EECS 678 - Fall 2020

Midterm Preparation Questions

1 Chapter 1

1. Explain the User's and System's view of the operating system.
2. Explain the operating system goals.
3. What is a multiprogramming OS? How does it differ from a batch OS? How does it differ from a time-sharing OS?
4. How does an OS ensure protection between two user processes?
5. How does an OS ensure its own protection as well as that of the other system resources?
6. Sort the following levels of storage hierarchy by their access time and size.
   (a) Registers, (b) cache, (c) disk
7. What factors restrict the number of architected registers?
8. What factors restrict the size of the cache? Why have multiple levels of caches?
9. Explain if true or false:
   (a) The architecture registers are managed by the hardware.
   (b) The user can directly manage the disk.
10. The OS provides a virtualization. Explain.

2 Chapter 2

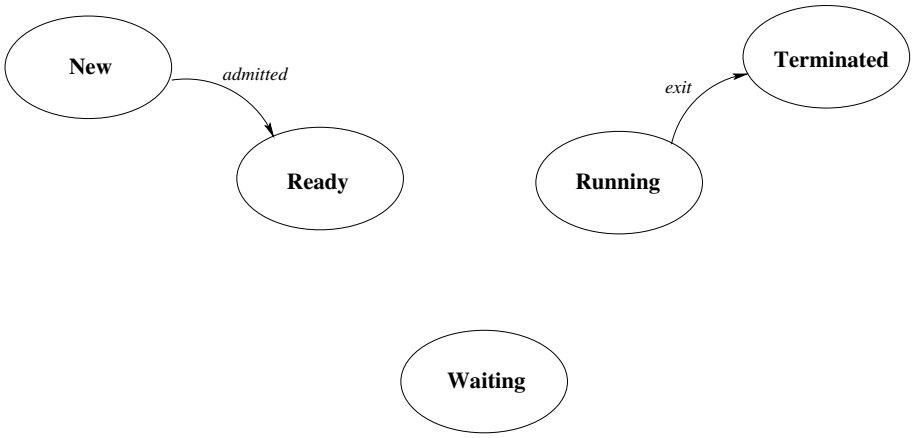
1. What are *system calls*? How do they differ from other function calls?
2. What is an API? How is an API call different from a system call? Which one would you prefer to use in your high-level program?
3. What are the limitations of passing function arguments in registers? How are these limitations overcome?
4. Discuss the definition of both policy and mechanism during software design and implementation.
5. What are the characteristics (in 1-2 lines), advantages and disadvantages of the following kinds of OS designs:
   (a) monolithic design
   (b) layered approach
   (c) microkernel approach.
6. What is virtualization? How is virtualization different from abstraction?
7. What are the benefits of virtualization?

3 Chapter 3

1. Define process. How is a process different from a *program*?

2. Illustrate (draw) the process address space. What component of your high-level C program goes into each section of the process address space?

3. The process state diagram below shows all the possible process states, but only two process transitions. Complete the figure by illustrating how the process changes states on the following events: (a) interrupt, (b) scheduler dispatch, (c) I/O or event completion, and (d) I/0 or event wait.



4. Which of the following typical kinds of CPU instructions should be privileged? Explain your reasoning briefly.
   (a) Set value for hardware timer
   (b) Read hardware clock
   (c) Set a value in the shared memory region.
   (d) Switch from user to OS execution mode
   (e) Turn off hardware interrupts.

5. Describe what a kernel must do to switch context between two processes. What are the benefits and drawbacks of context switching for an OS?

6. What is the purpose of the `fork` and `exec` system calls?

7. Using appropriate system calls, have the program below create a new process, and execute the program `/bin/ps`. It is not important to get all the function arguments and their positions correct.

```
int main()
{
   pid_t pid;

   /*create a new process*/
   pid =
   if(pid == 0){
```

```
   }
   else if(pid < 0){



   }
   else if (pid > 0){



   }
}
```

8. Compare the shared memory and message passing IPC models as regards: (a) characteristics (b) efficiency (c) ease of use. Explain

9. Consider the program below:

```
int main()
{
   pid_t pid;
   int shared_num = 0;

   /* create a pipe */
   ...

   /* create a process */
   pid = fork();
   if(pid > 0){
      /* parent process */
      /* write numbers to the shared variable*/
      int i;
      for(i=0 ; i<10000 ; i++)
         shared_num += 1;

      wait();
      fprintf(stdout, "Shared variable = %d\n", shared_num);
   }
   else{
      /* child process */
      int i;
      for(i=0 ; i<10000 ; i++)
         shared_num -= 1;
   }
}
```

   Use pipes to synchronize access to the shared variable so that the final result printed in the parent process is 0. (You need to first understand why the answer printed may be different from 0).

10. Compare shared memory based IPC with message passing based IPC.
11. What are the advantages and limitations of pipes IPC? How do some other IPC mechanisms address these drawbacks?
12. List and explain in a line the system calls used to setup a shared memory segment.

4 Chapter 4

1. Draw the process address space of a process with two threads. Indicate the stack, heap, text, and data space of the two threads. How is the process address space different if we had two processes instead of two threads?
2. What the (a) advantages and (b) disadvantages of threads over processes?
3. Differentiate between (a) user-level and (b) kernel-level threads. What are their advantages and disadvantages over each other?
4. What is the main disadvantage of the many-to-one multithreading model?
5. What *system call* is used to create threads in Linux?
6. Write a simple program that contains two global variables `num1` and `num2`. The program then: (a) creates two threads, (b) Thread-1 performs `num1+num2` and returns the result, (c) Thread-2 performs `num1-num2` and returns the result, (d) Main thread displays the two returned results and exits.
7. How do system calls *fork* and *exec* operate if invoked from multi-threaded programs? (You may use Linux as an example.)
8. Explain the two mechanisms of thread cancellation.
9. How are *synchronous* and *asynchronous* signals handled in a multi-threaded program?

5 Chapter 5

1. What is a critical section? What are the three conditions to be ensured by any solution to the critical section problem?
2. Consider the following set of processes P1 and P2:

```
P1: {                          P2:{
  shared int x;                  shared int x;

  x = 7;                         x = 7;
  while (1) {                    while (1) {
    x--;                           x--;
    x++;                           x++;
    if (x != 7) {                  if (x != 7) {
      printf(``x is %d'',x)          printf(``x is %d'',x)
    }                              }
  }                              }
}                              }
```

Note that the scheduler in a uniprocessor system would interleave the instructions of the two processes executing, without restriction on the order of the interleaving.

Show an execution (i.e., trace the sequence of inter-leavings of statements) such that the statement 'x is 7' is printed.

3. Understand Peterson's solution to the the synchronization problem.

4. Provide the pseudo code definitions of the following hardware atomic instructions:
(a) TestAndSet

5. A general synchronization solution using locks looks as follows:

```
int mutex;
init_lock(&mutex);

do {
   lock(&mutex);

   // critical section

   unlock(&mutex);

   // remainder section
}while(TRUE);
```

Provide the definitions of *init_lock*, *lock*, and *unlock* when using (a) TestAndSet

Chapter 1 - OS Introduction

1. **Introduction – Outline:**

   The major objectives of this chapter are:
   1. To provide a grand tour of the major components of operating systems, and
   2. To describe the basic organization of computer systems.

2. **What is an Operating System?:**

   - OS provides an abstraction. Example, The OS abstracts the tracks/sectors for disk storage with the file system abstraction. The user does not need to know where each data byte is saved on the physical storage medum.
   - The embedded device may not use an OS, but might still need to interact with some IO devices, like the display, but that will be handled by the application itself.

3. **User's View of the Operating System:** The OS goal here is to make the user's life easier.

   - From the user's viewpoint, the OS provides an easier environment to work in than that provided by pure hardware, which includes the processor and the I/O devices. The OS also provides more powerful instruction and functions that those provided by the hardware, eg., write(fileno, buf, len);
   - Note the difference between library call and system call. While the library call is provided by the language *runtime*, the system call is provided by the OS.
   - Maintaining memory and program state are complex operations that are abstracted by the OS.
   - When multiple users are accessing the same computational resources then, the OS manages the CPU time, memory and I/O resources, schedules them fairly among all the users of the system. Other users should not be able to view or modify other user's program state. Each user feels that she has the computer all to herself.
     Fairness: If some user program goes into an infinite loop, then the other programs should still run unaffected.
     Performance: The OS should not take very long for its own activities, to the detriment of application performance.

4. **System's view of the Operating System:** Hardware resource allocation, management and resource monitoring.

   From the system;s point of view it is not critical to make the user's life easier, but more important to use hardware resources efficiently.

   The OS is most intimately involved with managing the hardware resources. Conflict resolution is important whenever the system has multiple simultaneous programs executing, from one or multiple users.

   What is fair? Higher priority processes, but who gets higher priority? Real-time processes, critical system applications, high-paying users on batch systems.

   The view of the OS as a control program only provides a slight variation over the previous view.

5. **What Does an Operating System Do?** A good analogy of an OS is the government. It does no useful work on its own, but provides an environment for other people and industries to do their work.

   However, we still do not have a very good definition of the OS. Some OSes are less that 1Mb and others are in GB sizes.

   OS vendors need to be careful regarding what they ship as part of the OS. Microsoft has got into numerous problems for shipping applications with the OS that the courts decided were not a core functionality of the OS, and thawrted competition for other vendors, like music player, and web browzer.

6. **Components of a Computer Systen:**

   - The OS is mostly written in software, although that is not necessary. In fact, there have been attempts to write OSes in hardware, particularly using FPGAs for embedded applications.
   - The programs provided by the OS are called *system calls*. Application programs access the hardware by invoking these system functions.

7. **Computer System Operation:**

   - Bootstrapping ("to pull oneself up by one's bootstraps") refers to techniques that allow a simple system to activate a more complicated system. The bootstrap program does not need to modify often, and hence is present on ROMs, EEPROMS, or firmware in general. It initializes all aspects of a system.
   - The first program is called *init* on Unix-based systems.
   - Software interrupts can be triggered by the software, either by implicitly (such as divide by zero errors or by calling privileged instructions), or explicitly by invoking a system call. These are called traps or exceptions. Exceptions are triggered by the hardware, and so are not associated with a specific software instruction.
   - Occurence of an interrupt stops normal execution of the current program. The interrupt vector contains the addresses of all the service routines for a fixed number of interrupts supported by that system. This interrupt vector table is a table of pointers stored in low memory addresses. This interrupt mechanism is used to perform all useful tasks on a modern computer.
     eg., the execve() function call to start a new program.
     eg., the timer interrupt to schedule programs, etc.

8. **Operating System Operations:** What is an interface? Communication link between 2 computer layers. We will talk more about the abstraction provided by the OS, and how it simplifies the hardware environment for the user in Chapter 2.

9. **Evolution of Operating Systems:**

   - **Batch OS:** Goal: Simplify OS structure.

   - **Multiprogramming OS:** Goal: Maximize system throughput / performance. Keep CPU busy all the time.
     Multiprogramming is also how people generally operate. We have several tasks on our hands, and if one is waiting for something, then we go on to the others.

- **Time-sharing OS:** Goal: Provide an interactive OS environment.
  Time sharing enables an interactive environment for program execution. Allows the OS to produce a virtualization in which each user believes that the entire computer system is dedicated to her use.

  Virtual memory allows execution of programs that are larger than the available physical memory. It also abstracts memory into a uniform array of storage, separating *logical memory* as seen by the user from physical memory. This is a very useful abstraction, and we will study it in more detail later in Chapters 8 ad 9.

10. **Operating System Tasks:**

11. **Process co-ordination and security:**

    - The OS needs to ensure that an error or malicious activity in one process does not affect any any program in a multiprocess/multiuser environment. For example, if a program got stuck in an infinite loop, then this loop could affect the operation in other concurrent processes. Without protection, the computer will be restricted to execute only one process at a time.
    - Kernel mode is also called system mode, privileged mode, or supervisor mode. Mode bit is provided by the hardware. System executing normal user applications is in user mode. User programs can invoke system calls to request OS to perform a privileged service. The system then transitions to system mode. Instructions that can affect the correct operations of other processes in the system are called privileged instructions. We will see an example on the next slide with the system timer.
    - MS-DOS was written for the 8088, which had no mode bit, and hence no dual mode. DOS therefore allowed user programs unrestricted access to the system, and could erase the entire OS, if the program were not careful. Most current architectures provide the mode bit, and OSes take advantage of it to provide security.
    - In Unix-like systems, "fork" and "execve" are C library functions that in turn execute instructions that invoke the "fork" and "execve" system calls. Making the system call directly in the application code is more complicated and may require embedded assembly code to be used (in C and C++) as well as knowledge of the application binary interface; the library functions are meant to abstract this away.

12. **Transition from User to Kernel Mode:** Timer ensures that the OS remains in control of the CPU resources. Timer can prevent a single user program from running too long, and being unfair to other time-shared programs. All instructions managing the timer setting, decrement, reset, etc. are privileged instructions.

13. **Process Management:**

    - All programs running on the computer, web-browser, word, games, compiler, etc. run as processes. A static program when initialized for execution or doing useful work, is a process. Thus, we can start multiple instrances of a web-browser as distinct processes, although they are running the same program.
    - Resources needed by a process include: cpu, memory and I/O, files, and well as initialization data. Typically system has many processes, some user, some operating system running concurrently on one or more CPUs. Concurrency is achieved by multiplexing the CPUs among the processes / threads.
    - We will also study threads, which are light-weight processes, sharing some resources.

14. **Process Management Activities:** We will discuss mechanisms for performing all these process management activities in this class.

15. **Memory Management:** Memory can be considered as a large array of addresses, each address storing a byte or a word of data. The CPU can only address data and instructions from memory, not from disk, or from the cache. Addresses produced by the CPU are memory addresses. The only other option is to use registers.

    Why memory management? Because memory is limited, and is often less than the combined requirement of multiple processes wanting simultaneous access to it.

    Memory management is to manage the requirement of the multiple processes present in memory at the same time. This is to improve CPU utilization, and user response.

16. **Storage Management:**

    - Providing storage is important because users cannot use memory to store information for long: memory is limited in size, and is volatile.
    - Several kinds of mass storage media are available for permanant storage of often-used information, as well as for backups of archieval data. Different media have different properties that are difficult and cumbersome to remember. OS makes computer system easier to use by abstracting away details of the various storage media such as magnetic disk, magnetic tape, optical disk, etc. Thus, we do not need to remember the various tracks and sectors locations on a disk where bits of our data might be stored. The OS provides a common abstraction to store information, called a file.
    - When files are shared among multiple users, providing access controls is important. At the same time, providing effecient access is very important.

17. **Storage Heirarchy:** The closer the storage level is to the CPU the better its performance. Smaller size allows faster access, and less power consumption. Faster process technology makes the component more expensive.

    Note that registers and caches are not typically managed by the OS, but the cache may be in some cases. Also note that a data item is backed up all the way, it appears in multiple locations simultaneously, which makes it important to maintain its consistency among the various storage location.

18. **I/O subsystem:**

    - File system mamagement can also be also part of the I/O subsystem. *Drivers* are the OS programs used to manage each I/O device. That is why you are required to install drivers for any new device you want to use with your computer, disks, USB mouse, keyboard, monitors, etc.
    - Buffering is done while printing data to file to reduce overhead of small writes.
    - Disk caches are used to enable faster disk access.
    - Spool refers to the process of placing data in a temporary working area for another program to process. Spooling is useful because devices access data at different rates. Spooling allows one program to assign work to another without directly communicating with it. The most common spooling application is print spooling.

19. **Protection and Security:**

    - We already saw the concepts of address spaces and mode bits and timer for process-level protection.

- A computer virus attaches itself to a program or file. it actually cannot infect your computer unless you run or open the malicious program. A virus cannot spread without human action; worm like virus, but can travel autonomously.
- setuid and setgid are Unix access rights flags that allow users to run an executable with the permissions of the executable's owner or group. setuid and setgid are needed for tasks that require higher privileges than those which common users have, such as changing their login password. Some of the tasks that require elevated privileges may not immediately be obvious, though such as the ping command, which must send and listen for control packets on a network interface.

20. **Summary - Operating Systems:** The OS frequently creates the VM that presents a different view of the system that reality.

    A time-shared OS provides the view of multiple CPUs. Virtual memory provides a view of infinite memory capacity. File system abstraction provides a view of simple, flexible, random-access data access. Insecure networks are made secure as well as reliable by implementing protocols such as TCP/IP. Distributed OS, or multi-core OS takes multiple processors or systems and provides the view of a single high-performance system.

Introduction – Outline

- Goals of an operating system
- Computer system operation
- Types of operating systems
- Operating system tasks and functions

OS GOALS

What is an Operating System?

- A program that acts as an intermediary between users and the computer hardware.
- Who needs an OS?
  - all general purpose computers use an OS
  - OS makes a computer easier to use
- A better question: Who does not need OS?
  - Some specialized systems that usually do one thing (OS can be embedded in the application).
    - Microwave oven control, MP3 players, etc.
- Operating System's role:
  - User viewpoint
  - System viewpoint

User's View of the Operating System

- Make it easier to write programs
  - provide an abstraction over the hardware
- Provide more powerful instructions than the ISA
  - e.g., write(fileno, buf, len);
- Make it easy to run programs
  - loading program into memory, initializing program state, maintaining program counter, stopping the program
  - instead user types: *gcc hello.c* and then *./a.out*
- Utilities for multi-user mode operation
  - resource management, security, fairness, performance

System's View of the Operating System

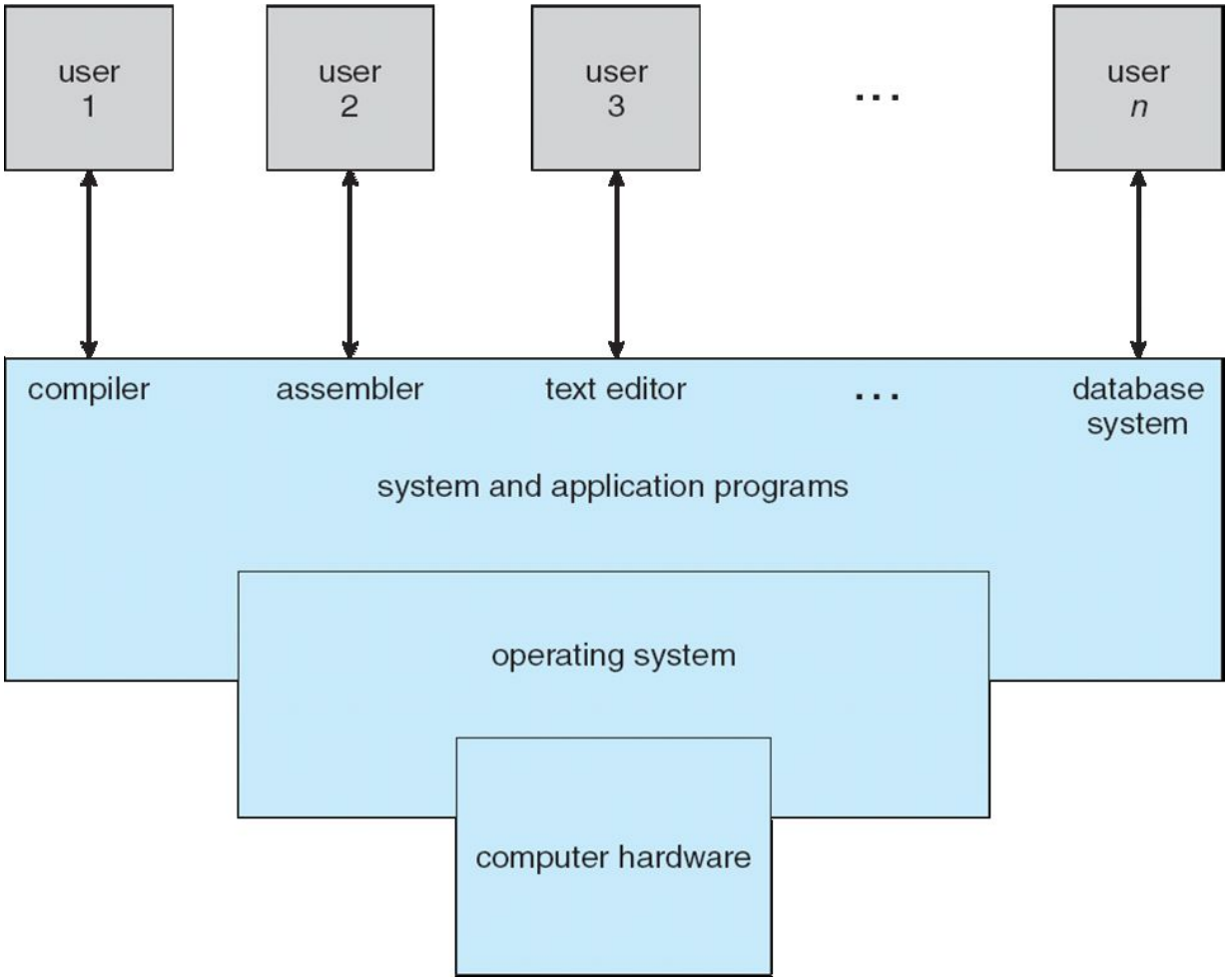
- OS as a resource allocator
  - manage CPU time, memory space, file storage space, I/O devices, network, etc.
  - fairly resolve conflicting requests for hardware resources from multiple user programs
- OS as a control program
  - control the various I/O devices and user programs

What Does an Operating System Do?

- OS is a program that acts as an intermediary between a user of a computer and the computer hardware.
  - provide an environment for easy, efficient program execution
- Operating system goals:
  - execute user programs and make solving user problems easier
  - make the computer system convenient to use
  - use the computer hardware in an efficient manner
- Definition
  - everything that a vendor ships when you order an *OS* !
  - program that is always running on the computer (*kernel)*

computer system operation

Components of a Computer System



Components of a Computer System

- Hardware
  - provides basic computing resources (CPU, memory, I/O devices)
- Operating system
  - controls and coordinates use of hardware among various applications and users
- Application programs
  - define the ways in which the system resources are used to solve the computing problems of the users (word processors, compilers, web browsers, database systems, video games)
- Users

Computer System Operation

- *Bootstrap program* for computer to start running
  - initialize CPU registers, device controllers, memory
  - locate and load the OS kernel
- OS starts executing the first program
  - waits for some event to occur (interrupt-driven)
- Occurrence of an interrupt (exceptions and traps)
  - processor checks for occurrence of interrupt
  - transfers control to the interrupt service routine, generally, through the *interrupt vector* table (IVT)
  - IVT is at a fixed address in memory (known to CPU)
  - execute the associated interrupt service routine
  - return control to the interrupted program

Operating System Operation

- OS hides the complexity and limitations of hardware (hardware interface) and creates a simpler, more powerful abstraction (OS interface).
- The bootstrap program locates the OS kernel and loads into memory.
- OS sits quietly, waits for events
  - hardware interrupts (sent to the CPU over the system bus)
  - software interrupts (software error or system call)

Operating System Types

Evolution of Operating Systems

- Batch systems
  - earliest operating systems
  - each user submits one or more *jobs*
  - collect a *batch* of jobs
  - process the batch one job at a time, one after the other
- Problems
  - Job 2 cannot start until job 1 finishes
  - I/O activity stalls the CPU; CPU under-utilized
  - No interactivity

Evolution of Operating Systems

- Multiprogramming
  - a single job cannot keep CPU and I/O devices busy at all times
  - OS is given multiple runnable jobs at a time
  - when current job has to wait (for I/O for example), OS switches to another job
  - multiple jobs kept in memory simultaneously
  - I/O and CPU computation can overlap
  - CPU-bound Vs. I/O bound jobs
  - OS goal: maximize system throughput

Evolution of Operating Systems

- Time sharing *(multitasking)*
  - logical extension of multiprogramming
  - CPU switches jobs so frequently that users can interact with each job while it is running
  - very fast response time
  - each user has at least one program executing in memory
  - OS scheduler decides which job will run if several jobs are ready to run at the same time
  - If processes don't fit in memory, *swapping* moves them in and out to run
  - *virtual memory* allows execution of processes not completely in memory
  - OS goal: optimize user response time

Evolution of Operating Systems

- Distributed Operating Systems
  - for physically separate, heterogeneous, networked computers
  - Hardware: computers with networks
  - OS goal: ease of resource sharing among machines
- Virtualization
  - OS itself runs under the control of an *hypervisor*
  - VM have own memory management, file system, etc.
  - VM is key feature of some operating systems
    - Windows server 2008, HP Integrity VM
  - hypervisor no longer optional
    - POWER5 and POWER6 from IBM

Operating System Tasks

Operating System Tasks

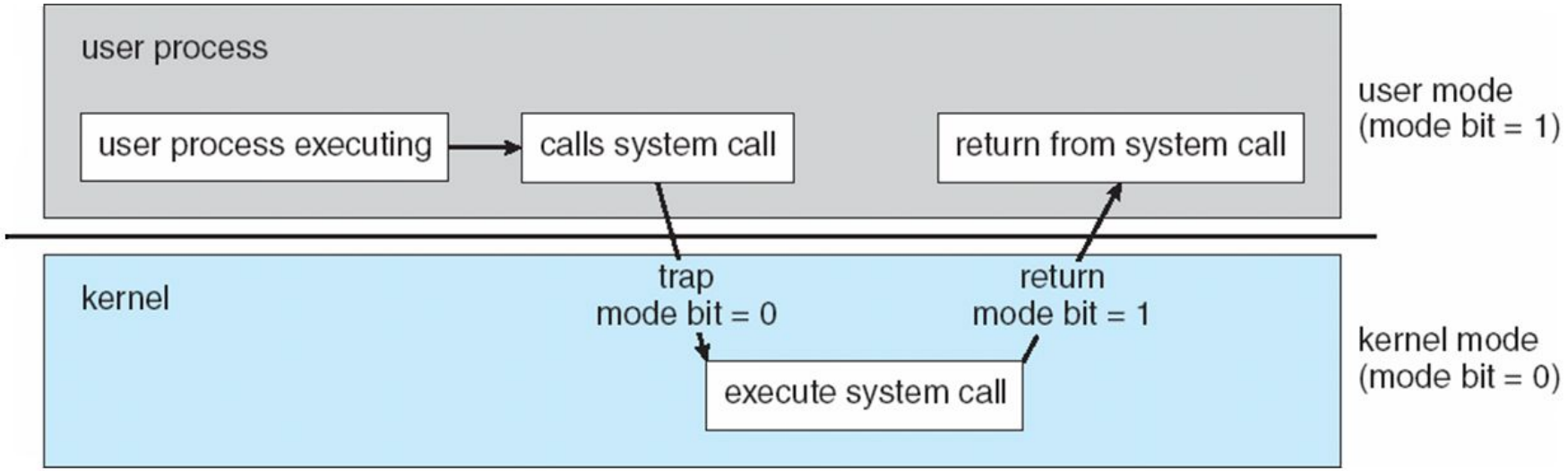
- Process co-ordination and security
- Process management
- Memory management
- Storage management
- Other issues in protection and security

Process Co-ordination & Security

- Applications should not crash into one-another
  - *address space:* all memory addresses that an application can touch
  - address space of one process is separated from address space of another process
- Applications should not crash the OS
  - *dual-mode* operation (*user* mode and *kernel* mode)
  - distinguish when system is running in user or system mode
  - *privileged* instructions only operate in kernel mode
  - system calls / returns change mode

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - OS sets *timer* interrupt after specific period
  - hardware decrements counter
  - when counter zero generate an interrupt
  - set up before scheduling process to regain control or terminate program that exceeds allotted time



Process Management

- Process
  - is a program in execution
  - is a unit of work within the system
  - program is a *passive entity*, process an *active entity*
  - needs resources to accomplish its task
    - termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
  - execute instructions sequentially, until completion
- Multi-threaded process has one program counter per thread.

Process Management Activities

- The operating system is responsible for the following activities in connection with process management
  - process scheduling
  - suspending and resuming processes
  - providing mechanisms for process synchronization
  - providing mechanisms for process communication
  - providing mechanisms for deadlock handling

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
  - optimizing CPU utilization and computer response to users
- Memory management activities
  - keeping track of which parts of memory are currently being used and by whom
  - deciding which processes (or parts thereof) and data to move into and out of memory
  - allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of info. storage
  - abstracts physical properties to logical storage unit – file
  - medium controlled by device (i.e., disk drive, tape drive)
    - varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control is provided to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

Storage Hierarchy

- Performance of various levels of storage depends on
  - distance from the CPU, size, and process technology used
- Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

Protection and Security

- Protection – mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of the system against internal and external attacks
  - denial-of-service
  - worms
  - viruses
  - identity theft
  - theft of service

Protection and Security (2)

- Systems generally first distinguish among users, to determine who can do what
  - user identities (user IDs, security IDs) include name and associated number, one per user
  - user ID then associated with all files, processes of that user to determine access control
  - group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
  - privilege escalation allows user to change to effective ID with more rights

summary

Summary – Operating System

- OS is the software layer between the hardware and user programs.
- OS is the ultimate API.
- OS is the first program that runs when the computer boots up.
- OS is the program that is always running.
- OS is the resource manager.
- OS is the creator of the virtual machine.

Summary – Operating System (2)

Reality	***Abstraction***
A single CPU	Multiple CPUs
Limited RAM capacity	Infinite capacity
Mechanical disk	File system
Insecure and unreliable networks	Reliable and secure
Many physical machines	A single machine

Preview Test: Quiz-Ch1

Test Information

Description	Quiz on Chapter 1 topics
Instructions	
Multiple Attempts	Not allowed. This test can only be taken once.
Force Completion	This test can be saved and resumed later.

QUESTION 1

5 points Saved

True or False (5 points): I completed and submitted the survey assigned in the first class.

- (●) True
- ( ) False

QUESTION 2

3 points Saved

Fill-in the blanks (3 points): Only write the "single letter" (upper-case, no spaces, no quotes) indicated for each option.

The three main components of the computer hardware are: B , D , and H .

(Choose from: 'A' - Operating system ; 'B' - CPU ; 'C' - Monitor ; 'D' - Memory ; 'E' - Disk ; 'F' - Keyboard ; 'G' - Storage devices ; 'H' - I/O devices ; 'G' - Mother Board ; 'H' - Network ; 'I' - Printer, 'O' - Other)

Question Completion Status:

QUESTION 3

1 points Saved

True or False (1 point): After startup, the operating system is idle until there is an interrupt.

- (●) True
- ( ) False

QUESTION 4

1 points Saved

True or False (1 point): The memory location of the interrupt vector table (IVT) must be known to the OS, but not necessarily to the CPU.

- ( ) True
- (●) False

QUESTION 5

2 points Saved

Write 'T' for all that apply, 'F' for false (upper-case and without quotes or extra spaces): The OS uses the *address space* protection mechanism to:

(a) ensure applications do not crash into each other --

T

(b) ensure applications do not crash into the OS --

F

QUESTION 6

1 points Saved

Select from options A, B or C (1 point):

The primary broad goal(s) of OS services designed to support the *user's view* of the operating system is:

C

A - enable efficient use of hardware resources

B - enable fair use of the hardware resources

C - make it easier to use or interact with the computer hardware

Question Completion Status:

Click Save and Submit to save and submit. Click Save All Answers to save all answers.

Save All Answers Save and Submit

Chapter 2 - OS Structures

1. **Chapter 2: Operating System Structures:** In this chapter we will try to answer the following questions.

