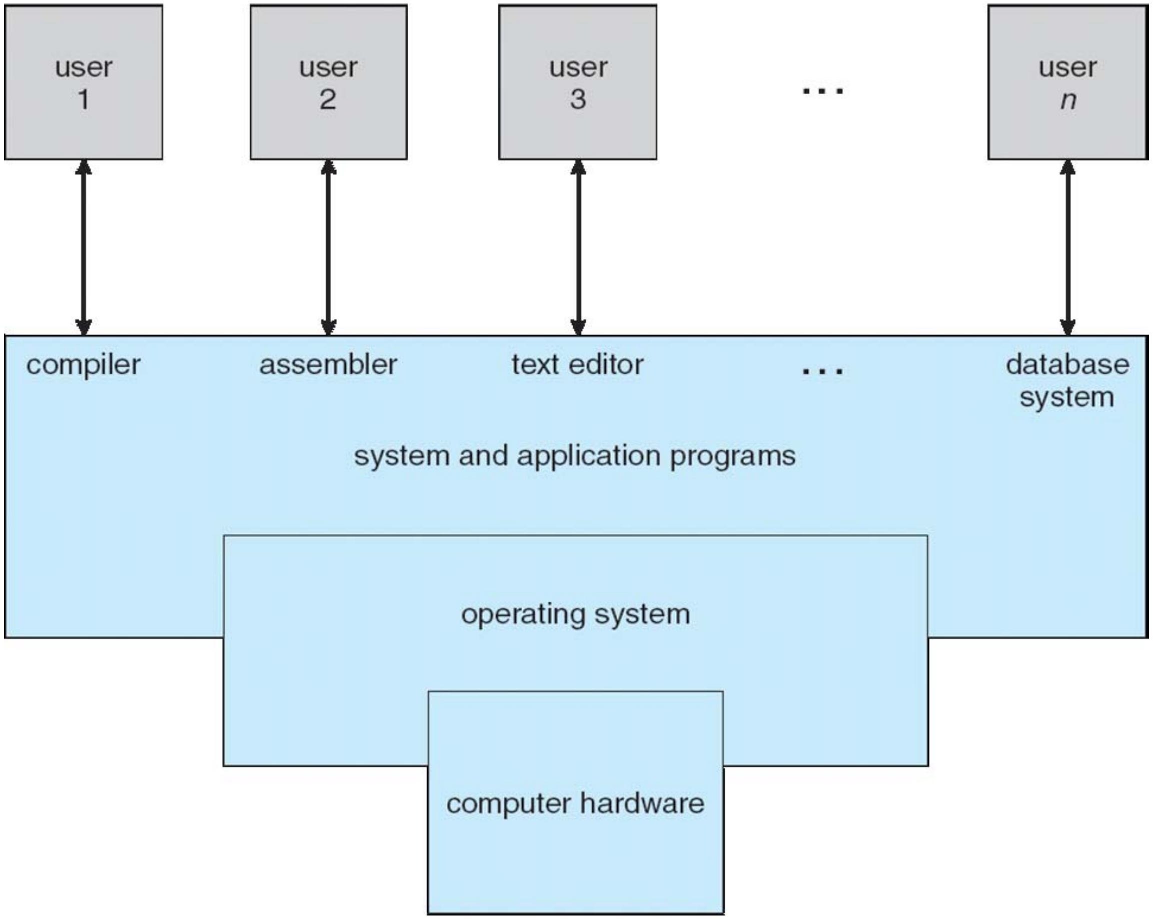
Final Exam Review

Computer System Structure

- Computer system can be divided into four components:
  - 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
    - People, machines, other computers

Four Components of a Computer System

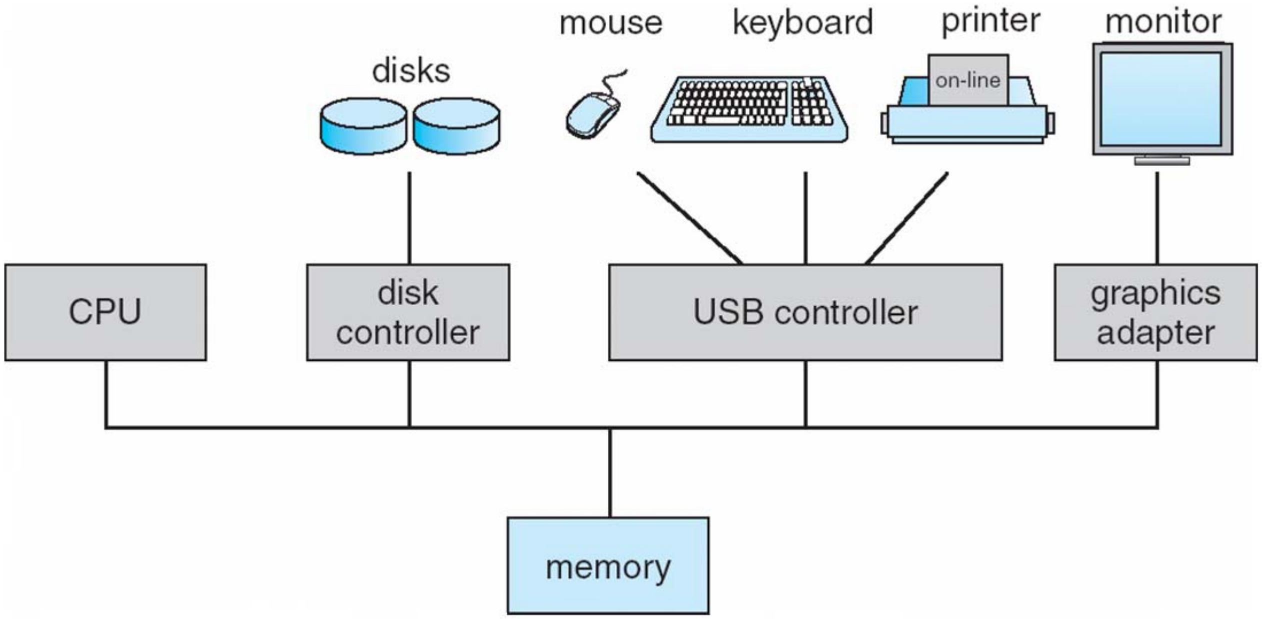


Operating System Definition

- OS is a **resource allocator**
  - Manages all 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

Computer System Organization

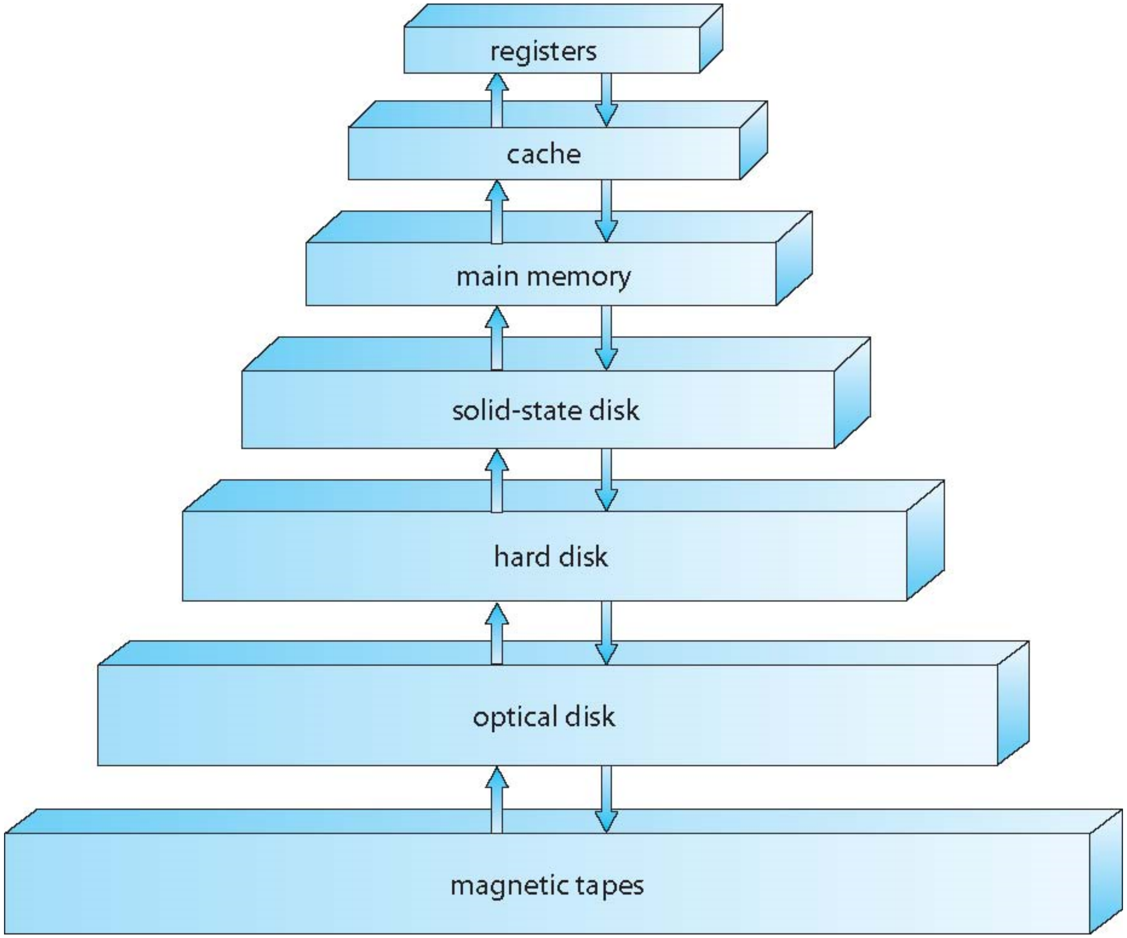
- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles. A memory controller synchronizes access to the memory.



Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy



Caching

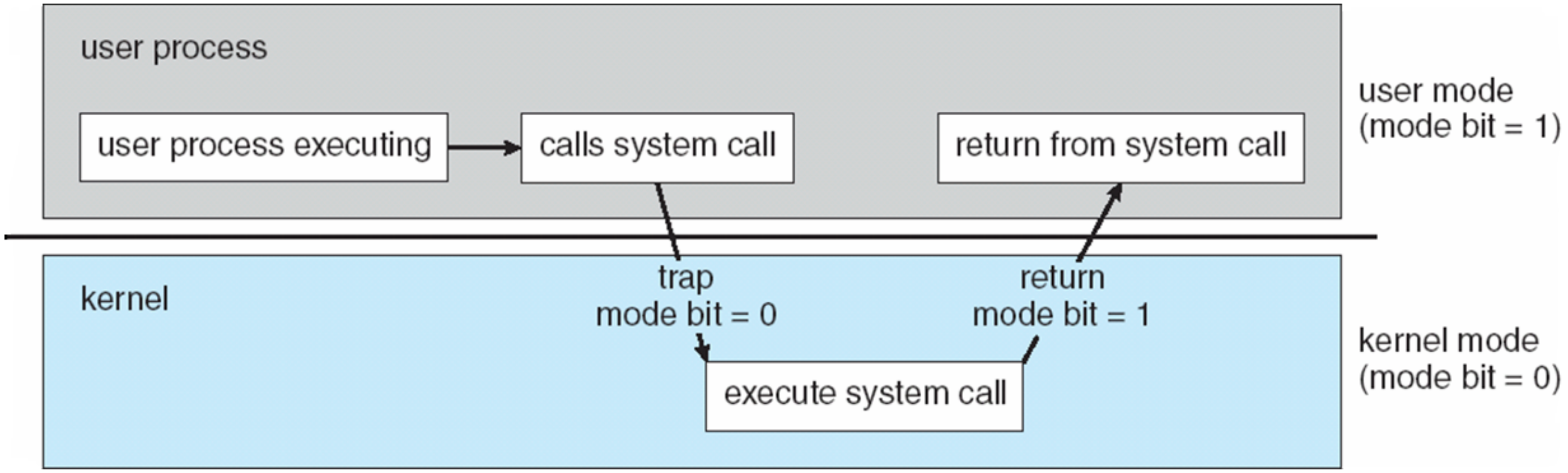
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Operating System Structure

- **Multiprogramming** (**Batch system**) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing** (**multitasking**) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - 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

Operating-System Operations (cont.)

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware (e.g., CS register in CPU)
    - 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



Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** **(CLI)**, **Graphics User Interface** **(GUI)**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

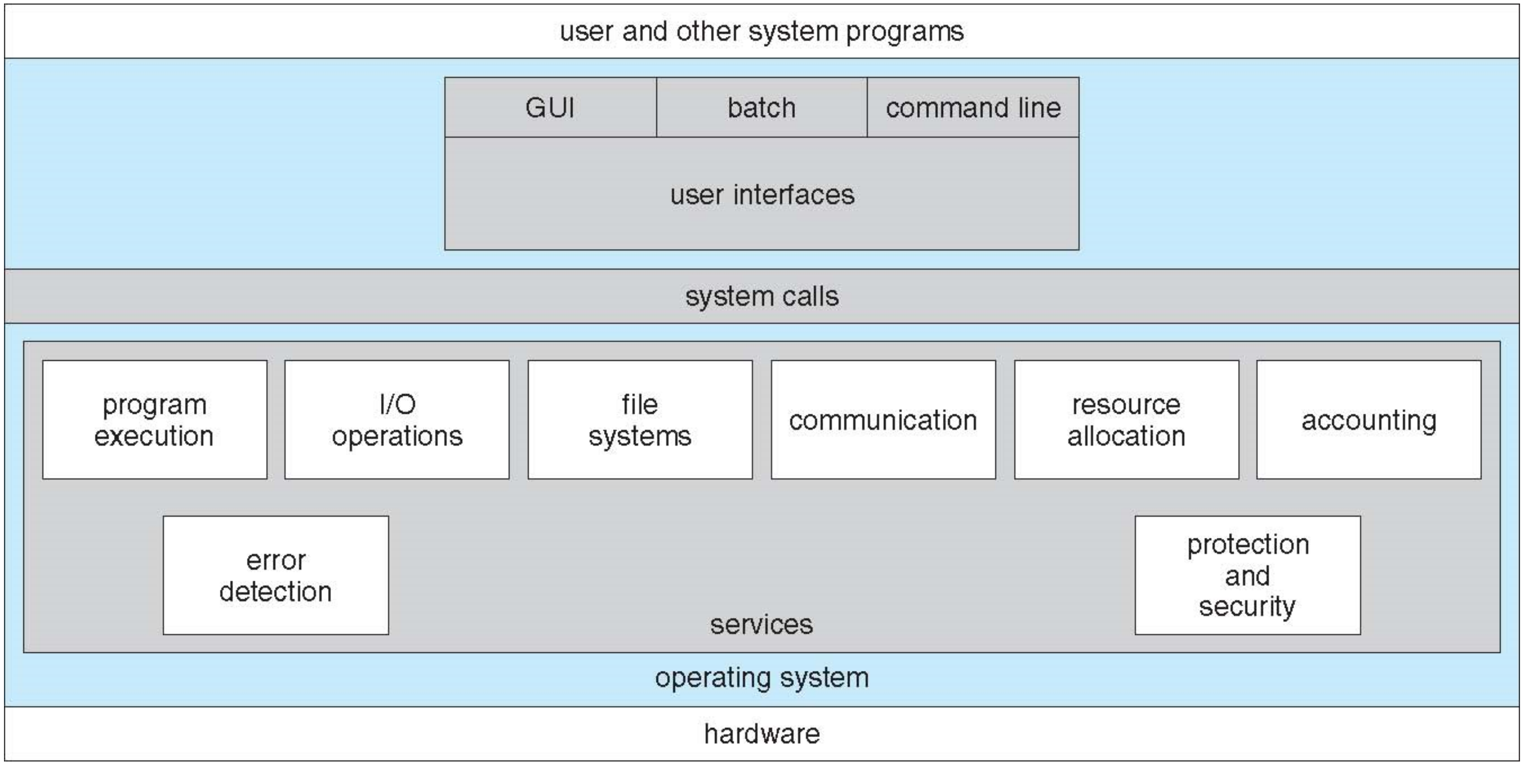
Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user’ s and programmer’ s abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources -   CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



user and other system programs

GUI | batch | command line

user interfaces

system calls

program execution

I/O operations

file systems

communication

resource allocation

accounting

error detection

protection and security

services

operating system

hardware

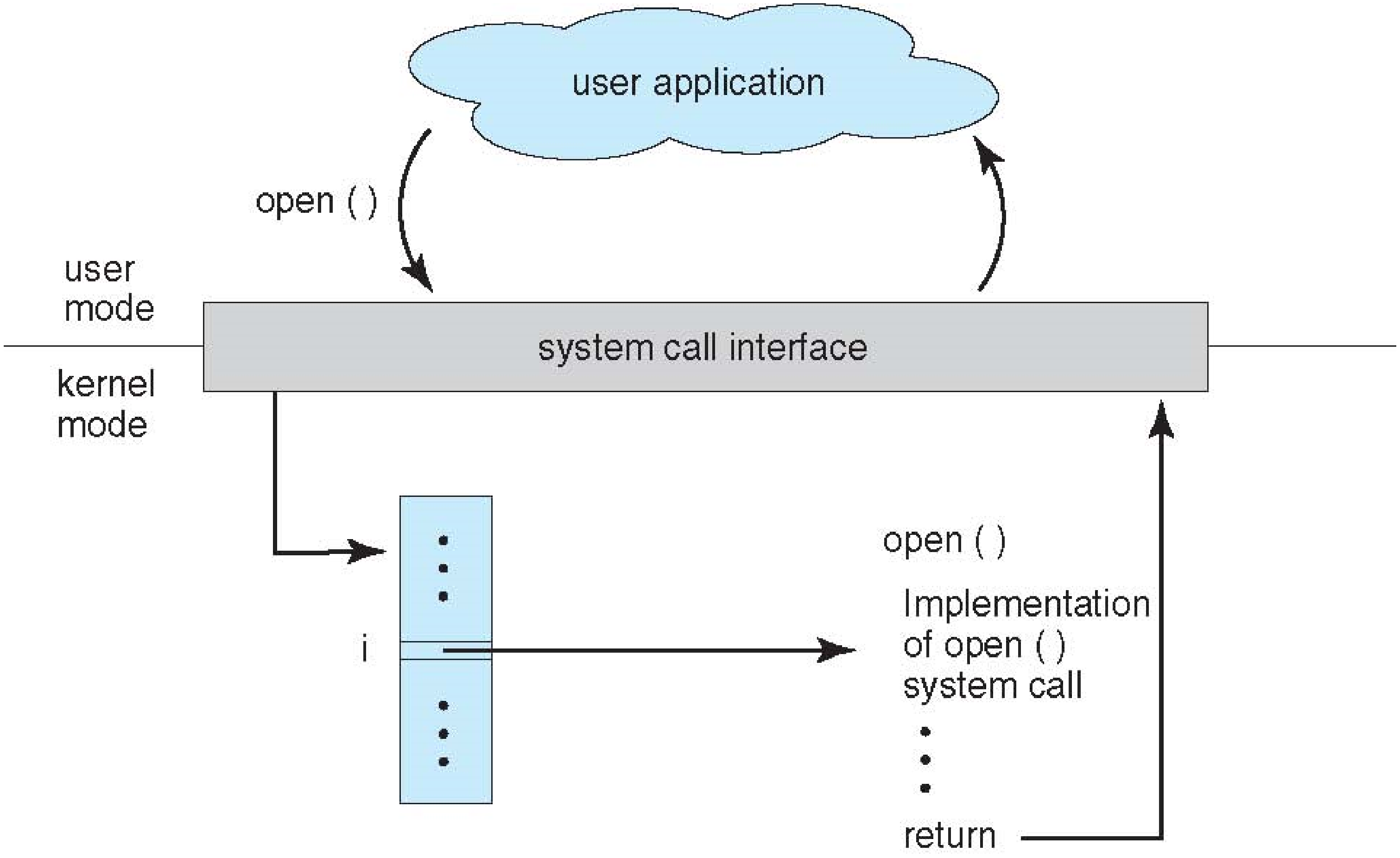
System Calls

- Programming 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 Programming Interface (API)** rather than direct system call use
- Three most common APIs are Windows 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)

System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



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
- Start the design by defining goals and specifications
- Highest level: affected by choice of hardware, type of system
- The requirements can be divided into **User** and **System** 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

Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?

