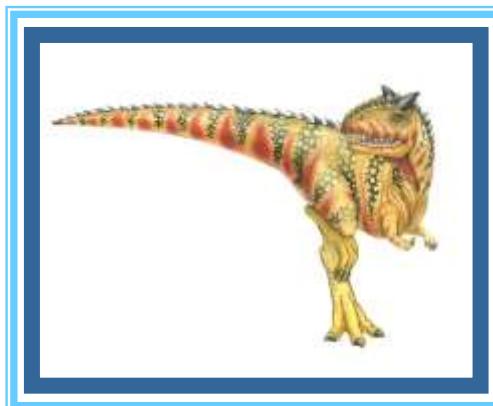


# Chapter 3: Processes





# Outline

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

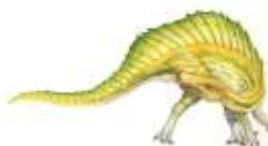




# Objectives

---

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.





# Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- 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 Concept (Cont.)

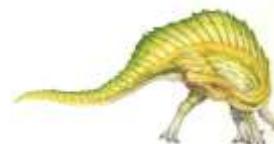
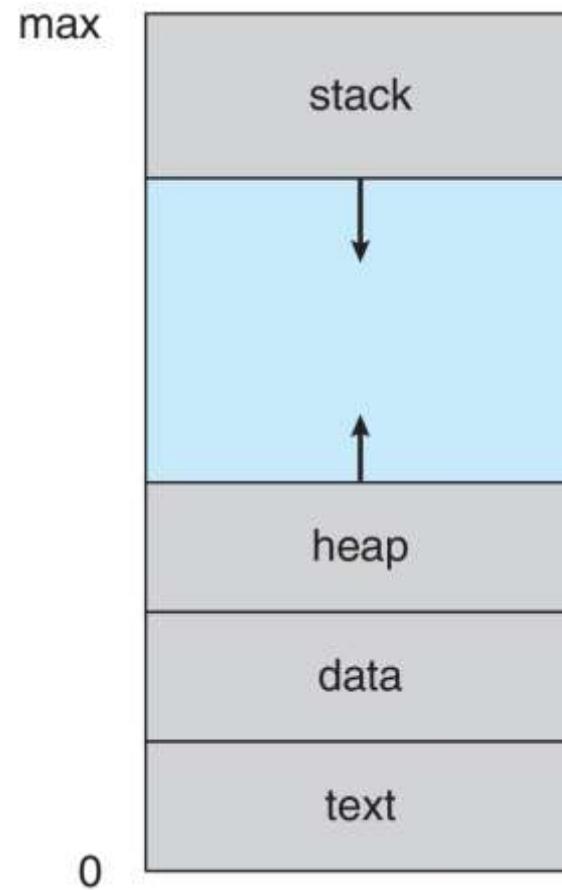
---

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes 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



