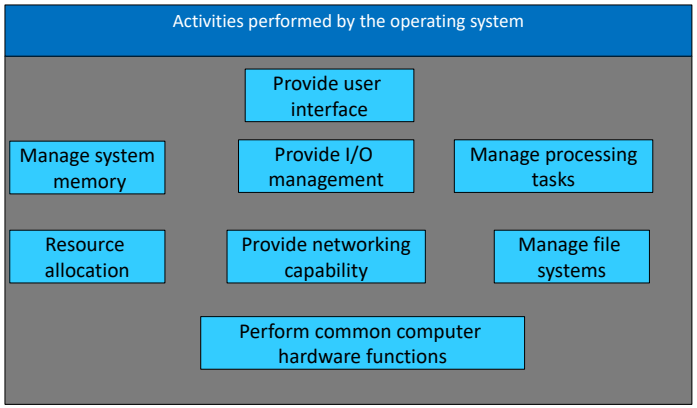
Introduction

Q1: What is a Computer System?
Q2: What is an Operating System?
Q3: What are the Goals of an OS?

What Operating Systems Do (Cont'd)



What Operating Systems Do?

Types of OS	OS goals
OS for Single User System	**Ease of use**
OS for Multi User Systems (**mainframe** or **minicomputer**)	**Maximize resource utilization**
Users of dedicate systems (workstations)	**Compromise between individual usability and resource utilization**
Handheld computers	**optimized for usability and battery life**
Embedded computers	**Run without user intervention (with little or no user interface)**

Operating System Definition

- OS is a resource allocator
  - Manages all hardware resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer
- OS provides abstractions
  - Hides the details of the hardware
  - Provides an interface that allows a consistent experience for application programs and users

Operating System Definition

What are common abstractions provided by the OS?

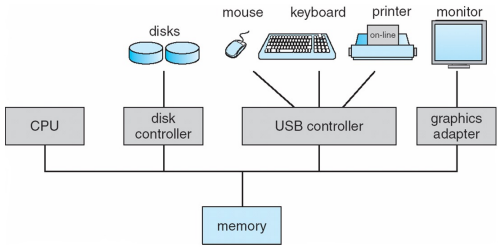
- A program has exclusive access to the CPU(s) and other hardware devices
- A program has unbounded access to memory
- Directories and files
- Reliable communication between programs and computers
- No errors in: execution, communication, device interaction

Interrupts

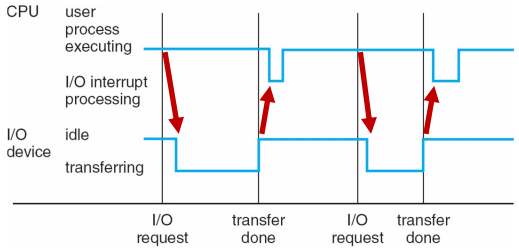
- An operating system is **interrupt driven**
  - An interrupt transfers control from the currently executing program to the appropriate interrupt service routine
  - Interrupt architecture must save the address of the interrupted instruction, as well as the register state
  - A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request

Computer System Organization

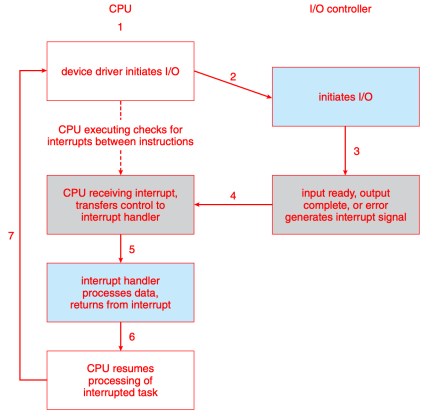
- A modern computer-system
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