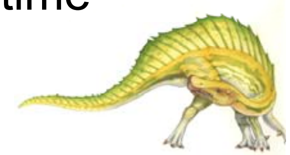
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of **software engineering**

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstrcation
  - Microkernel -Mach

Process Concept

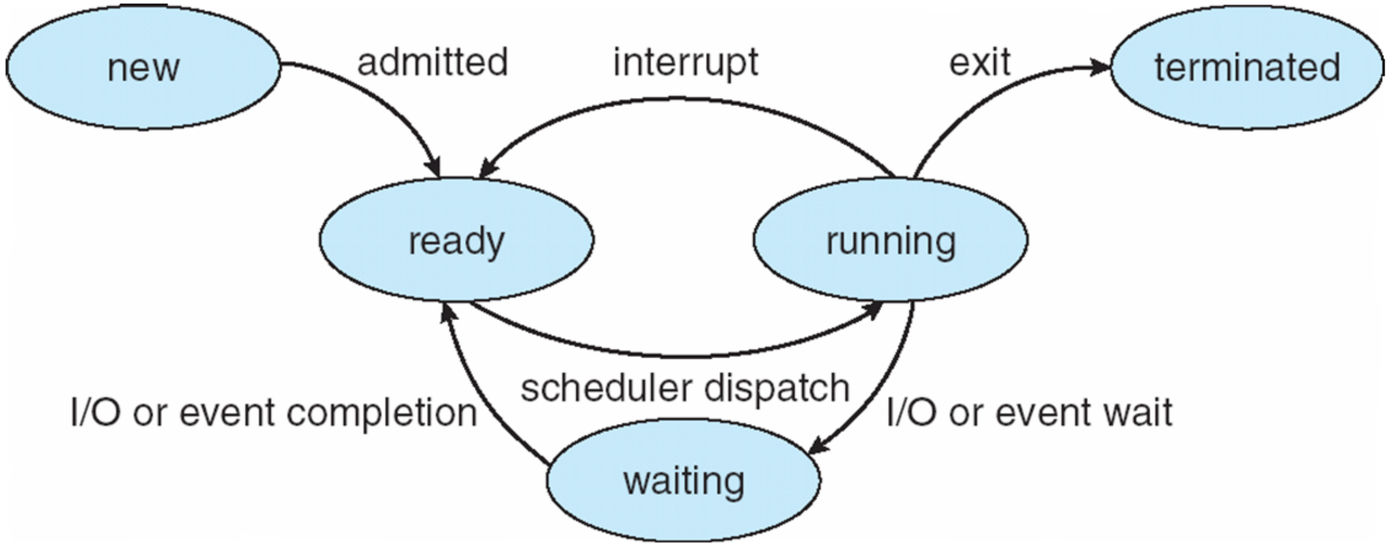
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time



Process State

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

Diagram of Process State

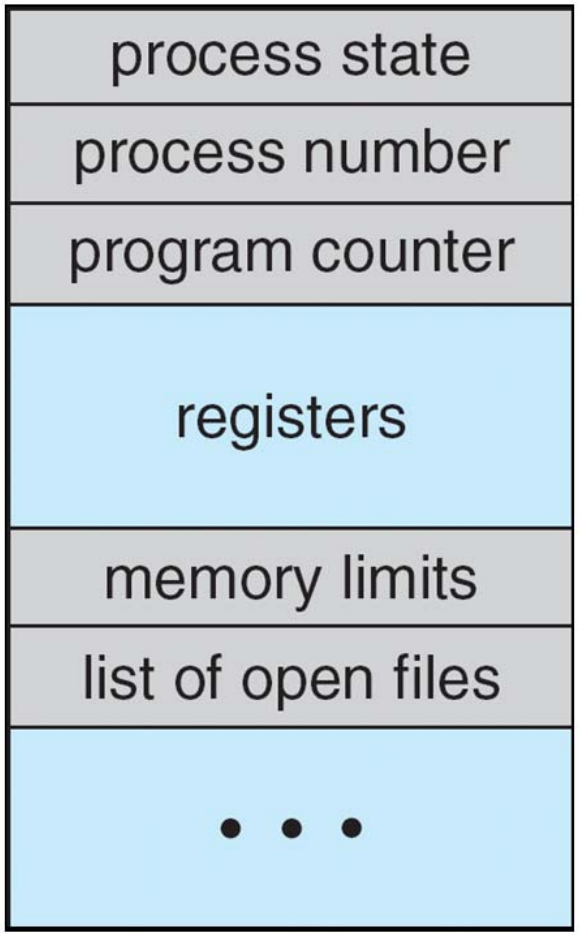


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



process state

process number

program counter

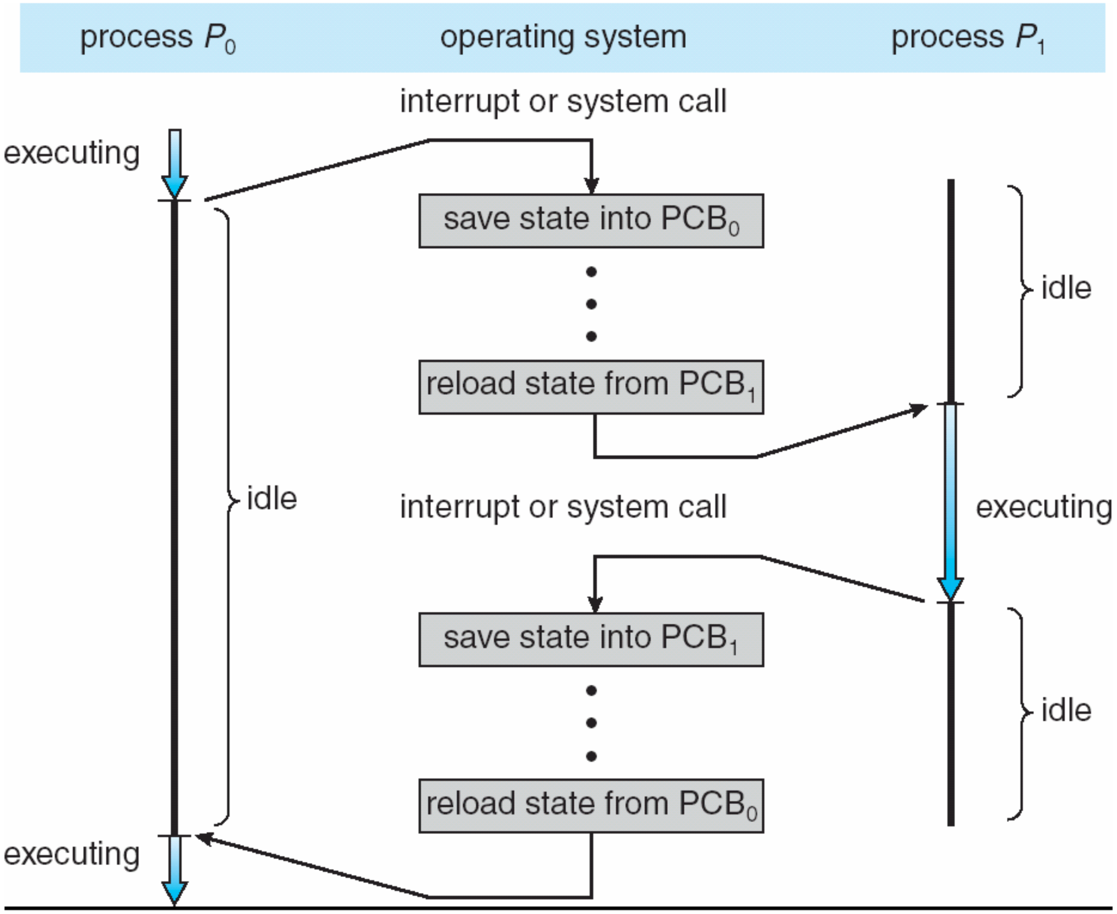
registers

memory limits

list of open files

• • •

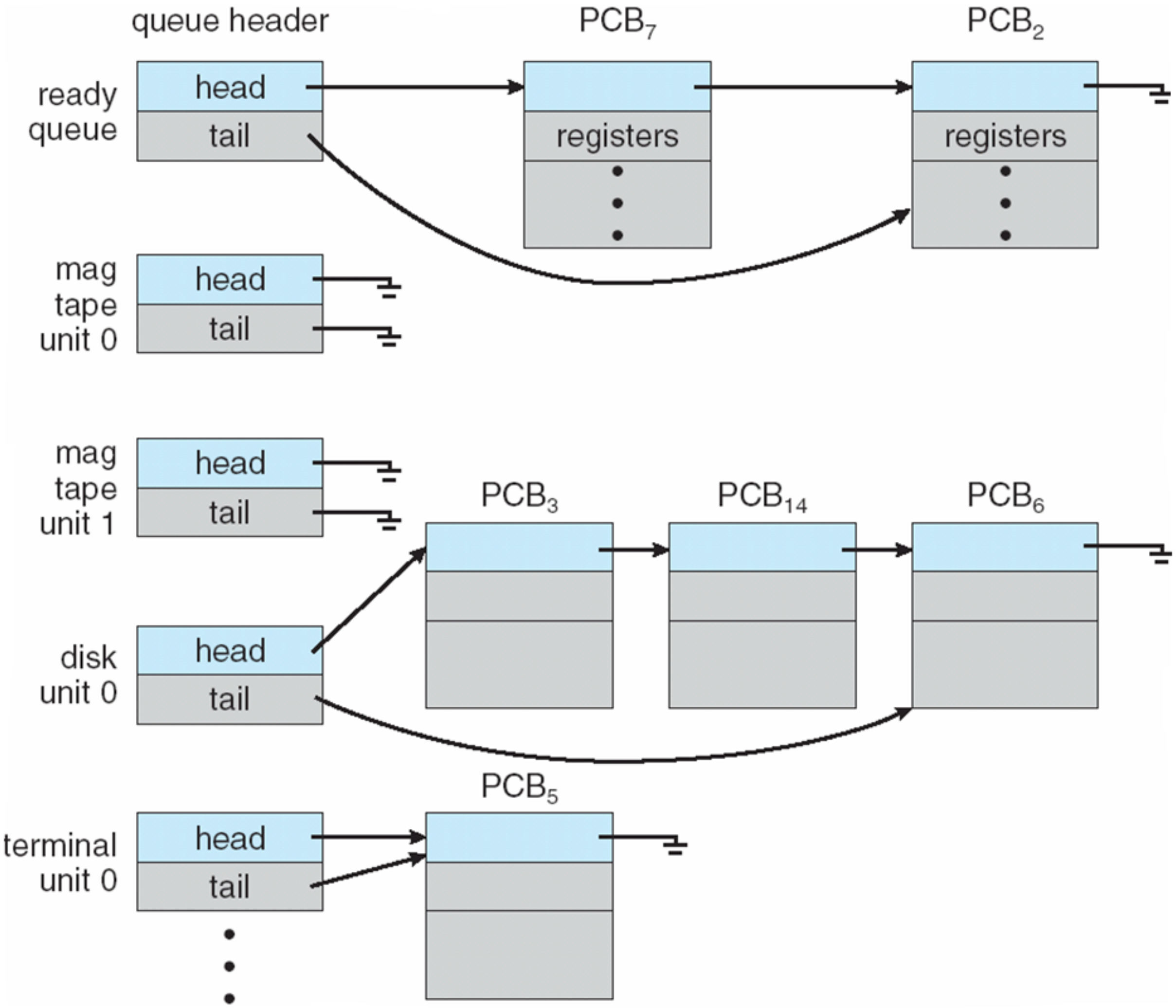
CPU Switch From Process to Process



Process Scheduling

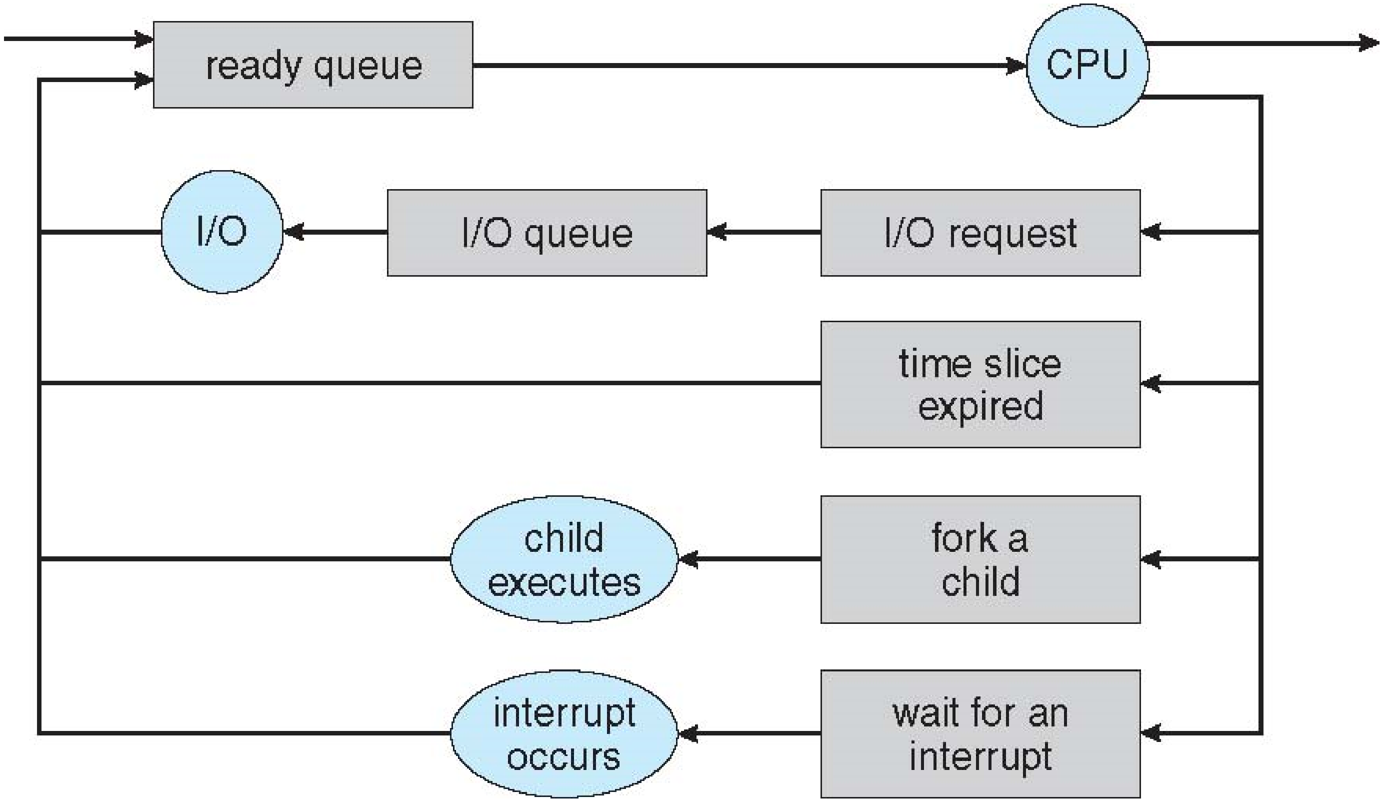
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing 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

- **Queueing diagram** represents queues, resources, flows



Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **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
- Long-term scheduler strives for good ***process mix***

Operations on Processes

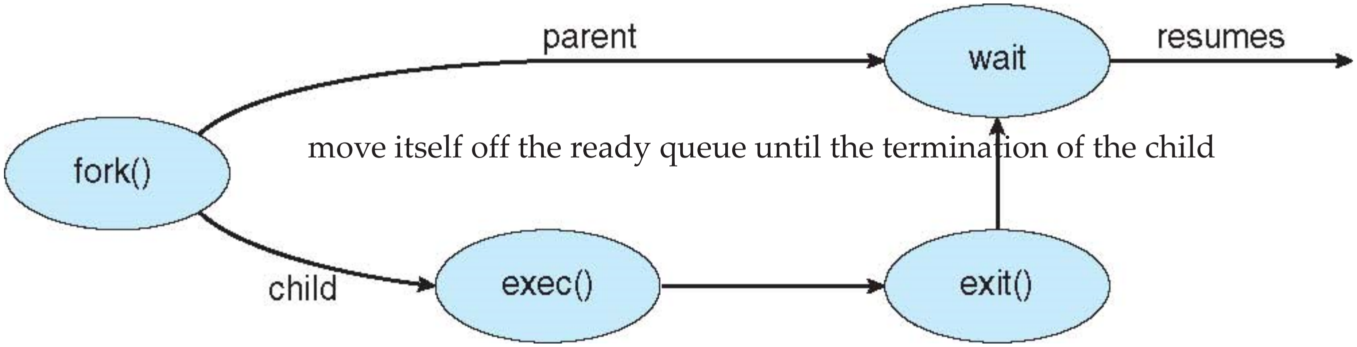
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process 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
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