   - What are the services provided by an OS ?
   - What are system calls ?
   - What are some common categories of system calls ?
   - What are the principles behind OS design & implementation ?
   - What are common ways of structuring an OS ?
   - How are VMs and OS related ?

   Detailed discussion on many of these issues will continue in the remaining portion of the semester.

2. **Operating System Services:**

   - Operating system services may change from one OS to the other. For example, some embedded OS may not allow user interaction. However, many services remain the same.
   - CLI uses text commands, as those issued from a shell in linux. Batch interface executes files containing commands, like commands written and execuetd from makefiles or shell scripts. GUI is well-known using windowing system, mouse pointer, menus, etc.
   - Trivia: Who invented the first OS graphical interface?
   - I/O devices include disks, CD, DVD, USB devices, printer, etc. How are I/O devices protected? For ease of operation, efficiency, and protection, I/O cannot be directly controlled in user-mode.
   - Several varieties of file systems are available, like the disk file system (FAT, NTFS), flash file system (flash devices), database file systems (in addition to hierarchical management, files are identified by other characteristics, like type of file, topic, author), network file system providing access to files on a remote server (FTP client).

3. **Operating System Services (2):**

   - The processes may be on the same computer, or on distinct computers separated by some network. Shared memory can be be simultaneously accesses by multiple processes, across distinct address spaces. We will study microkernel OS that rely on messages for communication between user processes.
   - The OS detects and reports errors to the user. In Unix, a return value of zero means correct execution, higher values report increasing severness of errors.

4. **Operating System Services (3):** Various scheduling algorithms may be employed to determine access to different devices. CPU cycles uses time-shared allocation, printer uses spooling algorithms, etc.

   If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

5. **A View of Operating System Services:** Figure shows a view of operating system services and how they inter-relate.

6. **System Calls:**

   - Programs in user mode do not have *direct* access to system resources, such as I/O devices, which are shared.
   - The API interface handles the low-level details of passing information to the kernel and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode.
   - Programs using an API (and not system calls) will work on any system that supports the same API.
   - System call implementation requires some architecture-specific mechanism. RISC architectures generally only provide an interrupt mechanism. CISC architectures may provide a direct *syscall* instruction to quickly transfer control to the kernel for a system call without the overhead of an interrupt.
   - Example: Linux 2.5 began using this on the x86, where available; formerly it used the INT instruction, where the system call number was placed in the EAX register before interrupt 0x80 was executed.
   - Can you explain the difference between a system call Vs. Library call?

7. **Example of System Calls:** Even simple programs make heavy use of system calls, typically thousands of such calls executed per second.

8. **API – System Call – OS Relationship:** A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

   The caller does not worry about how the system call is implemented. The caller obeys the API, which hides the details of the OS interface. The API call massages the arguments as desired by the system call. Linux has between 300-350 system calls.

9. **Standard C Library Example:** `printf()` is part of the libc API library. It calls the `write` system call to write the passed message to the specified I/O device. These are passed as arguments.

10. **System Call Parameter Passing:** Register method is not general enough to handle all cases. The argument passing specifications as given as part of the ABI to maintain system portability.

11. **Types of System Calls:** Some system calls may be present in multiple categories. Some important system calls:
    create process – fork()
    terminate process – exit()
    allocate and free memory – brk()
    etc.

12. **System Programs:** System programs shipped with the OS may vary with the OS, and may provide different functionalities. These programs are outside the kernel, and are not traditionally a part of the OS. System programs can be user interfaces to system calls, or may be considerably more complex.

    Note: Make sure you can differentiate between system calls, system programs, and application programs.

    Note: How many of these (Unix) system programs do you recognize?

13. **Role of Linker and Loader:** This figure shows the various systems tools to transform a program written in a high-level language to a binary executable.

    - **Compiler:** Transform a high-level language program to assembly program. (Given, test1.c; `gcc -S test1.c` will generate assembly program, `test1.S`.
    - **Assembler:** Assembly code to object code. (`gcc -c test1.c` will generate `test1.o`, a binary file. Use "objdump" to disassemble object files.)
    - **Linker:** Resolve external references, and generate executable file. (`gcc -o test1.exe test1.c` will generate the executable file, `test1.exe`)
    - **Loader:** is a part of the OS, that creates a *process* by loading the program into memory, allocating resources, etc.

14. **OS Design and Implementation:**

    - Lets see problems in designing and implementing an OS. No unique solution is presented, just some engineering principles.
    - At the highest design level, is the type of the OS. Note the requirement for each type of system. For example,
      Batch OS – does not does very fast switching between jobs, can reduce switch overhead.
      Single-user OS – may need to worry less about about security, etc.
    - Very important to separate policy from mechanism. Mechanisms determine how to do something, policies decide what will be done. The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later. Best to devise mechanisms so that several different user or system policies can be configured just by changing arguments.
    - With good compilers higher-level language programs can deliver performance very close to assembly programs. Compiler technology is constantly improving, which means that recompiled systems should perform better. Profiling can be used to only improve the critical sections of an OS.

15. **Operating System Structure:**

    - OS is large and complex piece of software. Proper engineering is essential. An OS is modularized into distinct components with well-defined interfaces. The OS should fulfil the essential software engineering design principles. We will see how these components are combined to work together.
    - Many OS have a simple monolithic structure that makes it hard to maintain and update. DOS started off as a simple OS then became very popular and had to be extended to other environments. Similarly, UNIX also encompasses a massive amount of functionality in the kernel. A better or poor OS structure plays little bearing on the popularity of the OS !!

16. **Operating System Structure (2):** Application programs in DOS had the power to change anything.

    While many modern OSes do not use the BIOS once started, DOS used BIOS interrupts extensively. Modern OS installs its own interrupt handlers. BIOS operates in 16-bit real mode, making it diffuclt to work with 32 and 64-bit OS.

17. **Operating System Structure (3):** Unix consists of 2 separable parts: the kernel and system programs. The kernel consists of everything below the system-call interface and above

the physical hardware. It provides the file system, CPU scheduling, memory management, and other operating-system functions. Thus, the structure is highly monolithic with a large number of functions for one level.

18. **Operating System Structure (4):**

    - The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface. Each layer defines data structures and functions to be invoked by upper-level layers.
    - Achieves information hiding since lower level routines can be implemented as seen fit as long as the interface they provide to the upper levels remains unchanged.
    - Each layer can be debugged individually, and when verifying layer n, all lower layers can be assumed correct, since they have already been debugged and verified.
    - Interdependece between layers is an issue. For example, device drivers for the backing store must be at a lower level then memory management routines, because memory management requires those drivers. However, backing store drivers need to be above CPU scheduler, since scheduler can schedule around the driver if it is waiting for I/O, however, if scheduling tables are large then they themselves might require swapping.
    - Inter-layer calls may require argument massaging at every level, and additional calls.

19. **Operating System Structure (5):** Shows layered OS with N layers. Layer 0 is hardware, layer N is the user interface. Note that most designs try to adopt a layered approach, here we are just stressing more conspicuous layers, rather than just 2 used in Unix.

20. **Operating System Structre (6):** Microkernels only provide limited process, memory management and communication facilities. The user-level programs never interact directly, but exchange messages with the kernel.

    New services are added to the user space and do not need modifications to the kernel. Its small size results in ease of porting, reliability, security.

    Tru64 Unix implements a Mach kernel, but provides a Unix user interface. Windows NT started as a microkernel but performance reasons drove it to include more within the kernel.

21. **Operating System Structure (7):**

    - Most modern operating systems implement kernel modules. Each kernel section has defined, protected interfaces like in a layered approach, but any module can talk to any other module. Thus, no restriction of only talking to the lower layer.
    - Modules can talk directly without passing messages to the kernel, hence more efficient.
    - Loadable kernel modules in Linux are loaded (and unloaded) by the modprobe command. They are located in /lib/modules and have had the extension .ko ("kernel object") since version 2.6. The lsmod command lists the loaded kernel modules.
    - The modules allow easy extension of the operating systems's capabilities as required. So, this is more flexible approach to handle the OS changes at run-time.
    - A related concept is statically Vs. dynamically linked libraries.

22. **Operating System Structure (8):** Solaris design: Organized around a core kernel, with seven types of loadable kernel modules.

23. **Operating System Structure (9):** Most modern operating systems are actually not one pure model. Hybrid combines multiple approaches to address performance, security, usability needs May contain Linux and Solaris kernels in kernel address space, so monolithic, plus

modular for dynamic loading of functionality Windows mostly monolithic, plus microkernel for different subsystem personalities.

Apple Mac OS X – hybrid, layered, Aqua UI plus Cocoa programming environment Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions) Mac OS X is a hybrid design, with a layered approach and microkernel design. The lowest layer uses the Mach microkernel, with the next layer based on BSD Unix. The top layer include application programs, GUIs, etc. and can talk directly with the microkernel or the kernel.

Android VM – Based on Linux kernel but modified to provide process, memory, device-driver management. Also adds power management.

24. **Virtual Machines:**

    - Why are we discussing VMs in this class? Because in future systems, the OS may not be in control of the computer resources, even though it may not realize it. The hypervisor will do the job of resource management, the OS will provide the user interface.
    - A multi-user OS provides the illusion that each user in the sole person operating the machine. This is not real, therefore most general-purpose OSes are actually running a virtual machine.
    - The VM running on an x86 machine can expose an interface for a PowerPC machine. Then VM will need to simulate the instructions of the PowerPC binary on the x86 machine.
    - The hypervisor provides scheduling, timer services, multiplexes hardware resources, memory management, etc.
    - In this course we will look at multiprogramming OS, and virtual memory, which are examples of virtualization. The same concept of virtualization extended to cover all aspects of a machine is called a virtual machine.

25. **Virtual Machines (2):** The virtual machine implementation is also called the hypervisor. Each process running on the hypervisor, can run its own OS. This provides complete separation between the applications running on different VMs.

26. **Virtual Machine History and Benefits:** Some VM bebefits include:

    - A virus in one OS can affect application in the VM, but should not affect the host or other VMs.
    - Applications running on different VMs communicate as if on different machines by sharing a file-system volume, or by using a virtual network.
    - Bugs in the OS can be damaging to data. So, OSes can be tested in an isolated VM environment. At the same time other users can keep using the system on the older, original OS.
    - Applications can be tested on multiple platforms without the need to maintain multiple hardware configurations.
    - Currently, a major application of VMs is server consolidation. Another big application is application portability by writing programs for a virtual machine, such as Java or MS-.Net.

27. **VMware Architecture:** VMware runs on traditional OSes such as Linux and Windows as a user application. Allows running several different guest OSes as independent virtual machines.

28. **Java Virtual Machine:** Java is a popular OO language that compiles its code to a machine-independent bytecode format in the .class files. The JVM is a specification for an abstract computer, with its own ISA. Loader loads the bytecodes and verifies their sanity, such as out-of-bound jumps, pointer arithmetic, etc. Since the ISA is different that the native ISA, we need interpretation of the bytecodes. The JVM is more than a pure interpreter, since it provides verification, memory management (garbage collection), JIT compilation, etc.

Preview Test: Homework-Ch2

Test Information

Description Homework on Chapter 2 topics

Instructions

Multiple Attempts Not allowed. This test can only be taken once.

Force Completion This test can be saved and resumed later.

QUESTION 1

1 points Saved

(True or False): The current Linux kernel (for example, on the EECS cycle servers) uses the *syscall* (rather than the *software trap)* system call implementation.

- (●) True
- ( ) False

QUESTION 2

1 points Saved

(True or False): Passing function arguments in registers is faster than passing arguments on the stack.

- (●) True
- ( ) False

QUESTION 3

1 points Saved

True or False: Each virtual machine (created by the hypervisor) can expose a virtual interface that is *identical* to that of the real underlying hardware.

- (●) True
- ( ) False

QUESTION 4

1 points Saved

Choose and write only the single upper-case letter indicated for each option (e.g., 'A', or 'B', or 'C', or 'D'; no quotes):
The API provided by the OS to users and application programs is called the A .

The options are:
A - System Calls
B - System Programs

Question Completion Status:

QUESTION 5

2 points Saved

(Choose and write only the single upper-case letter indicated for each option):
The Linux OS uses a hybrid design that combines aspects of the A and D approaches.

Options are:
A - monolithic
B - layered
C - microkernel
D - modular

QUESTION 6

2 points Saved

(Choose and write only the single upper-case letter indicated for each option):
The C is a system program that takes an assembly code file as input and outputs object code. The D is a system program that resolves external references.

Options are:
A - Preprocessor
B - Compiler
C - Assembler
D - Linker
E - Loader

QUESTION 7

2 points Saved

Write 'T' for true and 'F' for false (upper-case and no quotes): A *modular* OS design has the following properties:

(a) moves much of the functionality from kernel space to user space -- F

(b) requires modules to communicate by passing messages across the OS interface -- F

(c) allows users to add or remove kernel services dynamically -- T

(d) is not as small as a microkernel OS design -- T

QUESTION 8

2 points Saved

Write 'T' for true and 'F' for false (upper-case and no quotes):
An operation by the currently running process to get the process-id will cause the following:

(a) context switch -- F

(b) mode switch -- T

Question Completion Status:

Click Save and Submit to save and submit. Click Save All Answers to save all answers.

Save All Answers Save and Submit

Chapter 2: Operating-System Structures

- What are the services provided by an OS ?
- What are system calls ?
- What are some common categories of system calls ?
- What are the principles behind OS design & implementation ?
- What are common ways of structuring an OS ?
- How are VMs and OS related ?

Operating System Services

- Operating-system services that are helpful to the user
  - user interface – *almost all* operating systems have a user interface (UI)
    - Command-Line (CLI)
    - Graphics User Interface (GUI)
    - Touch-Screen Interface
  - program execution – load in memory and run a program
    - end execution, either normally or abnormally (indicating error)
  - I/O operations – allow interaction with I/O devices
    - provide efficiency and protection
  - file-system manipulation – provide uniform access to mass storage
    - create, delete, read, write files and directories
    - search, list file Information
    - permission management.

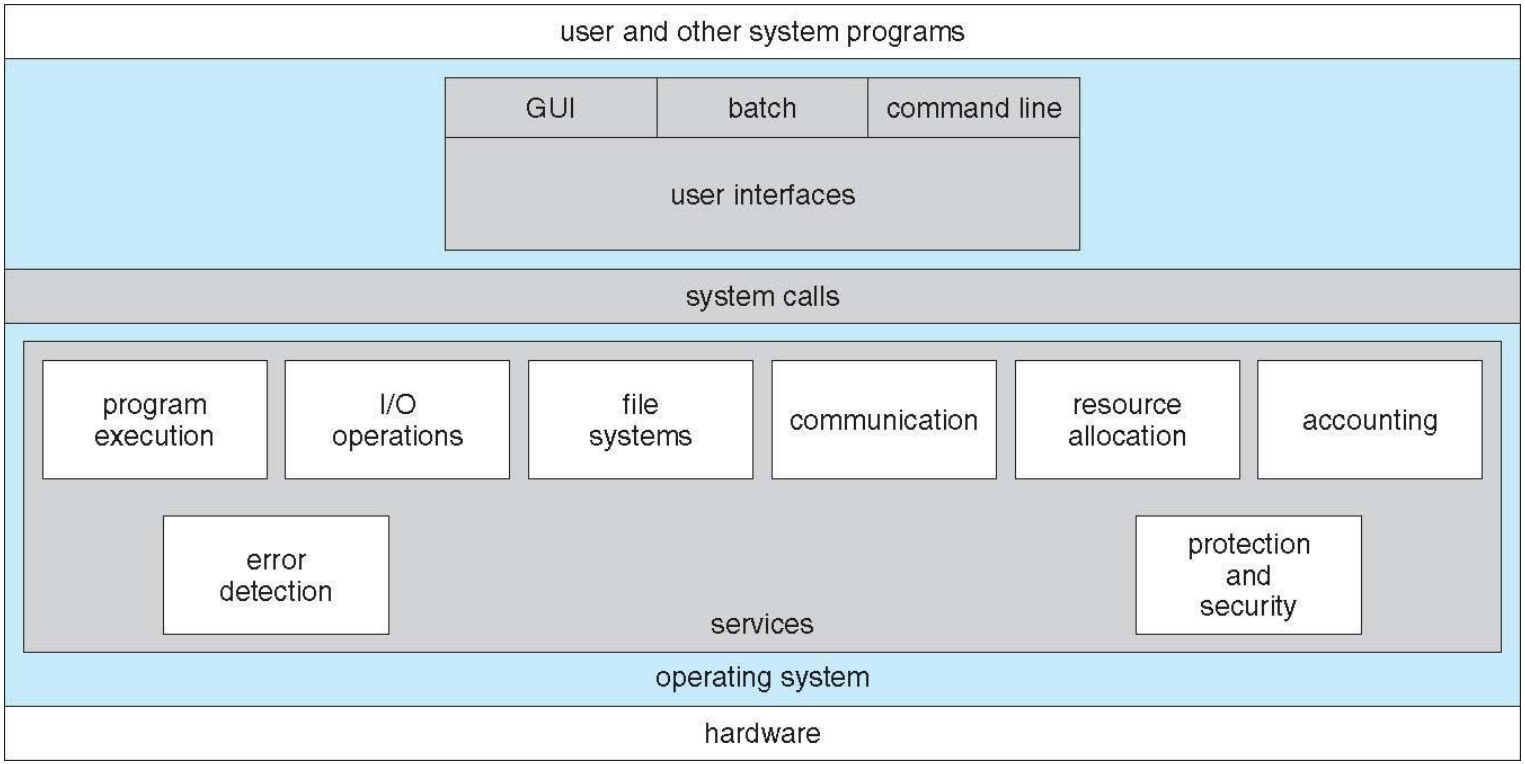
Operating System Services (2)

- Operating-system services that are helpful to the user (cont...)
  - inter-process communication – exchange information among processes
    - shared memory, POSIX *shm_open()*
    - message passing, microkernel OS, RPC, CORBA, etc.
  - error detection – awareness of possible errors
    - CPU and memory hardware (pwer failure, memory fault)
    - I/O device errors (printer out-of-paper, network connection failure)
    - user program (segmentation fault, divide-by-zero)

Operating System Services (3)

- Operating system services for efficient system operation
  - resource allocation – providing access to shared resources in multiuser system
    - CPU cycles, main memory, file storage, I/O devices
  - Accounting – keep track of system resource usage
    - for cost accounting
    - accumulating usage statistics (for profiling, etc.)
  - Protection and security – restrict access to computer resources
    - ensure that all access to system resources is controlled (*protection*)
    - protect system from outsiders (*security*)
    - user authentication, file access control, address space restrictions, etc.

A View of Operating System Services

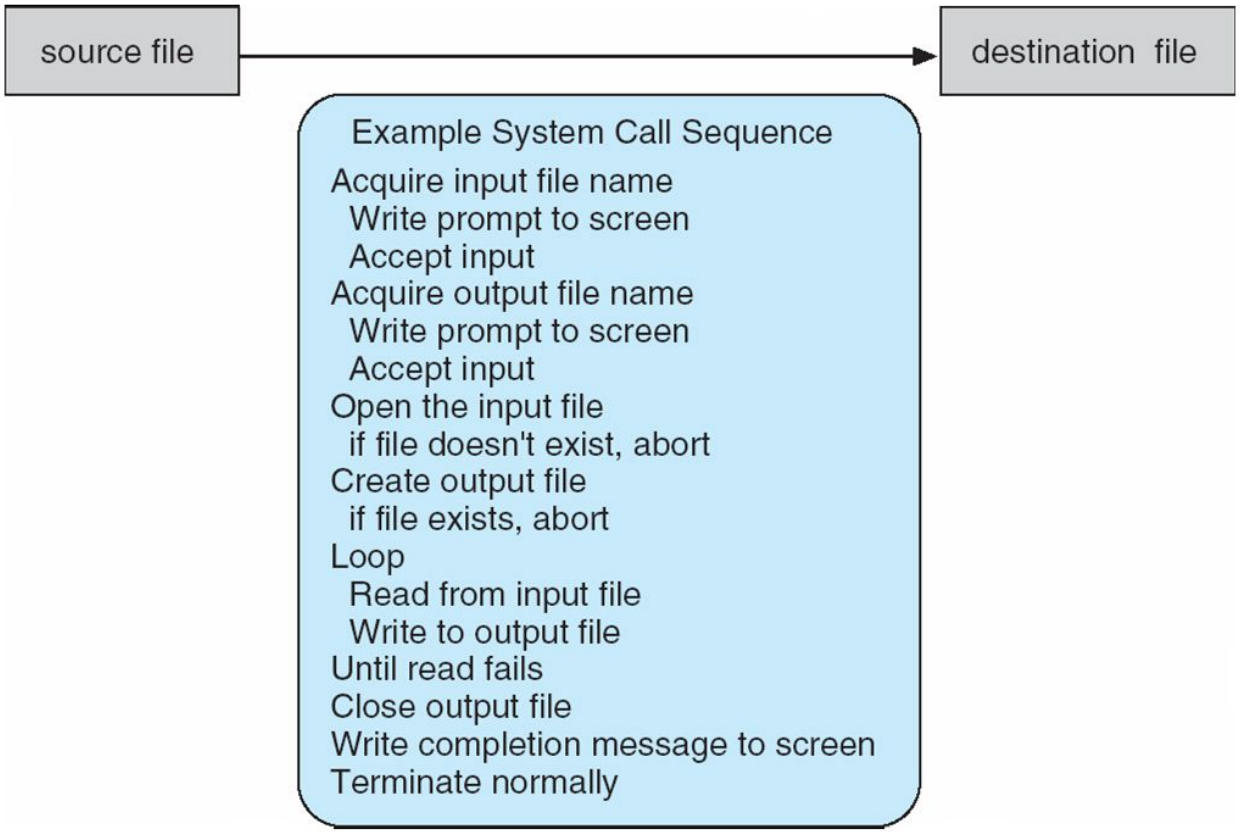


System Calls

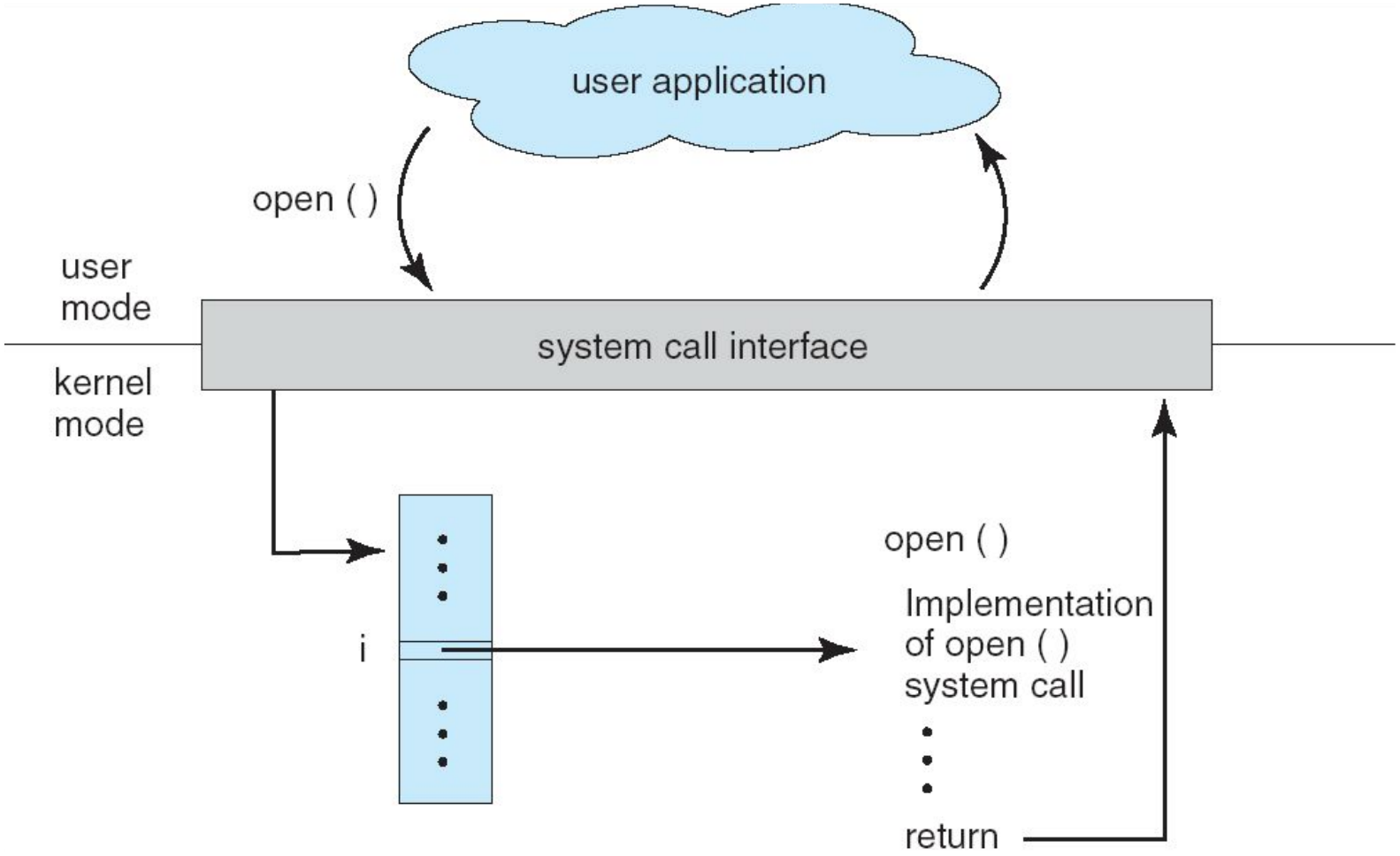
- Programming interface to the services provided by the OS
  - request privileged service from the kernel
  - typically written in a high-level *system* language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
  - provides a simpler interface to the user than the system call interface
  - reduces coupling between kernel and application, increases portability
- Common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- Implementation
  - software trap, register contains system call number
  - *syscall* instruction for fast control transfer to the kernel

Example of System Calls

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

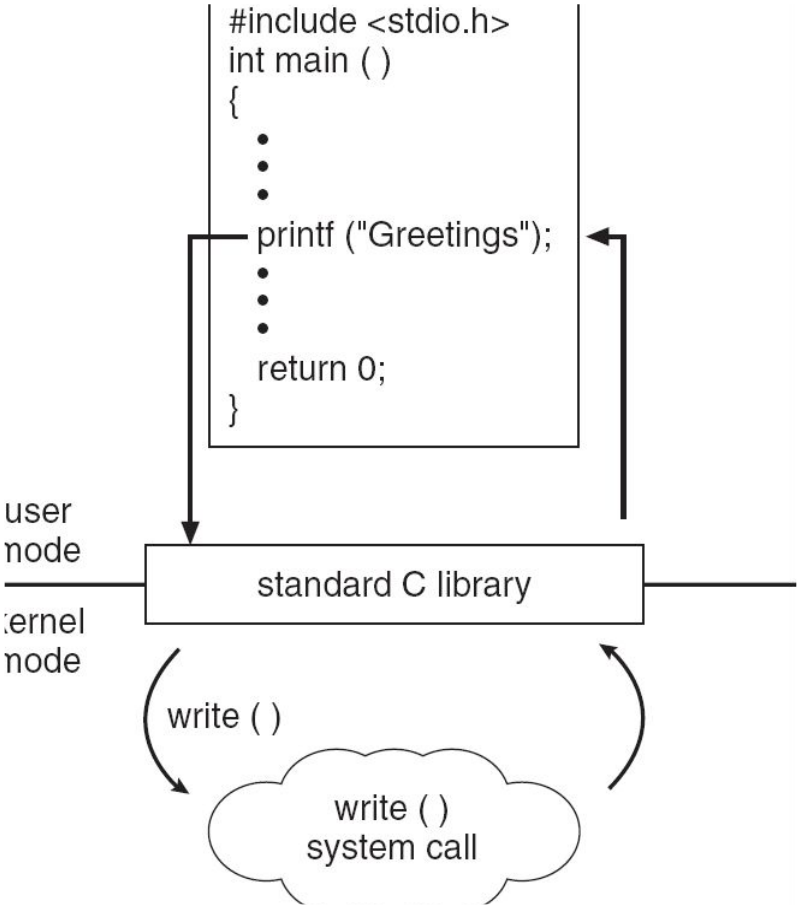


API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Pass additional information to the system call.
- Three general methods used to pass parameters to the OS
  - pass the parameters in *registers*
    - simplest, fastest
    - what if more parameters than registers ?
  - store arguments in a block on stack
    - pass stack location in a register
  - parameters *pushed* on the *stack* by the program and *popped* off the stack by the operating system
- Pure register method is hardly ever used
  - block and stack methods do not limit the number or length of parameters being passed

Types of System Calls

- Process control
  - create process, terminate process, get/set process attributes, wait event, signal event, allocate and free memory
- File management
  - create, delete, open, close, read, write a file, get/set file attributes
- Device management
  - request, release, read, write, reposition device, get/set device attributes
- Information maintenance
  - get/set time/date, get/set process/file/device attributes
- Communications
  - create/delete connection, send/receive messages
- Protection
  - set/get file/device permissions, allow/deny system resources

Examples of System Calls

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	WriteConsole()
		`ioctl()`
		`read()`
		write()

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

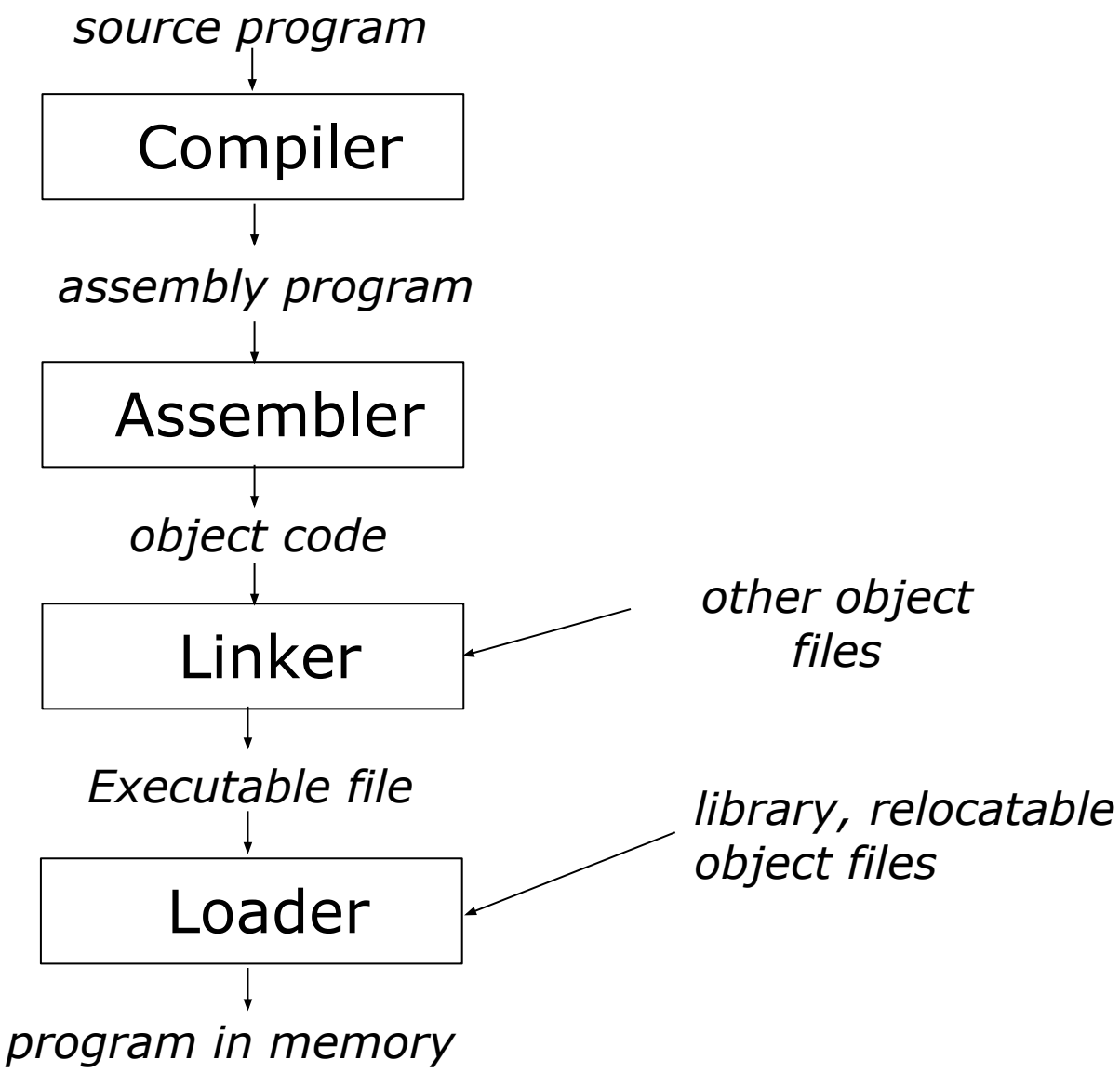
System Programs

- User-level utility programs shipped with the OS
  - ease the job of program development and execution
  - not part of the OS *kernel*
- System programs can be divided into:
  - file manipulation
  - status information
  - file modification
  - programming language support
  - program loading and execution
  - communications
  - application programs