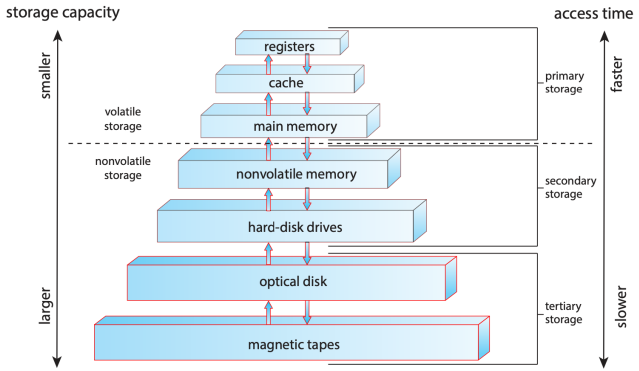
Interrupt Timeline



Interrupt-driven I/O cycle



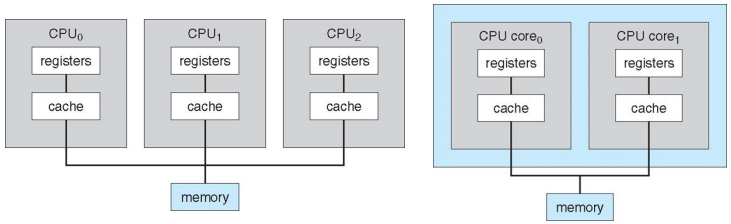
Storage Hierarchy



Storage Definitions

- Bit: contains a value of 0 or 1
- Byte: 8 bits. Fundamental unit of memory
- Word: multiple bytes (system dependent)
  - In modern laptops: 8 bytes
- 2^10 bytes: kilobyte
- 2^20 bytes: megabyte
- 2^30 bytes: gigabyte
- 2^40 bytes: terabyte

Multiprocessing Architectures



Symmetric Multiprocessor:
loosely coupled, multiple chips

Dual-core Processors:
tightly coupled, single chip

Operating System Operations

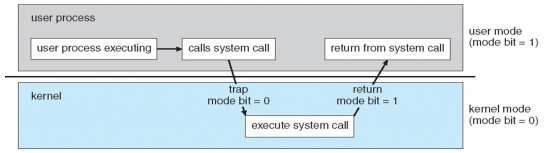
- **Interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap):**
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

Transition from User to Kernel Mode

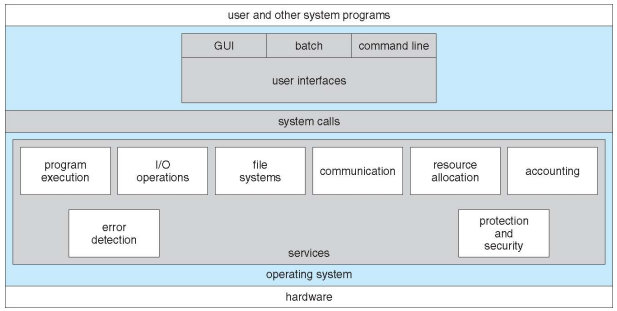
- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Operating-System Operations (Cont'd)

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user



A View of Operating System Services



System Calls

- Systems calls provide an interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

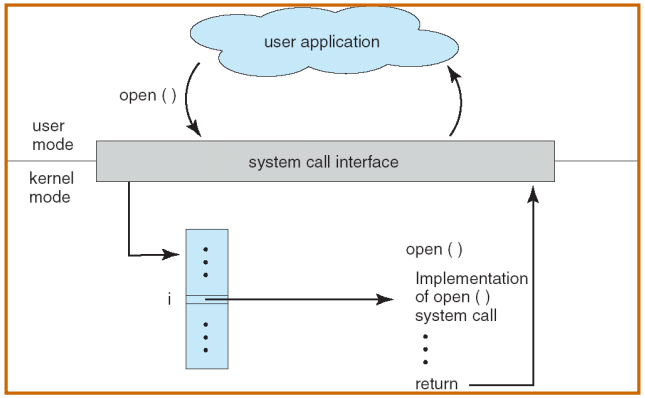
Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Affected by choice of hardware, type of system