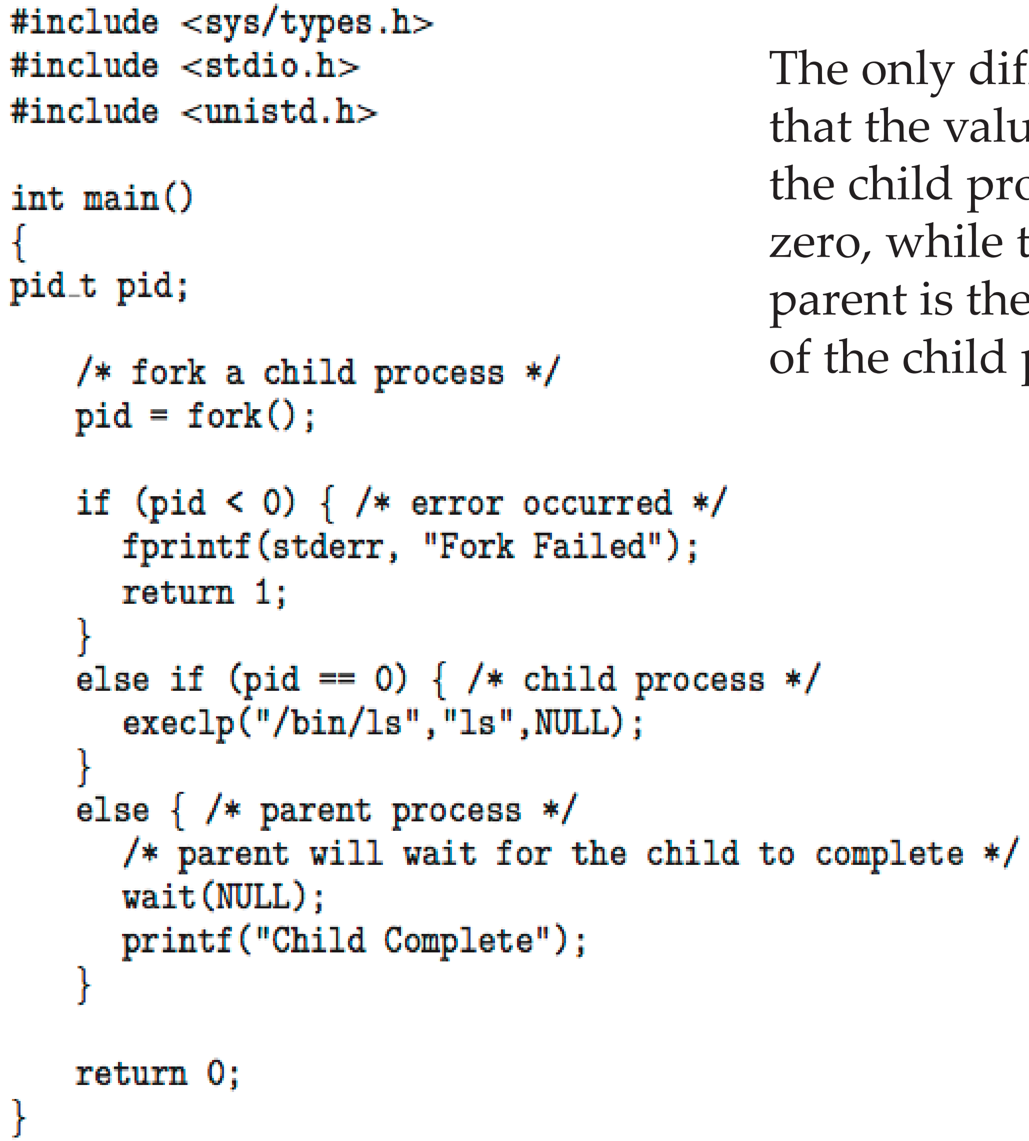
Process Creation (Cont.)

- Address space
  - Child duplicate of parent (has the same program as the parent)
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process. The new process consists of a copy of the address space of the original process.
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



move itself off the ready queue until the termination of the child

C Program Forking Separate Process



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 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");
    }

    return 0;
}
```

The only difference is that the value of pid for the child process is zero, while that for the parent is the actual pid of the child process.

Process Termination

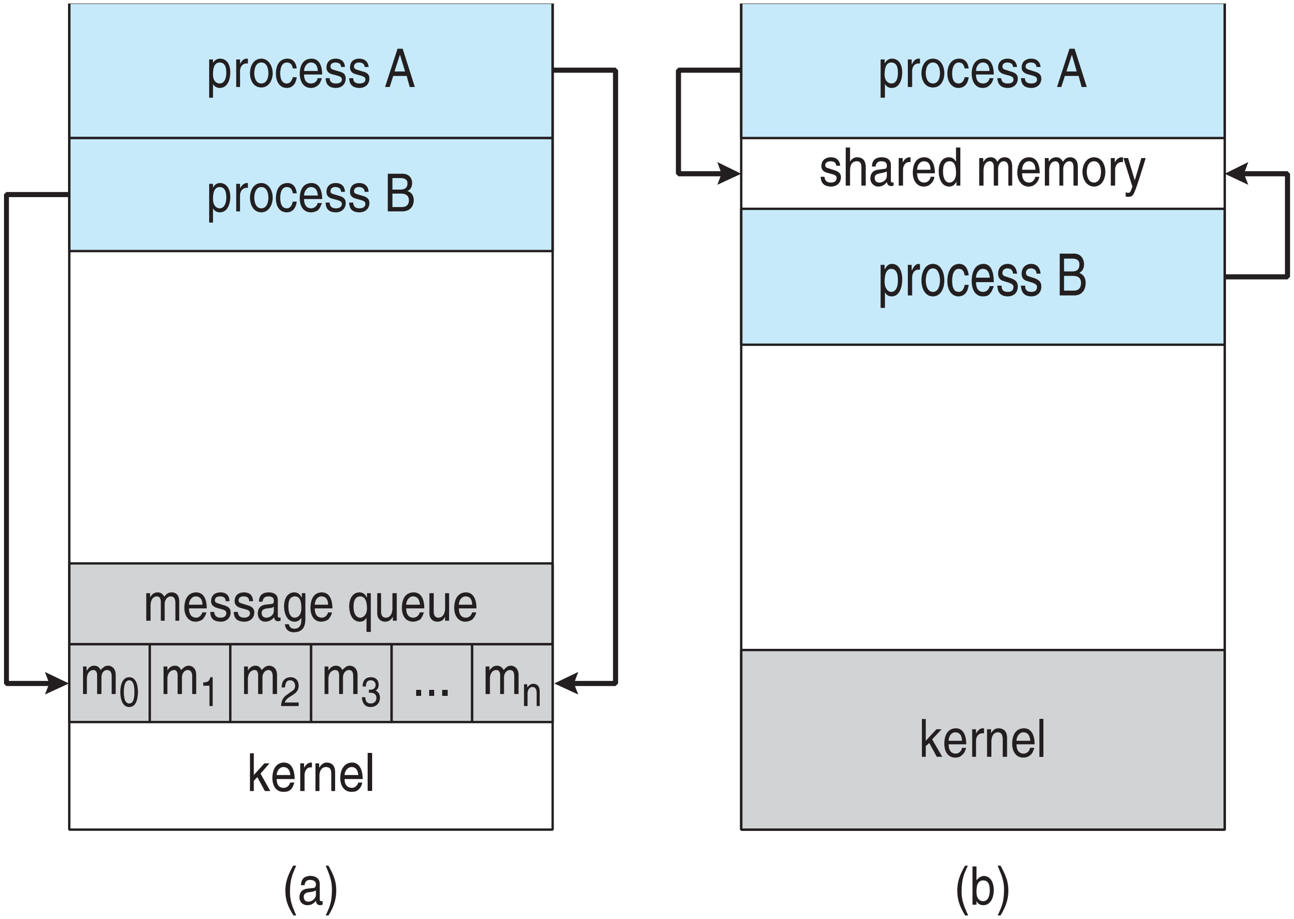
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing (shared files)
  - Computation speedup (parallel subtasks)
  - Modularity (system function divided into separate processes)
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

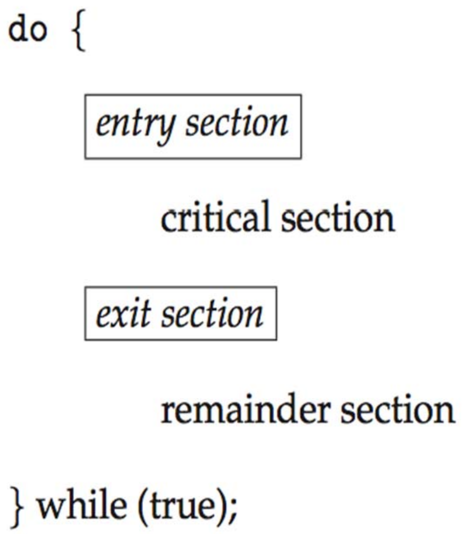
S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 – 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this problem
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i



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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $\boldsymbol{P_i}$ 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 the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson’ s Solution

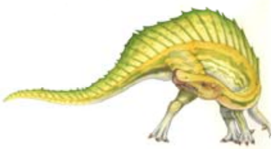
- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose 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 $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
               critical section
    flag[i] = false;
               remainder section
 } while (true);
```

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- 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
  - Or swap contents of two memory words



test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- 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**

acquire() and release()

-
  ```
  acquire() {
      while (!available)
         ; /* busy wait */
      available = false;
  }
  ```

-
  ```
  release() {
      available = true;
  }
  ```

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

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ to happen before $\boldsymbol{S_2}$

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

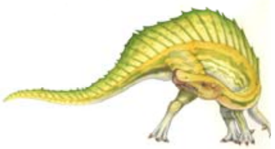
- Can implement a counting semaphore ***S*** as a binary semaphore

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
       P0                       P1
wait(S);                  wait(Q);
wait(Q);                  wait(S);
 ...                        ...
signal(S);                 signal(Q);
signal(Q);                 signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
       /* produce an item in next_produced */
      ...
    wait(empty);
    wait(mutex);
       ...
       /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
 Do {
      wait(full);
      wait(mutex);
          ...
      /* remove an item from buffer to next_consumed */
          ...
      signal(mutex);
      signal(empty);
          ...
      /* consume the item in next consumed */
          ...
  } while (true);
```

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
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

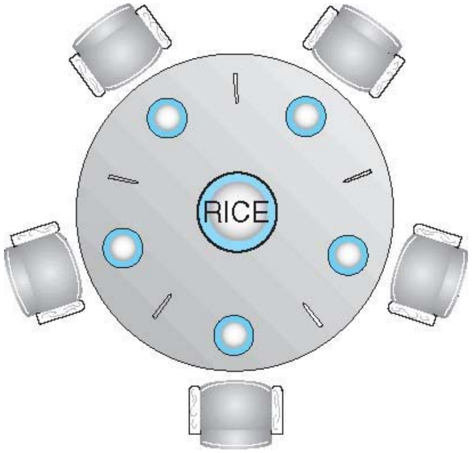
```
      do {
             wait(rw_mutex);
                ...
             /* writing is performed */
                ...
          signal(rw_mutex);
    } while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
       wait(mutex);
       read_count++;
       if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
       /* reading is performed */
        ...
    wait(mutex);
       read count--;
       if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try 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
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                    //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                      //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Monitors

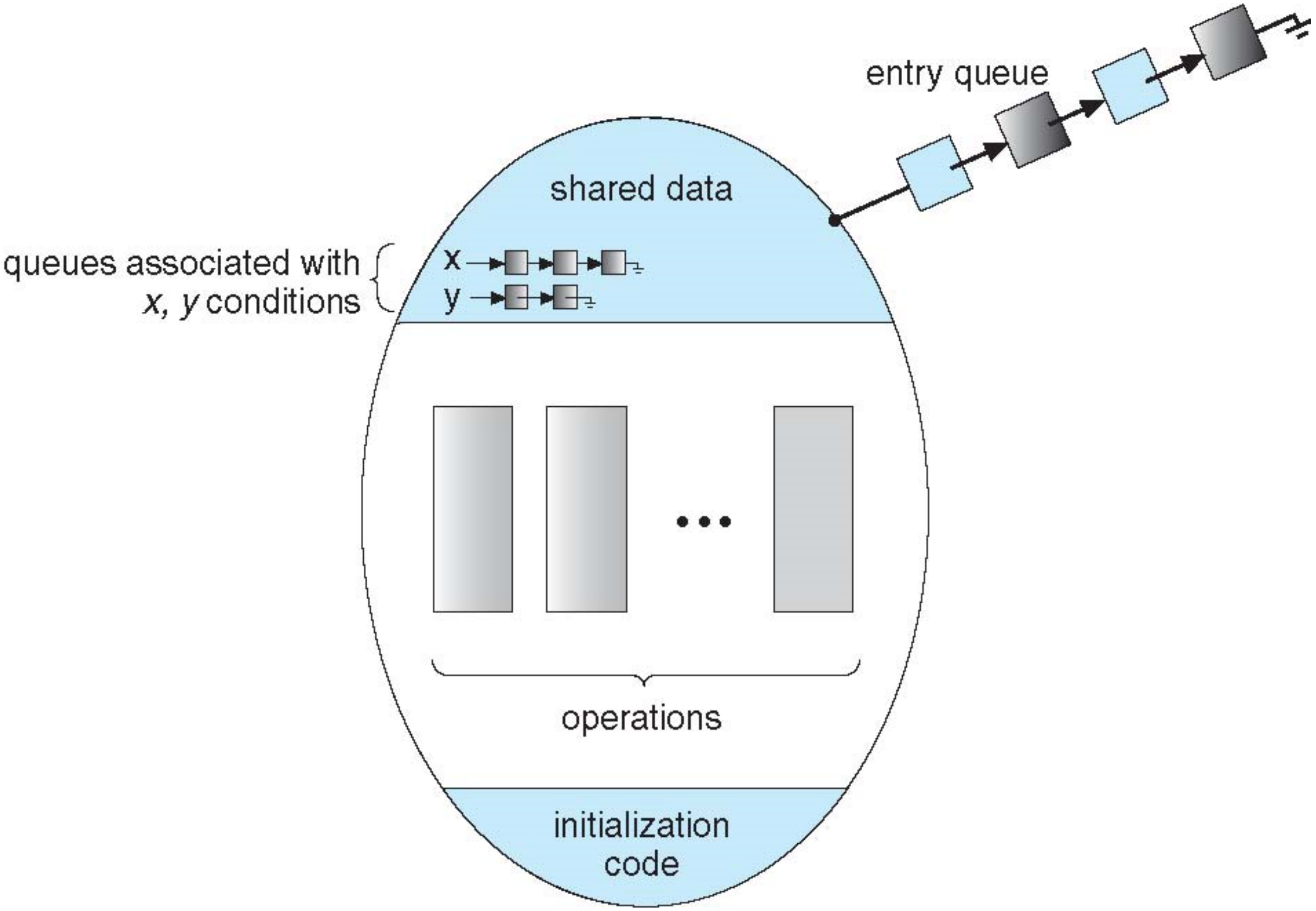
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
}

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

Solution to Dining Philosophers (Cont.)

```
   void test (int i) {
           if ((state[(i + 4) % 5] != EATING) &&
           (state[i] == HUNGRY) &&
           (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
            self[i].signal () ;
           }
   }

       initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
        }
}
```

Solution to Dining Philosophers (Cont.)

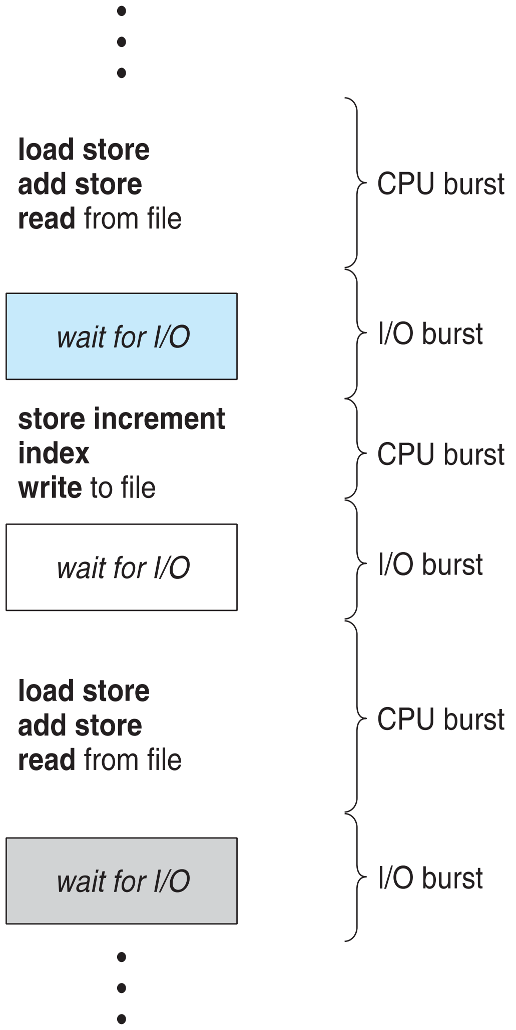
- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
          DiningPhilosophers.pickup(i);

               EAT

          DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

CPU Scheduling - Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



load store
add store
read from file

CPU burst

wait for I/O

I/O burst

store increment
index
write to file

CPU burst

wait for I/O

I/O burst

load store
add store
read from file

CPU burst

wait for I/O

I/O burst

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **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, not output (for time-sharing environment)

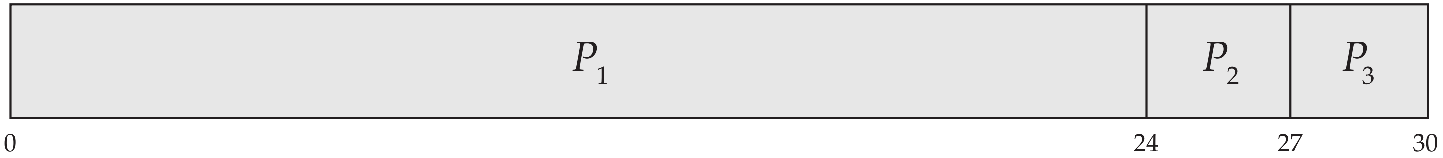
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

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: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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 user

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. $t_n =$ actual length of n^{th} CPU burst
  2. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n .$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - 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

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue

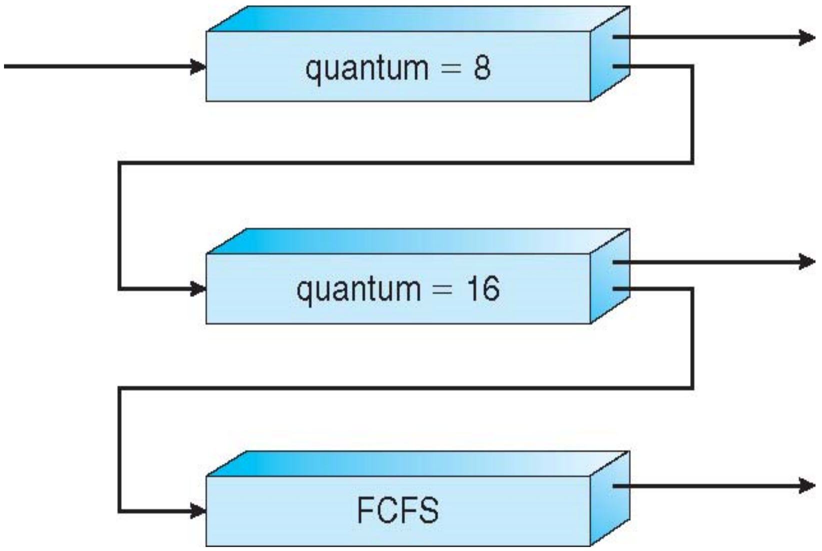
- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., 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; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Real-Time Scheduling Is Not Fair

- Main goal of an RTOS scheduler is to meet task deadlines, instead of throughput, latency and response time, etc.