System Programs (2)

- File management
  - mkdir, cp, rm, lpr, ls, ln, etc.
- Status information
  - date, time, ds, df, top, ps, etc.
- File modification
  - editors such as vi and emacs, find, grep, etc.
- Programming language support
  - compilers, assemblers, debuggers, such as gcc, masm, gdb, perl, java, etc.
- Program loading and execution
  - ld
- Communications
  - ssh, mail, write, ftp

Role of Linker and Loader



OS Design and Implementation

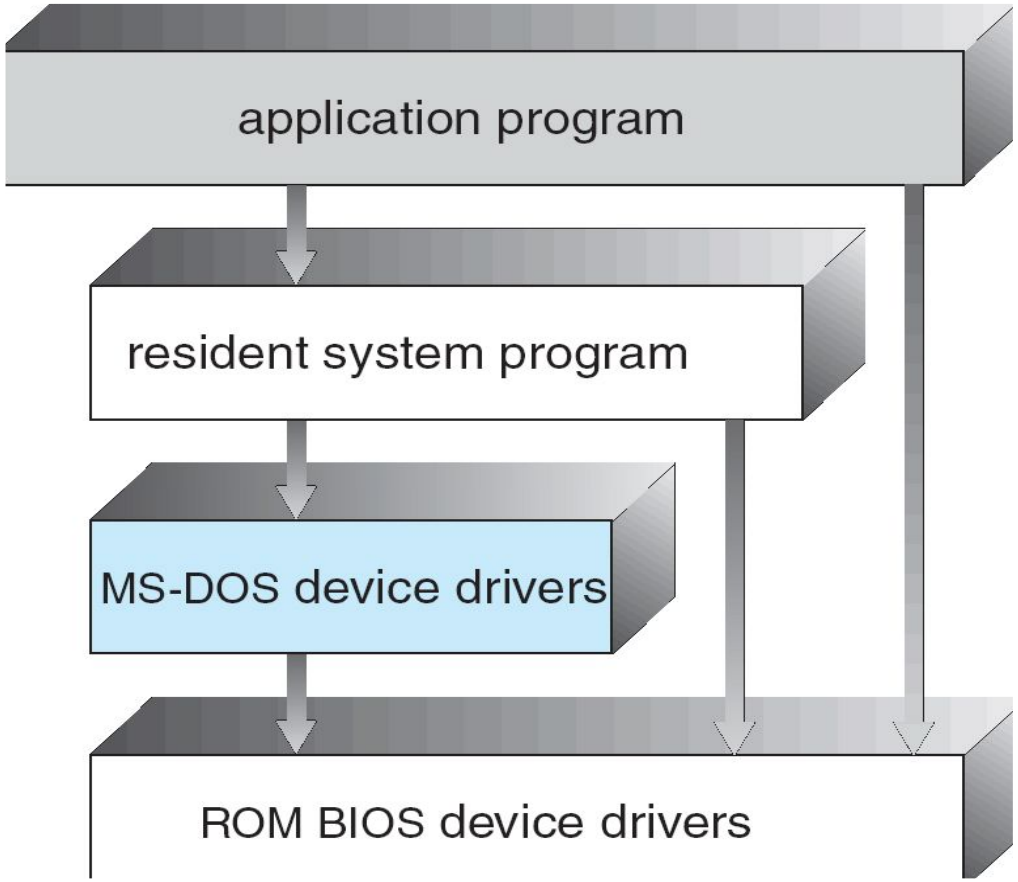
- Design
  - type of system – batch, time-shared, single/multi user, distributed, real-time, embedded
  - user goals – convenience, ease of use and learn, reliable, safe, fast
  - system goals – ease of design, implementation, and maintenance, as well as flexible, reliable, error-free, and efficient
- Mechanism
  - **policy –** what will be done?
  - **mechanism –** how to do it?
- Implementation
  - higher-level language – easier, faster to write, compact, maintainable, easy to debug, portable
  - assembly language – more efficient

Operating System Structure

- Engineering an operating system
  - modularized, maintainable, extensible, etc.
- Simple Structure
  - Characteristics
    - monolithic
    - poor separation between interfaces and levels of functionality
    - ill-suited design, difficult to maintain and extend
  - Reasons
    - growth beyond original scope and vision
    - lack of necessary hardware features during initial design
    - guided more by initial hardware constraints than by sound software engineering principles
    - eg., MS-DOS, UNIX

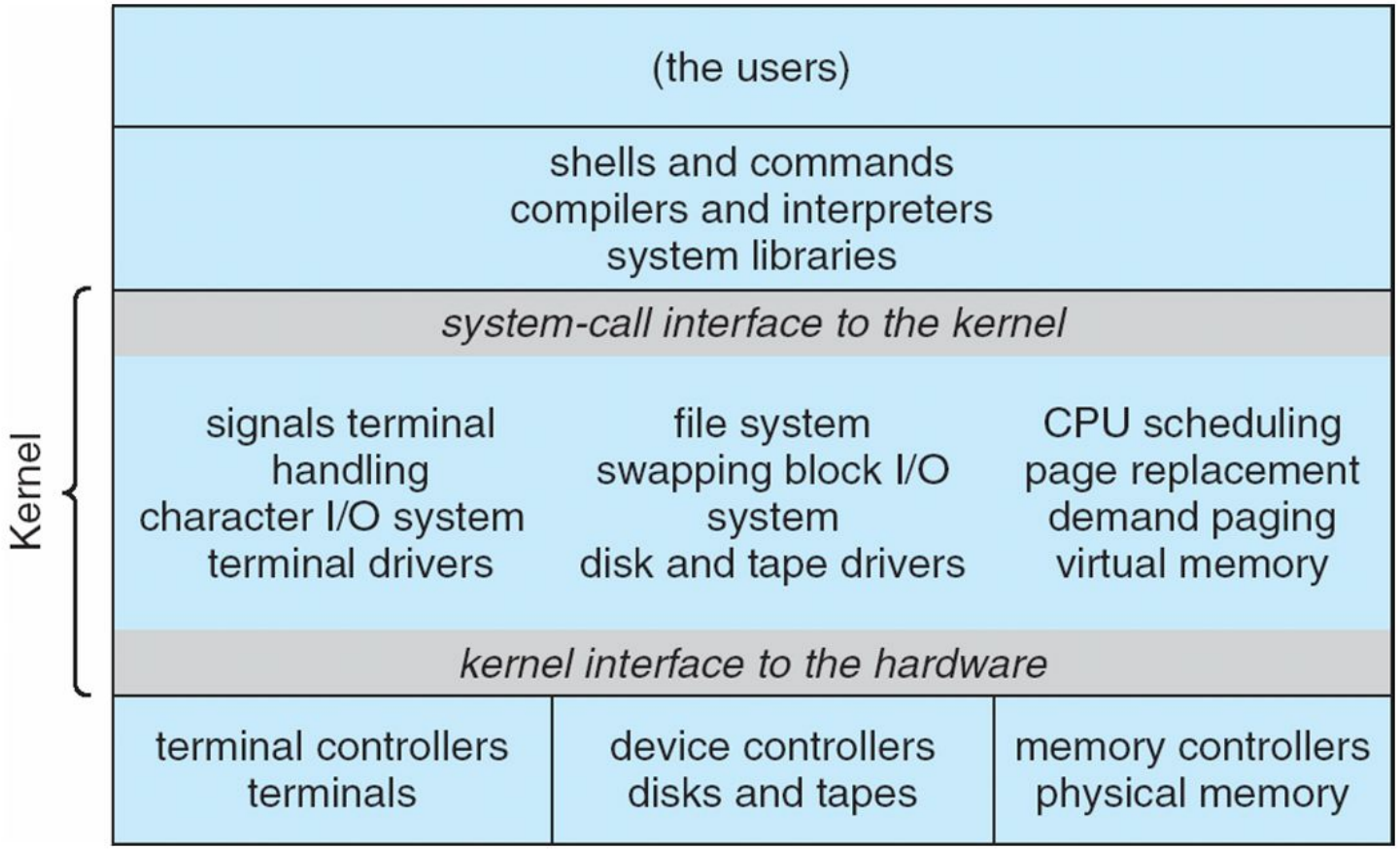
OS Structure - Monolithic

- MS-DOS layer structure:



OS Structure - Monolithic

- Traditional UNIX system structure

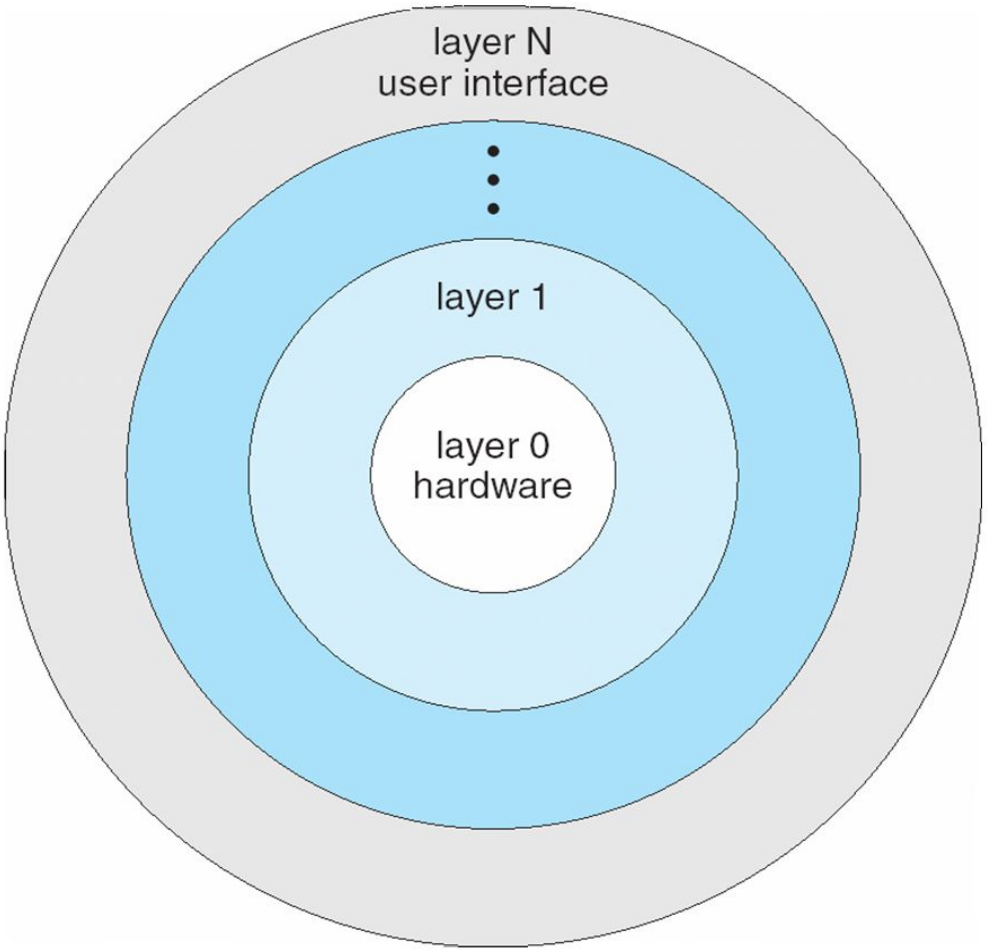


OS Structure - Layered

- Layered approach
  - OS division into a number of layers (levels)
  - upper layers use functions and services provided by lower-level layers
  - Benefits
    - more modular, extensible, and maintainable design
    - achieves information hiding
    - simple construction, debugging, and verification
  - Drawbacks
    - interdependencies make it difficult to cleanly separate functionality among layers
      - eg., backing-store drivers and CPU scheduler
    - less efficient than monolithic designs

OS Structure - Layered

- Layered Operating System

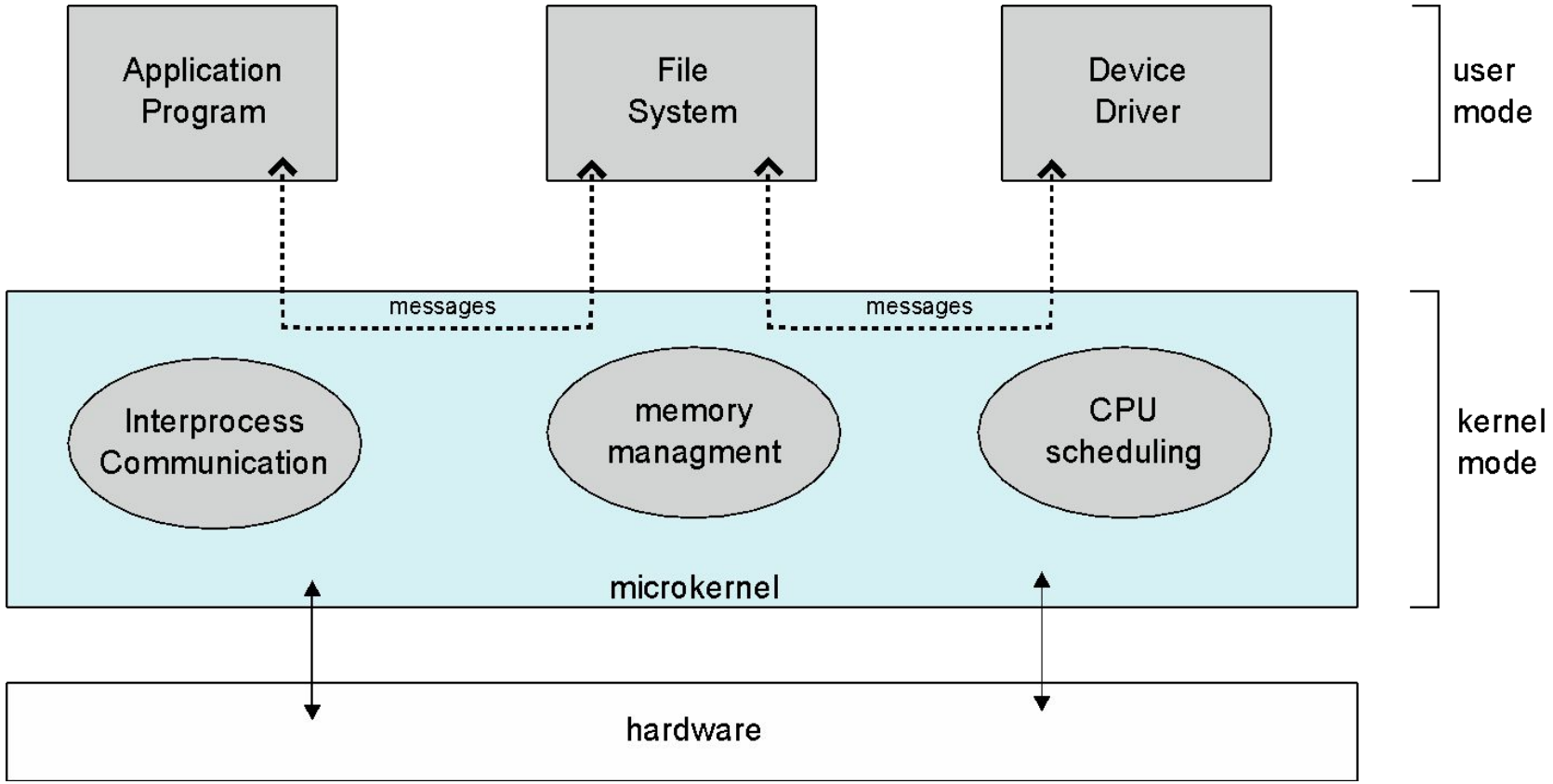


OS Structure - Microkernels

- Microkernel System Structure
  - moves as much functionality from the kernel into "*user*" space
  - communicate between user modules using message passing
  - Benefits
    - easier to extend (user level drivers)
    - easier to port to new architectures
    - more reliable (less code is running in kernel mode)
    - more secure
  - Drawbacks
    - no consensus regarding services that should remain in the kernel
    - performance overhead of user space to kernel space communication

Operating System Structure (7)

- Microkernel system structure

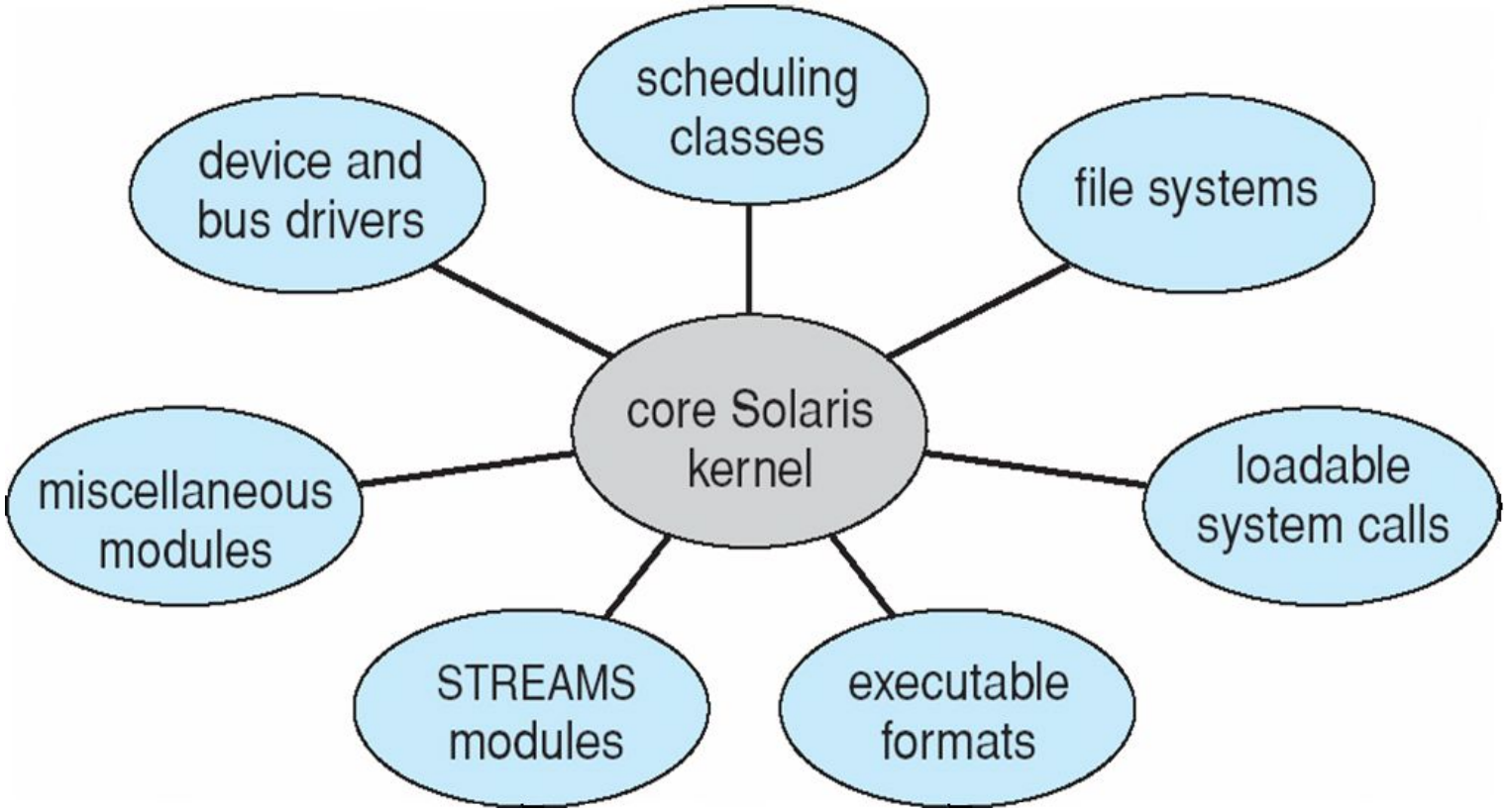


OS Structure - Modules

- Modules
  - uses object-oriented approach
  - kernel provides core functionality, like communications, device drivers
  - additional services are modules linked dynamically
  - services talk directly over interfaces bypassing the kernel
  - Benefits
    - advantages of layered structure but with more flexible
    - advantages of microkernel approach, without message passing overhead
  - Drawbacks
    - not as clean a design as the layered approach
    - not as small a kernel as a microkernel
    - but, achieves best of both worlds as far as possible

OS Structure - Modules

- Solaris modular approach

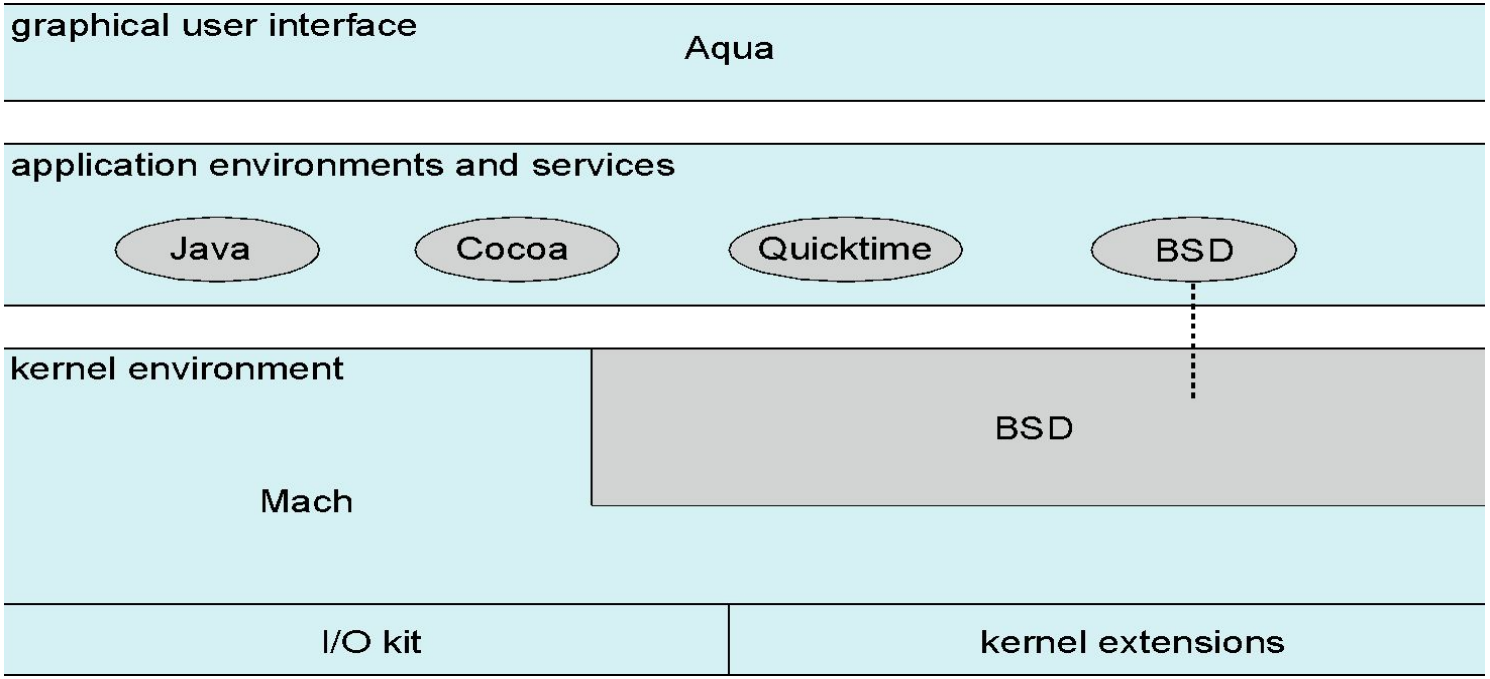


OS Structure – Hybrid Systems

- Hybrid operating systems
  - combine multiple approaches to address performance, security, usability
- Linux
  - Monolithic, since OS is in a single address space
  - Modular, since can be extended dynamically
- Windows
  - Monolithic, but some microkernel aspects
- Hybrid OS – Android OS structure
  - modified Linux kernel for process, memory, device driver management
  - Runtime provided higher-level libraries and ART runtime
  - Uses *bionic*, rather than *glibc*

OS Structure – Hybrid Systems

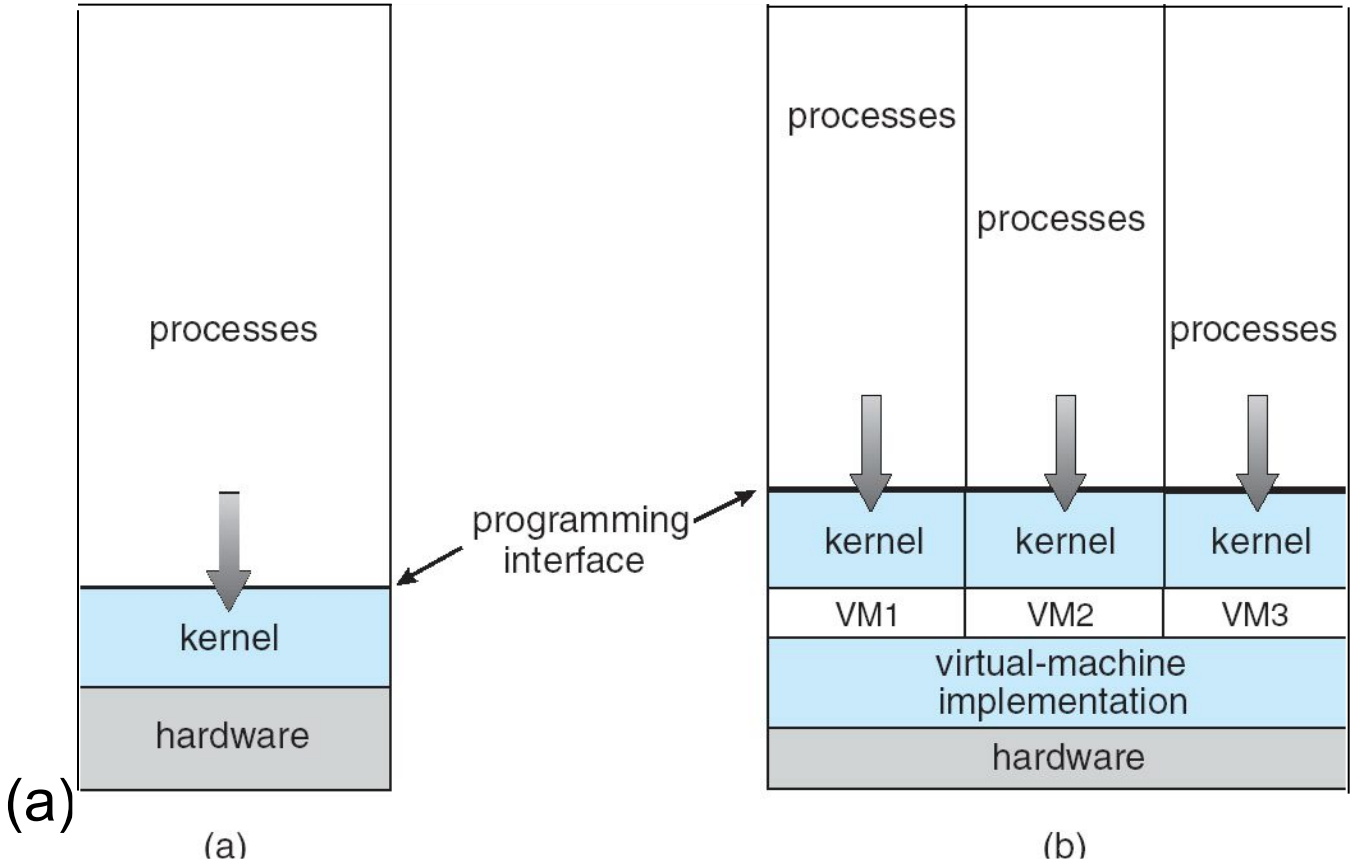
- Example – Apple Mac OS X
  - hybrid, layered
  - Mach microkernel and BSD Unix, plus I/O kit, and dynamically loadable modules for kernel extensions



Virtual Machines

- Generally, exposes a *virtual* interface different from the *physical* real
  - time sharing, multi-user OS as a virtual machine ?
  - abstraction Vs. virtualization ?
- Traditionally, exposes an interface of *some* hardware system
  - includes CPU, memory, disk, network, I/O devices, etc.
  - interface need not be identical to the underlying hardware
- A virtualization layer, called *hypervisor,* takes over control of the host hardware resources
  - creates the illusion that a process has its own computer system
  - each guest provided with a (virtual) copy of underlying computer
  - each guest process can then run another OS and application programs