✓ Design Goals

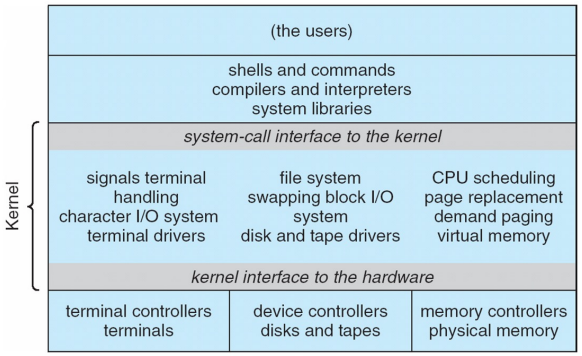
- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

API – System Call Interface – OS Relationship



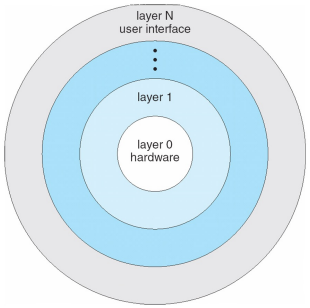
Traditional UNIX System Structure

Beyond simple but not fully layered



Layered Approach

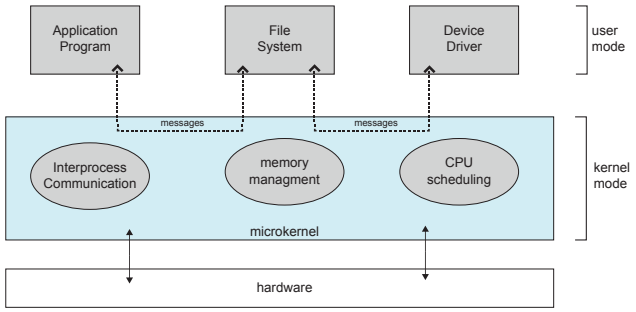
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Modules

- Many modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, etc

Microkernel System Structure



Virtual Machines

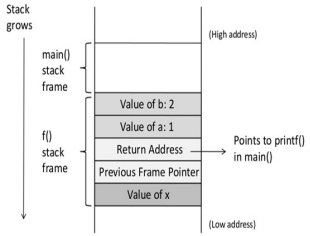
- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- Each **guest** provided with a (virtual) copy of underlying computer

Process VS. Program

- Program is ***passive*** entity stored on disk (***executable file***), process is ***active***
  - A program becomes a process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



- Arguments: beginning of the stack frame
- Return address
- Previous frame pointer
- Local variables

Processes and Memory

On process creation, the process is effectively given its own memory space

- Text: storage of code
- Data: global variables (preallocated space)
- Heap: dynamically allocated space
- Stack: local variable storage

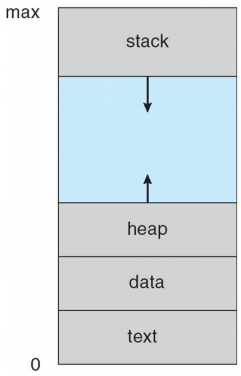
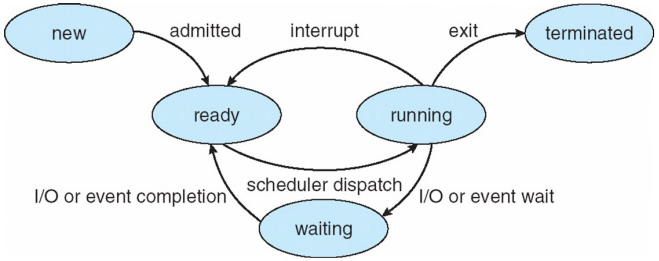