- If you have five homework assignments and only one is due in half an hour, you work on that one first

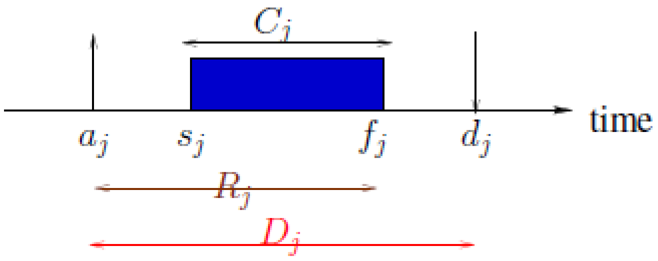
- Fairness does not help you meet deadlines

Real-Time Scheduling Policies

- Important Questions for real-time scheduling
  - What scheduler is guaranteed to meet all task deadlines for a given workload?
  - Given a scheduler, how do we know that it will work for a given workload?
  - Is there an “optimal” scheduler independent of workload?

Timing parameters of a job J_j

- Arrival time (a_j) or release time (r_j) is the time at which the job becomes ready for execution
- Computation (execution) time (C_j) is the time necessary to the processor for executing the job without interruption.
- Absolute deadline (d_j) is the time at which the job should be completed.
- Relative deadline (D_j) is the time length between the arrival time and the absolute deadline.
- Start time (s_j) is the time at which the job starts its execution.
- Finishing time (f_j) is the time at which the job finishes its execution.
- Response time (R_j) is the time length at which the job finishes its execution after its arrival, which is f_j - a_j.



Feasibility of Schedules and Schedulability

- A schedule is feasible if all jobs can be completed according to a set of specified constraints.

- A set of jobs is schedulable if there exists a feasible schedule for the set of jobs.

- A scheduling algorithm is optimal if it always produces a feasible schedule when one exists (under any scheduling algorithm).

Evaluating A Schedule

- For a job J_j:
- Lateness L_j: delay of job completion with respect to its deadline.

$$L_j = f_j - d_j$$

- Tardiness E_j: the time that a job stays active after its deadline.

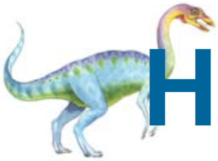
$$E_j = max\{0, L_j\}$$

- Laxity (or Slack Time)(X_j): The maximum time that a job can be delayed and still meet its deadline.

$$X_j = d_j - a_j - C_j$$

Metrics of Scheduling Algorithms (for Jobs)

- Given a set ***J*** of *n* jobs, the common metrics are to minimize:
  - Average response time: $\sum_{J_j \in \mathbb{J}} \frac{f_j - a_j}{|\mathbb{J}|}$
  - Makespan (total completion time): $\max_{J_j \in \mathbb{J}} f_j - \min_{J_j \in \mathbb{J}} a_j$
  - Total weighted response time: $\sum_{J_j \in \mathbb{J}} w_j(f_j - a_j)$
  - Maximum latency: $L_{\max} = \max_{J_j \in \mathbb{J}} (f_j - d_j)$
- Number of late jobs: $N_{late} = \sum_{J_j \in \mathbb{J}} miss(J_j),$ where *miss(J$_j$)* = *0* if $f_j <= d_j$, and *miss(J$_j$)* = *1* otherwise.



Hard/Soft Real-Time Systems

- Hard Real-Time Systems
  - If any hard deadline is ever missed, then the system is incorrect
  - The tardiness for any job must be 0
  - Examples: Nuclear power plant control, flight control
- Soft Real-Time Systems
  - A soft deadline may occasionally be missed
  - Various definitions for “occasionally”
    - minimize the number of tardy jobs, minimize the maximum lateness, etc.
  - Examples: Telephone switches, multimedia applications
- We mostly consider hard real-time systems in this lecture.

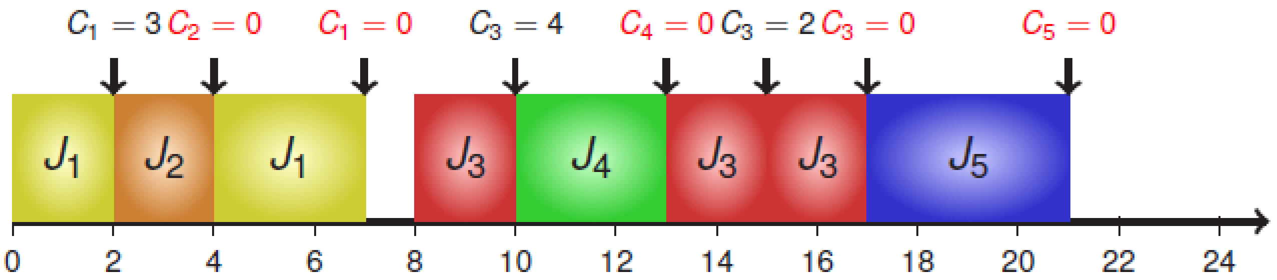
An Example: Shortest-Job-First (SJF)

- At any moment, the system executes the job with the shortest remaining time among the jobs in the ready queue.

	J_1	J_2	J_3	J_4	J_5
a_j	0	2	8	10	15
C_j	5	2	6	3	4
d_j	6	8	20	14	22

Exercise

What is the average response time of the above schedule?

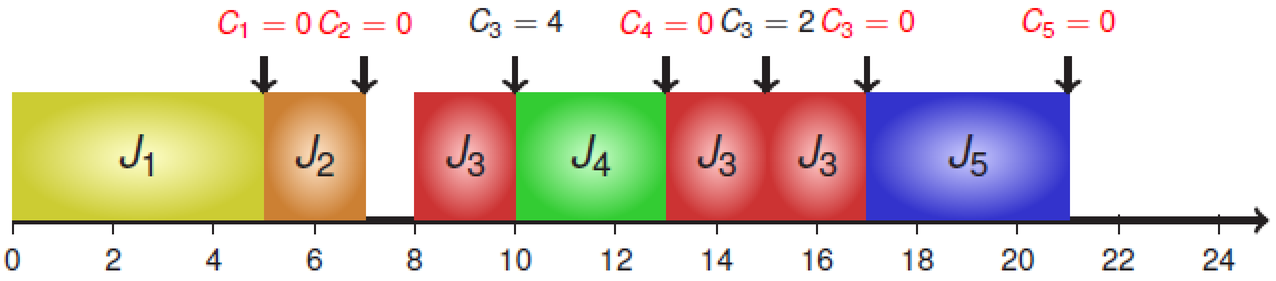


An Example: Earliest-Deadline-First (EDF)

- At any moment, the system executes the job with the earliest absolute deadline among the jobs in the ready queue.

	J_1	J_2	J_3	J_4	J_5
a_j	0	2	8	10	15
C_j	5	2	6	3	4
d_j	6	8	20	14	22

Exercise
What is the average response time of the above schedule?



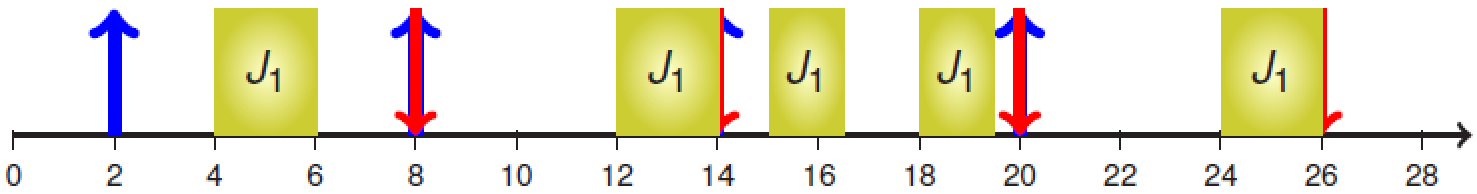
Recurrent Task Models

- When jobs (usually with the same computation requirement) are released recurrently, they can be modeled by a recurrent task
- Periodic Task t_i:
  - A job is released exactly and periodically by a period T_i
  - A phase φ_i indicates when the first job is released
  - A relative deadline D_i for each job from task t_i
  - (φ_i, C_i , T_i, D_i) is the specification of periodic task t_i, where C_i is the worst-case execution time.
- Sporadic Task t_i:
  - T_i is the minimal time between any two consecutive job releases
  - A relative deadline D_i for each job from task t_i
  - (C_i, T_i, D_i) is the specification of sporadic task t_i , where C_i is the worst-case execution time.
- Aperiodic Task: Identical jobs released arbitrarily.

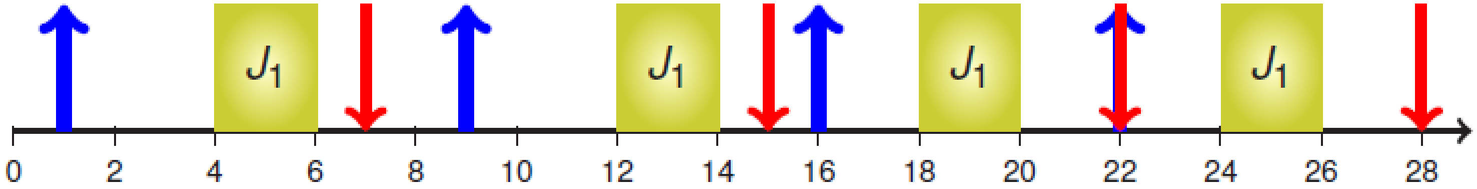
Examples of Recurrent Task Models

Periodic task: $(\phi_i, C_i, T_i, D_i) = (2, 2, 6, 6)$

↑ release ↓ deadline



Sporadic task: $(C_i, T_i, D_i) = (2, 6, 6)$



Feasibility and Schedulability for Recurrent Tasks

- A schedule is feasible if all the jobs of all tasks can be completed according to a set of specified constraints.
- A set of tasks is schedulable if there exists a feasible schedule for the set of tasks.
- A scheduling algorithm is optimal if it always produces a feasible schedule when one exists (under any scheduling algorithm).

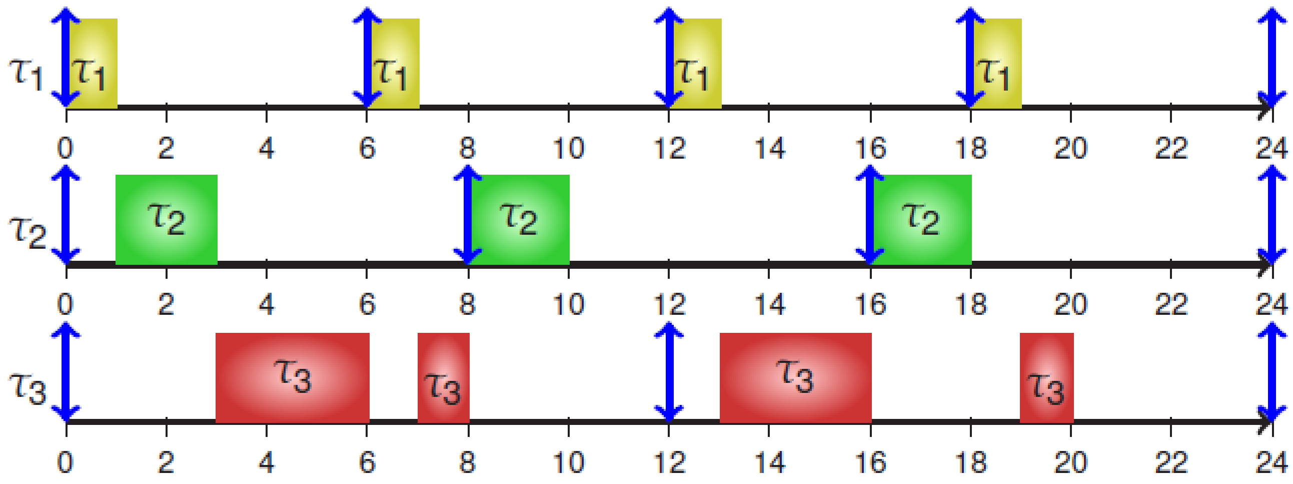
Static-Priority Scheduling

- Different jobs of a task are assigned the same priority.
  - π_i is the priority of task t_i.
  - HP_i is the subset of tasks with higher priority than t_i.
  - Note: we will assume that no two tasks have the same priority.
- We will implicitly index tasks in decreasing priority order, *i.e.*, t_i has higher priority than t_k if $i < k$.
- Which strategy is better or the best?
  - largest execution time first?
  - shortest job first?
  - least-utilization first?
  - most importance first?
  - least period first?

Rate-Monotonic (RM) Scheduling

- Priority Definition: A task with a smaller period has higher priority, in which ties are broken arbitrarily.

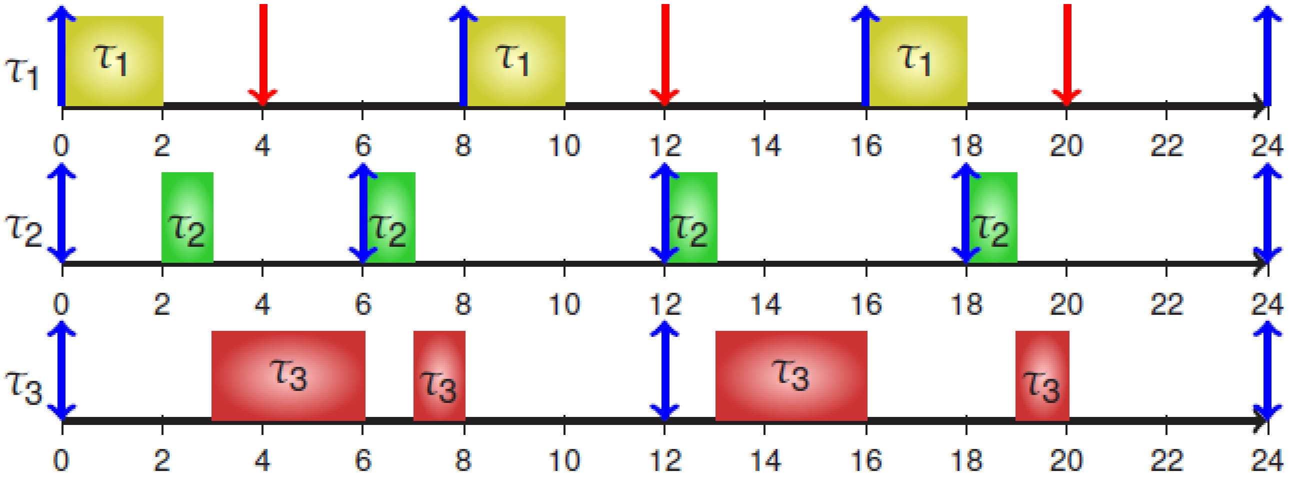
- Example Schedule: t_1 = (1, 6, 6), t_2 = (2, 8, 8), t_3 = (4, 12, 12). [(C_i, T_i, D_i)]



Deadline-Monotonic (DM) Scheduling

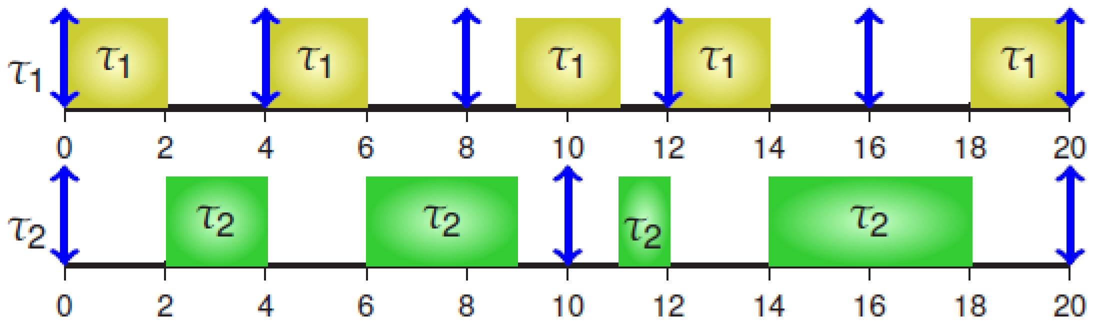
- Priority Definition: A task with a smaller relative deadline has higher priority, in which ties are broken arbitrarily.

- Example Schedule: t_1 = (2, 8, 4), t_2 = (1, 6, 6), t_3 = (4, 12, 12). [(C_i, T_i, D_i)]



Optimality (or not) of RM and DM

- Example Schedule: t_1 = (2, 4, 4), t_2 = (5, 10, 10)



- The above system is schedulable.
- No static-priority scheme is optimal for scheduling periodic tasks: However, a deadline will be missed, regardless of how we choose to (statically) prioritize t_1 and t_2.

- Corollary: Neither RM nor DM is optimal

Optimality Among Static-Priority Algorithms

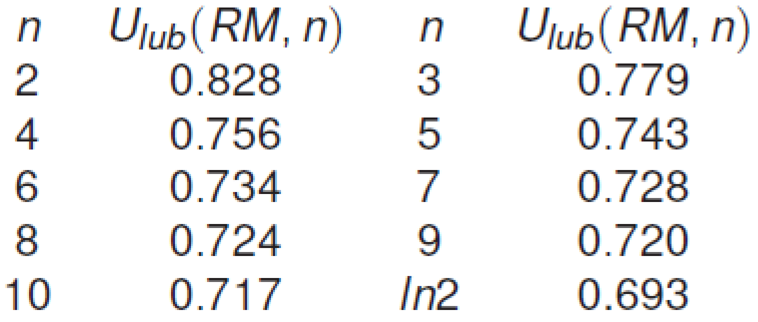
- Theorem: A system of T independent, preemptable, synchronous periodic tasks that have relative deadlines equal to their respective periods can be feasibly scheduled on one processor according to the RM algorithm whenever it can be feasibly scheduled according to any static priority algorithm.