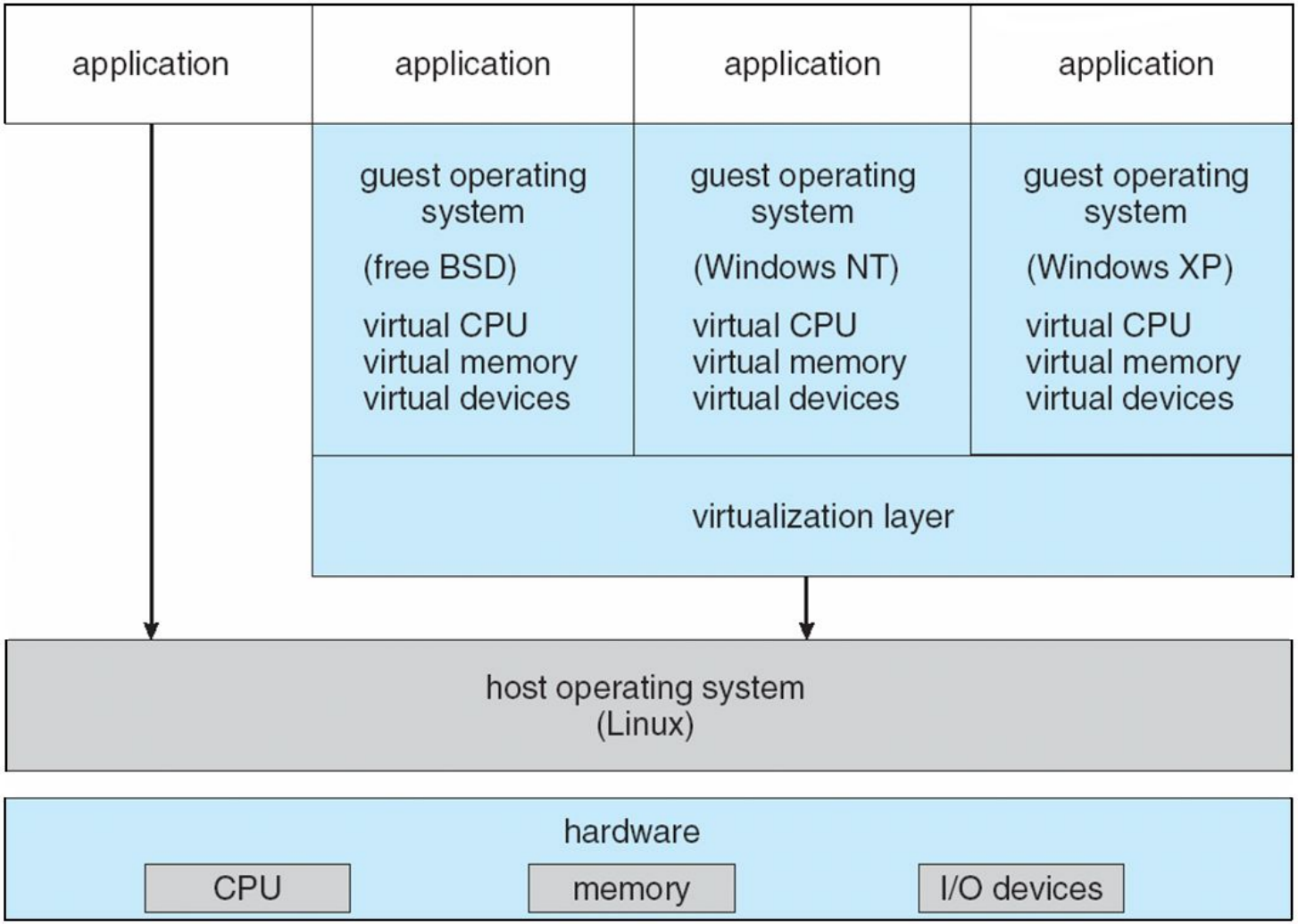
Virtual Machines (2)



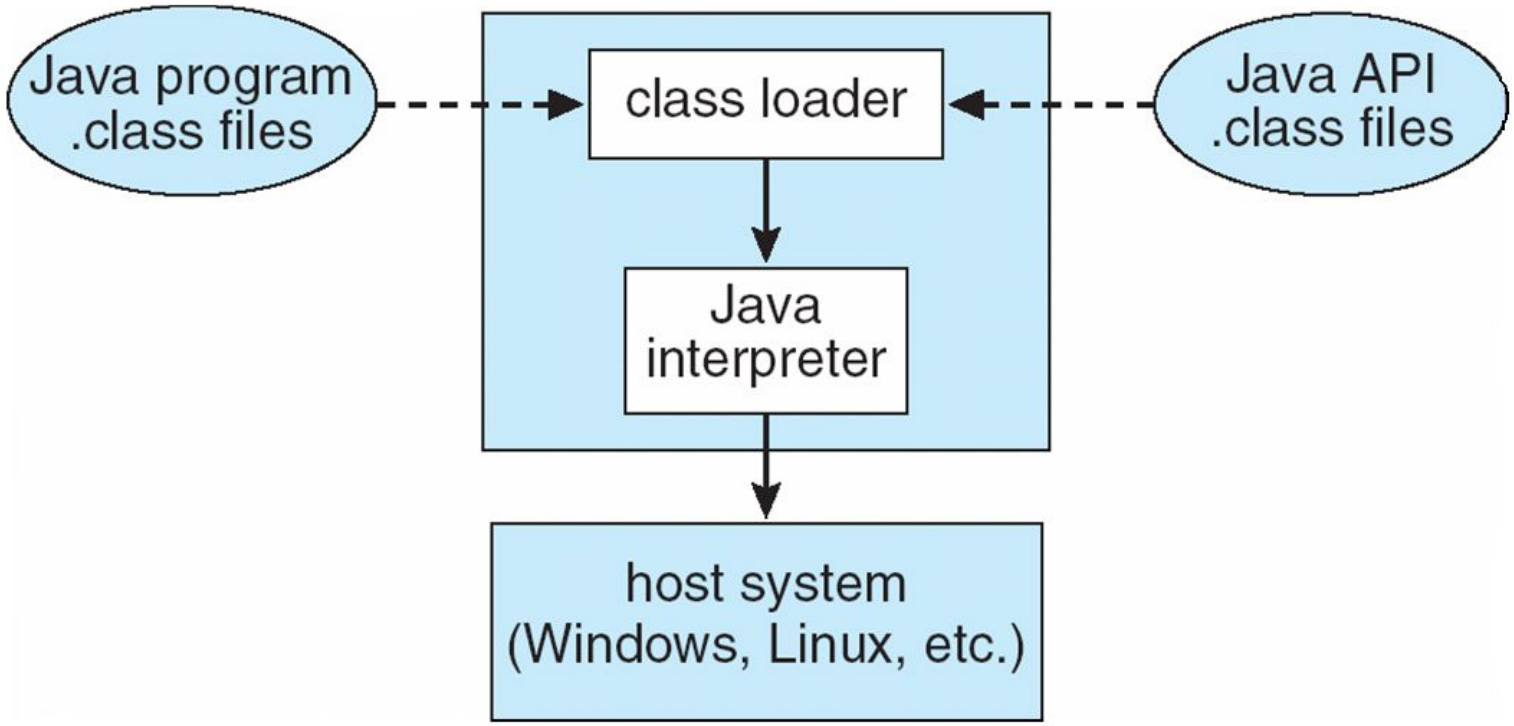
Virtual Machines History and Benefits

- History
  - introduced by IBM for their IBM 360/370 line of machines
  - exposed an interface that was identical to the underlying machine
  - ran the single-user, time-sharing CMS operating system on each VM
- Benefits
  - ability to enable multiple execution environments (different operating systems) to share the same hardware
  - application programs in different VMs *isolated* from each other
    - provides protection; can make sharing and communication difficult
  - useful for development, testing (particularly OS)
  - testing cross-platform compatibility
  - consolidation of many low-resource use systems onto fewer busier systems
  - process virtual machines (Java) provide application portability

VMware Architecture



The Java Virtual Machine



Chapter 3 - Process

1. **Chapter 3: Processes:** In this chapter we will introduce processes, and common operations performed on processes, scheduling and inter-process communication mechanisms.

2. **Process concept:** Job is the old reference for a process.

   Process is an active entity containing a program counter specifying the next instruction to be executed.

3. **Process in Memory:** Text section contains the program code. Data section holds the global variables and statics. (Initialized globals/statics in data section, uninitialized in .bss section). Dynamically allocated data is maintained in the heap. Function activation records (local variables and some other information) are maintained on the stack. Heap and stack grow towards each other.

   All these sections exist in memory in what is called the address space of a process. The OS maintains a PCB containing information about each process.

   We can use the 'size' command to get sizes of various sections in an executable.

   Question: Why does it not show the sizes of stack and heap?

4. **Process State:** OS maintains a process state in a time-shared system. A process may wait for I/O to be completed or some other event to happen. A time-shared OS can then schedule some other ready process to run while this process is waiting.

5. **Diagram of a Process State:** A process may transition between different states during its execution.

6. **Process Control Block:** PCB contains all the information required to swap out a running process, and restart execution of a process that was waiting for some event. It is a repository for any process-specific information.

   The OS needs to hold at least the following types of information in the PCB.
   PC (program counter) – to resume execution.
   Registers – CPU only contains one set of registers, time-sharing systems may run multiple processes.
   Scheduling info priority or other parameters based on scheduling algorithm used.
   Memory management may include values of base and limit registers, page table, segment table information, etc.
   Accounting info includes amount of CPU and real time used, time limit, job or process numbers, etc.
   I/O information includes list of I/O devices allocated to the process, list of open files, etc.

7. **Process Control Block:** Figure shows some fields of a PCB.

8. **CPU Switch from Process to Process:** The PCB holds all necessary process state information to allow this switch. Note that the time spent in saving and restoring process state information in the PCB is OS overhead during which no useful work is done. All OS try to minimize this overhead.

   Some processors may be able to hold the state of multiple processes simultaneously.

9. **Process Scheduling:** Scheduling is only necessary in the presence on multiple processes in a multiprogrammed or time-shared system. Remember that even if there is only user program running, there may be several other OS programs running at the same time.

   Each device may have its own device queue. Process migration can be explained from the next figure.

10. **Ready Queue and Various I/O device queues:** Queues are generally linked lists, which makes managing the lists easy, since processes frequently move between various queues. Only one process in the ready queue may be allocated to the CPU at any instance.

    If a running process quits, waits, or is interrupted (like by OS timer), it may move to one of the device queues or go back to the end of the ready queue.

11. **Representation of Process Scheduling:** This figure is called a *queuing diagram.* Each rectangular box represents a queue. Circles represent the resources that represent a queue. Arrows indicate the flow of processes in the system.

    A running process could:
    1. issue an I/O request and migrate to the I/O queue.
    2. process could voluntarily wait (maybe for child's completion)
    3. scheduler should forcibly remove process from the CPU.

12. **Schedulers:** Many general-purpose OSes including Unix and MS-Windows do not use multiple schedulers. They often do nothing or depend on some system-defined constant to limit the number of active processes in the system at any one point in time.

    I/O-bound process spends more time doing I/O than computations, many short CPU bursts. CPU-bound process spends more time doing computations; few very long CPU bursts.

    Getting the right mix of ready CPU-bound and I/O bound processes is important: if all I/O bound then ready queue will always be empty, if all CPU-bound then I/O queue will be empty and devices will go under-utilized.

    The main difference between the two schedulers is the frequency of invocation, which determines the algorithm used in the two schedulers.

13. **Context Switch:** Context switch happens frequently on most machines today. Therefore, it is important to limit overhead of the switch. A mode switch may not necessarily involve a context switch.

    Context switch overhead depends on the machine (memory speed, number of registers that must be copied, existence of special instructions, etc.) SPARC can hold the states of multiple processes in memory simultaneously.

    Check out the output of the '`time`' command that shows 'user' and 'system' time for each process. These times are spent in different modes, but in the same process context.

14. **Process Creation:** Each OS starts a primordial process during system bootup. This is the *init* process on Unix.

    A process needs resources like CPU time, memory, files, I/O devices, to accomplish its work. Restricting the child to a subset of the parents resources may limit the parent creating excess processes. On Unix, each child gets its own resources from the OS, file and I/O descriptors are copied to the child address space from the parent.

    In Unix parent and child start executing concurrently, but parent can wait for the child to finish execution by explicitly calling the '`wait`' system call.

    On Unix the child gets a duplicate of the parent address space.

15. **Process Creation Example on Unix:** The child process overlays its address space with the Unix command /bin/ls, using a version of the exec() system call.

16. **Process Creation:** Figure shows parent waiting for child to terminate, as in the example before. See fork.c

17. **Process Termination:** If the parent terminates before the child, then the child becomes a child of the init process under Unix. Thus parent exiting does not terminate the child, and init can collect its return status.

18. **Interprocess Communication:** A process is independent if it does not affect and cannot be affected by any other process in the system. If the process affects or is affected by other processes in the system, then the process is called co-operating process.

    Remember this is communication within the same OS between processes active at the same time.

    Use case scenarios for IPC:
    1. Several users may be interested in the same piece of information, like a shared file.
    2. Speedup can be increased with parallel computation, breaking tasks into parallel subtasks. Any such speedup can only be achieved if there are multiple CPUs in the system.
    3. Modularity by dividing the system functions into separate processes. User working on several tasks at the same time, such as editing, printing, compiling, may find if easier to divide work into multiple processes and share information among them.
    4. Convenience – instead of computing the same data in two processes, they can use IPC.

19. **Producer Consumer Problem:** The producer consumer problem can be addressed using various forms of inter-process communication using shared memory. It is a very common type of abstraction. The unbounded buffer cannot be implemented in most systems today.

20. **Models of IPC:** There are two primary IPC models:

    - Shared Memory: Note that this requires over-riding the address space security mechanisms setup by the OS to keep processes separate. Communication is faster since system calls are only required during the initial stage of setting up the shared region. Later, there is direct communication. Accessing shared memory is similar to accessing normal local memory, so is more convenient. With shared memory, both processes writing to the same shared region at the same time, will overwrite each other. So, synchronization may be required.
    - Message Passing: There is more overhead, since messages are copied from the sender to the kernel, and then to the receiver address space. So, sending small amounts of data is preferred. For the same reason, communication is slower, since system calls are needed for each message sent and received. Messages do not overwrite each other, so their is less concern about synchronization.

21. **Models of IPC (2):** Figure shows a schematic.

    - Shared memory: Shared region is generally created in the address space of the process initiating the region. Other interested processes must attach this region to their address space. The form of data and the location in the address space are determined by the communicating processes and not by the OS. The OS removes the restrictions on address space sharing. The processes are also responsible for synchronizing accesses, a very important topic that we will study later as well.

- Message Passing: We can see that the data is copied from the sender to the kernel to the receiver. This copying along with the explicit system calls for each data transfer, makes this approach more expensive than shared memory for processes on the same system. However, this technique is important in distributed environments where the communicating processes may reside on different computers, and thus cannot share address spaces.

22. **Message Passing:** Can be used for communication between non-local processes. The link can be implemented in a variety of ways, including shared memory, hardware bus, network, etc. These implementation issues vary according to the system.

    We will next look at naming techniques to see how links can be established; links can be established first before all communication (like TCP), or for each message sent/received (like UDP)? Link between more than 2 processes, such as one sender and multiple receivers, can create synchronization problems. Any receiver may be able to pick up the message sent by the sender.

    Number of links depends on naming scheme for the links, and system implementation. Link capacity can either be specified by the creator of the link, or can be a system-defined constant.

    Fixed size messages makes it easier for system implementation, but makes it harder for the programmer, unless the link creator makes large links to accommodate all messages. Conversely true for variable size messages.

23. **Message Passing – Naming:** The process only needs to know the identity of the other process to communicate. There is also a scheme with asymmetric naming, such that the sender specifies the receiver queue, but the receiver does not. Then, multiple processes can send to the same queue.

    With hard-coding the process id, we are in trouble if the process id changes. Therefore, we can use the other indirect naming scheme.

24. **Message Passing - Naming (2):** Without synchronization, the multiple receiver problem may be handled by:
    1. Allow a link to be associated with at most two processes
    2. Allow only one process at a time to execute a receive operation
    3. Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

    Mailbox in the process space has a default owner, and mailbox is deleted when process terminates. If mailbox in OS, then may have no owner. Explicit deletion of mailbox may be necessary to release resources.

25. **Message Passing (3):** Synchronization:
    Different combinations of send and receive are possible. When both are blocking, synchronization problem disappears, since send() sends a message and waits until the receive() gets it. Similarly, the receiver does not proceed until it gets a message. The '`select`' system call can be used to implement a non-blocking receive.

    Sent messages reside in a temporary queue. Zero capacity – sender must be blocking.

26. **Interprocess Communication in Linux:** Unix and its many variants provide several mechanisms for interprocess communication.

27. **Pipes**

- Pipes are the earliest form on IPC in Unix systems. They are also the simplest.
- Two-way communication may be achieved via two pipes.
- Ordinary pipes in Unix, also called anonymous pipe, can only be setup between parent and child processes. Another mechanism, called named pipes, that we study later, gets rid of this limitation.
- Since pipes can only be setup between parent-child, inter-computer communication is not allowed.
- If process exits before collecting the data, then it will be lost.
- Data only collected in the order it is entered. We will later study message queues to relax this limitation.

28. **Simple Example Using Pipes:** Pipe is constructed in the same process. The standard Unix system calls, read and write, are used for communication with pipes. The pipe is destroyed when the process ends.

29. **IPC Example Using Pipes:** This works because child copies the entire parent address space. So, file descriptors fd[0] and fd[1] are copied in the child space as well.

30. **Pipes Used for Process Synchronization:** A useful property of pipes is that the read is blocking, until something is written in the write end of the pipe. This property can be used to implement a naive synchronization mechanism.

31. **Pipes Used in Unix Shells:** Unix command-line shells provide several small system programs. A common activity is constructing more complex commands from these simple commands. Pipes are an irreplaceable tool to achieve this.

    ps is used to display status of processes running on the system.
    -e option is used to display all processes.
    -f is used to print additional process status information.
    By itself the screen scrolls. So, *more* is used to view command output one screen at a time. Output of ps is input to more.

32. **Pipes Used in Unix Shells (2):** *dup* is used to duplicate the specified file descriptor. The duplicated descriptor takes on the lowest file descriptor number available. We can also use dup2 to explicitly specify the new file descriptor number.

    Pipe will also be covered during Lab-4.

33. **FIFO (Named Pipes):**

    - Named pipes attempt to over come some limitations of normal pipes.
    - Several process can use the FIFO for communication, it appears in the file system as any other file.
    - If a process opens the FIFO file for reading, it is blocked until another process opens the file for writing. The same goes the other way around.
    - Reading from an empty FIFO does not return a EOF value.
    - FIFOs consume system resources, and must be explicitly removed.
    - FIFOs are half-duplex; file cannot be opened in read/write mode.

34. **Producer Consumer Example with FIFO:** Producer Code: The *open()* blocks on until someone opens the same fifo file for reading.

    The second argument to open specify the file access permissions. A FIFO can also be created on the command line using the mknod() command. Write to the fifo from the command line using standard file commands.

35. **Producer Consumer Example with FIFO (2):** Consumer Code: Consumer code is similar. If producer quits, then read() returns 0. Thus, reader can tell when all writers have closed. If there are multiple writers then read() does not return a zero unless all writers have closed!

    We can have multiple readers and writers for the same FIFO. Any communication then will need to be synchronized. There is no way using FIFOs to specify that a message is for some specific reader. Messages can only be read in FIFO order.

36. **Message Queues:** Message queues provide a way to specify a message type, and messages of only that kind can be removed from the queue in FIFO order, even if there are messages of some other type ahead in the queue.

37. **Message Queue Example:** *key* is specified to be 0. It is just a long integer. It is unique for each queue.

    IPC_CREAT is or'ed with the permissions on the queue. Queue length is initialized by the system, but can be reset using msgctl().

    As long as the first word in the buffer is a long, the rest of the buffer can contain anything of any length. In our example program, we have a character array of 1000 bytes.

    mtype is the type of the message. This can be used during msgrcv() to only retrieve messages of a particular type from the queue in a FIFO order. msgsnd() can block if there is not enough space in the queue.

    We specify 0 as fourth argument in msgrcv(), so that we get the first message from the queue.

    If we don't terminate the queue then it will exist even after process termination, and even if no process uses it, and consume system resources. We can view present queues using ipcs command.

38. **Message Passing in Unix Systems:** This slide answers some of the questions regarding message passing that we had posed earlier, in the context of Linux message queues.

39. **Memory Sharing in Unix:** Processes can also communicate using *Shared memory.*

40. **Shared Memory Example:** This is a naive example showing how a single process uses shared memory on Linux.

    For real IPC, multiple processes may share the shared region which will cause race conditions. So, some kind of synchronization mechanism will be necessary. Therefore, we delay such an example until we learn about process synchronization.

    `shmat()` returns a void pointer, and here we are casting it to a char pointer.

41. **Shared Memory Example (2):** The first argument to shmget is again the key (which can get using `ftok()`). It can also be specified to IPC_PRIVATE for a always new anonymous memory segment.

Access to file region is as for file access for user, group, and others. Linux does not implement execute permissions.

We can specify other commands as well with `shmctl()`. (Read man page for `shmtl()`).

ipcs command can be used to check the status of the message queues and shared memory segments in the system.

42. **Unix Domain Sockets:** Similar to pipes, but allows full-duplex communication. Sockets are most commonly used on the Internet. However, here we will first study Unix domain sockets.

    Like pipes, sockets are a special file in the file system. However, the communication does not use the file interface. Thus, we won't use open(), close(), read(), write() system calls.

    Unix domain sockets are very similar to Internet sockets, with minor differences.

    Main reason for using connectionless sockets is speed. Setting up a connection, and providing for error-free communication involves a lot of overhead. Connectionless communication, which may lose some packets, may be enough for some activities, like streaming video and audio, Even ftp uses UDP, but has its own higher level protocol providing error-free communication, maintaining the same order of data sent.

43. **Unix Domain Sockets – System Calls:** `Socket()`: domain can be used to specify local communication, or a host of protocols like IPv4, IPv6, Novell IPX, etc.
    'type' can be used to specify sequences, reliable, connection-oriented SOCK_STREAM, or SOCK_DGRAM, etc.
    'Protocol' is mostly 0, but can be used if the socket type supports more than one protocol.

    `bind()`: bind socket to a unique address in the Unix domain, a special type of file.

    `listen()`: If there are this many connections waiting to be accepted, additional clients will generate the error ECONNREFUSED.

    `accept()`: The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket.
    The argument '`addr`' is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer.
    The '`addrlen`' argument is a value-result argument. It should initially contain the size of the structure pointed to by addr; on return it will contain the actual length (in bytes) of the address returned.

44. **Socket Example – Echo Server:** See the examples in the notes.

45. **Communications in Client Server Systems:** All the interprocess communication mechanisms we have seen only allow communication on the same system. We might need to share information between processes in a distributed environment, where multiple computers are linked using a network. Client-server mechanisms help achieve this. They are most commonly based on a message passing paradigm.

    Sockets can use several standard protocols, as we have just seen, for their communication. Further examples using Sockets over Internet will be discussed in the Labs. The book has an example of using Sockets in Java.

46. **Sockets:** Instead of using the file interface, as done by pipes, socket communications use the socket interface.

47. **Remote Procedure Calls:** RPC is built on top of an IPC mechanism that we have seen earlier. Client and server typically do not share an address space, so message-passing is used.

    Messages in common IPC mechanisms may be just bare packets of data. RPC daemon listens to a particular port for requests. Port is simply a number denoting the procedure to be invoked.

    Sever kernel may provide a rendezvous mechanism to locate the correct port on the server. This rendezvous daemon is also called a matchmaker. Client queries this matchmaker daemon.

    Message received by the server will include the service name, and the associated parameters.

    Marshalling is required due to differing architectures on the client and the server. One particularly annoying difference is the architecture byte-order (big-endian or little-endian). RPCs use a standard machine-independent representation, such as external data representation (XDR).

48. **Marshalling Parameters:** Stub is the client-side routine converting all data to a machine-neutral representation. Skeleton converts the marshalled arguments into machine-specific format for the server. The return value may be marshalled again by the skeleton before communicating it to the client.

Chapter 3 - Process

1. **Chapter 3: Processes:** In this chapter we will introduce processes, and common operations performed on processes, scheduling and inter-process communication mechanisms.

2. **Process concept:** Job is the old reference for a process.

   Process is an active entity containing a program counter specifying the next instruction to be executed.

3. **Process in Memory:** Text section contains the program code. Data section holds the global variables and statics. (Initialized globals/statics in data section, uninitialized in .bss section). Dynamically allocated data is maintained in the heap. Function activation records (local variables and some other information) are maintained on the stack. Heap and stack grow towards each other.

   All these sections exist in memory in what is called the address space of a process. The OS maintains a PCB containing information about each process.

   We can use the 'size' command to get sizes of various sections in an executable.

   Question: Why does it not show the sizes of stack and heap?

4. **Process State:** OS maintains a process state in a time-shared system. A process may wait for I/O to be completed or some other event to happen. A time-shared OS can then schedule some other ready process to run while this process is waiting.

5. **Diagram of a Process State:** A process may transition between different states during its execution.

6. **Process Control Block:** PCB contains all the information required to swap out a running process, and restart execution of a process that was waiting for some event. It is a repository for any process-specific information.

   The OS needs to hold at least the following types of information in the PCB.
   PC (program counter) – to resume execution.
   Registers – CPU only contains one set of registers, time-sharing systems may run multiple processes.
   Scheduling info priority or other parameters based on scheduling algorithm used.
   Memory management may include values of base and limit registers, page table, segment table information, etc.
   Accounting info includes amount of CPU and real time used, time limit, job or process numbers, etc.
   I/O information includes list of I/O devices allocated to the process, list of open files, etc.

7. **Process Control Block:** Figure shows some fields of a PCB.

8. **CPU Switch from Process to Process:** The PCB holds all necessary process state information to allow this switch. Note that the time spent in saving and restoring process state information in the PCB is OS overhead during which no useful work is done. All OS try to minimize this overhead.

   Some processors may be able to hold the state of multiple processes simultaneously.

9. **Process Scheduling:** Scheduling is only necessary in the presence on multiple processes in a multiprogrammed or time-shared system. Remember that even if there is only user program running, there may be several other OS programs running at the same time.

   Each device may have its own device queue. Process migration can be explained from the next figure.

10. **Ready Queue and Various I/O device queues:** Queues are generally linked lists, which makes managing the lists easy, since processes frequently move between various queues. Only one process in the ready queue may be allocated to the CPU at any instance.

    If a running process quits, waits, or is interrupted (like by OS timer), it may move to one of the device queues or go back to the end of the ready queue.

11. **Representation of Process Scheduling:** This figure is called a *queuing diagram.* Each rectangular box represents a queue. Circles represent the resources that represent a queue. Arrows indicate the flow of processes in the system.

    A running process could:
    1. issue an I/O request and migrate to the I/O queue.
    2. process could voluntarily wait (maybe for child's completion)
    3. scheduler should forcibly remove process from the CPU.

12. **Schedulers:** Many general-purpose OSes including Unix and MS-Windows do not use multiple schedulers. They often do nothing or depend on some system-defined constant to limit the number of active processes in the system at any one point in time.

    I/O-bound process spends more time doing I/O than computations, many short CPU bursts. CPU-bound process spends more time doing computations; few very long CPU bursts.

    Getting the right mix of ready CPU-bound and I/O bound processes is important: if all I/O bound then ready queue will always be empty, if all CPU-bound then I/O queue will be empty and devices will go under-utilized.

    The main difference between the two schedulers is the frequency of invocation, which determines the algorithm used in the two schedulers.

13. **Context Switch:** Context switch happens frequently on most machines today. Therefore, it is important to limit overhead of the switch. A mode switch may not necessarily involve a context switch.

    Context switch overhead depends on the machine (memory speed, number of registers that must be copied, existence of special instructions, etc.) SPARC can hold the states of multiple processes in memory simultaneously.

    Check out the output of the '`time`' command that shows 'user' and 'system' time for each process. These times are spent in different modes, but in the same process context.

14. **Process Creation:** Each OS starts a primordial process during system bootup. This is the *init* process on Unix.

    A process needs resources like CPU time, memory, files, I/O devices, to accomplish its work. Restricting the child to a subset of the parents resources may limit the parent creating excess processes. On Unix, each child gets its own resources from the OS, file and I/O descriptors are copied to the child address space from the parent.

    In Unix parent and child start executing concurrently, but parent can wait for the child to finish execution by explicitly calling the '`wait`' system call.

    On Unix the child gets a duplicate of the parent address space.

15. **Process Creation Example on Unix:** The child process overlays its address space with the Unix command /bin/ls, using a version of the exec() system call.

16. **Process Creation:** Figure shows parent waiting for child to terminate, as in the example before. See fork.c

17. **Process Termination:** If the parent terminates before the child, then the child becomes a child of the init process under Unix. Thus parent exiting does not terminate the child, and init can collect its return status.

18. **Interprocess Communication:** A process is independent if it does not affect and cannot be affected by any other process in the system. If the process affects or is affected by other processes in the system, then the process is called co-operating process.

    Remember this is communication within the same OS between processes active at the same time.

    Use case scenarios for IPC:
    1. Several users may be interested in the same piece of information, like a shared file.
    2. Speedup can be increased with parallel computation, breaking tasks into parallel subtasks. Any such speedup can only be achieved if there are multiple CPUs in the system.
    3. Modularity by dividing the system functions into separate processes. User working on several tasks at the same time, such as editing, printing, compiling, may find if easier to divide work into multiple processes and share information among them.
    4. Convenience – instead of computing the same data in two processes, they can use IPC.

19. **Producer Consumer Problem:** The producer consumer problem can be addressed using various forms of inter-process communication using shared memory. It is a very common type of abstraction. The unbounded buffer cannot be implemented in most systems today.

20. **Models of IPC:** There are two primary IPC models:

    - Shared Memory: Note that this requires over-riding the address space security mechanisms setup by the OS to keep processes separate. Communication is faster since system calls are only required during the initial stage of setting up the shared region. Later, there is direct communication. Accessing shared memory is similar to accessing normal local memory, so is more convenient. With shared memory, both processes writing to the same shared region at the same time, will overwrite each other. So, synchronization may be required.
    - Message Passing: There is more overhead, since messages are copied from the sender to the kernel, and then to the receiver address space. So, sending small amounts of data is preferred. For the same reason, communication is slower, since system calls are needed for each message sent and received. Messages do not overwrite each other, so their is less concern about synchronization.

21. **Models of IPC (2):** Figure shows a schematic.

    - Shared memory: Shared region is generally created in the address space of the process initiating the region. Other interested processes must attach this region to their address space. The form of data and the location in the address space are determined by the communicating processes and not by the OS. The OS removes the restrictions on address space sharing. The processes are also responsible for synchronizing accesses, a very important topic that we will study later as well.

- Message Passing: We can see that the data is copied from the sender to the kernel to the receiver. This copying along with the explicit system calls for each data transfer, makes this approach more expensive than shared memory for processes on the same system. However, this technique is important in distributed environments where the communicating processes may reside on different computers, and thus cannot share address spaces.

22. **Message Passing:** Can be used for communication between non-local processes. The link can be implemented in a variety of ways, including shared memory, hardware bus, network, etc. These implementation issues vary according to the system.

    We will next look at naming techniques to see how links can be established; links can be established first before all communication (like TCP), or for each message sent/received (like UDP)? Link between more than 2 processes, such as one sender and multiple receivers, can create synchronization problems. Any receiver may be able to pick up the message sent by the sender.

    Number of links depends on naming scheme for the links, and system implementation. Link capacity can either be specified by the creator of the link, or can be a system-defined constant.

    Fixed size messages makes it easier for system implementation, but makes it harder for the programmer, unless the link creator makes large links to accommodate all messages. Conversely true for variable size messages.

23. **Message Passing – Naming:** The process only needs to know the identity of the other process to communicate. There is also a scheme with asymmetric naming, such that the sender specifies the receiver queue, but the receiver does not. Then, multiple processes can send to the same queue.

    With hard-coding the process id, we are in trouble if the process id changes. Therefore, we can use the other indirect naming scheme.