Diagram of Process State



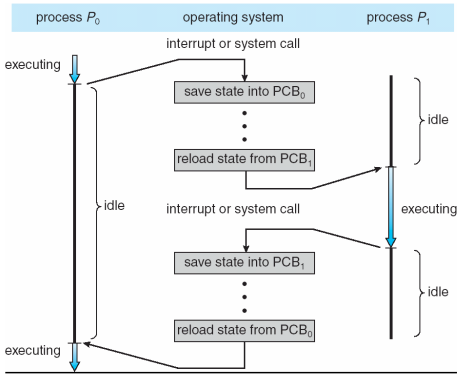
Process Scheduling

- Our goals are to:
  - Maximize CPU use
  - Give processes the CPU time that they need
- **Process scheduler** selects among available processes for next execution on CPU

Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

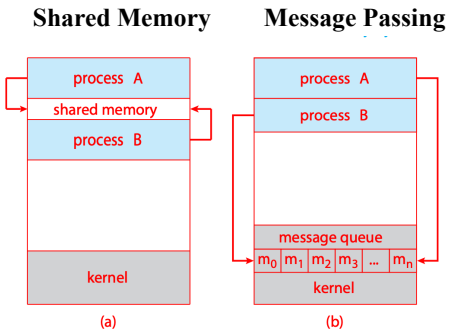
CPU Switching from One Process to Another



Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Communication Models



Pipes

- Act as a conduit allowing two processes on the same computer to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

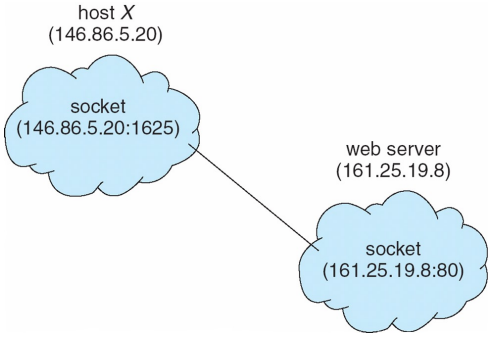
Producer-Consumer Problem

- Producer: process generates data through some mechanism
- Consumer: process uses data generated by another

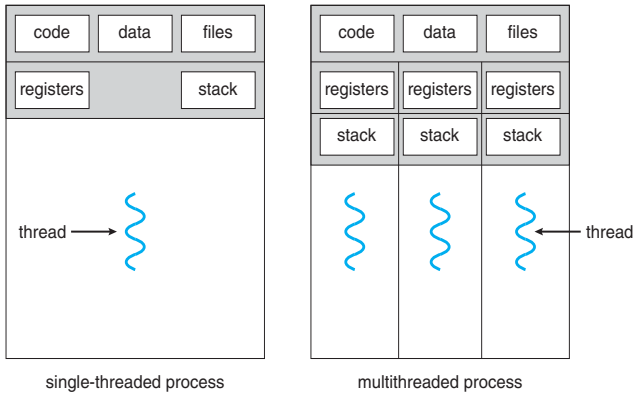
Types of Pipes

- Ordinary pipes: cannot be accessed from outside the process that created it.
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes: can be accessed without a parent-child relationship.

Socket Communication



Process vs Threads within a Process



Threads

- Memory is shared!
  - Program
  - Data (globals, heap, etc)
- Separate execution context
  - Program counter
  - Registers
  - Stack

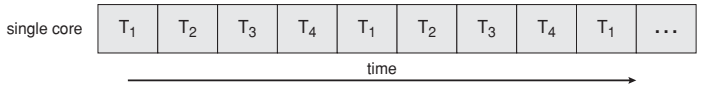
We refer to an execution context as a ***thread***

Benefits of Multi-Thread Programming

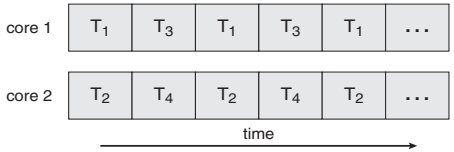
- **Responsiveness:** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing:** threads share resources of process, easier than shared memory or message passing
- **Economy:** cheaper than process creation, and thread switching has lower overhead than context switching
- **Scalability:** process can take advantage of multiprocessor architectures