- Exercise: Complete the proof.

- Note: When $D_i \le T_i$ for all tasks, DM can be shown to be an optimal static-priority algorithm using similar argument. Proof left as an exercise.

Liu and Layland Bound

- Theorem: [Liu and Layland] A set of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be scheduled on a processor according to the RM algorithm if its total utilization U is at most n($2^{1/n}$ - 1). In other words, $U_{lub}(RM, n) = n(2^{1/n} - 1) >= 0.693$.



n	$U_{lub}(RM, n)$	n	$U_{lub}(RM, n)$
2	0.828	3	0.779
4	0.756	5	0.743
6	0.734	7	0.728
8	0.724	9	0.720
10	0.717	$ln2$	0.693

Utilization-Based Test for EDF Scheduling

- Theorem: A task set T of independent, preemptable, periodic tasks with relative deadlines equal to their periods can be feasibly scheduled (under EDF) on one processor if and only if its total utilization U is at most one.

Relative Deadlines Less than Periods

- Theorem: A task set T of independent, preemptable, periodic tasks with relative deadlines equal to or less than their periods can be feasibly scheduled (under EDF) on one processor if:

$$\sum_{k=1}^{n} \frac{C_k}{\min\{D_k, T_k\}} \leq 1.$$

- Note: This theorem only gives sufficient condition.

Comparison between RM and EDF (Implicit Deadlines)

RM

- Low run-time overhead: *O(1)* with priority sorting in advance
- Optimal for static-priority
- Schedulability test is NP-hard (even if the relative deadline = period)
- Least upper bound: 0.693
- In general, more preemption

EDF

- High run-time overhead: *O(log n)* with balanced binary tree
- Optimal for dynamic-priority
- Schedulability test is easy (when the relative deadline = period)
- Least upper bound: 1
- In general, less preemption

Deadlocks

- **Deadlock** = condition where multiple threads/processes wait on each other

process A

```
printer->wait();
disk->wait();
  do stuffs …
disk->signal();
printer->signal();
```

process B

```
disk->wait();
printer->wait();
 do stuffs …
printer->signal();
disk->signal();
```

Binary semaphore: printer, disk. Both initialized to be 1.

Deadlocks - Terminology

- **Deadlock**:
  - Can occur when several processes compete for finite number of resources simultaneously
- **Deadlock prevention** algorithms:
  - Check resource requests & availability
- **Deadlock detection**:
  - Finds instances of deadlock when processes stop making progress
  - Tries to recover

- **Note: Deadlock ≠ Starvation**

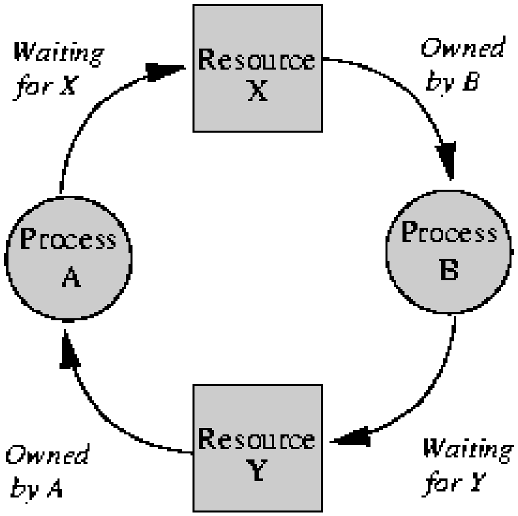
When Deadlock Occurs

All of below *must* hold:

1. **Mutual exclusion**:
   - An instance of resource used by one process at a time
2. **Hold and wait**
   - One process holds resource while waiting for another; other process holds that resource
3. **No preemption**
   - Process can only release resource *voluntarily*
   - No other process or OS can force thread to release resource
4. **Circular wait**
   - Set of processes $\{t_1, \ldots, t_n\}$: t_i waits on t_{i+1}, t_n waits on t_1

Deadlock Detection: Resource Allocation Graph

- Define graph with vertices:
  - Resources = $\{r_1, \ldots, r_m\}$
  - Processes/threads = $\{t_1, \ldots, t_n\}$
- *Request edge* from process to resource $t_i \rightarrow r_j$
  - Process requested resource but not acquired it
- *Assignment edge* from resource to process $r_j \rightarrow t_i$
  - OS has allocated resource to process
- Deadlock detection
  - No cycles → no deadlock
  - Cycle → might be deadlock

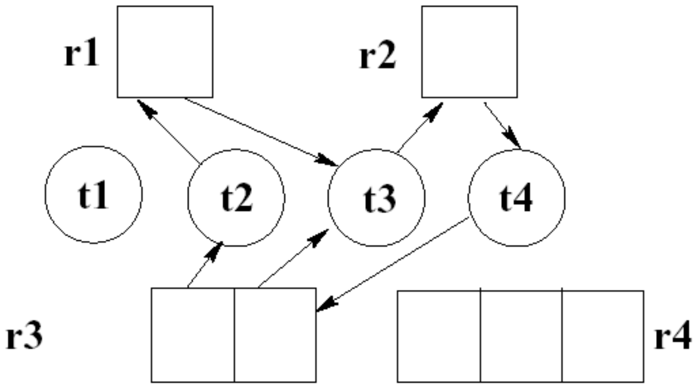


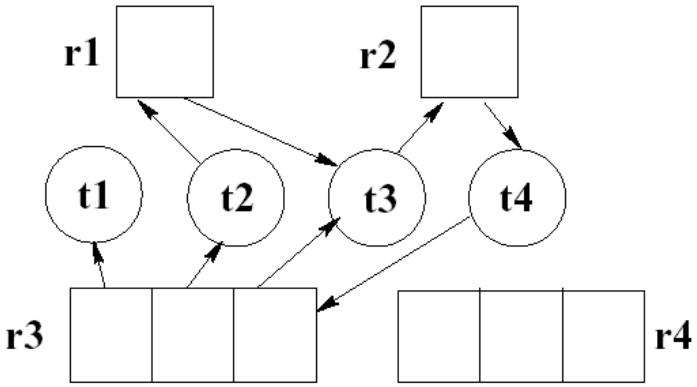
Deadlock Detection: Multiple Instances of Resource

- What if there are *multiple instances* of a resource?
  - Cycle → deadlock *might* exist
  - If any instance held by process outside cycle, progress is possible when process releases resource

Deadlock Detection

- Deadlock or not?





Detecting & Recovering from Deadlock

- Single instance of resource
  - Scan resource allocation graph for cycles & break them!
  - Detecting cycles takes O(n^2) time
    - DFS with back edge
    - $n = |T| + |R|$
  - When to detect:
    - When request cannot be satisfied
    - On regular schedule, e.g. every hour
    - When CPU utilization drops below threshold

Detecting & Recovering from Deadlock (cont'd)

  - How to recover? - break cycles:
    - Kill all processes in cycle
    - Kill processes one at a time
      - Force each to give up resources
    - Preempt resources one at a time
      - Roll back thread state to before acquiring resource
      - Common in database transactions

- Multiple instances of resource
  - No cycle → no deadlock
  - Otherwise, check whether processes can proceed

Deadlock Prevention

- Ensure at least one of necessary conditions doesn't hold
  - **Mutual exclusion**
  - **Hold and wait**
  - **No preemption**
  - **Circular wait**

Deadlock Prevention with Resource Reservation

- With future knowledge, we can prevent deadlocks:
  - Processes provide advance information about maximum resources they may need during execution
- Resource-allocation *state*:
  - Number of available & allocated resources, maximum demand of each process

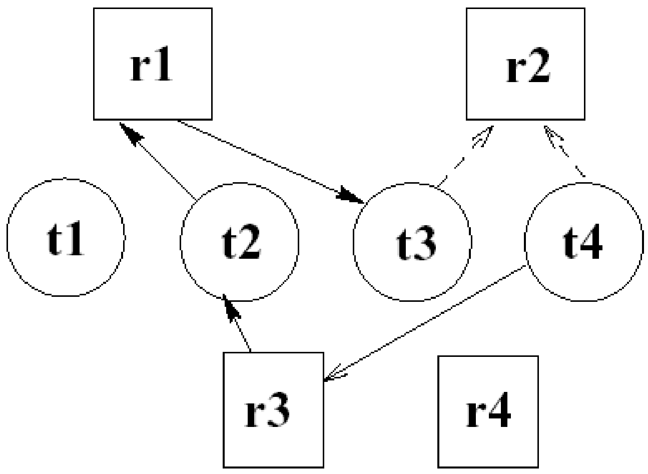
Deadlock Prevention with Resource Reservation (cont'd)

- Main idea: grant resource to process if new state is *safe*
  - Define sequence of processes $\{t_1, \ldots, t_n\}$ as *safe:*
    - For each t_i, the resources that t_i can still request can be satisfied by currently available resources plus resources held by all t_j, $j < i$
  - *Safe state* = state in which there is safe sequence containing all processes
- If new state unsafe:
  - Process waits, even if resource available

Guarantees no circular-wait condition

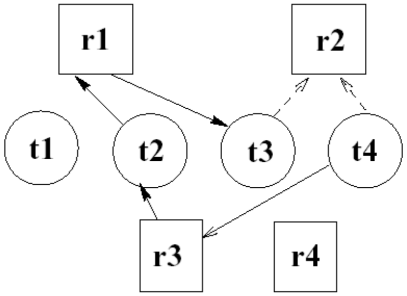
Single-Instance Resources: Deadlock Avoidance via Claim Edges

- Add *claim edges*:
  - Edge from process to resource that may be requested in future



Single-Instance Resources: Deadlock Avoidance via Claim Edges (cont'd)

- To determine whether to satisfy a request:
  - convert claim edge to allocation edge
  - No cycle: grant request
  - Cycle: unsafe state; Deny allocation, convert claim edge to request edge, block process



Banker's Algorithm

- Multiple instances
  - Each process must a priori claim maximum use
  - When a process requests a resource it may have to wait
  - When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

- **Available***:* Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available

- **Max***:* *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j

- **Allocation***:* *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j

- **Need***:* *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize:

   Work **=** ***Available***

   Finish **[*i*] =** ***false*** **for** ***i*** **= 0, 1, …,** ***n*****- 1**

2. Find an ***i*** such that both:

   (a) ***Finish*** **[*i*] =** ***false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish**[*i*] =** ***true***

   go to step 2

4. If ***Finish*** **[*i*] ==** ***true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request$_i$ = request vector for process P_i. If ***Request***$_i$ [j] = k then process P_i wants k instances of resource type R_j

1. If ***Request***$_i$ ≤ ***Need***$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request***$_i$ ≤ ***Available***, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   Available = ***Available*** – ***Request***$_i$;

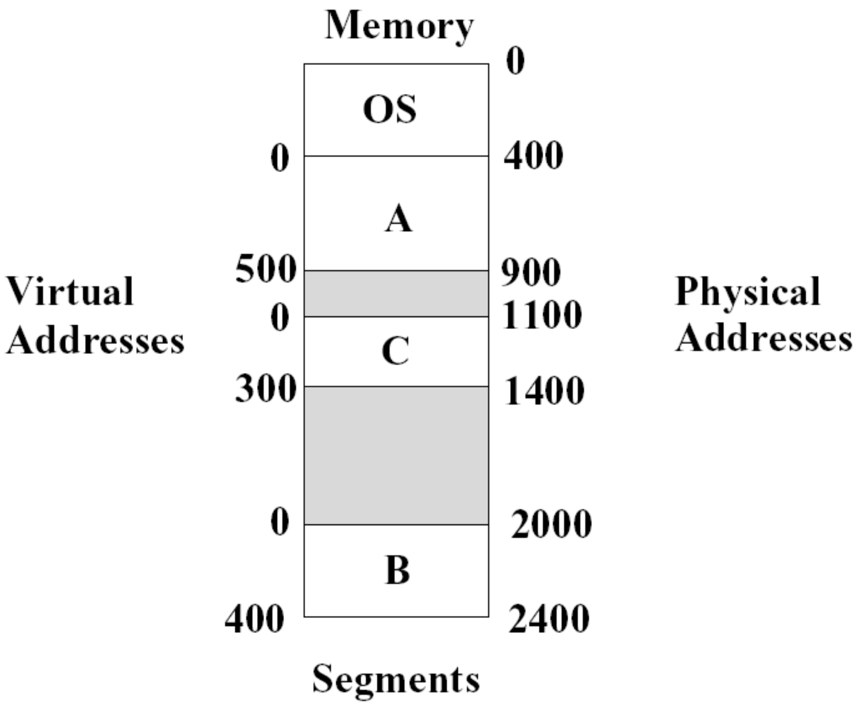
   Allocation$_i$ = ***Allocation***$_i$ + ***Request***$_i$;

   Need$_i$ = ***Need***$_i$ – ***Request***$_i$;

   - If safe ⇒ the resources are allocated to P_i
   - If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Memory Management: Terminology

- **Segment**: chunk of memory assigned to process
- **Physical address**: real address in memory
- **Virtual address**: address relative to start of process's address space

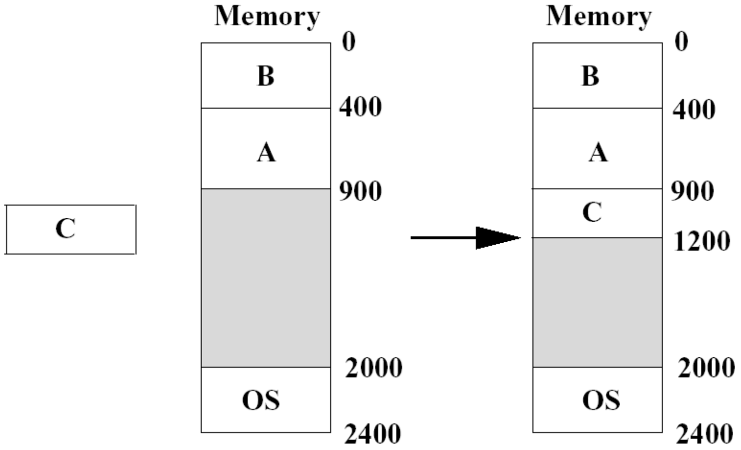


Multiprogramming Requirements

- **Transparency**
  - No process aware memory is shared
  - Process has no constraints on physical memory
- **Safety**
  - Processes cannot corrupt each other or OS
- **Efficiency**
  - Performance not degraded due to sharing

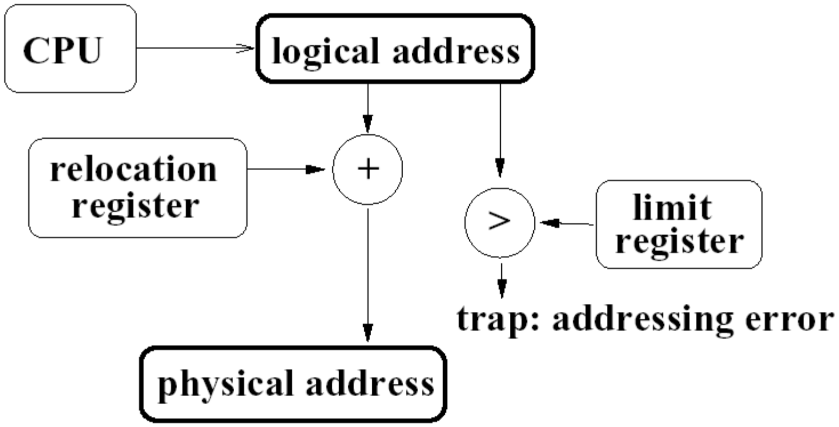
Contiguous memory allocation

- Put OS in high memory
- Process starts at 0
  - Max addr = memory size – OS size
- Load process by allocating contiguous segment for process
- Smallest addr = *base*, largest = *limit*



Address Translation

- Hardware adds relocation register (base) to virtual address to get physical address
- Hardware compares address with limit register
  - Test fails → trap



Pros & Cons

- Advantages
  - Simple, fast hardware
    - Two special registers, add & compare
- Disadvantages
  - Process limited to physical memory size
  - Degree of multiprogramming limited
    - All memory of active processes must fit in memory

Fragmentation

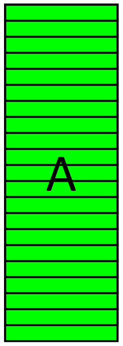
- Fragmentation = % memory unavailable for allocation, but not in use
- **External fragmentation:**
  - Large # of small holes s.t. even the total size satisfies a request; no contiguous chunk can be found
  - Caused by repeated unloading & loading
- **Internal fragmentation:**
  - Space inside process allocations
    - Unavailable to other processes

Compaction

- Can make space available by shuffling process space
  - Eliminate holes
  - Place free memory together
  - Cannot move a process if addresses are determined at compile or load time

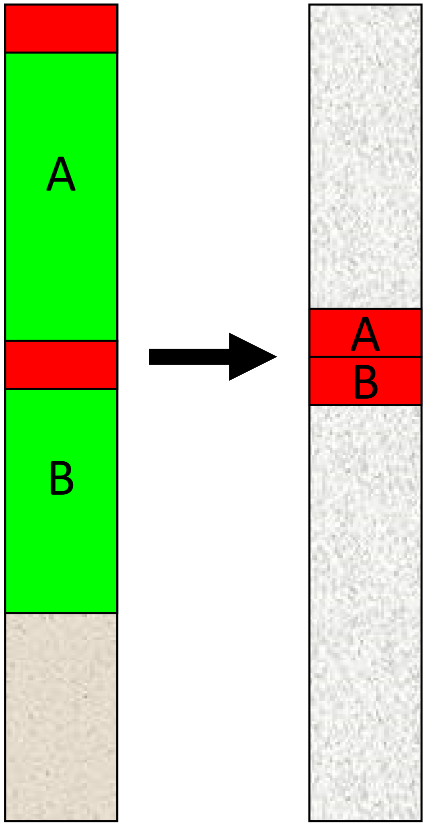
Alternative: Paging

- Divide logical memory into fixed-sized *pages* (4K, 8K)
- Divide physical memory into fixed-sized *frames*
  - Pages & frames same size
  - OS manages pages
    - Moves, removes, reallocates
- Disk space: blocks same size as frames
  - Pages copied to and from disk to frames



Paging Advantages

- Most programs obey 90/10 "rule"
  - 90% of time spent accessing 10% of memory
- Exploiting this rule:
  - Only keep "live" parts of process in memory