24. **Message Passing - Naming (2):** Without synchronization, the multiple receiver problem may be handled by:
    1. Allow a link to be associated with at most two processes
    2. Allow only one process at a time to execute a receive operation
    3. Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

    Mailbox in the process space has a default owner, and mailbox is deleted when process terminates. If mailbox in OS, then may have no owner. Explicit deletion of mailbox may be necessary to release resources.

25. **Message Passing (3):** Synchronization:
    Different combinations of send and receive are possible. When both are blocking, synchronization problem disappears, since send() sends a message and waits until the receive() gets it. Similarly, the receiver does not proceed until it gets a message. The '`select`' system call can be used to implement a non-blocking receive.

    Sent messages reside in a temporary queue. Zero capacity – sender must be blocking.

26. **Interprocess Communication in Linux:** Unix and its many variants provide several mechanisms for interprocess communication.

27. **Pipes**

- Pipes are the earliest form on IPC in Unix systems. They are also the simplest.
- Two-way communication may be achieved via two pipes.
- Ordinary pipes in Unix, also called anonymous pipe, can only be setup between parent and child processes. Another mechanism, called named pipes, that we study later, gets rid of this limitation.
- Since pipes can only be setup between parent-child, inter-computer communication is not allowed.
- If process exits before collecting the data, then it will be lost.
- Data only collected in the order it is entered. We will later study message queues to relax this limitation.

28. **Simple Example Using Pipes:** Pipe is constructed in the same process. The standard Unix system calls, read and write, are used for communication with pipes. The pipe is destroyed when the process ends.

29. **IPC Example Using Pipes:** This works because child copies the entire parent address space. So, file descriptors fd[0] and fd[1] are copied in the child space as well.

30. **Pipes Used for Process Synchronization:** A useful property of pipes is that the read is blocking, until something is written in the write end of the pipe. This property can be used to implement a naive synchronization mechanism.

31. **Pipes Used in Unix Shells:** Unix command-line shells provide several small system programs. A common activity is constructing more complex commands from these simple commands. Pipes are an irreplaceable tool to achieve this.

    ps is used to display status of processes running on the system.
    -e option is used to display all processes.
    -f is used to print additional process status information.
    By itself the screen scrolls. So, *more* is used to view command output one screen at a time. Output of ps is input to more.

32. **Pipes Used in Unix Shells (2):** *dup* is used to duplicate the specified file descriptor. The duplicated descriptor takes on the lowest file descriptor number available. We can also use dup2 to explicitly specify the new file descriptor number.

    Pipe will also be covered during Lab-4.

33. **FIFO (Named Pipes):**

    - Named pipes attempt to over come some limitations of normal pipes.
    - Several process can use the FIFO for communication, it appears in the file system as any other file.
    - If a process opens the FIFO file for reading, it is blocked until another process opens the file for writing. The same goes the other way around.
    - Reading from an empty FIFO does not return a EOF value.
    - FIFOs consume system resources, and must be explicitly removed.
    - FIFOs are half-duplex; file cannot be opened in read/write mode.

34. **Producer Consumer Example with FIFO:** Producer Code: The *open()* blocks on until someone opens the same fifo file for reading.

    The second argument to open specify the file access permissions. A FIFO can also be created on the command line using the mknod() command. Write to the fifo from the command line using standard file commands.

35. **Producer Consumer Example with FIFO (2):** Consumer Code: Consumer code is similar. If producer quits, then read() returns 0. Thus, reader can tell when all writers have closed. If there are multiple writers then read() does not return a zero unless all writers have closed!

    We can have multiple readers and writers for the same FIFO. Any communication then will need to be synchronized. There is no way using FIFOs to specify that a message is for some specific reader. Messages can only be read in FIFO order.

36. **Message Queues:** Message queues provide a way to specify a message type, and messages of only that kind can be removed from the queue in FIFO order, even if there are messages of some other type ahead in the queue.

37. **Message Queue Example:** *key* is specified to be 0. It is just a long integer. It is unique for each queue.

    IPC_CREAT is or'ed with the permissions on the queue. Queue length is initialized by the system, but can be reset using msgctl().

    As long as the first word in the buffer is a long, the rest of the buffer can contain anything of any length. In our example program, we have a character array of 1000 bytes.

    mtype is the type of the message. This can be used during msgrcv() to only retrieve messages of a particular type from the queue in a FIFO order. msgsnd() can block if there is not enough space in the queue.

    We specify 0 as fourth argument in msgrcv(), so that we get the first message from the queue.

    If we don't terminate the queue then it will exist even after process termination, and even if no process uses it, and consume system resources. We can view present queues using ipcs command.

38. **Message Passing in Unix Systems:** This slide answers some of the questions regarding message passing that we had posed earlier, in the context of Linux message queues.

39. **Memory Sharing in Unix:** Processes can also communicate using *Shared memory.*

40. **Shared Memory Example:** This is a naive example showing how a single process uses shared memory on Linux.

    For real IPC, multiple processes may share the shared region which will cause race conditions. So, some kind of synchronization mechanism will be necessary. Therefore, we delay such an example until we learn about process synchronization.

    `shmat()` returns a void pointer, and here we are casting it to a char pointer.

41. **Shared Memory Example (2):** The first argument to shmget is again the key (which can get using `ftok()`). It can also be specified to IPC_PRIVATE for a always new anonymous memory segment.

Access to file region is as for file access for user, group, and others. Linux does not implement execute permissions.

We can specify other commands as well with `shmctl()`. (Read man page for `shmtl()`).

ipcs command can be used to check the status of the message queues and shared memory segments in the system.

42. **Unix Domain Sockets:** Similar to pipes, but allows full-duplex communication. Sockets are most commonly used on the Internet. However, here we will first study Unix domain sockets.

    Like pipes, sockets are a special file in the file system. However, the communication does not use the file interface. Thus, we won't use open(), close(), read(), write() system calls.

    Unix domain sockets are very similar to Internet sockets, with minor differences.

    Main reason for using connectionless sockets is speed. Setting up a connection, and providing for error-free communication involves a lot of overhead. Connectionless communication, which may lose some packets, may be enough for some activities, like streaming video and audio, Even ftp uses UDP, but has its own higher level protocol providing error-free communication, maintaining the same order of data sent.

43. **Unix Domain Sockets – System Calls:** `Socket()`: domain can be used to specify local communication, or a host of protocols like IPv4, IPv6, Novell IPX, etc.
    'type' can be used to specify sequences, reliable, connection-oriented SOCK_STREAM, or SOCK_DGRAM, etc.
    'Protocol' is mostly 0, but can be used if the socket type supports more than one protocol.

    `bind()`: bind socket to a unique address in the Unix domain, a special type of file.

    `listen()`: If there are this many connections waiting to be accepted, additional clients will generate the error ECONNREFUSED.

    `accept()`: The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket.
    The argument '`addr`' is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer.
    The '`addrlen`' argument is a value-result argument. It should initially contain the size of the structure pointed to by addr; on return it will contain the actual length (in bytes) of the address returned.

44. **Socket Example – Echo Server:** See the examples in the notes.

45. **Communications in Client Server Systems:** All the interprocess communication mechanisms we have seen only allow communication on the same system. We might need to share information between processes in a distributed environment, where multiple computers are linked using a network. Client-server mechanisms help achieve this. They are most commonly based on a message passing paradigm.

    Sockets can use several standard protocols, as we have just seen, for their communication. Further examples using Sockets over Internet will be discussed in the Labs. The book has an example of using Sockets in Java.

46. **Sockets:** Instead of using the file interface, as done by pipes, socket communications use the socket interface.

47. **Remote Procedure Calls:** RPC is built on top of an IPC mechanism that we have seen earlier. Client and server typically do not share an address space, so message-passing is used.

    Messages in common IPC mechanisms may be just bare packets of data. RPC daemon listens to a particular port for requests. Port is simply a number denoting the procedure to be invoked.

    Sever kernel may provide a rendezvous mechanism to locate the correct port on the server. This rendezvous daemon is also called a matchmaker. Client queries this matchmaker daemon.

    Message received by the server will include the service name, and the associated parameters.

    Marshalling is required due to differing architectures on the client and the server. One particularly annoying difference is the architecture byte-order (big-endian or little-endian). RPCs use a standard machine-independent representation, such as external data representation (XDR).

48. **Marshalling Parameters:** Stub is the client-side routine converting all data to a machine-neutral representation. Skeleton converts the marshalled arguments into machine-specific format for the server. The return value may be marshalled again by the skeleton before communicating it to the client.

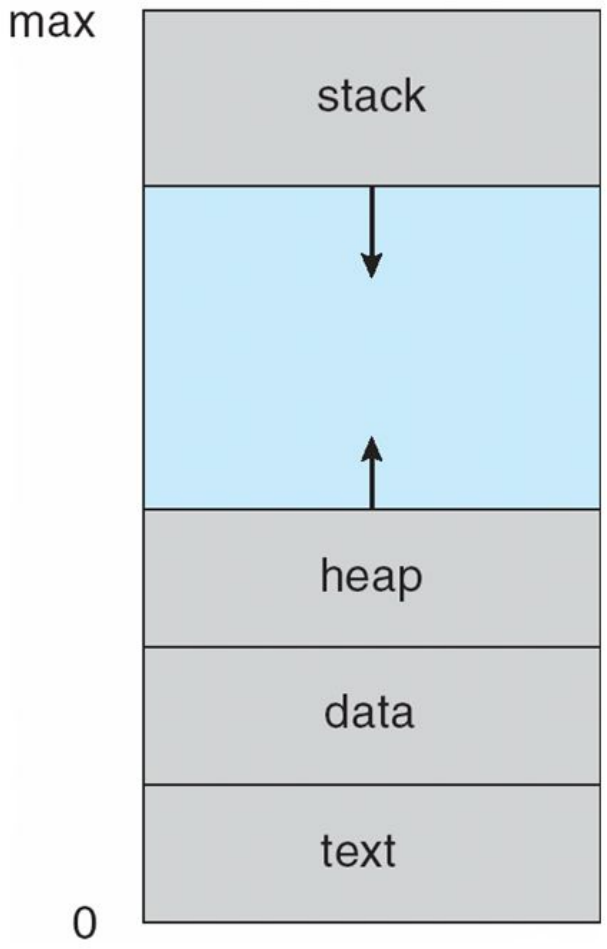
Chapter 3: Processes

- What is a process ?
- What is process scheduling ?
- What are the common operations on processes ?
- How to conduct process-level communication ?
- How to conduct client-server communication ?

Process Concept

- Process
  - is a program in execution
  - is an instance of a computer program being sequentially executed
  - process execution must progress in sequential fashion
  - process is also called a *job*
- Program Vs. process
  - program is a *passive* entity; process is an *active* entity
  - program only contains text; process is associated with code, data, PC, heap, stack, registers, and other information
  - program becomes a process when an executable file is loaded into memory
  - same program executed multiple times will correspond to different process each time

Process in Memory



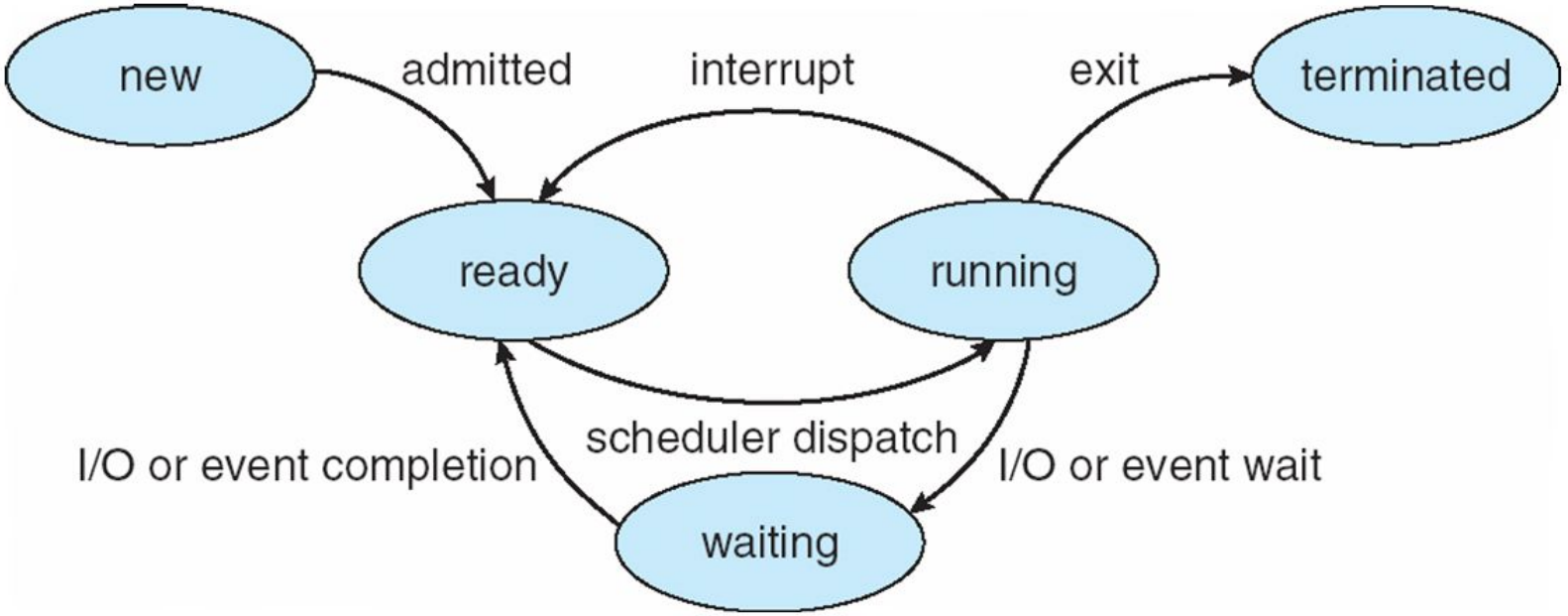
Regions in Process Memory

	Holds?	Allocation	Deallocation	Scope
Stack	Local variables (Function activation records)	Automatic (on function entry)	Automatic (on function exit)	Duration of function
Heap	Dynamically allocated data	Implicit or Explicit	Implicit or Explicit	Until deallocation
Data	Global and static data	On process creation	On process termination	During program execution
Text	Program binary instructions	N/A	N/A	N/A

Process State

- During execution, the process may be in one of the following *states*
  - new – process is being created
  - running – instructions are being executed
  - waiting – waiting for some event to occur
  - ready – waiting to be assigned a processor
  - terminated – process has finished execution
- Each processor can only run one process at any instant.

Diagram of Process State



Process Control Block (PCB)

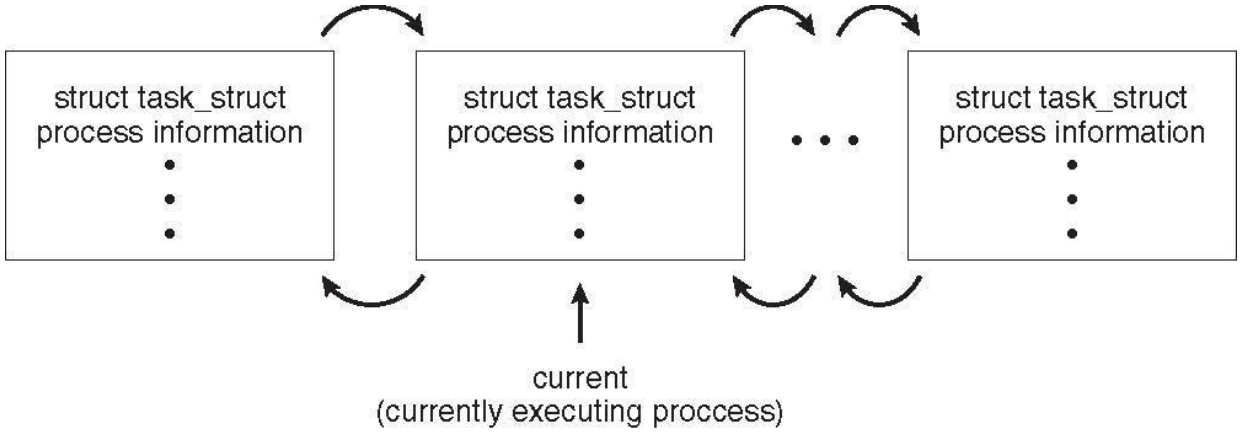
- PCB is representation of a process in an operating system.
  - maintains process-specific information
  - necessary for scheduling
- Information associated with each process
  - process state
  - program counter
  - CPU registers
  - CPU scheduling information
  - memory-management information
  - accounting information
  - I/O status information

Process Control Block (PCB) (2)

Process Representation in Linux

Represented by the C structure `task_struct`

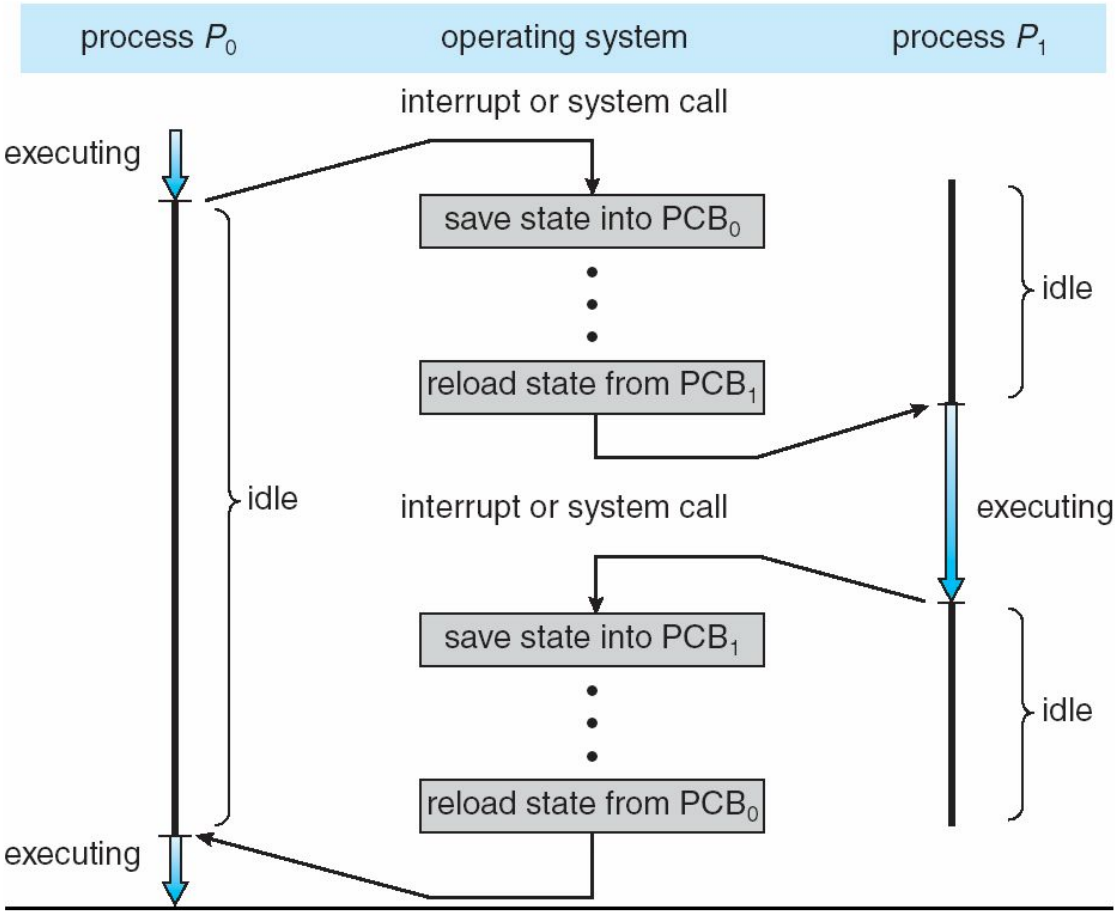
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Context Switch

- A *context switch* is the process of storing and restoring the state (context) of the CPU such that multiple processes can share a single CPU resource
  - for time-shared or multiprogramming environments
  - context of a process represented in the PCB
  - context switch involves a state *save* of the current process, and a state *restore* of the process being resumed next
  - switch from *user* to *kernel* mode or vice-versa is a mode switch
- Context-switch time is overhead
  - the system does no useful work while switching
  - overhead depends on hardware support
    - Sun UltraSPARC provides multiple banks of registers
    - Intel x86 processors also provide some support

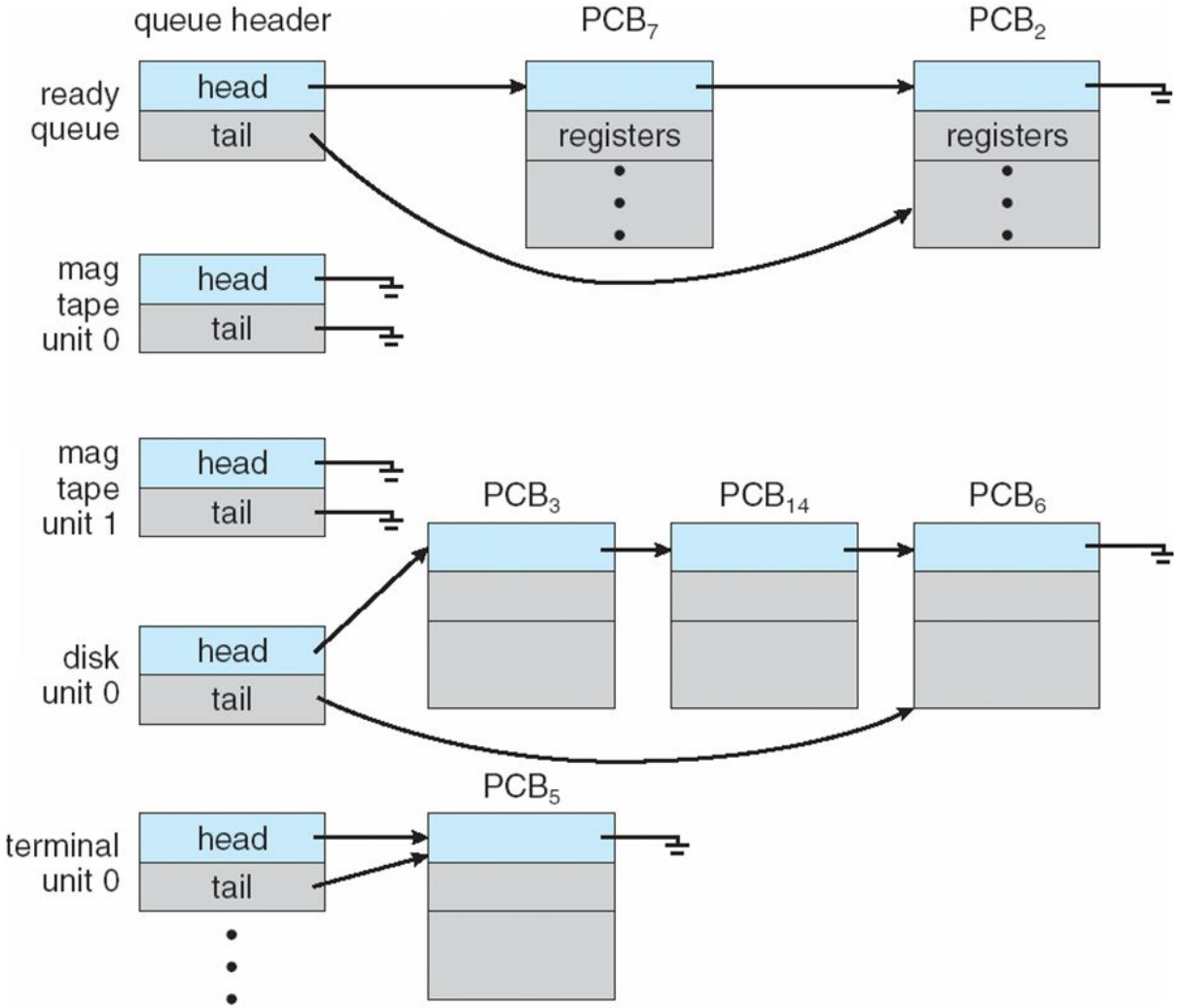
CPU Switch From Process to Process



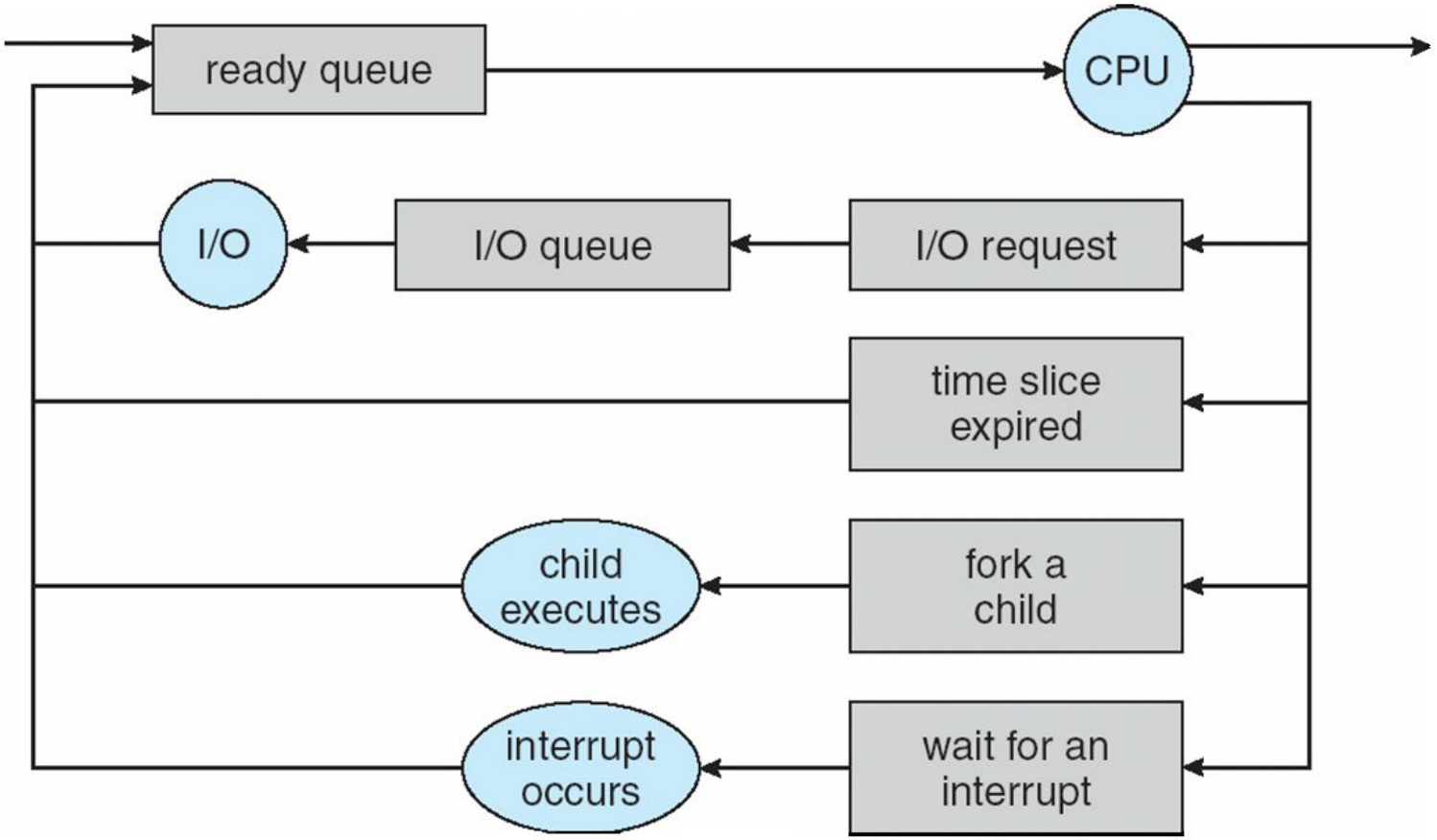
Process Scheduling

- Process scheduling selects the process to run on a CPU
  - maximizes CPU utilization in a multiprogramming OS
  - provides illusion of each process owning the system in a time-shared OS
- Terminology used in OS schedulers
  - **job queue** – set of all processes in the system
  - **ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

- Systems with a possibility of huge deluge of job requests may use multiple schedulers.
- **Long-term scheduler** (or job scheduler)
  - selects processes to be brought into the ready queue
  - controls the *degree of multiprogramming*
  - controls the mix of active CPU-bound and I/O-bound processes
  - invoked infrequently
  - can afford more time to make selection decision
- **Short-term scheduler** (or CPU scheduler)
  - selects the process to be executed next and allocates CPU
  - invoked frequently
  - necessary to limit scheduling overhead

Process Creation

- Any process can create other processes during its execution
  - operating systems have a *primordial* process
  - creating process called parent process
  - new process called child process
  - processes identified and managed via **a process identifier** (**pid**)
- Resource sharing options
  - parent and children share all resources
  - children share subset of parent's resources
  - parent and child share no resources (Unix)
- Execution options
  - parent and children execute concurrently (Unix)
  - parent waits until children terminate
    - on Unix, parent can *explicitly* call the wait() system call to wait for the child process to finish

Process Creation (Cont)

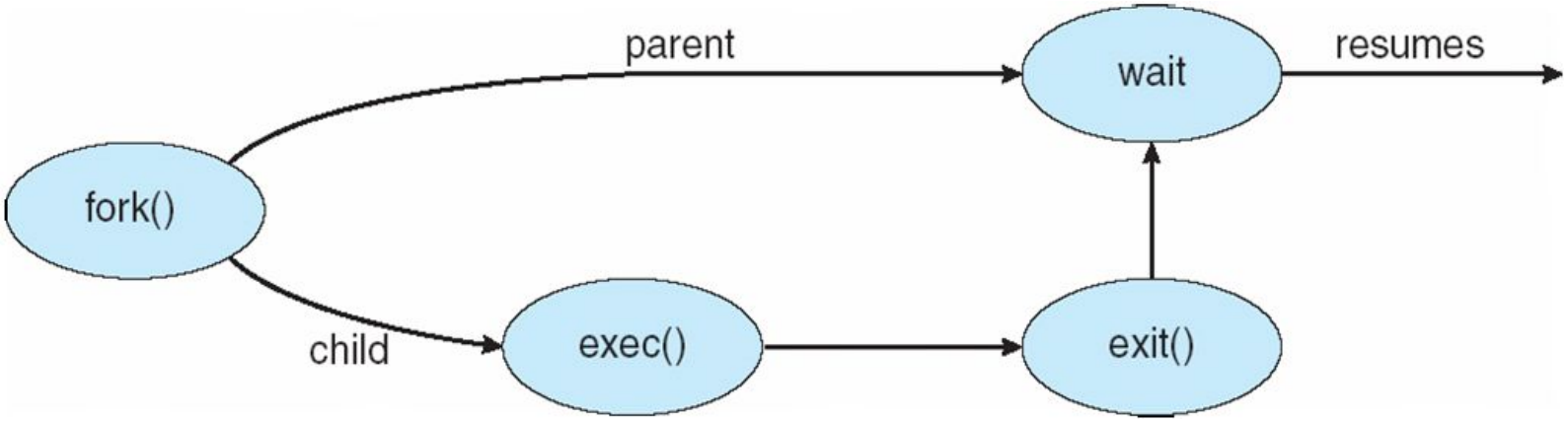
- Address space options
  - child duplicate of parent
  - child has a program loaded into it (by using exec)
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call replaces the process' memory space with a new program