Concurrency vs Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Amdahl's Law

Performance speedup with parallelization

- *S*: fraction of task that is necessarily serial (rest is parallel)
- N: number of processors/cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

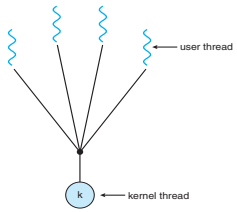
Multithreading Models

Relationship between user space threads and kernel threads. Options include:

- Many-to-One
- One-to-One
- Many-to-Many

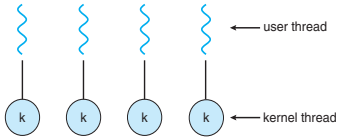
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



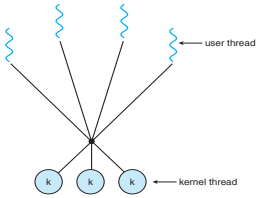
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the ***ThreadFiber*** package



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
  - API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads

Set up:

- Global variable (!): sum
- Function prototype: runner

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

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

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads

Parent:

- Create a single thread
  - Starts execution
- Join: parent waits for the child to exit

Child:

- Writes result to global variable

```
   /* get the default 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);

   printf("sum = %d\n",sum);
}

/* 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);
}
```


Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive.
- Otherwise, it is preemptive.
- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Pthreads

- Join requires specific thread ID
- If the thread has already quit by the time join() is called, then it returns immediately

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```


Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.

- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

Scheduling Criteria

A variety of metrics are possible:

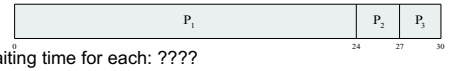
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

(1) First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order at time zero: P_1 , P_2 , P_3

The Gantt Chart for the schedule is:



- Waiting time for each: ????
- Average waiting time: ???

FCFS Scheduling (Cont'd)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

- Waiting time for all: $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

(2) Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal: gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the programmer to tell us

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

- Average waiting time = ????

Example of Shortest-Remaining-Time-First

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart

- Average waiting time = ??? msec

(3) Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - In Unix: smallest integer ≡ highest priority
  - Two versions:
    - Preemptive
    - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses, increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

- Average waiting time = ??? msec

Round Robin (RR) Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ Reduces to FIFO
  - q small $\Rightarrow$ All jobs must use multiple quanta to complete
    - q must be large with respect to context switch time, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

(5) Multilevel Queues

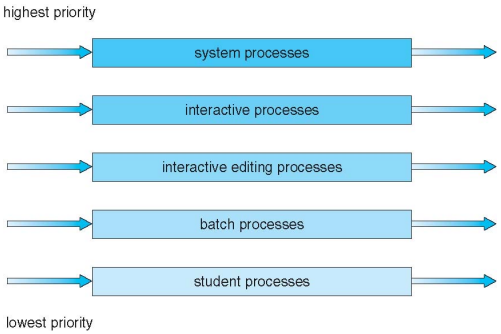
- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm. E.g.:
  - Foreground: RR
  - Background: FCFS

Multilevel Queues

Scheduling possibilities between the queues:

- Fixed priority scheduling
  - Serve all from foreground then from background
  - Possibility of starvation.
- Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes. For example:
  - 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



(6) Multilevel Feedback Queue

- A process can move between the various queues
  - Called: Aging
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

The Challenge of Concurrency

- Processes can execute concurrently
  - May be interrupted at any time, only partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

The Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a segment of code, called a **critical section**
  - Process may be changing common variables: updating a table, writing a file, etc
  - When one process is in the critical section, no other may be in its critical section
- ***Critical section problem:*** design a protocol for interaction and execution that enforces non-overlapping execution of critical sections

The Critical Section Problem

Critical section problem **-** One approach:

- Each process must ask permission to enter critical section in an **entry section** of code
- Process then executes critical section code
- Process then executes **exit section** of code
- Then, execute the **remainder section**