Paging Advantages

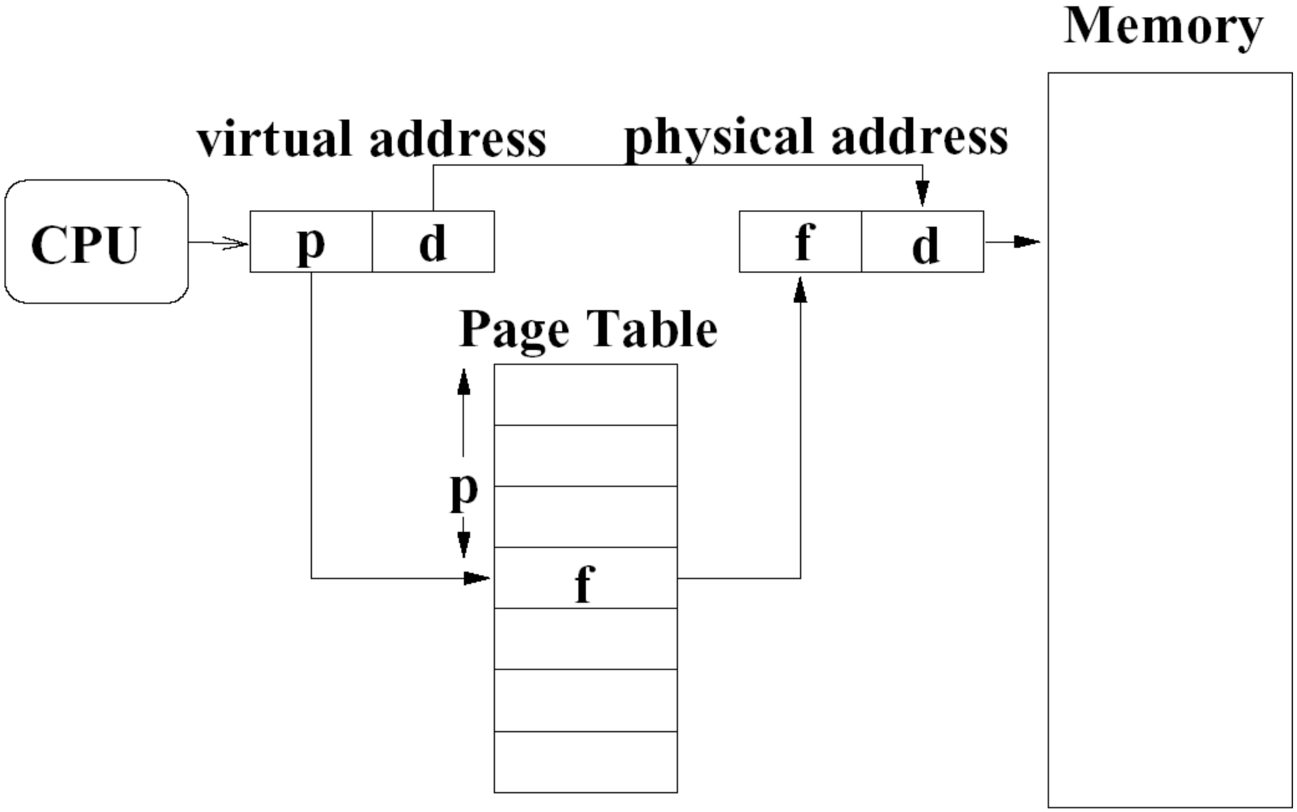
- “Hole-fitting problem” vanishes!
  - Logical memory contiguous
  - Physical memory not required to be
- Eliminates external fragmentation
  - But not internal (why not?)

- But: Complicates address lookup...

Paging Hardware

- Processes use virtual addresses
  - Addresses start at 0 or other known address
  - OS lays process down on pages
- MMU (memory-management unit):
  - Hardware support for paging
  - Translates virtual to physical addresses
  - Uses *page table* to keep track of frame assigned to memory page

Paging Hardware: Diagram



Paging Hardware: Intuition

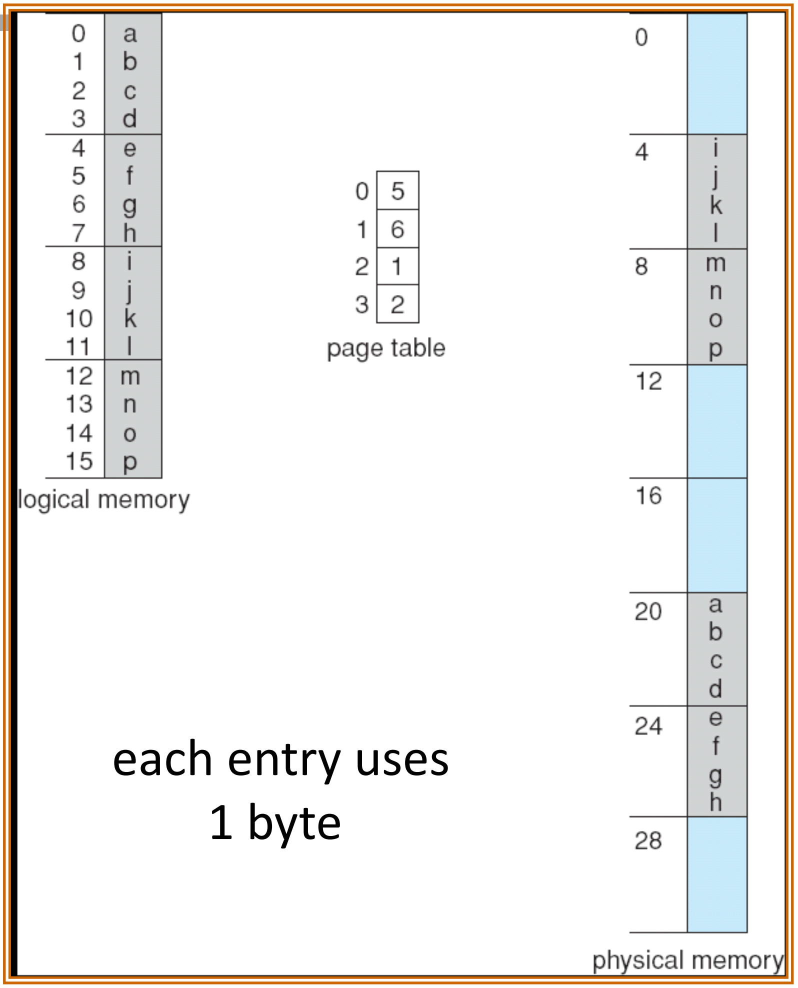
- Paging: form of dynamic relocation
  - Virtual address bound by paging hardware to physical address
- Page table: similar to a set of relocation registers
- Mapping – invisible to process
  - OS maintains mapping
  - H/W does translation
- Protection – provided by same mechanisms as in dynamic relocation

Address Translation: Example

- Assume 1 byte addressing, each page contains 4 bytes:
  - Length of p, d?
  - Given virtual address 0, 4, 10, 13, do virtual to physical translation

p	d
m-n	n

p: page number
d: page offset

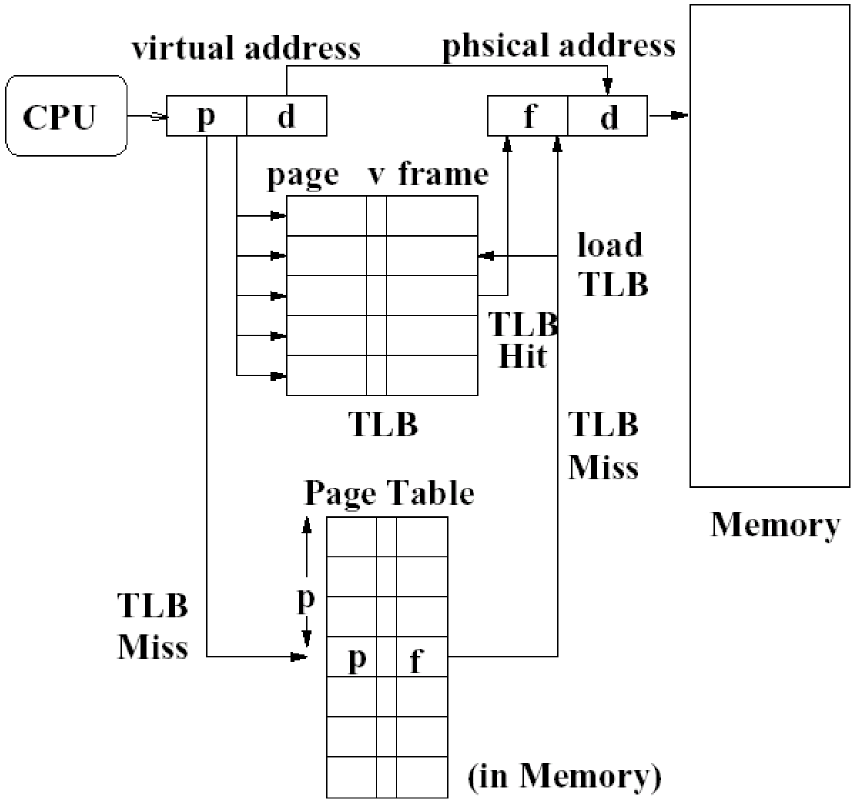


each entry uses 1 byte

Translation Lookaside Buffer (TLB)

- Small, fast-lookup hardware cache
- TLB sizes: 8 to 2048 entries

TLB: Diagram



- v = valid bit: entry is up-to-date

Effectiveness of TLB

- Processes exhibit locality of reference
  - **Temporal locality**: processes tend to reference same items repeatedly
  - **Spatial locality:** processes tend to reference items near each other (e.g., on same page)
- Locality in memory accesses → locality in address translation

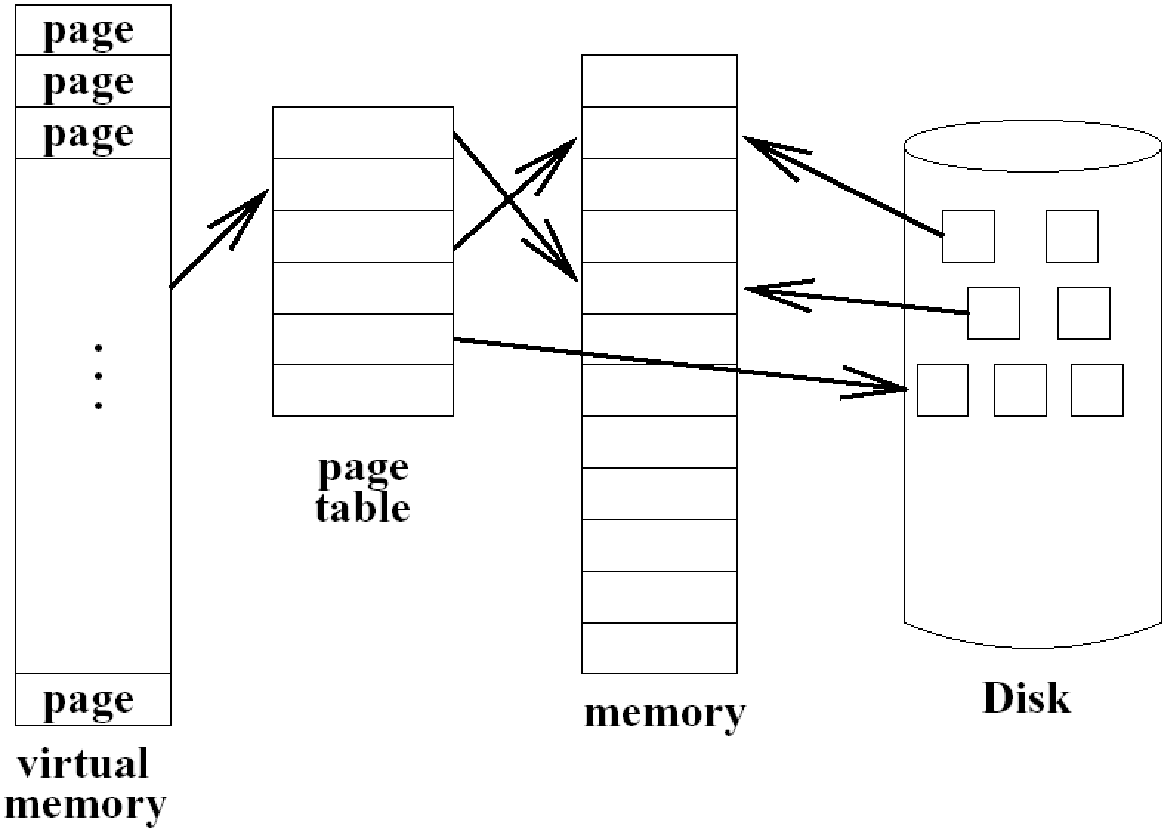
Managing TLB:
Process Initialization & Execution

- Process arrives, needs *k* pages
- If *k* page frames free, allocate;
  else free frames that are no longer needed
- OS:
  - puts pages in frames
  - puts frame numbers into page table
  - marks all TLB entries as invalid (*flush*)
  - starts process
  - loads TLB entries as pages are accessed,
    replaces entries when full

Managing TLB: Context Switches

- Extend Process Control Block (PCB) with:
  - Page table
  - Copy of TLB (optional)
- Context switch:
  - Copy page table to PCB
  - Copy TLB to PCB, Flush TLB (optional)
  - Restore page table
  - Restore TLB (optional)
- Use *multilevel paging* if tables too big (see text)

Demand-Paging Diagram



Key Policy Decisions

- Two key questions:
  - When do we read page *from* disk?
  - When do we write page *to* disk?

Reading Pages

- Read **on-demand**:
  - OS loads page on its first reference
  - May force an **eviction** of page in RAM
  - Pause while loading page = **page fault**
- Can also perform **pre-paging**:
  - OS *guesses* which page will next be needed, and begins loading it
  - Advantages? Disadvantages?
- Most systems just do demand paging

Page Replacement

- Process is given a fixed memory space of *n* pages
- Question:
  - process requests a page
  - page is not in memory, all *n* pages are used
  - which page should be evicted from memory?

Metric: Effective Access Time

- Let p = probability of page fault $(0 \leq p \leq 1)$
  ma = memory access time
- Effective access time =
  $(1 - p) * ma + p *$ **page fault service time**
  - Memory access = 200ns, page fault = 25ms: effective access time = (1-p)*200 + p*25,000,000

Evaluating Page Replacement Algorithms

- Average-case:
  - Empirical studies – real application behavior
- Theory: **competitive analysis**
  - Can't do better than optimal
  - How far (in terms of faults) is algorithm from optimal in worst-case?
    - **Competitive ratio**
    - If algorithm can't do worse than 2x optimal, it's **2-competitive**

Page Replacement Algorithms

- MIN, OPT (optimal)
- RANDOM
  - evict random page
- FIFO (first-in, first-out)
  - give every page equal *residency*
- LRU (least-recently used)
- MRU (most-recently used)

MIN/OPT

- Invented by Belady ("MIN"), now known as "OPT": optimal page replacement
  - Evict page to be accessed furthest in the future
- Provably optimal policy
  - Just one small problem...
- Requires predicting the future
  - Useful point of comparison

RANDOM

- Evict *any* page
- Works surprisingly well
- Theoretically: very good
- Not used in practice:
  takes no advantage of locality

LRU

- Evict page that has not been used in longest time (least-recently used)
  - Approximation of MIN if recent past is good predictor of future
  - A *variant* of LRU used in all real operating systems
- Competitive ratio: *n*, (*n:* # of page frames)
  - Best possible for deterministic algs.

FIFO

- First-in, first-out: evict *oldest* page
  - Also has competitive ratio *n*
- But: performs miserably in practice!
  - LRU takes advantage of locality
  - FIFO does not
- Suffers from **Belady's anomaly:**
  - More memory can mean more paging!

FIFO & Belady's Anomaly

- Request sequence

A B C D A B E A B C D E

- Q1: # of page faults when n=3?
- Q2: # of page faults when n=4?
- Q3: what are the results under LRU?

FIFO & Belady’s Anomaly

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	A	A	A	D	D	D	E			E	E	
frame 2		B	B	B	A	A	A			C	C	
frame 3			C	C	C	B	B			B	D	
frame 1	A	A	A	A			E	E	E	E	D	D
frame 2		B	B	B			B	A	A	A	A	E
frame 3			C	C			C	C	B	B	B	B
frame 4				D			D	D	D	C	C	C

- When n=3, 9 page faults
- When n=4, 10 page faults

LRU: No Belady's Anomaly

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	A	A	A	D	D	D	E			C	C	C
frame 2		B	B	B	A	A	A			A	D	D
frame 3			C	C	C	B	B			B	B	E
frame 1	A	A	A	A			A			A	A	E
frame 2		B	B	B			B			B	B	B
frame 3			C	C			E			E	D	D
frame 4				D			D			C	C	C

- When n=3, 10 page faults
- When n=4, 8 page faults

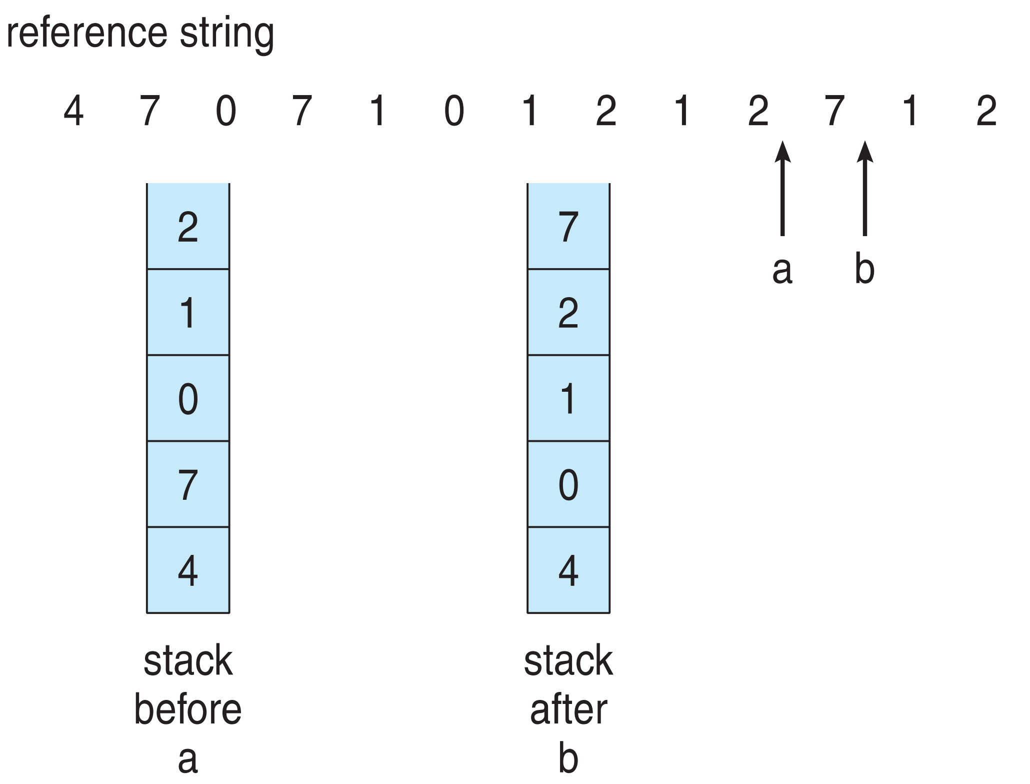
Why no anomaly for LRU?