Process Creation Example on Unix

```
int main()
{
      pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

Process Creation

- Parent waiting for child process to finish



Process Termination

- Process terminates after executing last statement
  - can explicitly invoke the **exit** system call to terminate
  - OS implicitly calls exit, if it is not explicitly invoked
  - child can pass return status to parent (via **wait**)
  - process resources are deallocated by operating system
- Parent may *explicitly* terminate execution of child processes (**abort**)
  - for example, if child has exceeded allocated resources
  - or, task assigned to child is no longer required
- If parent is exiting
  - on Unix-like OS, child runs independently and is unaffected; OS assigns it a *new* parent
  - some other OS may not allow child to continue if its parent terminates

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

- Communication within the same system.
- Processes may need to *co-operate* for several reasons
  - information sharing
  - computation speedup
  - modularity
  - convenience
- Cooperating process can affect or be affected by other processes
  - typically, by sharing data
- Cooperating processes need **interprocess communication** (**IPC**)

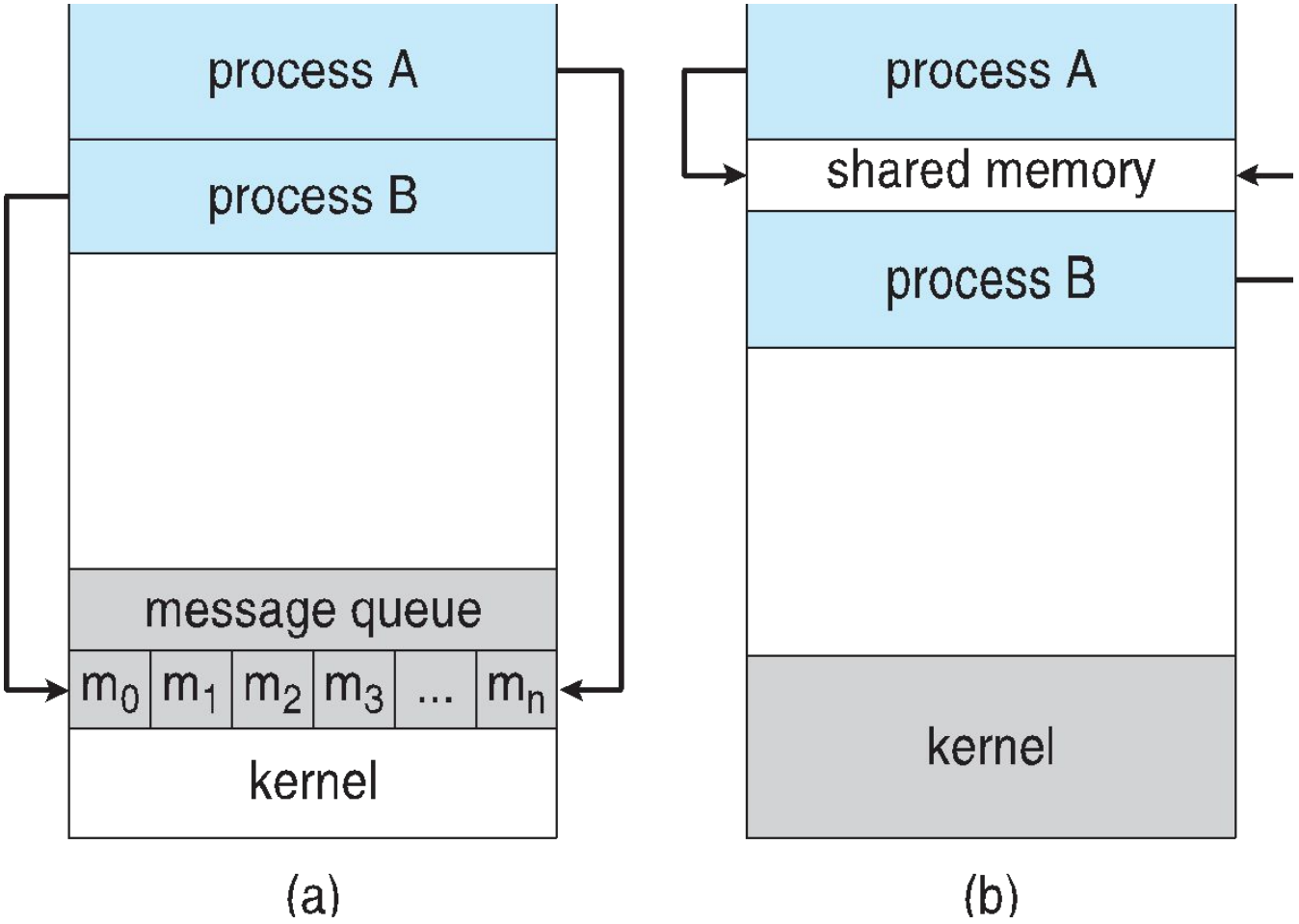
Producer-Consumer Problem

- Common paradigm for co-operating processes
  - *producer* process produces information
  - *consumer* process consumes the produced information
- Processes need synchronization
  - *consumer* cannot use information before it is produced by the *producer*
- Abstraction models
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Models of IPC

- Shared memory
  - share a region of memory between co-operating processes
  - read or write to the shared memory region
  - fast communication
  - convenient communication
- Message passing
  - exchange messages (*send* and *receive*)
  - typically, messages do not overwrite each other
    - no need for conflict resolution
  - typically, used for sending smaller amounts of data
  - slower communication
  - easy to implement (even for inter-computer communication)

Models of IPC (2)



Message Passing

- Another mechanism for interprocess communication
  - can be employed for client-server communication
- Message passing facility provides at least two operations:
  - **send** (*message*) and **receive** (*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation issues
  - how are links established?
  - can a link be associated with more than two processes?
  - how many links between every pair of communicating processes?
  - what is the capacity of a link?
  - fixed or variable sized message ?
  - is the link unidirectional or bi-directional?

Message Passing – Naming

- Direct communication
  - processes must name each other explicitly:
    - **send** (*P, message*) – send a message to process P
    - **receive**(*Q, message*) – receive a message from process Q
  - properties of communication link
    - links are established automatically
    - a link is associated with exactly one pair of communicating processes
    - between each pair there exists exactly one link
  - disadvantage
    - process identifiers are hard-coded

Message Passing – Naming (2)

- Indirect communication
  - messages are directed and received from mailboxes (also referred to as ports)
    - **send** (A, message) – send a message to mailbox A
    - **receive** (A, message) – receive a message from mailbox A
  - each mailbox has a unique id
  - processes can communicate only if they share a mailbox
  - properties of communication link
    - link may be associated with many processes
    - each pair of processes may share several communication links
    - link may be unidirectional or bi-directional
    - multiple receivers may need synchronization
  - mailbox can be held in the process address space or in the kernel

Message Passing (3)

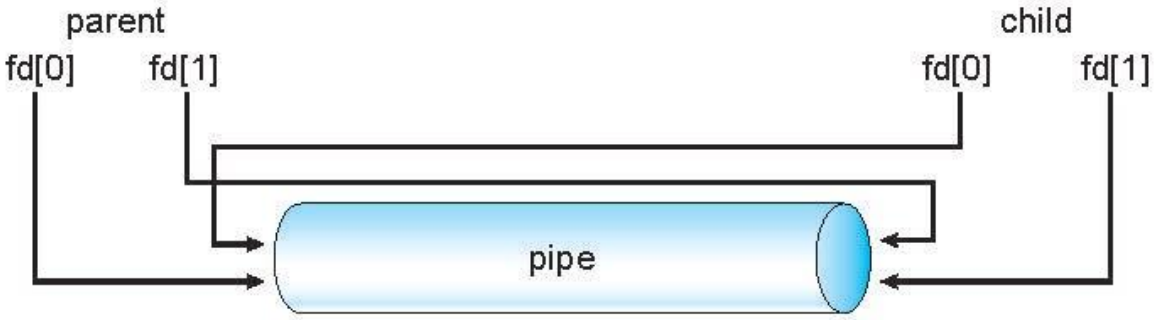
- Synchronization
  - message passing may be either blocking (synchronous) or
  - non-blocking (asynchronous)
  - **blocking send** has the sender block until the message is received
  - **blocking receive** has the receiver block until a message is available
  - **non-blocking** send has the sender send the message and continue
  - **non-blocking** receive has the receiver receive a valid message or null
- Buffering – queue of messages attached to the link
  - zero capacity – 0 messages
    - Sender must wait for receiver
  - bounded capacity – finite length of *n* messages
    - Sender must wait if link full
  - unbounded capacity – infinite length
    - Sender never waits

Interprocess Communication in Unix

- Provides multiple modes of IPC
  - pipes
  - FIFOs (names pipes)
  - message queues
  - shared memory
  - sockets

Pipes

- Most basic form of IPC on all Unix systems
  - also provides a useful command-line interface
- Conduit for two processes to communicate
  - ordinary pipes require parent-child relationship between communicating processes



Pipes

- Issues to be addressed
  - is communication unidirectional or bidirectional ?
    - Unix pipes only allow unidirectional communication
  - should communication processes be related ?
    - *anonymous* pipes can only be constructed between parent-child
  - can pipes communicate over a network
    - processes must be controlled by the same OS
- Pipes exist only until the processes exist
  - pre-mature process exit may cause data loss
- Data can only be collected in FIFO order

Simple Example Using Pipes

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678\n";

  /* open a pipe. fd[0] is opened for reading,
     and fd[1] for writing.*/
  pipe(fds);

  /* write to the write-end of the pipe */
  write(fds[1], s, strlen(s));

  /* This can be read from the other end of the pipe */
  read(fds[0], buf, strlen(s));

  printf("fds[0]=%d, fds[1]=%d\n", fds[0], fds[1]);
  write(1, buf, strlen(s));
}
```

IPC Example Using Pipes

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678. Pipe program 2\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```

Pipes Used for Process Synchronization

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = "EECS 678. Pipe program 3\n";

  /* create a pipe */
  pipe(fds);

  if (fork() == 0) {

    /* child process. */
    printf("Child line 1\n");
    read(fds[0], s, strlen(s));
    printf("Child line 2\n");
  } else {

    /* parent process */
    printf("Parent line 1\n");
     write(fds[1], buf, strlen(s));
    printf("Parent line 2\n");
  }
}
```

Pipes Used in Unix Shells

- Pipes commonly used in most Unix shells
  - output of one command is input to the next command
  - example: `/bin/ps -ef | /bin/more`
- How does the shell realize this command?
  - create a process to run `ps -ef`
  - create a process to run `more`
  - create a pipe from `ps -ef` to `more`
  - the standard output of the process to run `ps -ef` is redirected to a pipe streaming to the process to run `more`
  - the standard input of the process to run `more` is redirected to be the pipe from the process running `ps -ef`

FIFO (Named Pipes)

- Pipe with a name !
- More powerful than *anonymous* pipes
  - no parent-sibling relationship required
  - allow bidirectional communication
  - FIFOs exists even after creating process is terminated
- Characteristics of FIFOs
  - appear as typical files
  - only allow half-duplex communication
  - communicating process must reside on the same machine

Producer Consumer Example with FIFO

- Producer Code:

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // create FIFO file

  printf("waiting for readers...");
  fd = open(FIFO_NAME, O_WRONLY); // open FIFO for writing
  printf("got a reader !\n");

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){
    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

Producer Consumer Example with FIFO (2)

- Consumer code:

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // make fifo, if not already present

  printf("waiting for writers...");
  fd = open(FIFO_NAME, O_RDONLY); // open fifo for reading
  printf("got a writer !\n");

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```

Message Passing in Unix

- Linux uses indirect communication or mailboxes.
- Queues can be associated with multiple processes
  - synchronization may be required
- Communicating processes can use any number of queues
  - each queue is identified by a unique identifier
- Capacity of the link is system initialized
  - can be overridden by the user
- Messages are of a fixed size
  - specified by the buffer length
- Each communicating process can send and receive from the same queue.

Message Queue Example

```
int main()
{
  /* identifier for the message queue */
  int queue_id;
  /* send and receive message buffers */
  struct msg_buf send_buf, recv_buf;

  /* create a message queue */
  queue_id = msgget(0, S_IRUSR|S_IWUSR|IPC_CREAT);

  /* send a message to the queue */
  send_buf.mtype = 1;
  strcpy(send_buf.buffer, "EECS 678 Class");
  msgsnd(queue_id, (struct msg_buf *)&send_buf, sizeof(send_buf));

  /* get the message from the queue */
  msgrcv(queue_id, (struct msg_buf *)&recv_buf, sizeof(recv_buf), 0,
0);
  printf("%s\n", recv_buf.buffer);

  /* delete the message queue, and deallocate resources */
  msgctl(queue_id, IPC_RMID, NULL);
  return 0;
}
```

```
struct msg_buf{
   long mtype;
   char buffer[1000];
}
```

Message Queues Example (2)

- Message passing in Linux is done via message queues.
- msgget – create a new message queue
  - return existing queue identifier if it exists
- msgsnd – send a message to the queue
  - each message should be in a buffer like,
  - `struct msg_buf {`
  - `    long mtype;`
  - `    char mtext[1]; }`
  - nonblocking, unless no space in the queue
- msgrcv – receive message from the queue
  - `mtype` can be used to get specific messages
- msgctl – perform control operations specified by *cmd*
  - second argument, we use it to terminate queue

Memory Sharing in Unix

- Multiple processes share single chunk of memory.
- Implementation principles
  - uniquely naming the shared segment
    - system-wide or anonymous name
  - specifying access permissions
    - read, write, execute
  - dealing with race conditions
    - atomic, synchronized access
- Most *thread*-level communication is via shared memory.

Shared Memory Example

```
int main()
{
  int segment_id;
  char *shared_memory;
  const int size = 4096;

  /* allocate and attach a shared memory segment */
  segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);
  shared_memory = (char *) shmat(segment_id, NULL, 0);

  /* write and print a message to the shared memory segment */
  sprintf(shared_memory, "EECS 678 Class");
  printf("%s\n", shared_memory);

  /* detach and remove the shared memory segment */
  shmdt(shared_memory);
  shmctl(segment_id, IPC_RMID, NULL);

  return 0;
}
```

Shared Memory Example (2)

- shmget – create shared memory segment
  - IPC_PRIVATE specifies creation of new memory segment of size rounded to the system page size
  - access permissions as for normal file access
  - returns identifier of shared memory segment
- shmat – attach shared memory segment
  - must for every process wanting access to the region
  - segment identified by segment_id
  - system chooses a suitable attach address
- shmctl – performs the control operation specified by *cmd*
  - command is IPC_RMID to remove shared segment
- see program shared_memory2.c
- Read man pages!

Unix Domain Sockets

- Sockets
  - can be defined as an end-point for communications
  - two-way communication pipe
  - can be used in a variety of domains, including *Internet*
- Unix Domain Sockets
  - communication between processes on the same Unix system
  - special file in the file system
- Mostly used for client-server programming
  - client sending requests for information, processing
  - server waiting for user requests
  - server performing the requested activity and sending updates to client
- Socket communication modes
  - connection-based, TCP
  - connection-less, UDP

Unix Domain Sockets – System Calls

- socket ( ) - create the Unix socket
  - `int socket(int domain, int type, int protocol);`
  - domain is `AF_UNIX`
- bind ( ) - assign a name to a socket
  - `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
  - `my_addr` is `addrlen` bytes long
- listen ( ) - listen to incoming client requests
  - `int listen(int sockfd, int backlog);`
  - `backlog` specifies the queue limit for incoming connctions

Unix Domain Sockets – System Calls (2)

- accept ( ) - create a new connected socket
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - only for connection-based protocols
- recv ( ) - receive messages from socket
  - `ssize_t recv(int s, void *buf, size_t len, int flags);`
  - message placed in `buf`
- close ( ) - close the socket connection

Socket Example – Echo Server

- see socket_server.c
- see socket_client.c

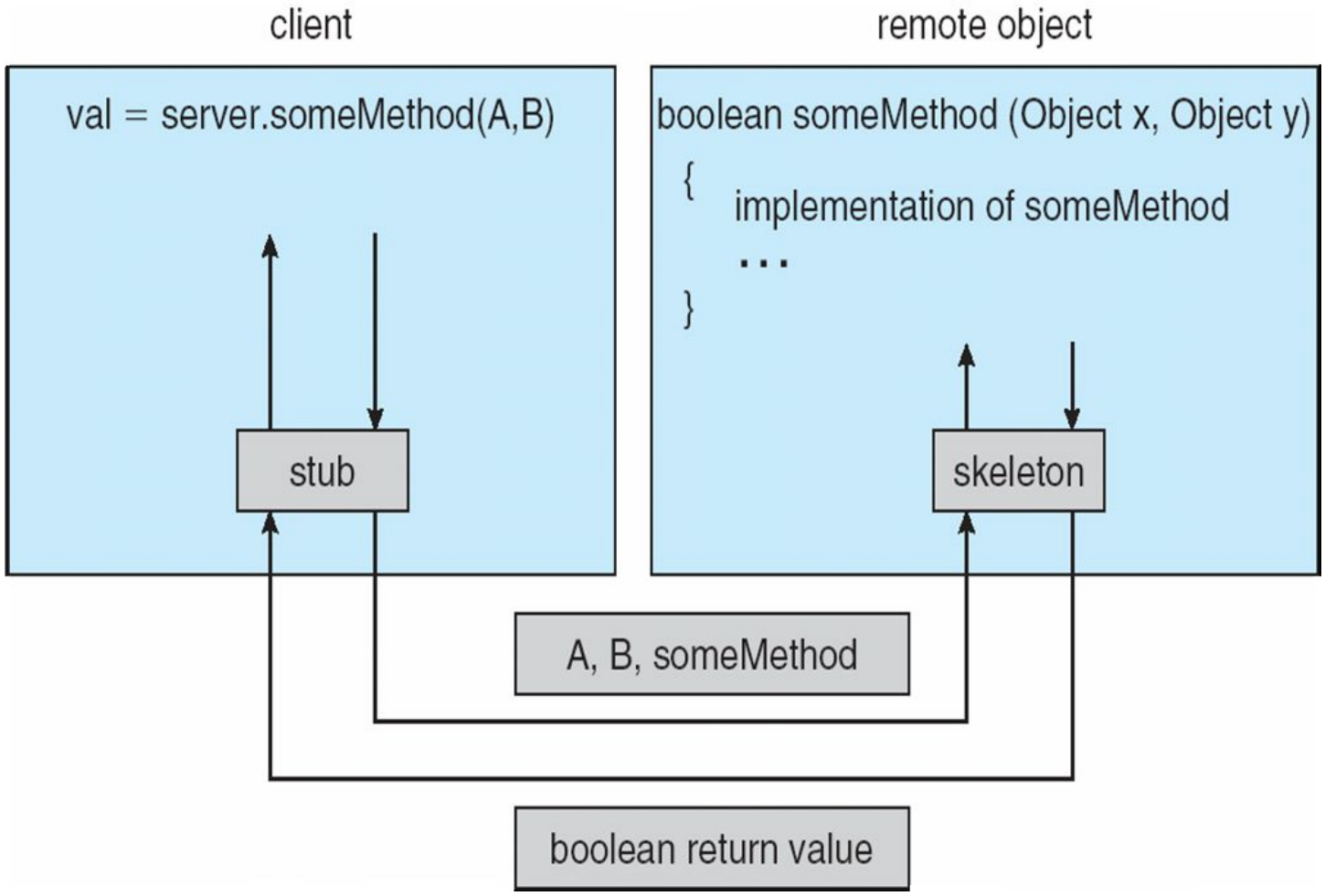
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

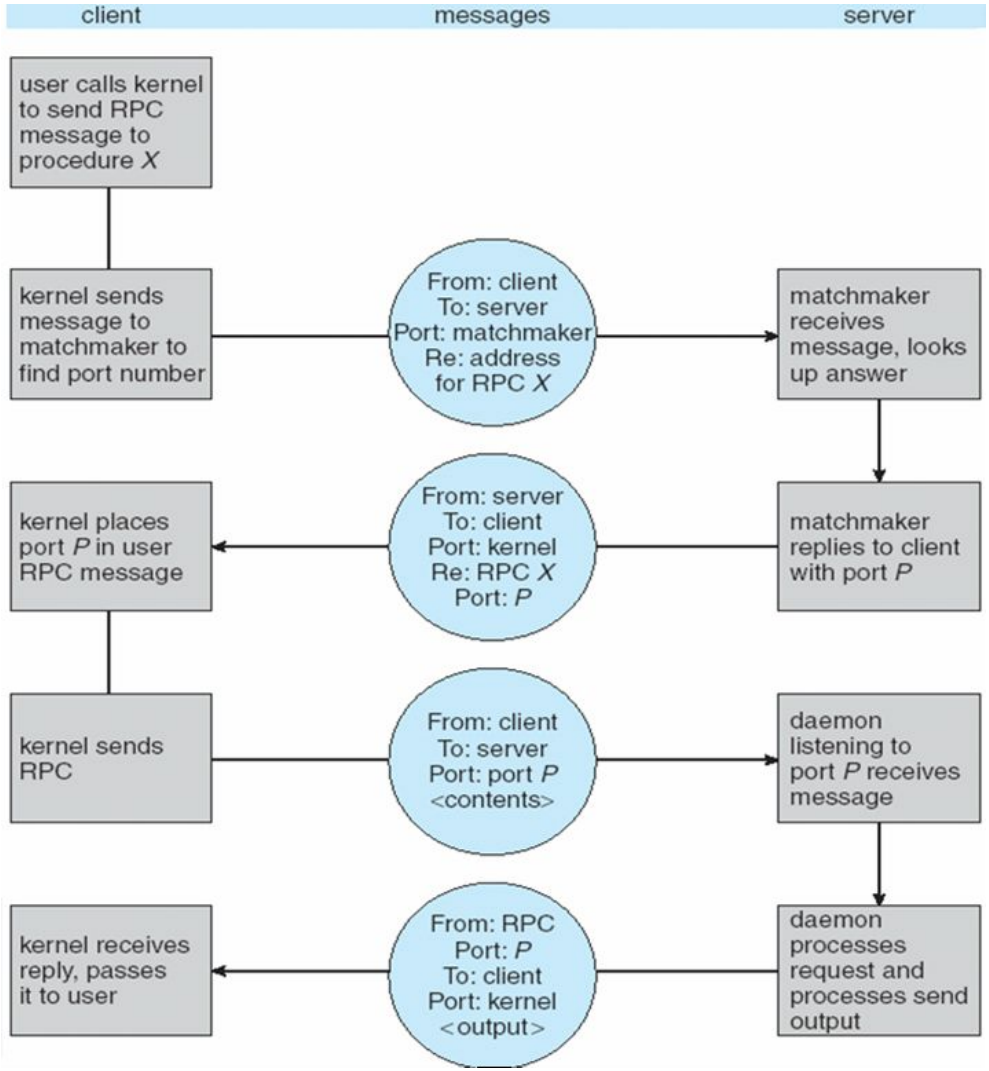
Remote Procedure Calls

- Remote procedure call (RPC) abstracts subroutine calls between processes on networked systems
  - subroutine executes in another address space
  - uses message passing communication model
  - messages are well-structured
  - RPC daemon on the server handles the remote calls
- Client-side *stub*
  - proxy for the actual procedure on the server
  - responsible for locating correct port on the server
  - responsible for *marshalling* the procedure parameters
- Server-side stub
  - receives the message; unpacks the marshalled parameters
  - performs the procedure on the server, returns result

Marshalling Parameters

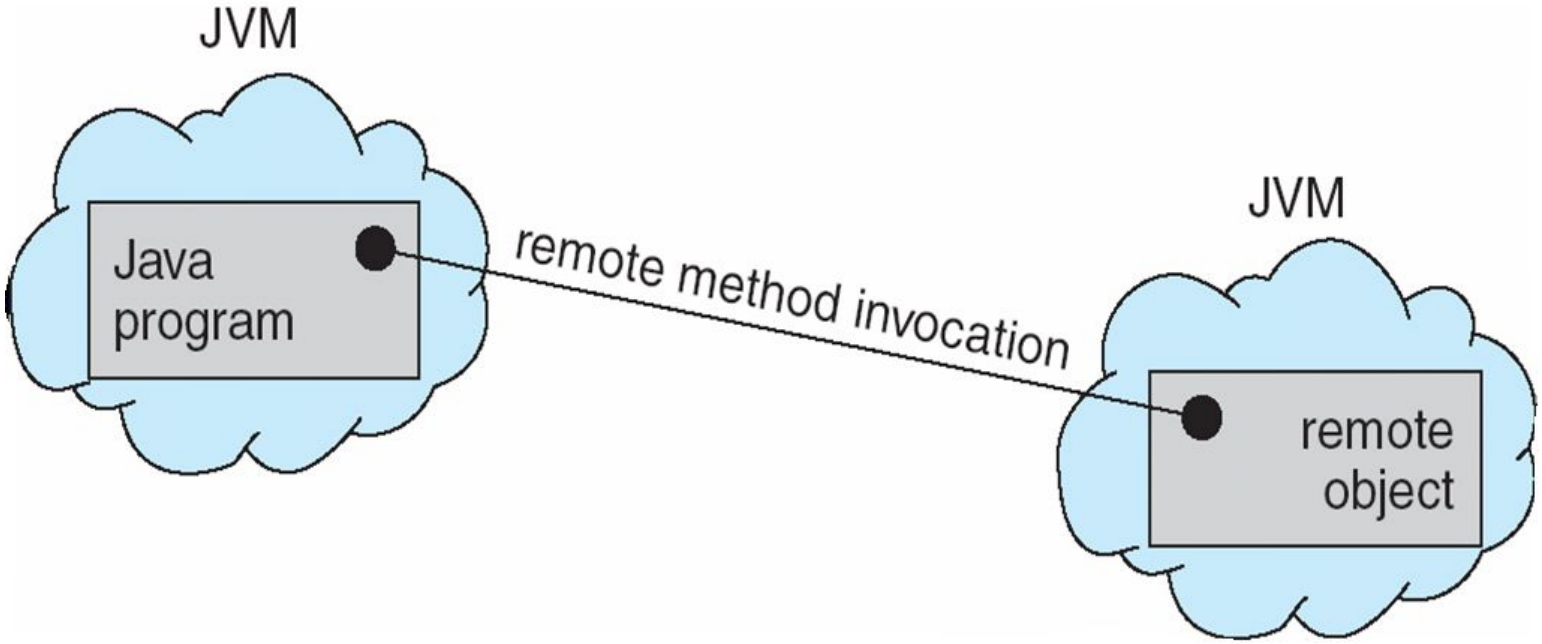


Execution of RPC



Remote Method Invocation

- Remote Method Invocation (RMI)
  - Java mechanism (API) to perform RPCs
  - Java remote method protocol (JRMP) only allows calls from one JVM to another JVM
  - CORBA is used to support communication with non-JVM code
  - client obtains reference to remote object, and invokes methods on them



Preview Test: Homework-Ch3

Test Information

Description

Instructions

Multiple Attempts Not allowed. This test can only be taken once.

Force Completion This test can be saved and resumed later.

QUESTION 1

1 points Saved

True or False (1 point): On function return, the OS automatically deallocates the stack space assigned to the function.

- ○ True
- ◉ False

QUESTION 2

1 points Saved

True or False (1 point): On Linux, the parent process by default waits for its child process to finish.

- ○ True
- ◉ False

QUESTION 3

1 points Saved

True or False (1 point): In the *shared memory* IPC model, "reads" to shared memory are typically blocking.

- ○ True
- ◉ False

QUESTION 4

1 points Saved

True or False (1 point): Any two processes running on the same OS can use the (anonymous) pipe IPC mechanism to communicate with each other.

- ○ True
- ◉ False

QUESTION 5

1 points Saved

True or False (1 point): The *shared memory* IPC model will typically allow faster communication than *message passing*.

- ◉ True
- ○ False

QUESTION 6

1 points Saved

Fill-in the blank (1 point). The answer can only be a single upper-case letter from 'A' - 'C'.

The following IPC mechanism allows sent messages to be received out-of-order -- C

Options are: 'A' - Anonymous Pipe ; 'B' - Fifo ; 'C' Message queue

Question Completion Status:

QUESTION 7

2 points Saved

Fill-in the blanks (1 point each): Each answer should only be a single-letter option from 'A' - 'D' (upper-case and without quotes).

The process address space is divided into 4 regions: 'A' - Stack, 'B' - Heap, 'C' - Data, 'D'- Text

Answer the following questions regarding the process-address space using an option from 'A' to 'D'.

Questions:

1. A dynamically allocated variable is always assigned space from the B region of the process address space.

2. The program counter (PC) points into the D region of the process address space.

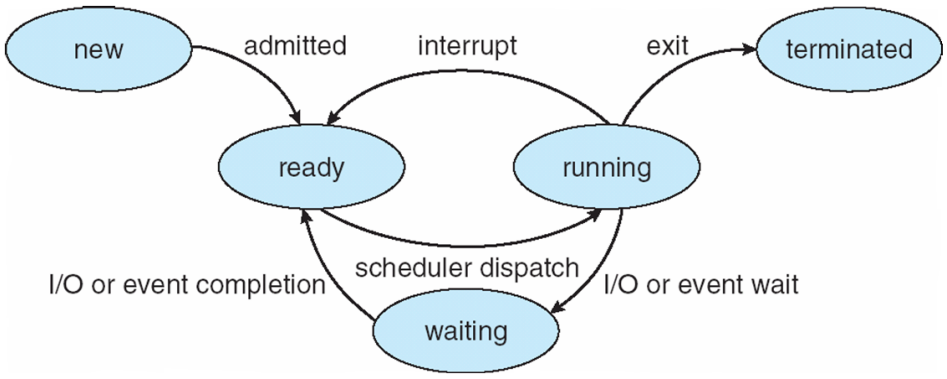
QUESTION 8

2 points Saved

Fill-in the blanks (1 point each): Each answer should only be a single-letter option from 'A' - 'G' (upper-case and without quotes).

Using the figure below, indicate the transition between process states that will happen for the specified process on the given events. Options are:

'A' = Admitted, 'B' - Interrupt, 'C' - Exit, 'D' - I/O or event completion, 'E' - I/O or event wait, 'G' - Scheduler dispatch



Questions:

1. For the currently running process when it issues the *printf* command to write a message to the screen -- D

2. For the process that is scheduled by the OS to run on the CPU in a time-shared OS -- G

QUESTION 9

2 points Saved

Answer the questions with a 'T' or 'F', upper-case and without the quotes (1 point each):
Consider the following program:

```
int main(){
   pid_t pid;

   printf("Process id is: %d\n", pid);
   pid = fork();     /* fork another process */

   if (pid == 0) { /* child process */
      execlp("/bin/ls", "ls", NULL);
      printf("Process exiting: %d\n", getpid());
   }
 if (pid > 0) { /* parent process */
 wait(NULL);
      printf ("Process exiting: %d\n", getpid());
   }

   exit(0);
}
```

Assume that,
(a) the "fork" and "exec" calls are successful,
(b) after the fork, the parent process runs before the child process, and

Question Completion Status:

Questions:

1. This program will print the line "Process id is: 11" -- F

2. The line "Process exiting: 11" is printed before the line "Process exiting: 10" -- F

QUESTION 10

3 points Saved

Fill-in the blanks (0.5 point each): Each answer should only be a single-letter option from 'A' - 'F' (upper-case and without quotes). The options can be repeated.

For the program given below, indicate what code will you insert at the marked spots to use the pipe IPC mechanism to synchronize the parent and child processes
From Child process
From Parent process

Options are: 'A' - pipe(fds) ; 'B' - read(fds[0], ...) ; 'C' - read(fds[1], ...) ; 'D' - write(fds[0], ...) ; 'E' - write(fds[1], ...) ; 'F' - Nothing to write here

Program:

```
int main(){

   char *s, buf[1024];
   int fds[2];
   char *s = "Pipe program for process synchronization\n";

A

   if (fork() == 0) {

F
      printf("From Child process\n");
E

   } else {

B
      printf("From Parent process\n");
F
   }

F
}
```

Click Save and Submit to save and submit. Click Save All Answers to save all answers.

Save All Answers Save and Submit

Chapter 4 - Threads

1. **Chapter 4: Threads – Outline:** Thread programming is steadily growing in prominence for a variety of reason. However, thread programming is both more labor-intensive, and error-prone. We will see what threads are, how to do thread-based programming, its advantages, disadvantages, OS and library provided abstractions, etc.

2. **Process Overview:** We studied processes in the last chapter, what did we learn?

   - Each program in execution is run as a separate process.
   - fork() duplicates the entire process address space, and exec() overwrites it with a new program.
   - OSes provide security among processes, but giving them then separate, isolated address spaces. This protection and isolation also makes it difficult to share information between processes.
   - Context switching is required for time-sharing OSes.
   - We looked at various IPC mechanisms. IPC is expensive due to over-riding OS security mechanisms during sharing memory, or invoking multiple, repeated system calls during message passing.

3. **Is the Process Abstraction Always Suitable ?**

   - Monolithic processes cannot exploit multicore processors or distributed environments. The process abstraction is also expensive and involves a lot of overhead.
   - There are several examples that illustrate the need to run multiple sequences of code concurrently:
     (a) word processing software may have several threads of control displaying graphics, accepting keyboard input, background spelling and grammar checking, etc.
     (b) Web browser displaying images or text, while another thread of control retrieves data from the network.
   - Creating processes and context switching between them involves calls in the OS kernel, so is expensive.

4. **Is the Process Abstraction Always Suitable ? (2):** So, what functionality do we really need for such applications (discussed on the last slide)?

   Single process not enough for single application on multicore machines. Data sharing in processes is very expensive, due to all the protection that is involved. Creating a lot of processes may be a drain on system resources. For several applications, full duplication may not be necessary, code and data regions can be shared to reduce overhead. Thus, using threads may also reduce memory pressure (less swapping).

5. **Threads to the Rescue:** Threads greatly minimize the overhead over processes.

   A process can be single-threaded or multi-threaded. Threads within a single process may not be very concerned about isolation from each other.

   Threads have become so ubiquitous that almost all modern computing systems use thread as the basic unit of computation.

6. **Thread Basics:**

- Thread is an independent flow on control within the same process. It is lightweight because most of the overhead has already been accomplished through the creation of a process.
- Threads can only exist as long as the process is alive, we can have zombie processes but no zombie threads.
- Because threads within the same process share resources: Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads. Two pointers having the same value point to the same data. Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

7. **Single and Multithreaded Processes:** We can see that the code, data, and file descriptors are shared. Shared memory is the primary mode of data sharing between threads. Each thread has its own context, so that it may be independently scheduled by the OS.

8. **Thread Benefits:** Some benefits include:

   - A multithreaded web browser could allow user interaction in one thread while an image was being loaded in another thread.
   - Threads exist in the same address space. Resource sharing is easier since no explicit programmer setup of IPC meachinsm is necessary.
   - Sharing resources reduce creation and context-switching overhead. Solaris creation of threads is 30 times cheaper than process creation.
   - A single-threaded process can only exploit one CPU. Multi-core are the direction of the future.
   - Threads are not very easy to program though. There are many things we should be aware of, and we will look at this in a later slide.

9. **Thread Programming in Linux:** Traditionally, hardware vendors implement their own propriety version of threads, making software portability a big issue for multithreaded programs. So, POSIX standard was developed.

   Mutex stands for mutual exclusion. Remember, threads share a lot of information.

   We will next look at an example that using functions from the first set of APIs. The remaining two (Mutexes and Condition Variables) will be studied in lab discussions.

10. **Pthreads Example:** The program calculates the summation of a non-negative integer in a separate thread.

    pthreads library is used. The library may be implemented as a separate library, or as part of libc.

    sum is a global, so it is in the data section.

    Pthreads thread starts execution in a new function. We can only pass it one argument. How to pass multiple arguments?

    i and *upper* defined in the function runner are thread-specific, assigned storage on the stack. The parameter type `void *` can be cast to the appropriate type.

11. **Pthreads Example – API:**

- Attribute objects provide clean isolation of the configurable aspects of threads. Attributes are specified only at thread creation time; they cannot be altered while the thread is being used. An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Attributes can be used to configure the scope, detach-state, stackaddr, stacksize, scheduling priority, and scheduling policy.
- This qualifier can be applied to a data pointer to indicate that, during the scope of that pointer declaration, all data accessed through it will be accessed only through that pointer but not through any other pointer. The 'restrict' keyword thus enables the compiler to perform certain optimizations based on the premise that a given object cannot be changed through another pointer.
- If the function called as the starting point of a thread returns, then an implicit call to pthread_exit is assumed. When a process exits, an implicit call to exit is assumed.
- An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.

12. **User Vs. Kernel Level Threads:**

    - **Use level threads:** Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher. Scheduling can also be made more efficient.

      Kernel is not aware of the user-level threads.
    - **Kernel-Level threads:** kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user-¿kernel-¿ user and loading of larger contexts, but initial performance measures indicate a negligible increase in time. Linux Pthreads uses the "clone" system call to create threads.
    - User-level threads are not ideal on multicore processors, since if one thread blocks on I/O, then the entire process blocks, since we only have one kernel thread. Having multiple kernel threads gets past this problem.

13. **Multithreading Models:**

14. **Many–to–One Model:** This model particularly shows the disadvantages of user level threads. I/O blocking causes problems, SMPs cannot be exploited.

15. **One–to–One Multithreading Model:** Creating a user thread requires creating a low efficient and bulky kernel thread. Restricts the number of threads that can be created due to this overhead.

16. **Many–to–Many Multithreading Model:** User library can create as many threads as needed.

17. **Two–level Multithreading Model:** Also allows a user-level thread to be bound to a kernel thread.

18. **Threading Issues:** Providing thread support will require answering some different questions than just using processes. We will discuss these issues on the next slides.

19. **Semantics of Fork() and Exec():** The semantics of fork and exec change in a multi-threaded program. One version of fork duplicates all threads, and the other only duplicates the calling thread.

    In Linux, fork only duplicates the calling thread. This may leave shared data, and locks managed by the other threads in an inconsistent state, and cause terrible issues later. Only use fork in a multi-threaded program, if you are going to immediately do an exec after the fork in the child process.
    Reference: *http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them*

    On Linux, `exec()`, the new program replaces the entire process, with all threads.

20. **Thread Cancellation:** Required if activity of the thread in no longer required: like several threads searching for same name in database, and one thread finds the record. Or, in browsers if user wants to stop the page loading, then all threads should stop.

    With deferred cancellation, the thread periodically checks if it should be canceled. Pthreads use deferred cancellation at points called cancellation points. Pthreads function to cancel threads is called pthread_cancel(). Thread can be set to ignore cancellation requests, or to change type to asynchronous or deferred (default) cancellation. On deferred cancellation, the cancel status is checked at certain calls, such as pthread_testcancel() or pthread_join, and cleanup handlers are then called to destroy thread-specific data.

21. **Signal Handling:**

    - How do you send a signal to a process? Use the `kill` or `raise` system calls.
    - Every signal has a default handler provided by the OS. The user process can override the default handler with its handler for most signals. The user-defined signal handler may even be used to completely ignore the signal. Some signals like SIGKILL cannot be overridden by the user process.
    - Synchronous signals: illegal memory access, divide by zero, etc.
      Asynchronous signals: timer interrupt, control-C, etc.
    - (a) Synchronous signals are typically delivered to the same process, asynchronous signals to another process.
      (b) Synchronous signals are generally delivered to a single thread that generates the signal.
      (c) Asynchronous signals vary: some like Control-C are delivered to all threads, for other signals, individual threads may be allowed to ignore them. Also, in many OSes, signals are only delivered once, so delivered to the first thread accepting the signal.

22. **Thread Pools:** Some applications like web-servers may require creation of a number of threads. The kernel does not allow creation of an unlimited number of threads since that could exhaust system resources, and cause more OS overhead in activities such as scheduling.

    Number of threads can also be dynamically adjusted as system state changes. Thread creation only happens at program startup and thread destruction only at program end.

23. **OpenMP** From Wikipedia:
    OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

24. **Linux Thread Implementation:** Arguments passed to clone() determine how much sharing needs to take place. Specifying all these parameters is essentially like creating a thread, if none are specified then it would be creating a process.

    Optional: For more details: `man -s7 pthreads` and `man nptl`

25. **Multicore Processors:** Modern processors often feature multiple *cores* on a single processor chip. We will discuss other details during the in-class Zoom sessions.

    What is *hyperthreading*?

26. **Single Core Vs. Multicore Execution:** Concurrent execution on (independent) tasks can result in significant performance gain on multi-core processors.

27. **Challenges for Multicore Programming:** There are many challenges programmers need to overcome when writing effective parallel programs for multicore processors.

    Dividing activities for balanced workload: Tasks need to carefully divided so all processors/core can be kept uniformly busy. A skewed distribution of concurrent tasks (for instance, one long task and several other short tasks) will waste hardware resources as some tasks finish, but the cores are idle while waiting for the longest task to finish.

    Data splitting: The data should be split such that the data needed by each task is located close to its processing.

    Any data dependency between tasks running concurrently will limit the gains from concurrent execution, as tasks routinely stall until the dependency is satisfied.

    Testing and debugging issues: A new category of errors, called race conditions, arise from parallel programming. We will study these further in the next chapter.

Chapter 4: Threads – Outline

- What are threads ? How do they compare with processes ?
- Why are threads important ?
- What are the common multithreading models ?
- What are the common multithreading libraries ?
- Discussion on threading issues.
- Examples of threads in contemporary OSes.

Process Overview

- The basic unit of CPU scheduling is a process.
- To run a program (a sequence of instructions), create a process.
- Process properties
  - `fork()` → `exec()` can be used to start new program execution
  - processes are well protected from each other
  - context-switching between processes is fairly expensive
  - inter-process communication used for information sharing and co-ordination between processes
    - shared memory
    - message passing

Is the Process Abstraction Always Suitable?

- Consider characteristics for a game software
  - different code sequences for different game objects
    - soldiers, cities, airplanes, cannons, user-controlled heroes, etc.
  - each object is more or less independent
- Problems
  - single monolithic process may not utilize resources optimally
    - can create a process for each object
  - action of an object depends on game state
    - sharing and co-ordination of information necessary
    - IPC can be expensive
  - number of objects proceed simultaneously
    - may involve lots of context switches
    - process context switches are expensive

Is the Process Abstraction Always Suitable? (2)

- Need ability to run multiple sequences of code (threads of control) for different object
  - individual process only offers one thread of control
- Need a way for threads of control to share data effectively
  - processes NOT designed to do this
- Protection between threads of control not very important
  - all in one application, anyway !
  - process is an over-kill
- Switching between threads of control must be efficient
  - context switching involves a lot of overhead
- Different threads of control may share most information
  - processes duplicate entire address space

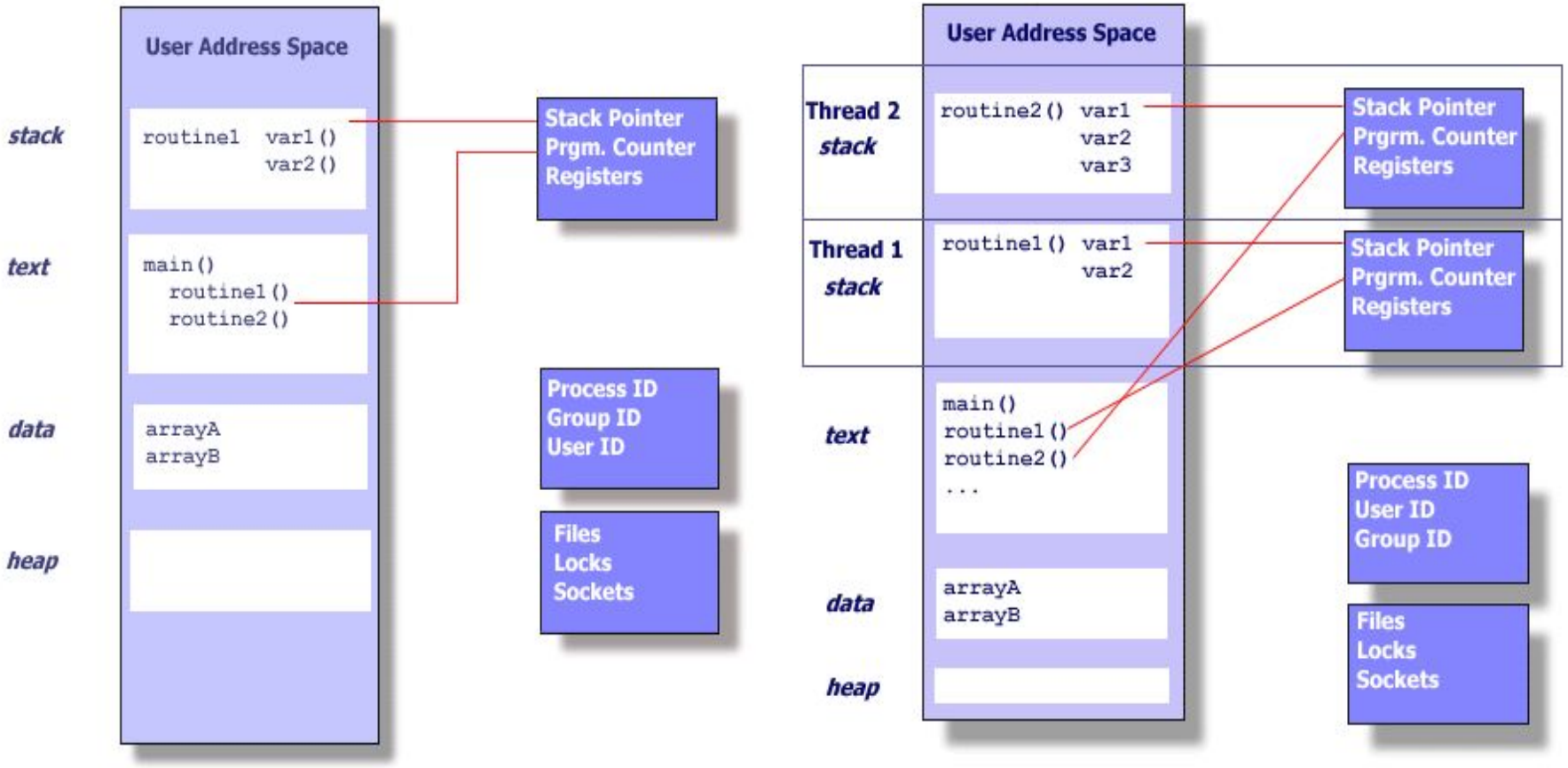
Threads to the Rescue

- *Threads* are designed to achieve all the above requirements !
  - do as little as possible to allow *concurrent* execution of a thread of control
- Thread is known as a *lightweight* process
  - only the necessary context information is re-generated
    - thread-context: PC, registers, stack, other misc. info
    - process-context: also includes data and code regions
  - threads are executed within a process
    - code and data shared among different threads
    - reduced communication overhead
  - smaller context
    - slightly faster context switching
  - a single address space for all threads in a process
    - reduced inter-thread protection

Thread Basics

- Thread – *a lightweight process*
  - have their own independent flow of control
  - share process resources with other sibling threads
  - exist within the context space of the same process
- Threads shared data
  - process instructions
  - most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - user and group id
- Threads specific data
  - thread id
  - registers, stack pointer
  - thread-specific data
  - stack (for activation records)
  - signal mask
  - scheduling properties
  - return value

Single and Multithreaded Process



source: https://computing.llnl.gov/tutorials/pthreads/

Thread Benefits

- Responsiveness
  - for an interactive user, if part of the application is blocked
- Resource Sharing
  - easier, via memory sharing
  - be aware of synchronization issues
- Economy
  - sharing reduces creation, context-switching, and space overhead
- Scalability
  - can exploit computational resources of a multicore CPU

Thread Programming In Linux

- Threads can be created using the *Pthreads* library
  - IEEE POSIX C language thread programming interface
  - may be provided either as user-level or kernel-level

- Pthreads API
  - *Thread* m*anagement* – functions to create, destroy, detach, join, set/query thread attributes
  - *Mutexes* – functions to enforce synchronization. Create, destroy, lock, unlock mutexes
  - *Condition variables* – functions to manage thread communication. Create, destroy, wait and signal based on specified variable values

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* data shared by all threads */
void runner(void param); /* thread function prototype */

int main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    pthread_attr_t attr /* set of thread attributes */

    if(atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >=0\n", atoi(argv[1]));
        return -1;
    }
    /* get the default thread attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    fprintf(stdout, "sum = %d\n", sum);
}
```

Pthreads Example (2)

```
.... (cont. from previous page...)

/* The thread will begin control in this function */
void *runner (void  *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for(i=1 ; i<=upper ; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthread Example – API Calls

- pthread_attr_init – initialize the thread attributes object
  - `int pthread_attr_init(pthread_attr_t *attr);`
  - defines the attributes of the thread created
- pthread_create – create a new thread
  - `int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);`
  - upon success, a new thread id is returned in `thread`
- pthread_join – wait for thread to exit
  - `int pthread_join(pthread_t thread, void **value_ptr);`
  - calling process blocks until thread exits
- pthread_exit – terminate the calling thread
  - `void pthread_exit(void *value_ptr);`
  - make return value available to the joining thread

User Vs. Kernel Level Threads

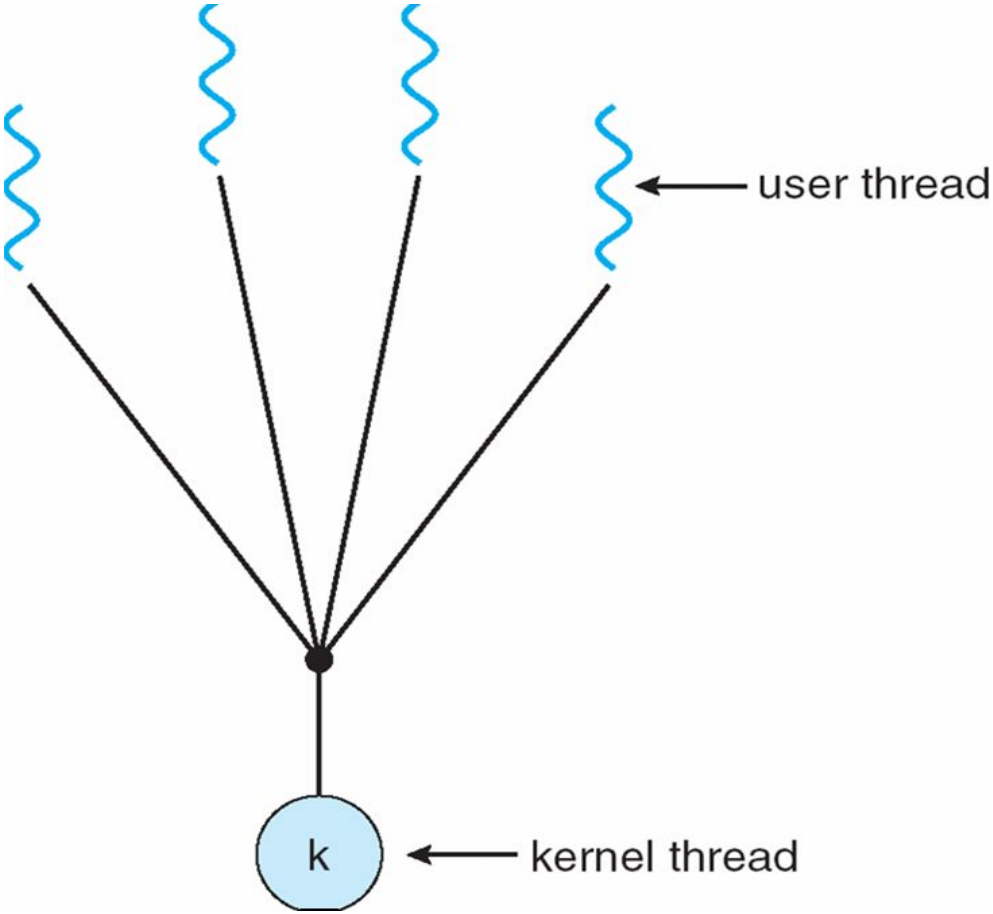
- User-level threads – manage threads in user code
  - advantages – efficient and flexible in space, speed, switching, and scheduling
  - disadvantages – one thread blocked on I/O can block all threads, difficult to automatically take advantage of SMP
  - examples of thread libraries – POSIX Pthreads, Windows threads, Java Threads, GNU portable Threads
- Kernel-level threads – kernel manages the threads
  - Advantages – removes disadvantages of user-level threads
  - Disadvantages – greater overhead due to kernel involvement
  - Examples – provided by almost all general-purpose OS
    - Windows, Solaris, Linux, Mac OS, etc.

Multithreading Models

- Relationships between user and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

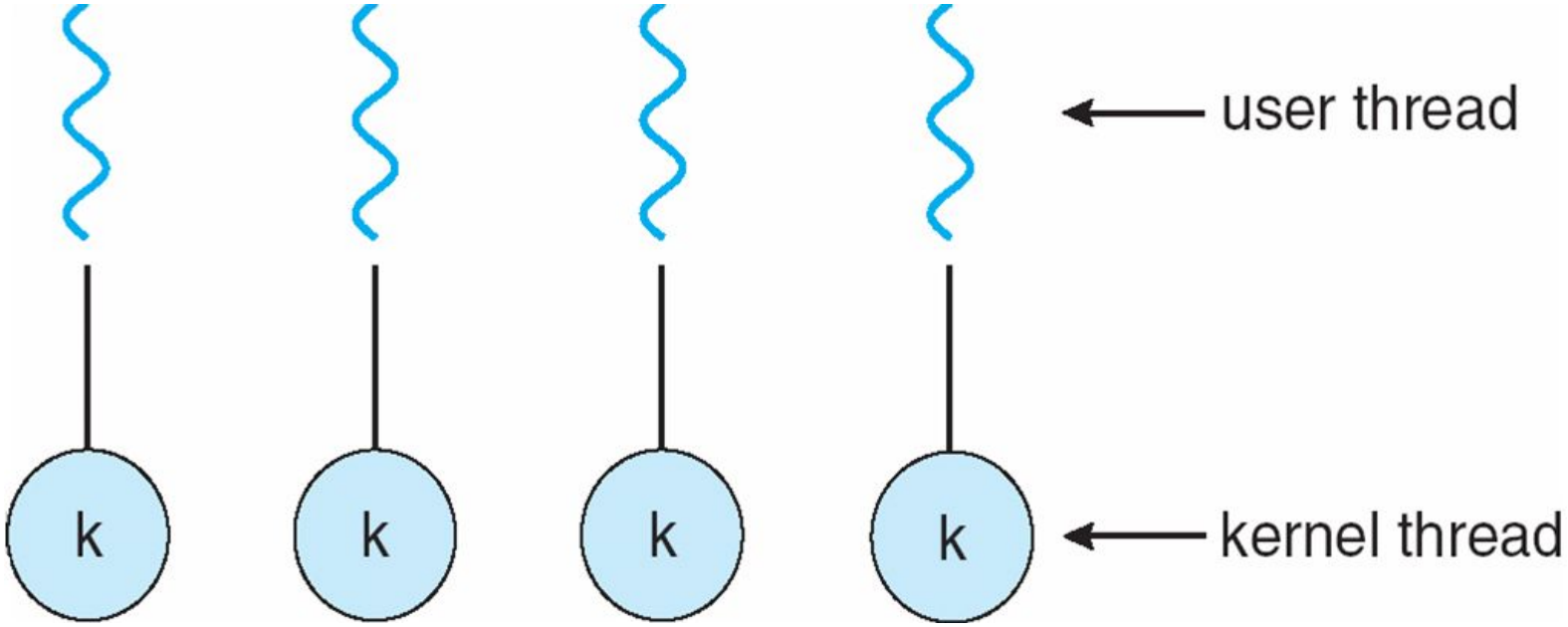
Many–to–One Multithreading Model

- Many user-level threads mapped to single kernel thread
  - examples – Solaris Green Threads, GNU Portable Threads



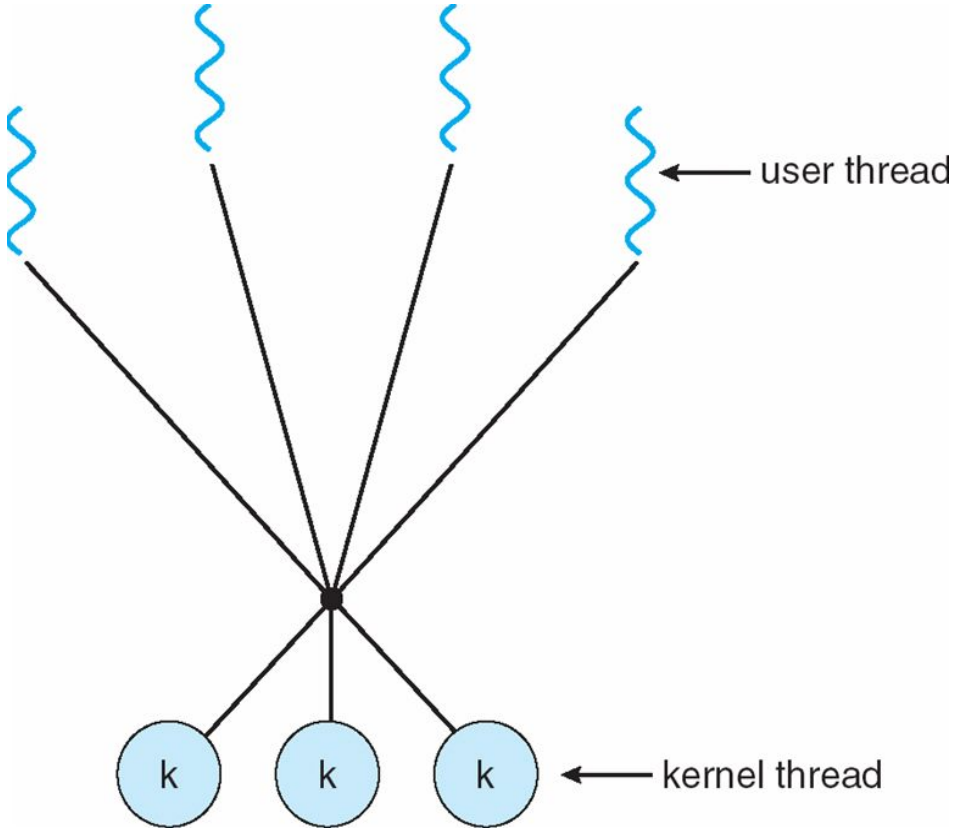
One–to–One Multithreading Model

- Each user-level thread maps to kernel thread
  - examples – Windows NT/XP/2000, Linux, Solaris 9 and later



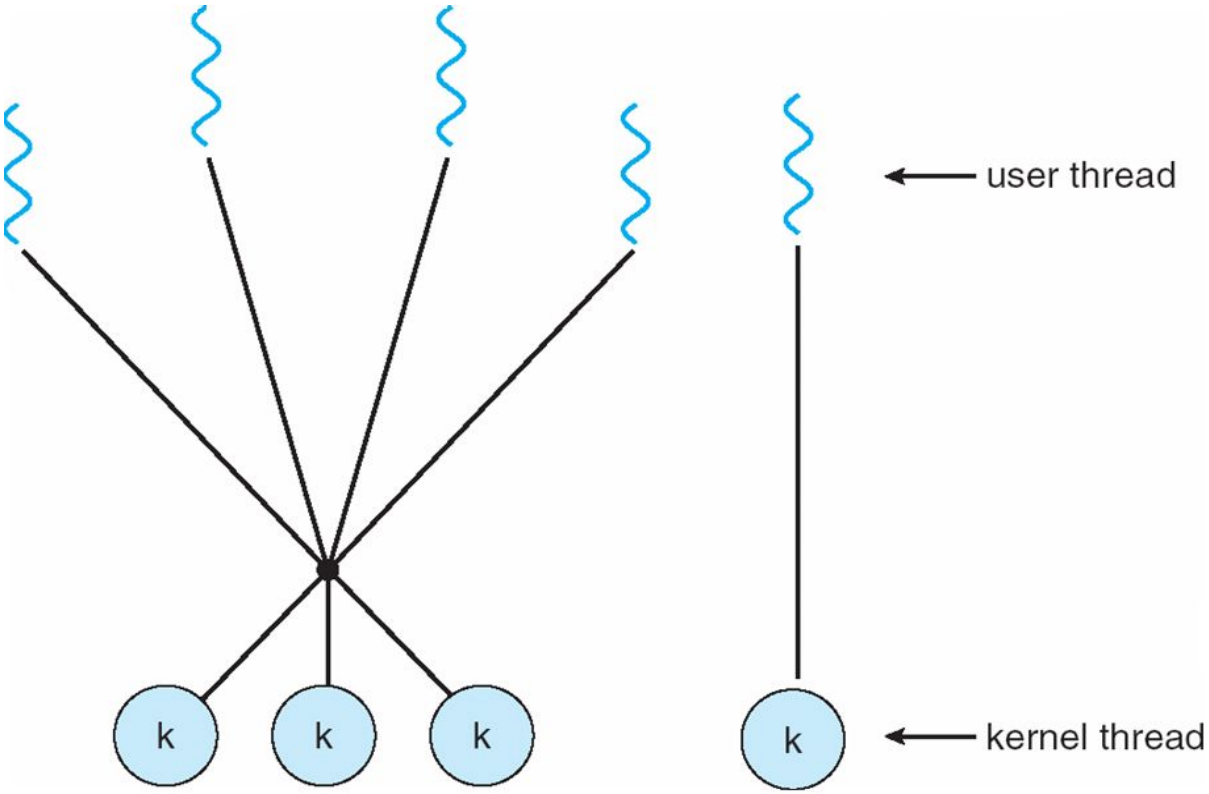
Many–to–Many Multithreading Model

- *m* user level threads mapped to *n* kernel threads
  - operating system can create a sufficient number of kernel threads
  - examples – Solaris prior to v9, Windows NT/2000 *ThreadFiber* package



Two-level Multithreading Model

- Similar to M:M, except that it also allows a user thread to be **bound** to kernel thread
  - examples – IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
- Signal handling
- Implicit Threading
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads ?
  - some systems provide two versions of fork()
  - Linux: only duplicate the thread calling fork()
- How about exec() ?
  - most systems maintain the semantics of exec()
  - Linux: complete process address space (all threads) are overwritten
- Observations
  - exec() called immediately after fork

    duplicating all threads is unnecessary

Thread Cancellation

- Terminating a thread before it has finished
- Asynchronous cancellation
  - terminates the target thread immediately
  - allocated resources may not all be freed easily
  - status of shared data may remain ill-defined
- Deferred cancellation
  - target thread terminates itself
  - orderly cancellation can be easily achieved
  - failure to check cancellation status may cause issues
- Linux supports both cancellation types
  - do: `man pthread_cancel`

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  - OS may deliver the signal to the appropriate process
  - OS or process handles the signal
- Types of signals
  - synchronous – generated by some event in the process
  - asynchronous – generated by an event outside the process
- Where to deliver a signal in multithreaded programs ?
  - deliver the signal to the thread to which the signal applies
  - deliver the signal to every thread in the process
  - deliver the signal to certain threads in the process
  - assign a specific thread to receive all signals for the process

Implicit Threading

- Writing correct multi-threaded programs
  - can cause latency and performance issues
  - is more difficult for programmers
- Use compilers and runtime libraries to create and manage threads (semi) automatically.
- Some example methods include
  - Thread pools
  - OpenMP
  - Grand central dispatch, MS Thread building blocks (TBB), java.util.concurrent package

Thread Pools

- Concerns with multithreaded applications
  - continuously creating and destroying threads is expensive
  - overshooting the bound on concurrently active threads
- Thread Pools
  - create a number of threads in a pool where they await work
  - number of threads can be proportional to the number of processors
- Advantages
  - faster to service a request with an existing thread than create a new thread every time
  - allows the number of threads in the application(s) to be bound to the size of the pool

OpenMP

- Compiler directives and an API for C, C++, FORTRAN
- Supports parallel programming in shared-memory environments
- *User* identifies parallel region
- Create as many threads as there are cores