Critical Sections in Code

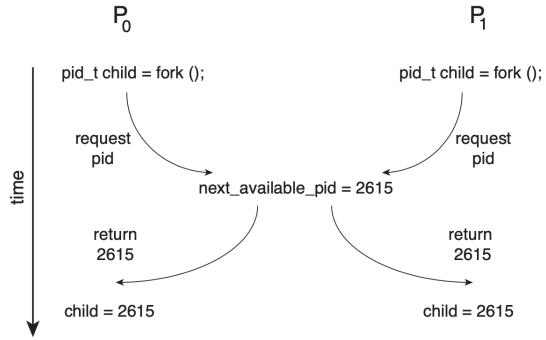
- General structure of process ***Pi***

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```


Properties of a Proper Solution to the Critical Section Problem

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then one of these processes must be allowed to proceed
3. **Bounded Waiting:** A process that is waiting to enter its critical section can only wait for a defined amount of time

Race Condition Example



Peterson's Solution: Two-Process Solution

- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **int turn;**
  - **Boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section.
  - **flag[i] = *true*** implies that process P_i is ready

Algorithm for Process P_i (other Process is P_j)

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);

     critical section

     flag[i] = false;
     remainder section
 } while (true);
```


Peterson's Solution

Provable that the three critical section requirements are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **flag[j] = false** or **turn = j**
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many modern microprocessors provide hardware support for implementing the critical section code
- Provide mechanism that implements a **lock**
  - Then, we use the lock to protect our critical sections:
    - Must "grab" the lock before starting to execute the critical section
    - After execution, must release the lock

Synchronization Hardware

- Uniprocessors: could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value simultaneously
  - Or swap contents of two memory words

Critical Section Solution: Using A Lock

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```


Mutex Locks

- Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires ***busy waiting***
  - This lock therefore called a ***spinlock***

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S: integer variable
  - Can only be accessed via two indivisible (atomic) operations: wait() and signal()
  - Originally called P() and V() by Dijkstra (1962)

Semaphores: Logical Definition

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

- Implementation guarantees safe access to S

Semaphores: Usage

- ***Binary semaphore***: integer value can range only between 0 and 1
  - Same as a mutex lock
- ***Counting semaphore***: integer value can range over an unrestricted domain
  - Can solve a wider range of synchronization problems
  - But, can still implement a binary semaphore

Example (1): Bounded-Buffer Problem

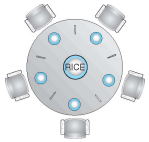
- Buffer that contains n entries
- Data structure is shared by both producers and consumers
- Must protect the buffer from being accessed by more than one process at once
- Want to avoid busy-waiting in two cases:
  - Producer busy-waiting if the buffer has no room for new items
  - Consumer is busy-waiting if the buffer has no items

Example (2): Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set; they do ***not*** perform any updates
  - Writers: can both read and write
- Problem:
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered … all involve some form of priorities

Example (3): Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- They don't interact with their neighbors
  - Occasionally each tries to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data are:
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Modeling Resource Contention

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **Request**
  - **Use** (exclusive)
  - **Release**

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Conditions for Deadlock