- “Stack” property:
  - Pages in memory for memory size of n are also in memory for memory size of $n+1$

Implementation of LRU Algorithm

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don’t have Belady’s Anomaly

Use Of A Stack to Record Most Recent Page References



reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

a b

2
1
0
7
4

stack
before
a

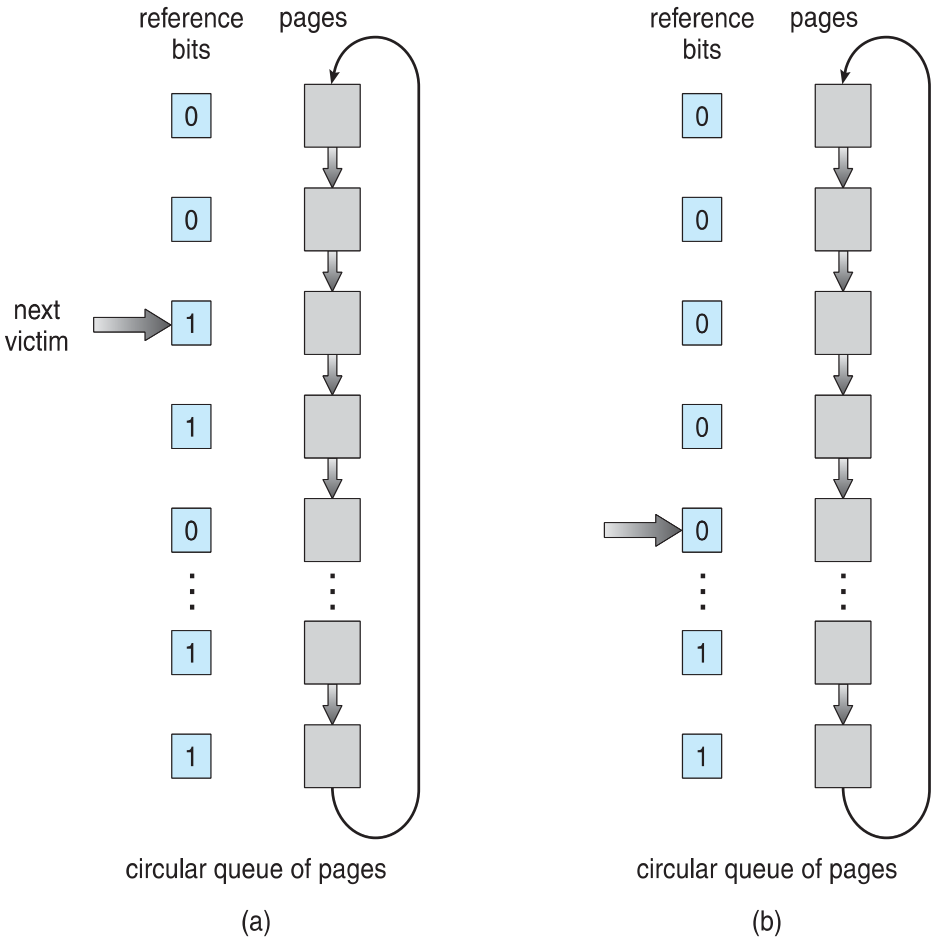
7
2
1
0
4

stack
after
b

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Allocation of Frames

- Each process needs ***minimum*** number of frames
  - Defined by the computer architecture
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

— s_i = size of process p_i

— $S = \sum s_i$

— m = total number of frames

— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

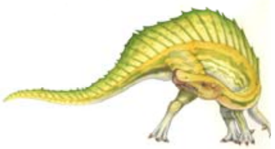
- Use a proportional allocation scheme using priorities rather than size

- If process ***P_i*** generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

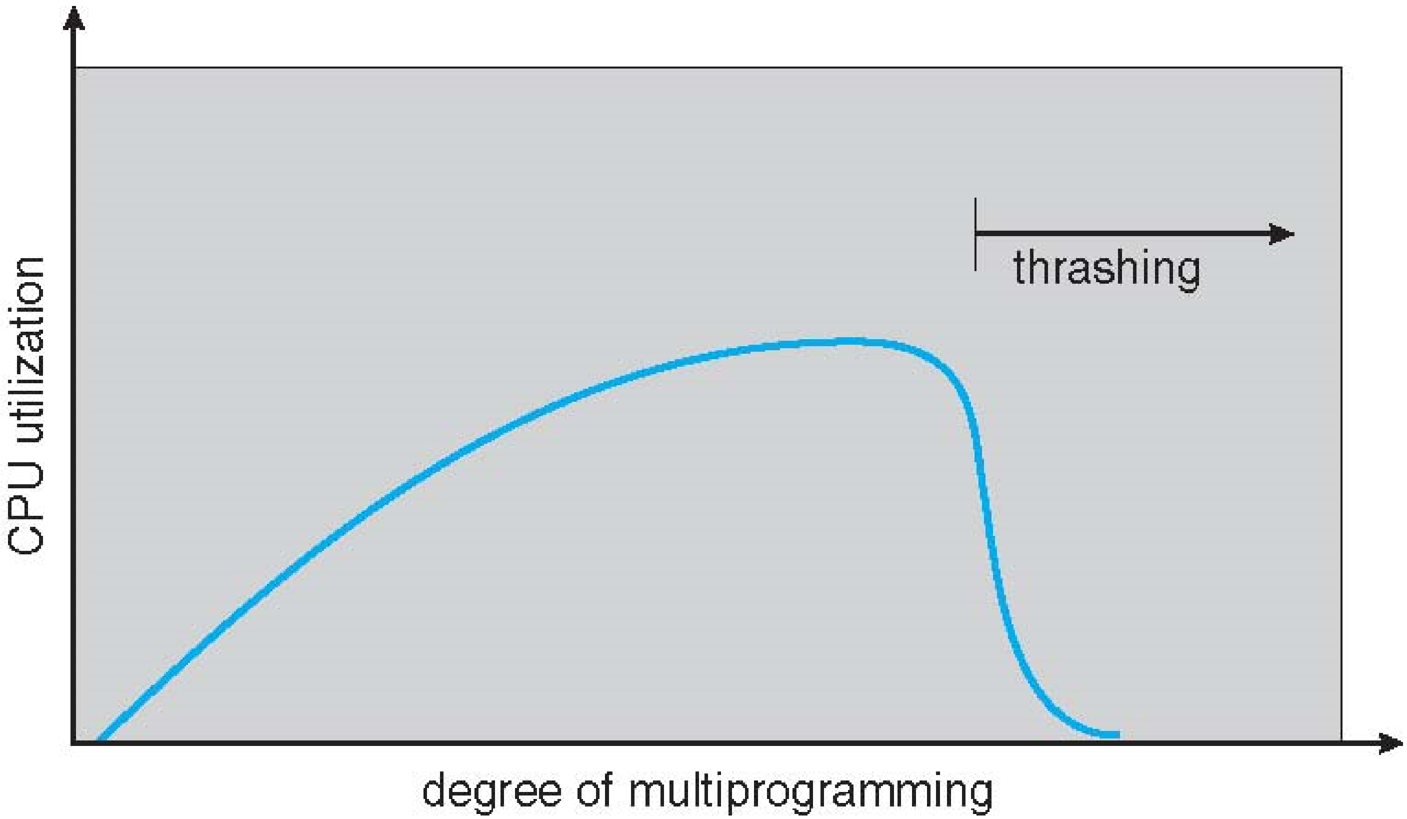


Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



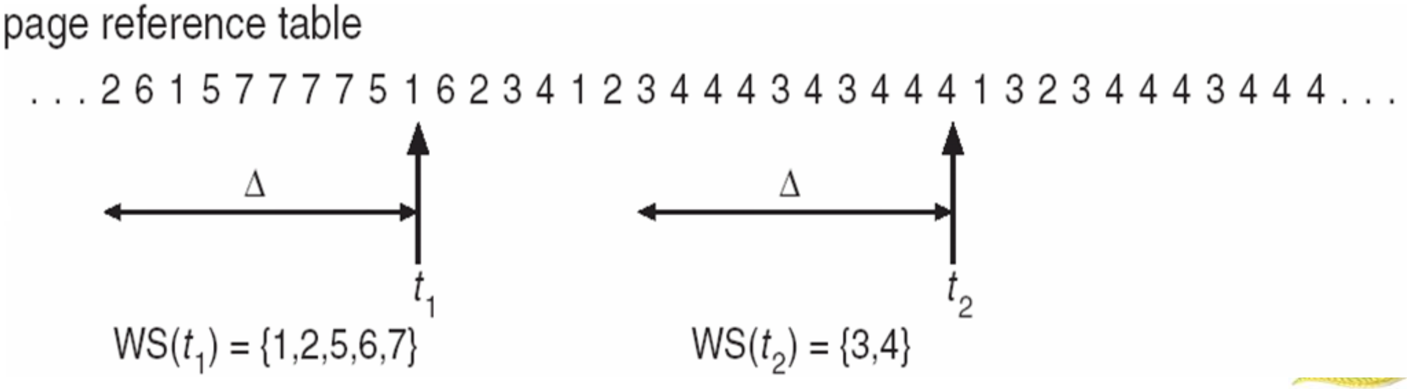
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - A locality is a set of pages actively used together
  - Process migrates from one locality to another
  - Localities may overlap
  - Localities are defined by the program structure and its data structure

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

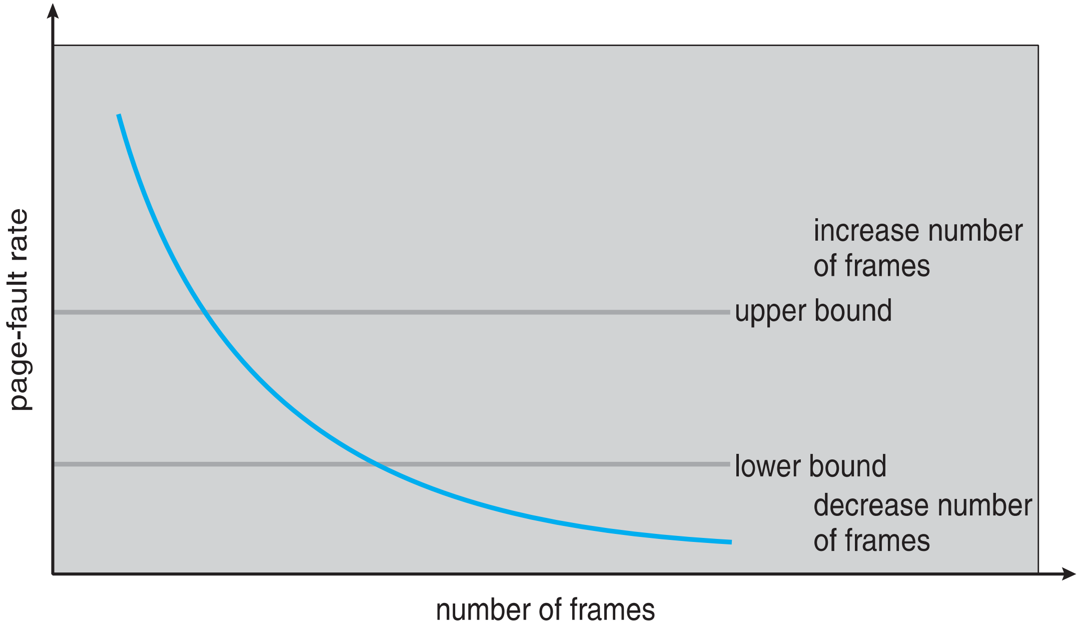
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes



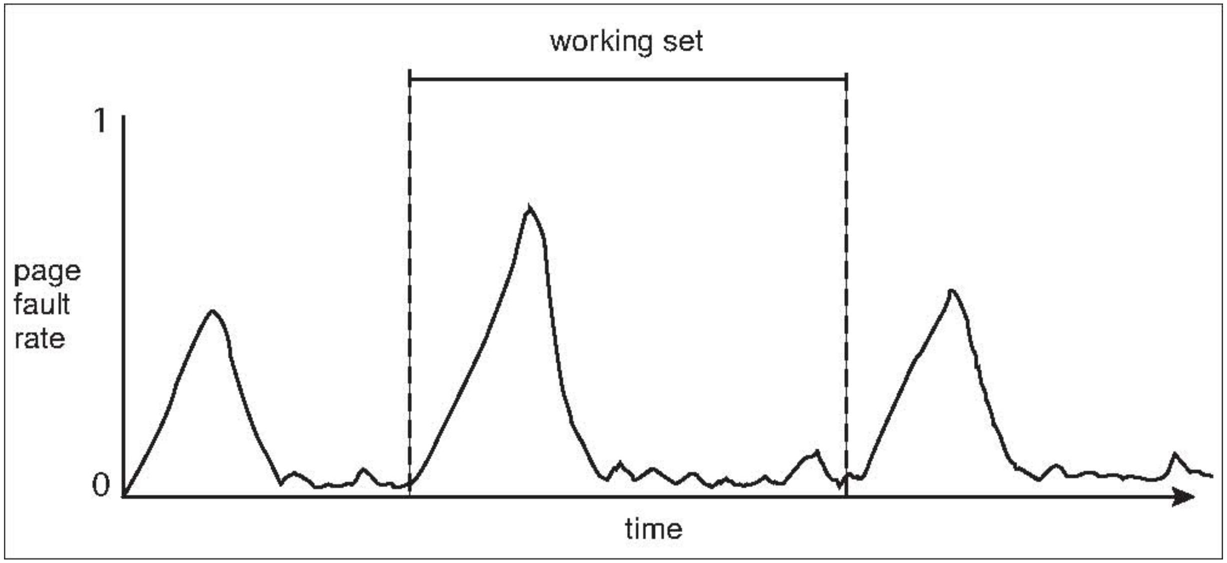
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Files: OS Abstraction

- Files: another OS-provided abstraction over hardware resources

OS Abstraction	Hardware Resource
Processes Threads	CPU
Address space	Memory
Files	Disk

File Concept

- The file system consists of two distinct parts:
  - A collection of files, each storing related data
  - A directory structure, which organizes and provides information about all the files in the system.
- File: Contiguous logical address space, mapped by the OS onto physical devices.
- Types:
  - Data
    - Numeric, character, binary
  - Program
- Contents (many types) is defined by file’s creator
  - text file,
  - source file,
  - executable file

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on the device (disk)
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – information kept for creation time, last modification time, and last use time.
  - Useful for data for protection, security, and usage monitoring
- Information kept in the directory structure (on disk), which consists of “inode” entries for each of the files in the system.

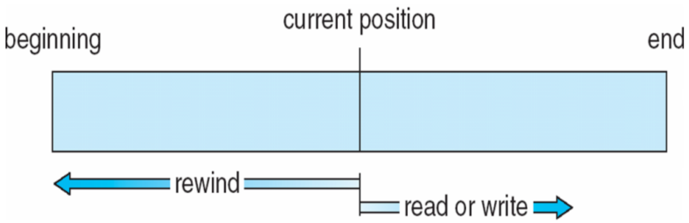
File Operations

- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- ***Open(F_i)*** – search the directory structure on disk for inode entry ***F_i***, and move the content of the entry to memory
- ***Close (F_i)*** – move the content of inode entry ***F_i*** in memory to directory structure on disk.

Access Methods

Sequential Access (based on tape model)

- Information in the file is processed in order, one record after the other.
- General structure



- Operations:
  - **read_next ()** – reads the next portion of the file and automatically advances a file pointer.
  - **write_next () –** append to the end of the file and advances to the end of the newly written material (the new end of file).
  - **reset** – back to the beginning of the file.

Access Methods

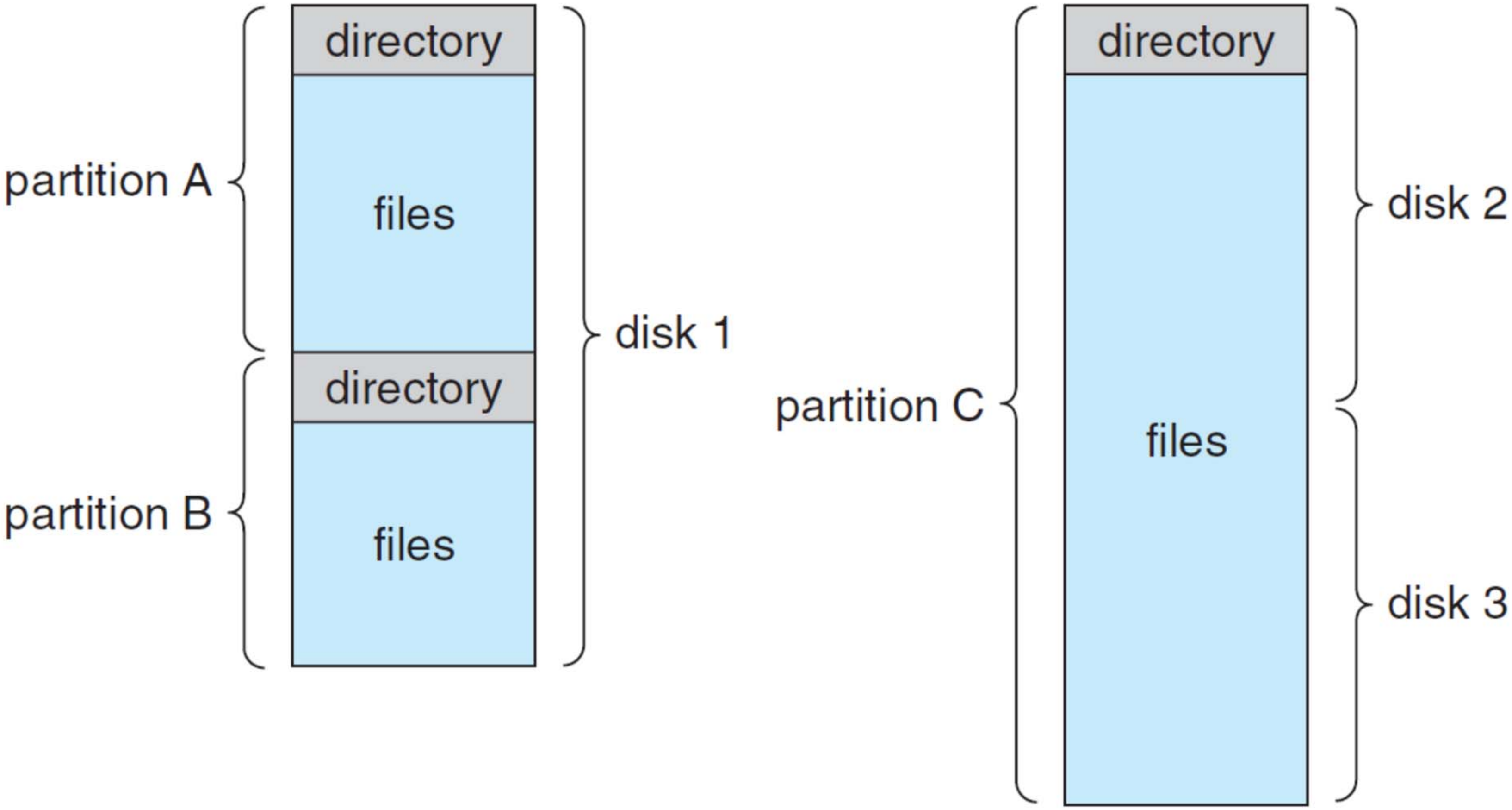
Direct Access (based on disk model)