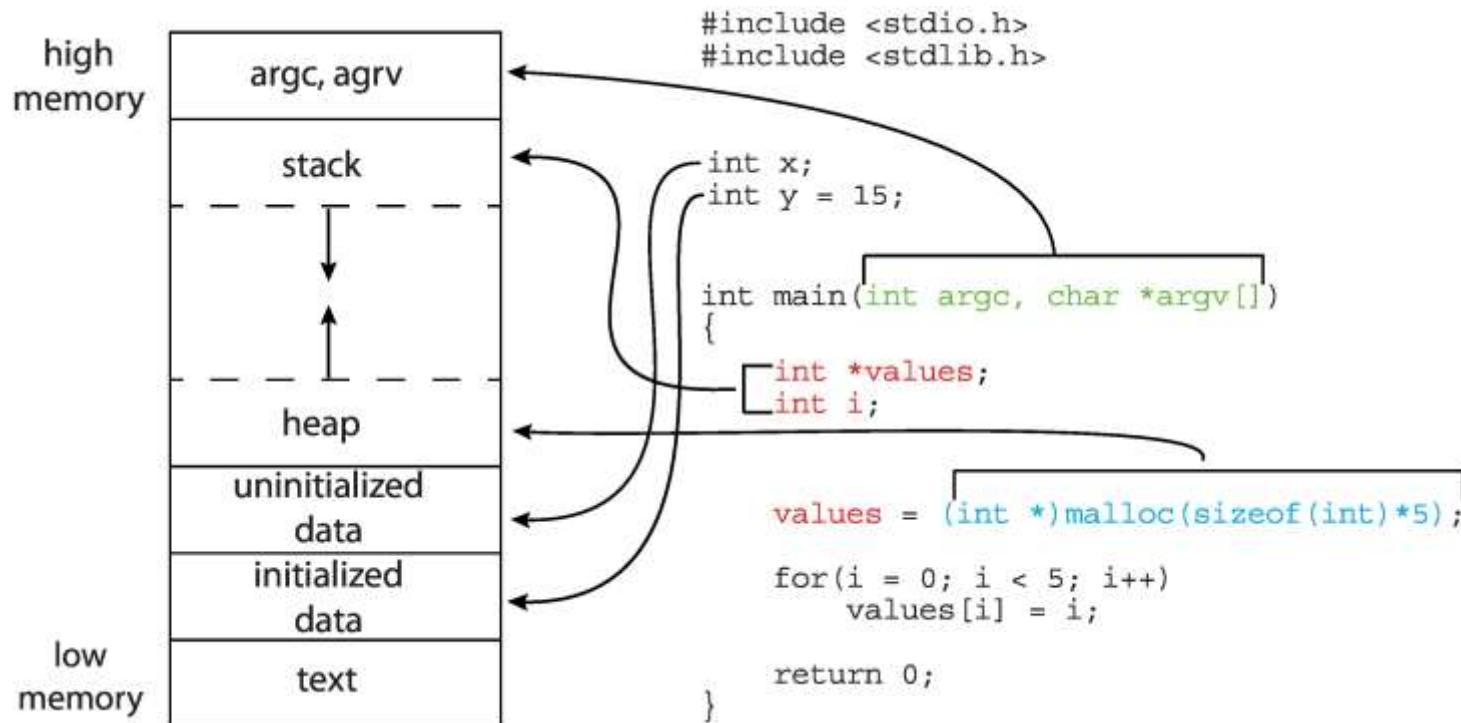


# Process in Memory





# Memory Layout of a C Program

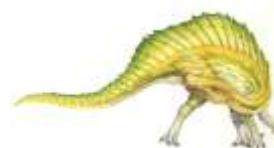




# Process State

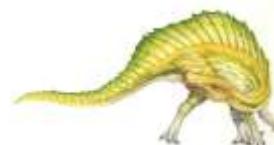
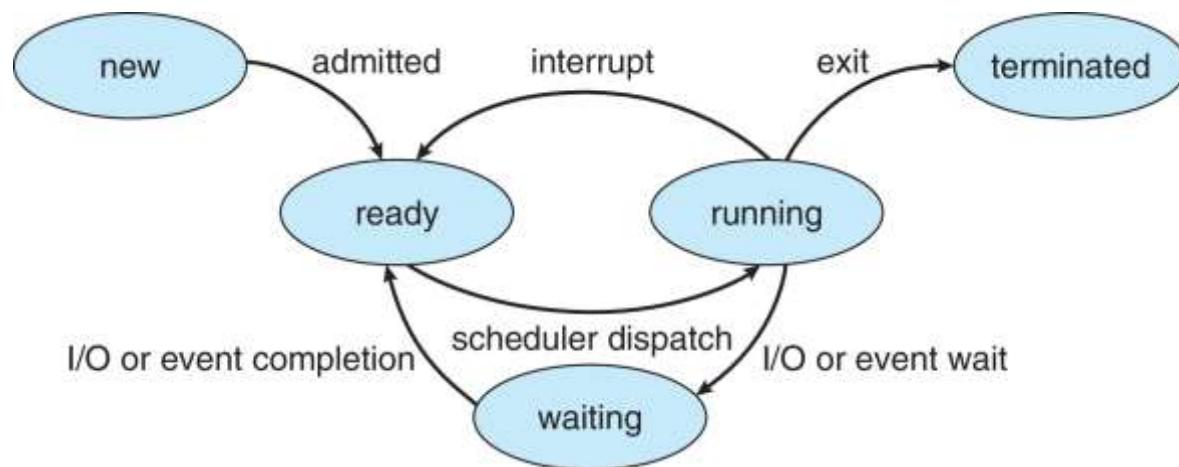
---