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: a process is holding onto a resource (R) while it is waiting for some other resource that can only be released after R is released

Resource Allocation Graph

- Vertices are of two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge**: directed edge $P_i \rightarrow R_j$
- **Assignment edge**: directed edge $R_j \rightarrow P_i$

Deadlock

How do we know if we have a deadlock?

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - If only one instance per resource type, then deadlock
  - If several instances per resource type, possibility of deadlock

Dealing with Deadlocks

- **Method 1:** Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- **Method 2:** Allow the system to enter a deadlock state and then recover
- **Method 3:** Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Approach: we don't allow one of the four **necessary conditions** to hold

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular wait

Deadlock Prevention

- Kernel can take preventative steps
  - Resource utilization could be poor
- Or the application programmer can take explicit steps
  - E.g., ordering of lock operations
  - Dealing with preemption
  - This approach relies on programmers doing the right thing
    - Generally, this is a bad idea…

Deadlock Avoidance

- Deadlock prevention techniques place a lot of restrictions on what can be done
  - In particular: allocation decisions are made using uniformly applied rules
- Next approach (avoidance): dynamically make allocation decisions on a case-by-case basis
  - Only allow an allocation to proceed if there is no opportunity in the current system for deadlock

Deadlock Avoidance

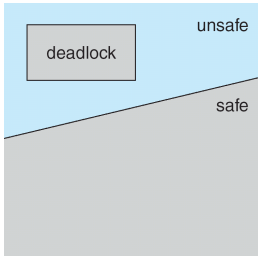
Process Model:

- Each process must declare up front the maximum number of resources of each type that it ***may need*** to complete execution
- Then, during execution, the process may request those resources as they are actually needed
  - Must respect the declared needs at the start

System State

Three possible situations:

- **Deadlock**: a circular wait has happened
- **Safe**: all processes can complete without deadlock occurring
- **Unsafe**: deadlock has not occurred, but if the right set of needs are requested, then deadlock will happen



System Allocation Algorithm

- Goal: always stay in a safe state
- When a new request is made by a process:
  - Kernel tests whether the new state will be safe or not
  - If safe, then allocation is allowed
  - If unsafe, then the process is placed in a waiting queue until a safe state can be achieved

Avoidance Algorithms

- All resources are single-instance:
  - We can just look at the resource allocation graph to determine whether a cycle can happen
- Multiple instances of some resources:
  - Use the **Banker's Algorithm** to determine safe vs unsafe

Banker's Algorithm

- Multiple instances of each resource
  - These are interchangeable instances
- Each process must claim the maximum use of resources before any requests can be made
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them and terminate in a finite amount of time

Banker's Example III

	Allocation	*Max*	*Available*
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

Banker's Example IV

New request by Process 1: 1,0,2

- Will we be in a safe state?

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Banker's Example IV

- What about (3,3,0) by T_4?
  - ✓ cannot be granted, since the resources are not available
- What about (0,2,0) by T_0?
  - ✓ cannot be granted, even though the resources are available, since the resulting state is unsafe

Programs and Memory

- To be executed, a program must be brought from disk into memory and placed within a process