- File is made up of fixed-length *logical records* that allow programs to read and write records rapidly in no particular order.
- File is viewed as a numbered sequence of blocks or records. For example, can read block 14, then read block 53, and then write block 7.
- Operations:
  - **read(n)** – reads relative block number n.
  - **write(n)** – writes relative block number n.
- Relative block numbers (to the beginning of the file) allow OS to decide where file should be placed

Other Access Methods

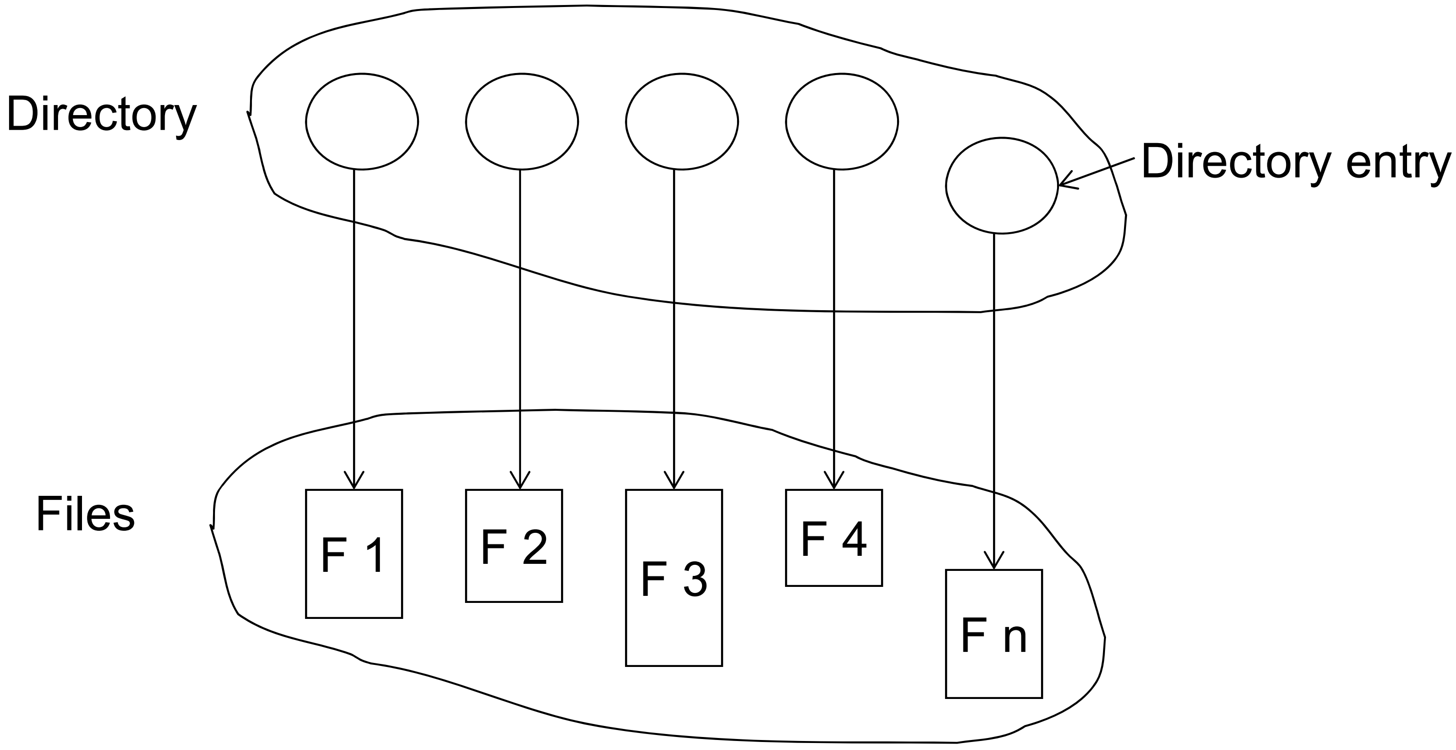
- Can be built on top of the direct-access methods
- Generally -- involve creation of an index for the file, containing pointers to the various blocks.
- Keep index in memory for fast determination of location of data to be operated on
- For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices: 10-digit UPC + 6-digit price = a 16-byte record. If the disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory.
- If too large, keep index (in memory) of the main index (on disk)

A Typical File System Organization



Directory Structure

- The directory can be viewed as a symbol table that translates file names into their directory
- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk

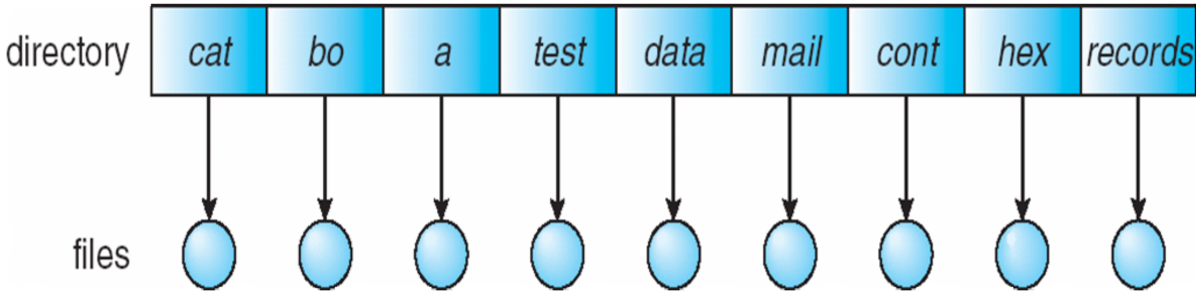
Directory Organization

The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties (e.g., all Java programs, all games, …)

Single-Level Directory

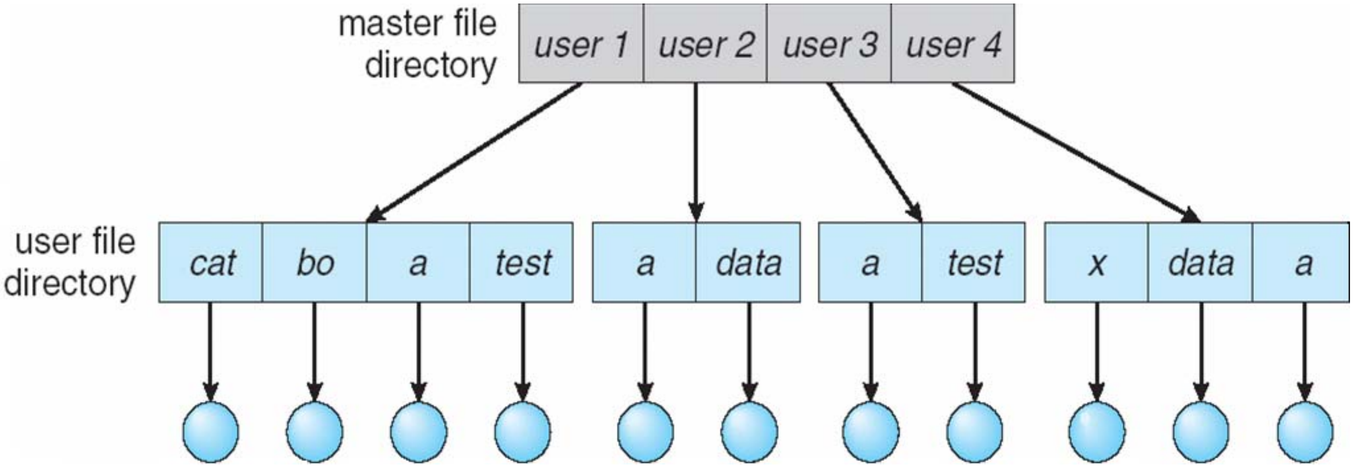
- A single directory for all users



- Naming problem: unique name rule is violated
- Grouping problem

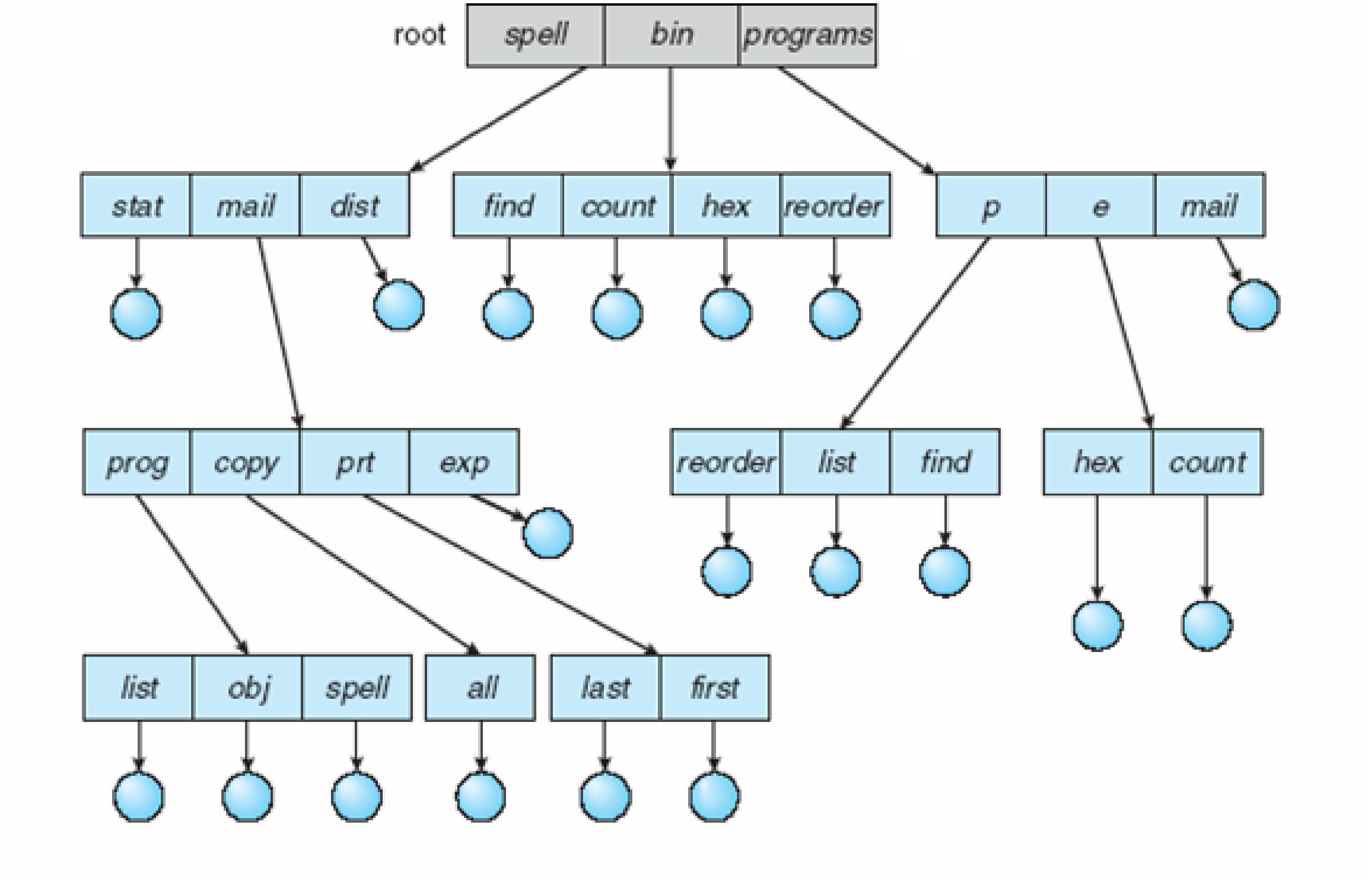
Two-Level Directory

- Separate directory for each user



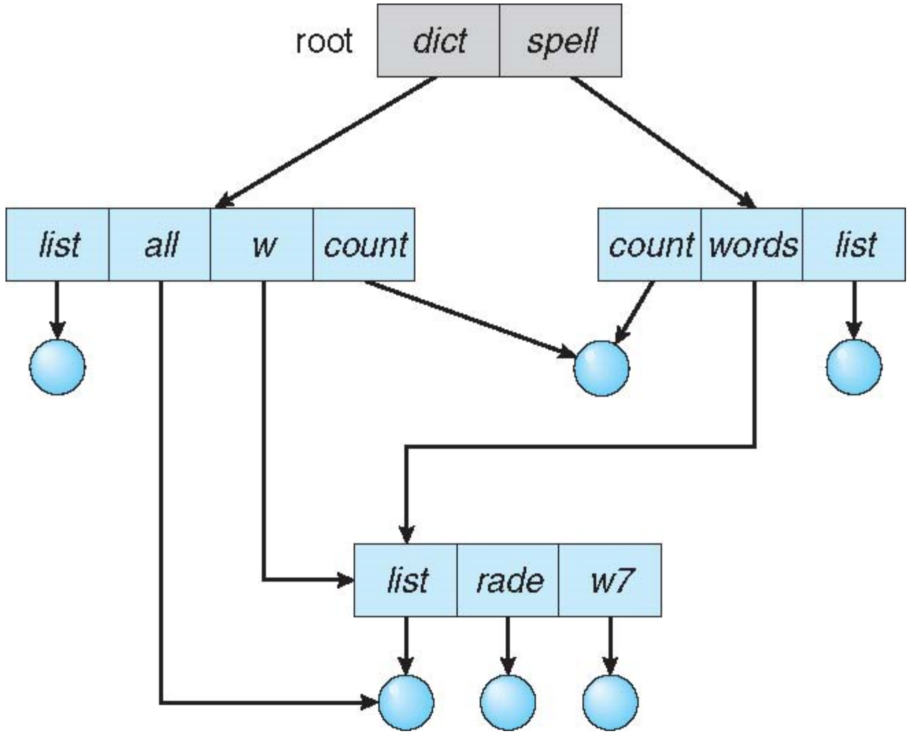
- Can have the same file name for different user
- Efficient searching
- User isolation: difficult for file sharing
- No grouping capability

Tree-Structured Directories

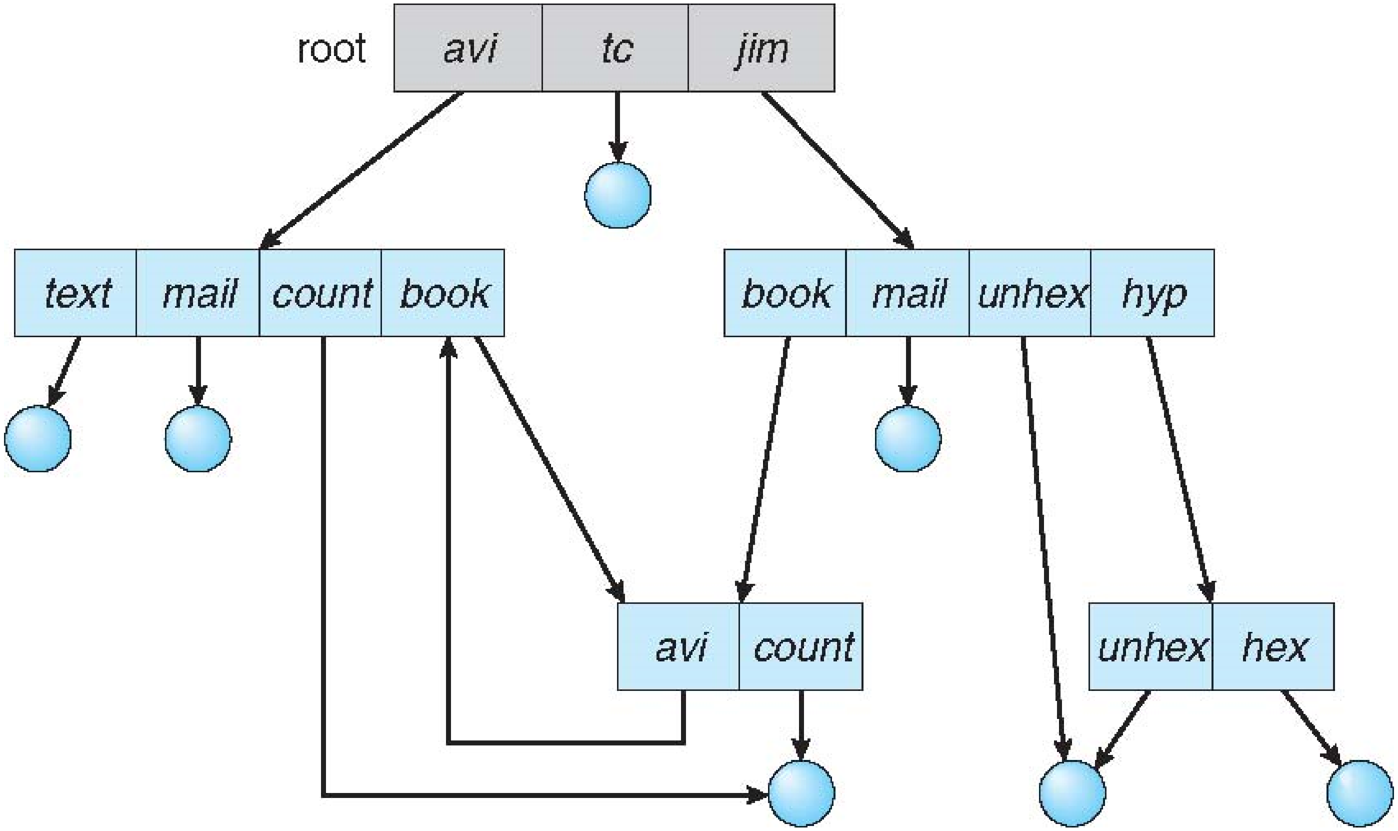


Acyclic-Graph Directories

Have shared subdirectories and files



General Graph Directory


root
avi
tc
jim
text
mail
count
book
book
mail
unhex
hyp
avi
count
unhex
hex