- 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





# Threads

---

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

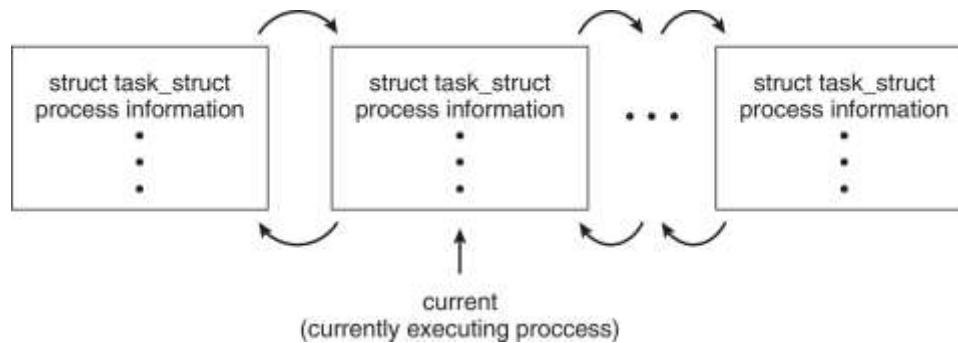




# Process Representation in Linux

Represented by the C structure `task_struct`

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





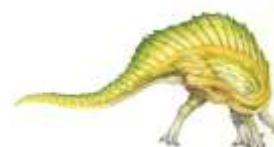
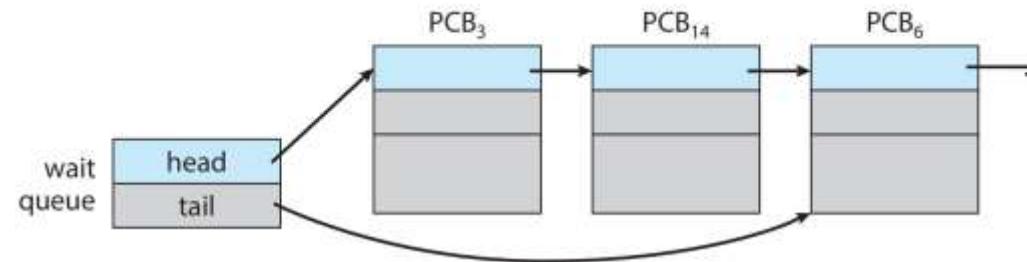
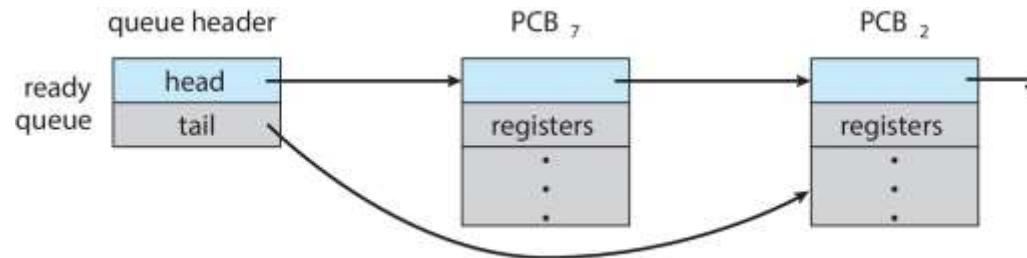
# Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