- Main memory and registers are the only storage that the CPU can access directly
- In order for the CPU to manipulate data, it must first be brought from memory into a register
- Memory unit only sees a stream of:
  - Address + read requests, or
  - Address + data and write requests

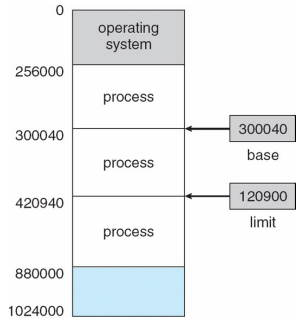
Multiple Processes in Memory

- Processes have (mostly) distinct parts of the physical memory
- Memory management is all about allocating processes to physical memory
- Our challenge is to:
  - Do this efficiently
  - Make good use of this valuable resource

Approach: Contiguous Allocation

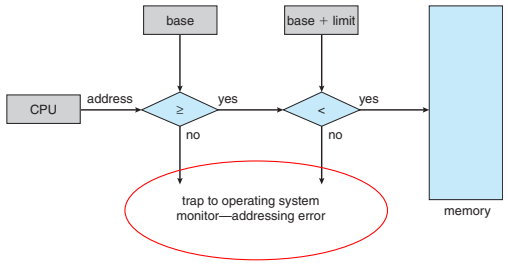
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection

Process is only allowed to access its own section of memory



Logical vs. Physical Address Space

Two different notions of address:

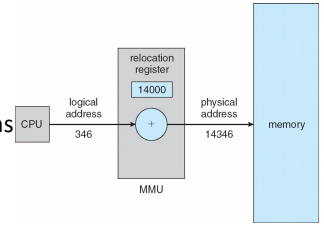
- **Logical address**:
  - Generated by the CPU; also referred to as **virtual address**
  - **Logical address space** is the set of all logical addresses generated by a program
- **Physical address**:
  - Address seen by the memory unit
  - **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that maps a virtual address to a physical one
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
- First approach:
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**

Using a Relocation Register

- Relocation register is part of the Process Control Block
  - Set when the process is brought onto the CPU
- Program "thinks" entirely in terms of the logical address
- Logical address is added to the value of the relocation register to generate the physical address



Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

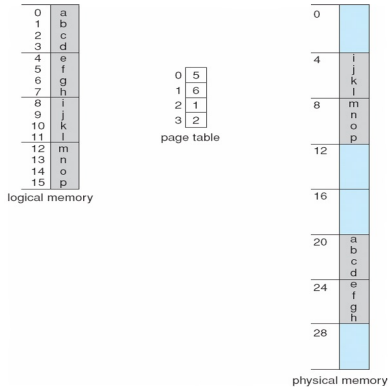
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Paging (Cont'd)

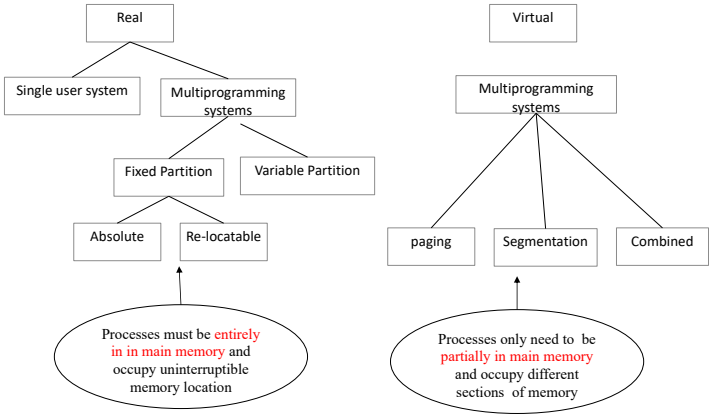
- Calculating internal fragmentation
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Paging (Cont'd)

We have no external fragmentation: any free frame can be allocated to a process that needs it

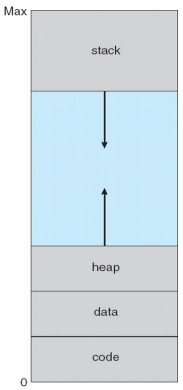
- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - ? pages + ? bytes
  - Internal fragmentation = ? bytes

Different Memory Organization



Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Practice exam

Q1: What does the following command mean?
chmod 777 readme.txt

Advantages of Virtual Memory

- Greater number of processes can execute concurrently because only part of the processes needs to be in main memory
- External fragmentation is eliminated in paging systems because every frame can be assigned to a process. In segmentation systems is still possible to have free memory where an entire segment cannot fit.
- Program can be larger in size than available main memory

Practice exam

Q2: What is a race condition?

Practice exam

Q3.

Consider the following processes with specified arrival and burst times:

Process	Arrival Time	Burst Time
P1	0	10
P2	2	4
P3	4	1
P4	5	4

Assume that the **Shortest-Remaining-Time-First** scheduling algorithm is being used.

What is the average wait time for the processes?

Practice exam

Q4: What are deadlock conditions?

Practice exam

Q5. Suppose that a program with a single core has a running time of T time units (it takes T time units to execute). If we instead allocate K cores to the program, what is the best case running time?

(A) T/K
(B) K*T
(C) K/T
(D) T
(E) K
(F) Answer not shown

Practice exam

Q6 Consider a simple paged memory system, with page sizes of 2^6 and a page table as follows:

0x5
0x7
0xC
0x2

Given a logical address of 0xB6, which physical address is accessed?

(A) 0x36 (B) 0x1F6 (C) 0x336
(D) 0xC36 (E) Answer not shown