```
#pragma omp parallel
```

- Run for loop in parallel

```
#pragma omp parallel for
for(i=0 ; i<N ; i++)
   c[i] = a[i] + b[i];
```

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

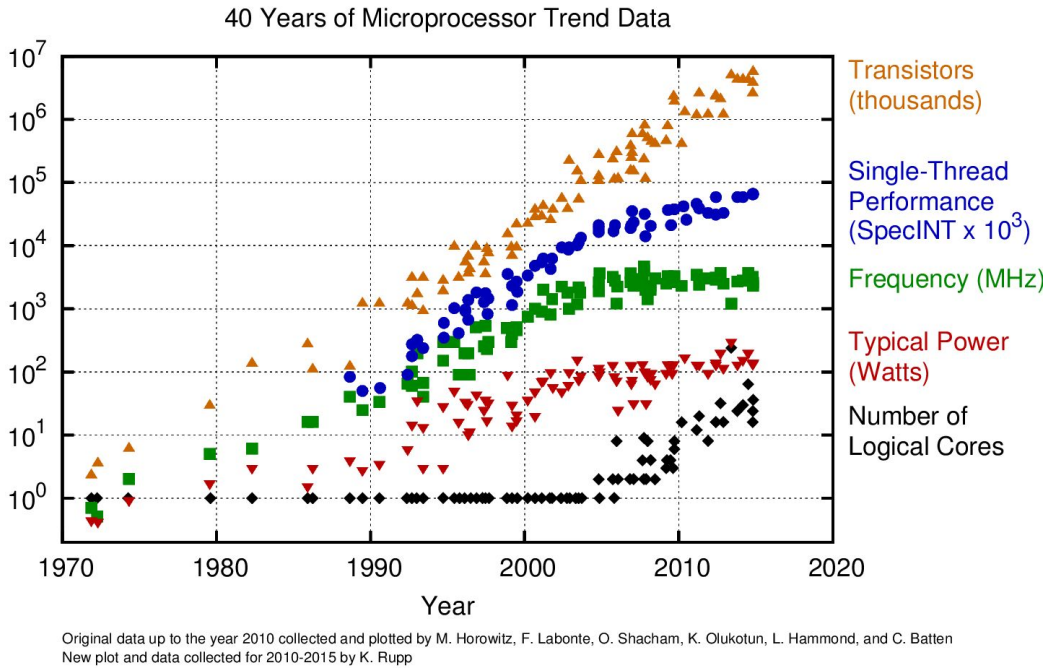
Linux Thread Implementation

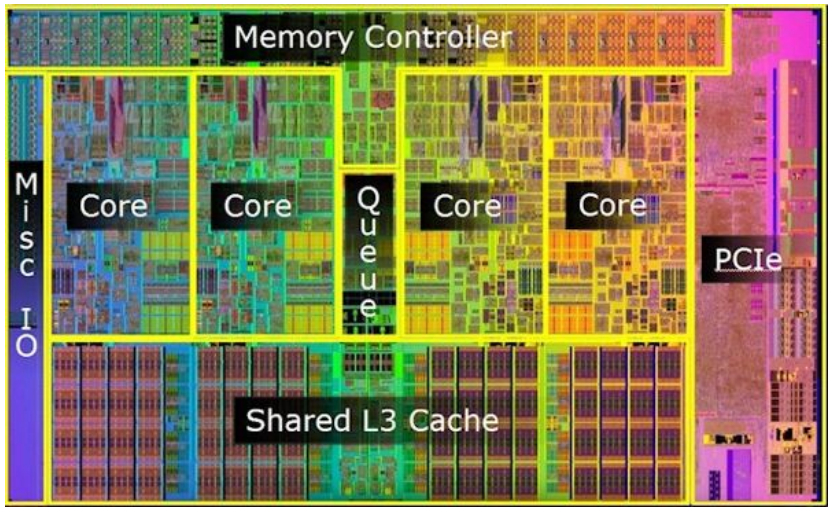
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

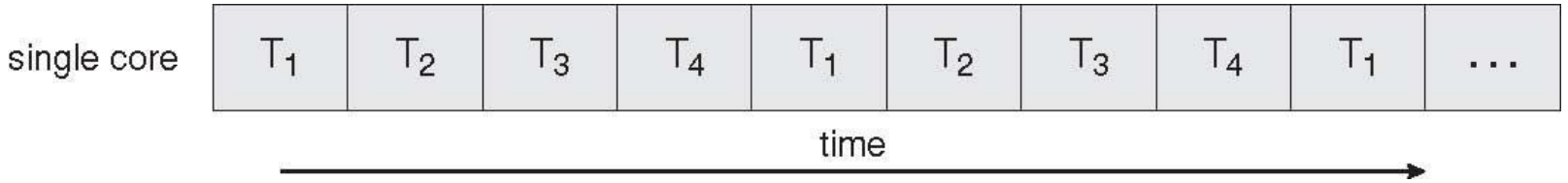
Multicore Processors

- Multiple processing cores on a single chip.
- Reasons for a shift to multicore processors
  - power wall
  - limits to frequency scaling
  - transistor scaling still a reality
- Multicore programming Vs. multicomputer programming
  - same-chip communication is faster
  - memory sharing is easier and faster

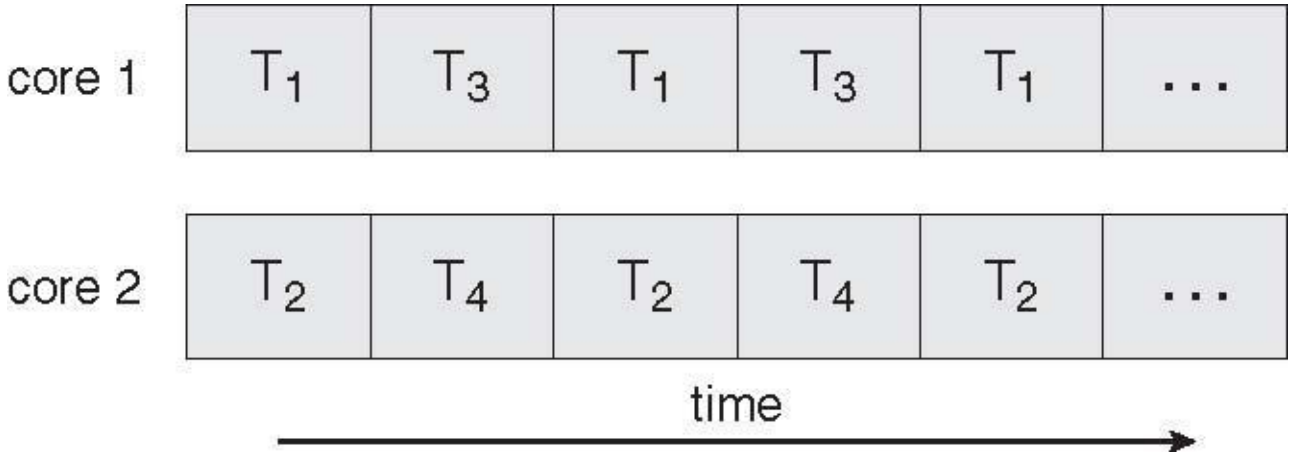




Single Core Vs. Multicore Execution



Single core execution



Multiple core execution

Challenges for Multicore Programming

- Dividing activities for balanced workload
- Data splitting
- Data dependency between tasks running concurrently
- Testing and debugging issues

Chapter 4 Threads - 1

Process "context switching" overhead is directly proportional to (or will depend on):

- [ ] size of process address space (stack, heap, etc.)
- [x] scheduler overhead
- [x] number of hardware registers
- [x] memory latency and bandwidth

Which of these statements about threads are true?

- [x] Threads are lighter-weight compared to process
- [x] Multiple threads will need less memory than multiple processes
- [ ] Multiple threads cannot run concurrently
- [x] Threads are not "protected" from each other (unlike processes)
- [ ] Each thread gets a different process id
- [x] Each thread gets it own stack partition
- [ ] Each thread gets it own heap partition
- [x] Each thread gets its own hardware context

The thread attributes set before thread creation can control the following:

- [x] Scheduling priority of the thread
- [ ] Size of the thread's heap region
- [ ] How long is the thread active

Calling "exit()" from a thread will kill the main thread and other sibling threads?

- (●) True
- ( ) False

Clear selection

Submit

Chapter 4 Threads - 1

Process "context switching" overhead is directly proportional to (or will depend on):

- [ ] size of process address space (stack, heap, etc.)
- [x] scheduler overhead
- [x] number of hardware registers
- [x] memory latency and bandwidth

Which of these statements about threads are true?

- [x] Threads are lighter-weight compared to process
- [x] Multiple threads will need less memory than multiple processes
- [ ] Multiple threads cannot run concurrently
- [x] Threads are not "protected" from each other (unlike processes)
- [ ] Each thread gets a different process id
- [x] Each thread gets it own stack partition
- [ ] Each thread gets it own heap partition
- [x] Each thread gets its own hardware context

The thread attributes set before thread creation can control the following:

- [x] Scheduling priority of the thread
- [ ] Size of the thread's heap region
- [ ] How long is the thread active

Calling "exit()" from a thread will kill the main thread and other sibling threads?

- (●) True
- ( ) False

Clear selection

Submit

Chapter 4 Threads - 1

Process "context switching" overhead is directly proportional to (or will depend on):

- [ ] size of process address space (stack, heap, etc.)
- [x] scheduler overhead
- [x] number of hardware registers
- [x] memory latency and bandwidth

Which of these statements about threads are true?

- [x] Threads are lighter-weight compared to process
- [x] Multiple threads will need less memory than multiple processes
- [ ] Multiple threads cannot run concurrently
- [x] Threads are not "protected" from each other (unlike processes)
- [ ] Each thread gets a different process id
- [x] Each thread gets it own stack partition
- [ ] Each thread gets it own heap partition
- [x] Each thread gets its own hardware context

The thread attributes set before thread creation can control the following:

- [x] Scheduling priority of the thread
- [ ] Size of the thread's heap region
- [ ] How long is the thread active

Calling "exit()" from a thread will kill the main thread and other sibling threads?

- (●) True
- ( ) False

Clear selection

Submit

Chapter 4 Threads - 2

The OS kernel controls the following properties/tasks for individual user-level threads:

- [ ] Thread creation and termination
- [ ] Thread scheduling on the CPU
- [x] Opening and closing files called from the thread
- [ ] Storing the thread's hardware state on a context switch

A program cannot realize concurrent thread-level execution on multicore machines with "kernel-level" threads.

- ( ) True
- (•) False

Clear selection

Kernel-level threads have the following properties and benefits over use-level threads:

- [x] threads can concurrently execute (if multiple CPU cores are available)
- [ ] each thread can open thread-specific files
- [x] required to support the one-to-one multithreading model
- [x] requires support from the OS kernel

On Linux, calling "fork()" from a thread will only duplicate a single flow of control in the child process

- (●) True
- ( ) False

Clear selection

On linux, with "deferred" cancellation a thread is required to terminate itself.

- (●) True
- ( ) False

Clear selection

Synchronous signals are always only delivered to the thread to which the signal applies

- (●) True
- ( ) False

Clear selection

Submit

Review Test Submission: Homework-Ch4

User	Prasad A Kulkarni
Course	2020Fall-EECS 678 Introduction to Operating Systems LEC
Test	Homework-Ch4
Started	10/7/20 2:47 PM LATE
Submitted	10/7/20 2:49 PM LATE
Due Date	10/7/20 2:00 PM
Status	Needs Grading
Attempt Score	0 out of 10 points
Time Elapsed	1 minute
Results Displayed	All Answers, Submitted Answers, Correct Answers, Incorrectly Answered Questions

Question 1

1 out of 1 points

True or False (1 point): The OS *protects* the address space of the parent thread from the address space of the child thread.

Selected Answer: ✓ False
Answers: True
✓ False

Question 2

1 out of 1 points

True or False (1 point): An *exit()* called from any thread will kill the entire process

Selected Answer: ✓ True
Answers: ✓ True
False

Question 3

1 out of 1 points

True or False (1 point): All asynchronous signal are always delivered to every thread in the process.

Selected Answer: False
Answers: True
False

Question 4

1 out of 1 points

True or False (1 point): A thread with *asynchronous* cancellation has no control over when and how it may be externally terminated.

Selected Answer: True
Answers: True
False

Question 5

1 out of 1 points

True or False (1 point): The system call *wait()* can be used by the main thread to wait for the child thread to exit.

Selected Answer: False
Answers: True
False

Question 6

3 out of 3 points

Answer 'T' for true and 'F' for false (upper-case and without quotes) (3 points):

While both *threads* and *processes* are "entities" that can enable multiple concurrent flows of control, creating *threads* is preferable to creating *processes* in the following situations:

(a) You require complete isolation and separation between the different entities -- [A]

(b) You require the entities to share much of the code and open files -- [B]

(c) You want to reduce the context switching overhead -- [C]

(d) You want the different entities to share the hardware context and register state - [D]

(e) You want to launch a new program in each entity -- [E]

(f) You want to reduce the overall memory consumption -- [F]

Specified Answer for: A F
Specified Answer for: B T
Specified Answer for: C T

Specified Answer for: D ✓ F

Specified Answer for: E ✓ F

Specified Answer for: F ✓ T

Correct Answers for: A		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	F	
Correct Answers for: B		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	T	
Correct Answers for: C		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	T	
Correct Answers for: D		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	F	
Correct Answers for: E		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	F	
Correct Answers for: F		
Evaluation Method	Correct Answer	Case Sensitivity
✓ *Exact Match*	T	

Question 7

2 out of 2 points

Answer 'T' for true and 'F' for false (upper-case and without quotes) (2 points):

The many-to-one multi-threading model has the following properties:

(a) does not need any (additional) support from the OS -- [A]

(b) can run the threads concurrently on different cores of a multi-core machine -- [B]

(c) one thread blocked (on I/O) blocks all other sibling threads --[C]

(d) requires separate stack space for each thread -- [D]

Specified Answer for: A ✓ T

Specified Answer for: B ✓ F

Specified Answer for: C ✓ T

Specified Answer for: D ✓ T

Correct Answers for: A		
Evaluation Method	Correct Answer	Case Sensitivity
Exact Match	T	
Correct Answers for: B		
Evaluation Method	Correct Answer	Case Sensitivity
Exact Match	F	
Correct Answers for: C		
Evaluation Method	Correct Answer	Case Sensitivity
Exact Match	T	
Correct Answers for: D		
Evaluation Method	Correct Answer	Case Sensitivity
Exact Match	T	

Wednesday, October 7, 2020 2:49:05 PM CDT

← OK

Chapter 5 Process Synchronization

1. **Process Synchronization – Outline:** Objectives of this chapter include: To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data. To present both software and hardware solutions of the critical-section problem.

2. **Why Process Synchronization ?** A process that can affect or be affected by other processes is called a cooperating process.

   Sharing may be achieved via shared memory. A context switch saves the local state of the process, and OS protection guards it from illegal access. But the shared process state in not guarded. In case of multicore processors two cooperating process may try to access shared data concurrently. We will see examples of concurrent shared data access producing different values shortly.

   So, you can well gather that this is an issue that only comes due to the sharing of data between processes!

3. **Producer-Consumer System Sharing Data:** Bounded buffer problem. Buffer can hold 10 items.

   Question: What is the shared data here? *counter* is the shared data variable. Both the consumer and producer update counter. [Buffer is only written in the producer.]

4. **Producer-Consumer System Sharing Data:** Although updates to counter seem to happen in one instruction, on several (RISC) architectures, they may require multiple assembly instructions. This is shown here.

5. **Race Condition:** Since we are only interested in the shared data, I have only shown those instructions here. Now, lets look at 2 possible execution orders for these six instructions that can produce incorrect values. These can happen due to context switches happening asynchronously, outside the control of the applications.

   The correct counter value should be 5. Howver, context switching on one processor or concurrent execution on multiple processors can produce different execution ordering.

   To guard against race condition, we need to ensure that only one process at a time can update the shared variable.

6. **Critical Section Problem:** We have seen some of the problems concurrent updating can lead to. Thus no two processes can be in their critical sections at the same time.

   Thus the protocol is quite simple. Each cooperating process should follow these steps. The OS or external library should ensure single access. Although the protocol appears simple, its implementation in real systems throws several interesting challenges. In the remainder of this chapter, we will see how to implement this protocol.

7. **Solution the the Critical Section Problem:** Progress: If no process is executing in its critical section, and some processes wish to enter their critical sections, then only those processes not executing in their remainder sections can participate in deciding which will enter the critical section, and the selection cannot be postponed indefinitely.

   Most commonly, this bound is defined in terms of how many other processes enter their critical section before this request is granted.

8. **Preemptive Vs. Non-Preemptive Kernels:** For example, Kernel data structure that maintains a list of open files can be modified by multiple processes simultaneously.

   Non-preemptive kernel: Process executing in kernel mode cannot be preempted. However, the process can voluntarily exit the kernel mode by finishing the function, or blocking on something. Cannot be used directly on multiprocessor systems since two processes can be executing on different processors and reach kernel state concurrently. On uniprocessors, this solution works very well. Drawback: A process running in kernel mode will never be preempted, so system may not be very responsive.

   Several OS use non-preemptive kernels and design it so that bad cases do not happen often.

9. **Peterson's Solution to the Critical Section Problem:**

10. **Peterson's Solution:** The eventual value of `turn` determines which process gets to enter the critical section.

    Note that the procedure sets turn to 'otherProcess' before busywaiting. This is actually quite cleaver. It's to ensure fairness. It is not silly, because if process #1 comes along while process #0 is in its critical section, process #1 sets 'turn' to 0 and waits politely. It is fair because it means that if two processes call 'enterCriticalSection' at about the same time, the last one to set 'turn' is the one which waits.

    Mutual exclusion P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is 1 and either flag[1] is 0 or turn is 0. In both cases, P1 cannot be in its critical section.

    Progress requirement If process P0 does not want to enter its critical section, P1 can enter it without waiting. There is not strict alternating between P0 and P1.

    Bounded waiting A process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

11. **Peterson's Solution – Notes:** This does not demand any architectural feature.

    The solution works correctly on any hardware in which references to main memory are atomic. The main disadvantage is that it requires busy waiting, continuous polling of a status variable, before gaining access to the critical region if another process is already in the critical region.

    However, a purely software approach for ¿2 processes may be inefficient. When working at the hardware level, Peterson's algorithm is typically not needed to achieve atomic access. Next, we will look at some hardware features that are normally available in most architectures today to address this problem.

12. **Lock Based Solutions:** Remember, only one process get get the lock. Other's must wait. We will now see some hardware based solutions to implement these lock-based protocols.

13. **Hardware Support for Lock-Based Solutions – Uniprocessors:** Hardware support are always better than a pure software based mechanism: more efficient, and easier! A classic example of harware support is for interrupts: during the execution of each instruction there is a test to see if a request for interrupt has arrived. If it has and is of sufficient priority, the instruction is suspended and the interrupt handled. Interrupts are made possible by (hidden) polling!

    Uniprocessor systems are a special case, where solutions might be easier, since two cooperating processes cannot really run at the same time, since we only have 1 CPU, but context

switches may result in their interleavings. Context switches happen on timer interrupts. Hardware support for disable/enable interrupts can provide a simple lock-based solution.

14. **Hardware Support for Lock-Based Solutions – Multiprocessors:** Multiprocessors are different from uniprocessors, so interrupt diabling is ineffective and costly. Costly, since messages need to be passed to all processors. Disabled interrupts may also affect the workings of some system resources, in particular clocks.

15. **TestAndSet Instruction:** This is the pseudo-code! Real hardware implementation is atomic.

    In multiprocessor-multicore systems, two or more processes executing this instruction

16. **using the same memory address** are guaranteed to occur in a serial but undefined order.

17. **Mutual Exclusion using TestAndSet:** We define three operations to provide mutual exclusion: initlock, lock, and unlock. The critical region is protected using the locking framework provided earlier.

    Mutex is a variable shared between the cooperating processes. We pass a pointer to this shared variable.

    The implementation is quite simple and with no system overhead since TestAndSet is a user mode instruction. It works because of the properties of the TestAndSet instruction. Notice that this solution involves Busy Waiting (polling).

18. **Swap Instruction:** Operates on the contents of two words. Also execuetd atomically.

19. **Mutual Exclusion Using Swap:** If the machine supports swap instruction, then mutual exclusion can be provided as shown here.

    Again, mutex is a shared variable. Only when mutex is 0, will key get the value 0 and break out of the loop. Mutex can only be 0, if no other process is currently in the critical section. Since, this instruction is atomic, only one swap can occur at a time.

    If two processes P and Q are trying to enter a critical region protected by a locked spinlock mutex, then there is no guaranty that they will be serviced in FIFO order. Their order of entry will be random. More serious is the problem that the process P may use the spinlock within a loop and use the lock repeatedly while Q is unable to acquire it.

20. **Bounded Waiting Solution:** Here we have inlined the init_lock, lock, and unlock operations in this code to be able to demonstrate the algorithms clearly. Note that we can also abstract the lock, unlock portions is desired.

    The array waiting[] is shared. (Is variable key shared? No!). Lock is shared. We can suppose that the processes are executted concurrently on two multi-processors.

    In cycle 4, suppose that Process 0 wins the race, and sets lock. Note that only one process can execute the testAndSet instruction at a time since it locks the (memory) bus during execution. (One one instruction is highlighted) Since key = 0, process 0 breaks out of the loop, and into the critical section. waiting[0] is set to false. As long as lock=TRUE, all future calls to TestAndSet will return TRUE.

    In cycle 9, if j == i, then we have completed the loop and no other process is waiting to enter the critical region. We then set lock = FALSE.

    On cycle 10, waiting[1] is set tp 0 by process 0. Therefore, process 1 can now exit the while loop, not because key==0, but because waiting[1]==0. This is different than what happened earlier on cycle 4.

Thus, if there are n processes, then each process has only to wait for at most (n-1) other processes to enter-exit their critical sections, before the last process gets its chance, after making a request.

Also realize, than for hardware designers, implementing atomic instructions such as TestAndSet and Swap is not easy for multiprocessors.

Process Synchronization – Outline

- Why do processes need synchronization ?
- What is the critical-section problem ?
- Describe solutions to the critical-section problem
  - Peterson's solution
  - using synchronization hardware
  - semaphores
  - monitors
- Classic Problems of Synchronization

Why Process Synchronization ?

- Processes may *cooperate* with each other
  - producer-consumer and service-oriented system models
  - exploit concurrent execution on multiprocessors
- Cooperating processes may share data (globals, files, etc)
  - imperative to maintain data *correctness*
- Why is data correctness in danger ?
  - process run asynchronously, context switches can happen at any time
  - processes may run concurrently
  - different orders of updating shared data may produce different values
- Process synchronization
  - to coordinate updates to shared data
  - order of process execution should not leave shared data in an inconsistent state
- *Only needed when processes share data !*

Producer-Consumer Data Sharing

Producer

```
while (true){

    /* wait if buffer full */
    while (counter == 10)
     ;  /* do nothing */

    /* produce data */
    buffer[in] = pdata;
    in = (in + 1) % 10;

    /* update number of
       items in buffer */
    counter++;
}
```

Consumer

```
while (true){

    /* wait if buffer empty */
    while (counter == 0)
     ;  /* do nothing */

    /* consume data */
    cdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of
       items in buffer */
    counter--;
}
```

Producer-Consumer Data Sharing

Producer

```
while (true){

    /* wait if buffer full */
    while (counter == 10)
     ;  /* do nothing */

    /* produce data */
    buffer[in] = sdata;
    in = (in + 1) % 10;

    /* update number of
       items in buffer */
    R1 = load (counter);
    R1 = R1 + 1;
    counter = store (R1);
}
```

Consumer

```
while (true){

    /* wait if buffer empty */
    while (counter == 0)
     ;  /* do nothing */

    /* consume data */
    sdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of
       items in buffer */
    R2 = load (counter);
    R2 = R2 – 1;
    counter = store (R2);
}
```

Race Condition

- Suppose *counter* = 5

Incorrect Sequence 1

```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R1);
counter = store (R2);
```

Final Value in counter = 4!

Incorrect Sequence 2

```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
counter = store (R1);
```

Final Value in counter = 6!

- Race condition is a situation where
  - several processes concurrently manipulate shared data, and
  - final shared data value depends on the order of execution

Critical Section Problem

- Region of code in a process *updating* shared data is called a critical region.
- Concurrent updating of shared data by multiple processes is dangerous.
- Critical section problem
  - how to ensure synchronization between cooperating processes ?
- Solution to the critical section problem
  - only allow a single process to enter and be in its critical section at a time
- Protocol for solving the critical section problem
  - request permission to enter critical section
  - indicate after exit from critical section
  - only permit a single process at a time

Solution to the Critical Section Problem

- Formally states, each solution should ensure
  - *mutual exclusion:* only a single process can execute in *its* critical section at a time
  - *progress:* selection of a process to enter its critical section should be fair, and the decision cannot be postponed indefinitely.
  - *bounded waiting:* there should be a fixed bound on how long it takes for the system to grant a process's request to enter its critical section
- Other than satisfying these requirements, the system should also guard against *deadlocks.*

Preemptive Vs. Non-preemptive Kernels

- Several kernel processes share data
  - structures for maintaining file systems, memory allocation, interrupt handling, etc.
- How to ensure OSes are free from race conditions ?
- Non–preemptive kernels
  - process executing in kernel mode cannot be preempted
  - disable interrupts when process is in kernel mode
  - what about multiprocessor systems ?
- Preemptive kernels
  - process executing in kernel mode can be preempted
  - suitable for real-time programming
  - more responsive

Peterson's Solution to Critical Section Problem

- Software based solution
- Only supports two processes
- The two processes share two variables:
  - int turn;
    - indicates whose turn it is to enter the critical section
  - boolean flag[2]
    - indicates if a process is ready to enter its critical section

Peterson's Solution

Process 0

```
do {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1]  &&  turn==1)
      ;
   // critical section

   flag[0] = FALSE;

   // remainder section
} while (TRUE)
```

Process 1

```
do {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0]  &&  turn==0)
      ;
   // critical section

   flag[1] = FALSE;

   // remainder section
} while (TRUE)
```

- Solution meets all three requirements
  - P0 and P1 can never be in the critical section at the same time
  - if P0 does not want to enter critical region, P1 does no waiting
  - process waits for at most one turn of the other to progress

Peterson's Solution – Notes

- Only supports two processes
  - generalizing for more than two processes has been achieved
- Assumes that the LOAD and STORE instructions are atomic
- Assumes that memory accesses are not reordered
- May be less efficient than a hardware approach
  - particularly for >2 processes

Lock-Based Solutions

- General solution to the critical section problem
  - critical sections are protected by locks
  - process must acquire lock before entry
  - process releases lock on exit

```
do {
        acquire lock;

        critical section

        release lock;

        remainder section

} while(TRUE);
```

Hardware Support for Lock-Based Solutions – Uniprocessors

- For uniprocessor systems
  - concurrent processes cannot be overlapped, only *interleaved*
  - process runs until it invokes system call, or is *interrupted*
- Disable interrupts !
  - active process will run without preemption

```
do {

        disable interrupts;
        critical section
        enable interrupts;

        remainder section
} while(TRUE);
```

Hardware Support for Lock-Based Solutions – Multiprocessors

- In multiprocessors
  - several processes share memory
  - processors behave independently in a peer manner
- Disabling interrupt based solution will not work
  - too inefficient
  - OS using this not broadly scalable
- Provide hardware support in the form of atomic instructions
  - atomic *test-and-set* instruction
  - atomic *swap* instruction
  - atomic *compare-and-swap* instruction
- Atomic execution of a set of instructions means that instructions are treated as a single step that cannot be interrupted.

TestAndSet Instruction

- Pseudo code definition of *TestAndSet*

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Mutual Exclusion using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

Swap Instruction

- Psuedo code definition of swap instruction

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Mutual Exclusion using *Swap*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    int key = TRUE;
    do {
        Swap(&key, mutex);
    }while(key == TRUE);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

- *Fairness not guaranteed by any implementation !*

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 0

lock=FALSE, key=FALSE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 1

lock=FALSE, key=FALSE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 2

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 3

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process 0 wins the race

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 4

lock=TRUE, key=FALSE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 5

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 6

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

j = 1

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 7

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != I) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 8

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != I) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 9

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != I) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 10

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != I) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 11

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // Critical Section

    j = (i + 1) % n;
    while ((j != I) && !waiting[j])
        j = (j+1) % n;

    if (j == i )
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // Remainder Section
} while (TRUE);
```

Cycle = 12

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0

Chapter 5 - Part 1

Process synchronization is only needed when multiple processes share data

- (●) True
- ( ) False

Clear selection

Process synchronization is not an issue on "single-core" systems

- ( ) True (since only one process can run at a time)
- (●) False (since context switches can still interleave process execution)

Clear selection

Solution to the critical section problem only allows a single process in the critical section at a time, even on multi-core systems

- (●) True
- ( ) False

Clear selection

With non-preemptive kernels, processes will typically spend more time executing in kernel mode

- (●) True
- ( ) False

Clear selection

In Peterson's solution, if Process-0 is already executing in its critical section, then Process-1 that wants to enter the critical section will wait in the "while" loop because:

- [x] flag[0] is true
- [ ] flag[1] is true
- [x] turn = 0
- [ ] turn = 1

Peterson's solution to the critical section problem satisfies the criteria of "bounded waiting"

- (●) True
- ( ) False

Clear selection

What is the value returned by "TestAndSet(mutex)" if mutex=FALSE?

- ○ true
- ◉ false

Clear selection

Submit