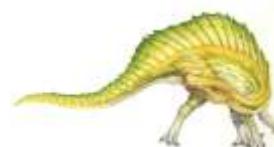
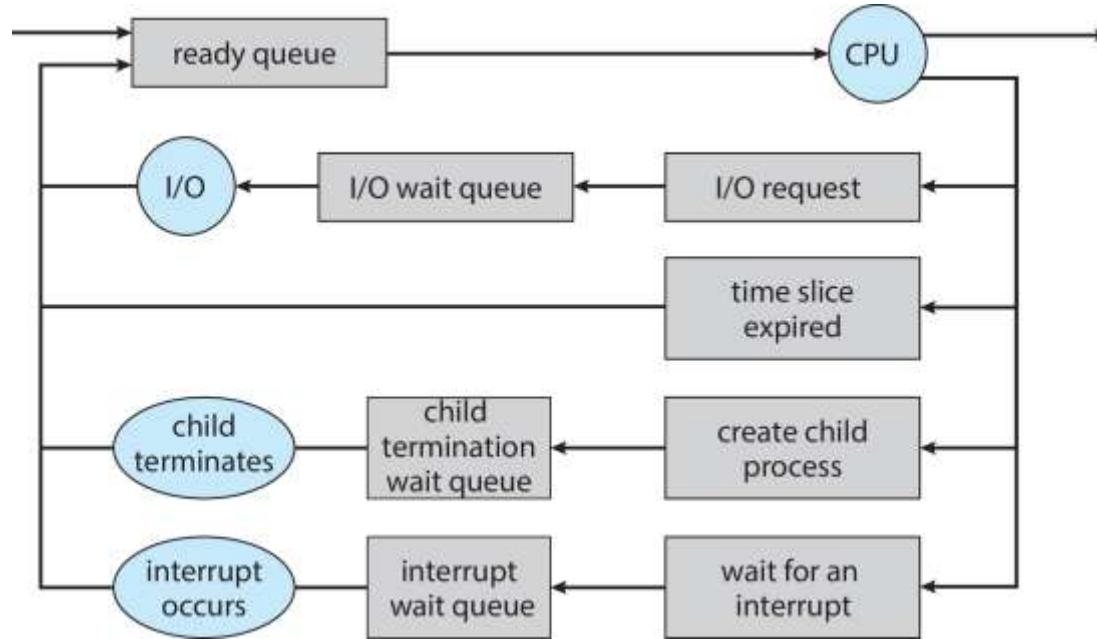


