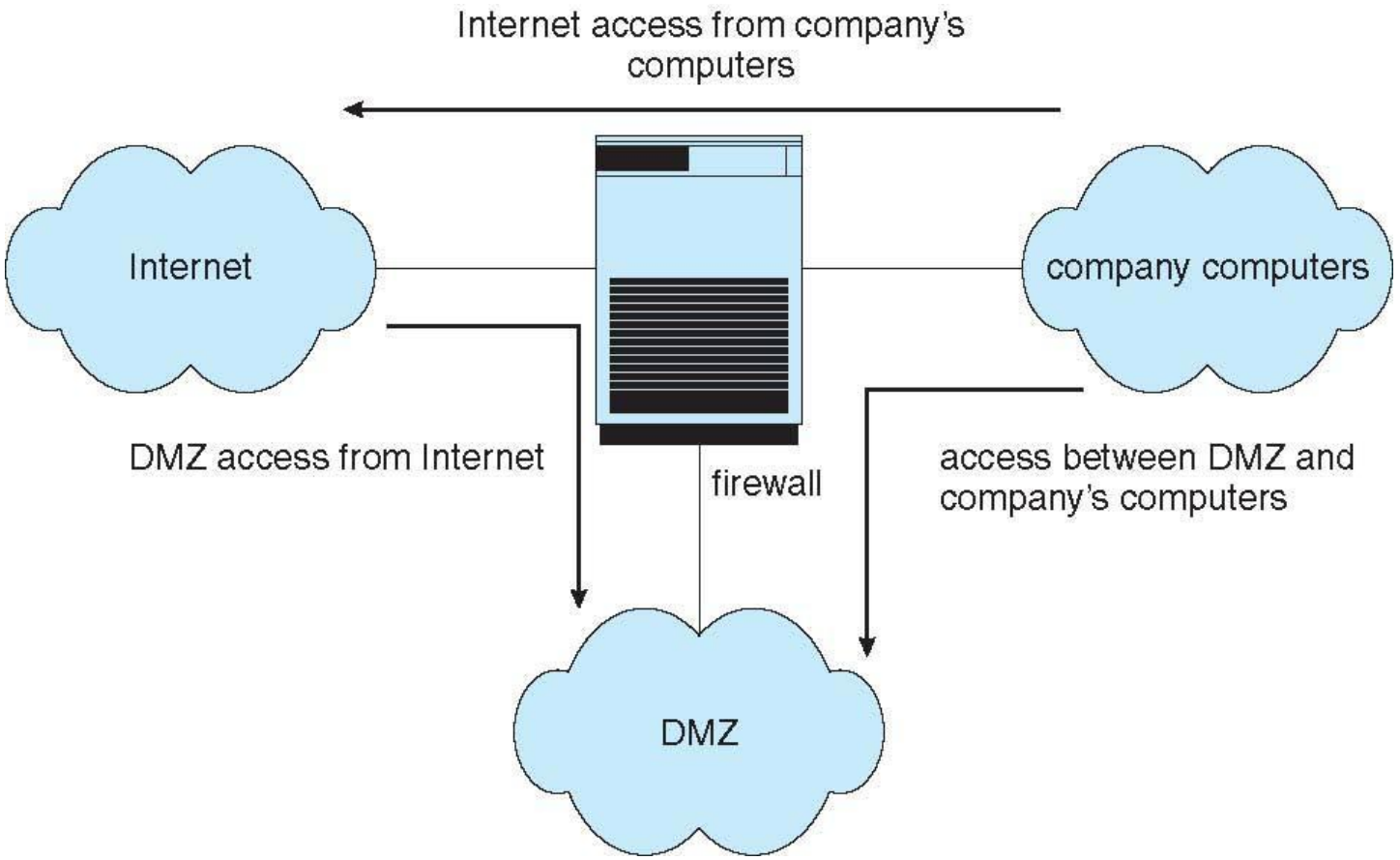
- Encrypt to avoid having to keep secret
  - But keep secret anyway (i.e. Unix uses superuser-only readably file `/etc/shadow`)
  - Use algorithm easy to compute but difficult to invert
  - Only encrypted password stored, never decrypted
  - Add “salt” to avoid the same password being encrypted to the same value
- One-time passwords
  - Use a function based on a seed to compute a password, both user and computer
  - Hardware device / calculator / key fob to generate the password
    - Changes very frequently
- Biometrics
  - Some physical attribute (fingerprint, hand scan)
- Multi-factor authentication
  - Need two or more factors for authentication
    - i.e. USB “dongle”, biometric measure, and password



Firewalling to Protect Systems and Networks

- A network **firewall** is placed between trusted and untrusted hosts
  - The firewall limits network access between these two **security domains**
- Can be tunneled or spoofed
  - Tunneling allows disallowed protocol to travel within allowed protocol (i.e., telnet inside of HTTP)
  - Firewall rules typically based on host name or IP address which can be spoofed
- **Personal firewall** is software layer on given host
  - Can monitor / limit traffic to and from the host
- **Application proxy firewall** understands application protocol and can control them (i.e., SMTP)
- **System-call firewall** monitors all important system calls and apply rules to them (i.e., this program can execute that system call)

Network Security Through Domain Separation Via Firewall



End of Chapter 15