# Ready and Wait Queues





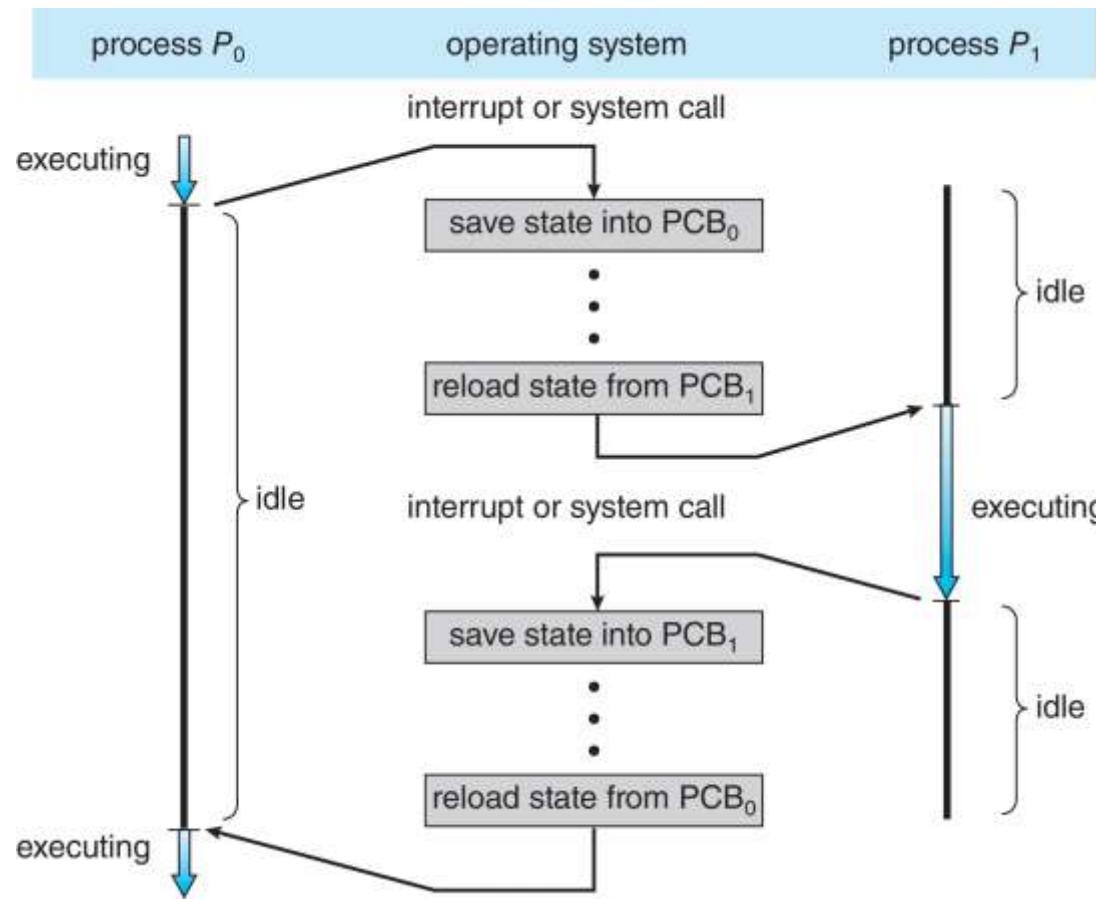
# Representation of Process Scheduling





# CPU Switch From Process to Process

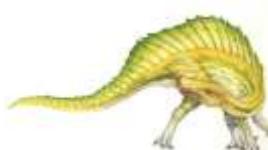
A **context switch** occurs when the CPU switches from one process to another.





# Context Switch

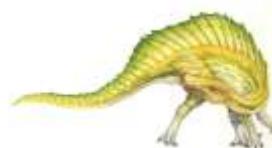
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once





# Multitasking in Mobile Systems

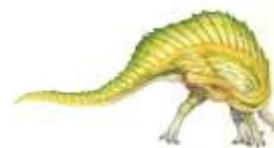
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use





# Operations on Processes

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





# Process Creation

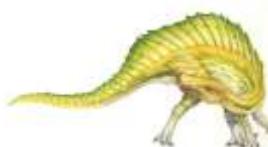
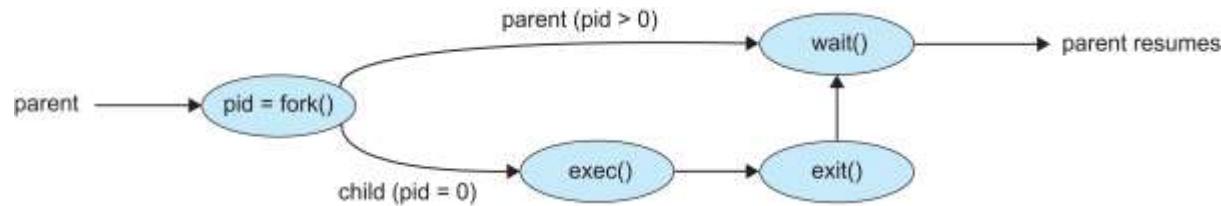
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate





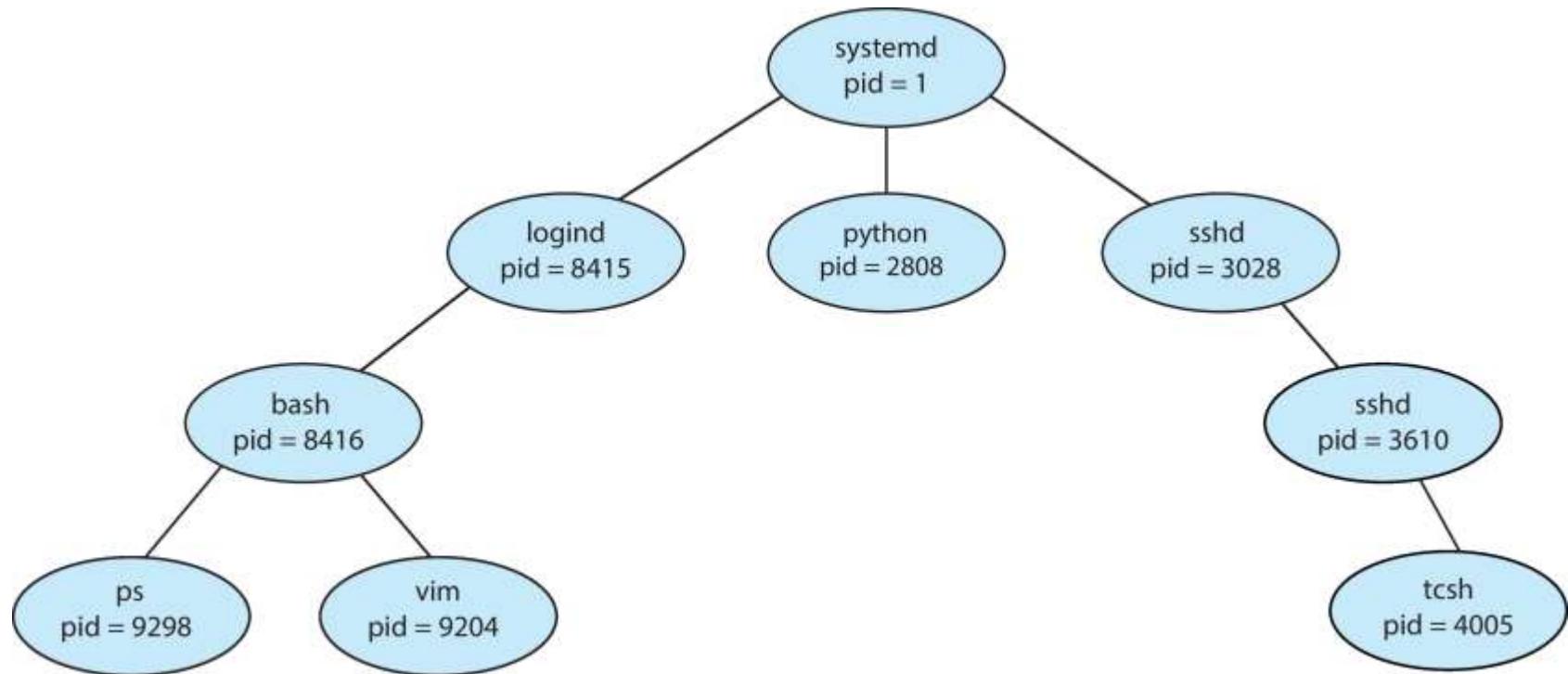
# Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
  - Parent process calls **wait()** waiting for the child to terminate





# A Tree of Processes in Linux





# 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;
}
```





# Creating a Separate Process via Windows API

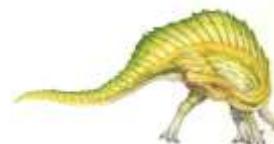
```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
                      NULL, /* don't inherit process handle */
                      NULL, /* don't inherit thread handle */
                      FALSE, /* disable handle inheritance */
                      0, /* no creation flags */
                      NULL, /* use parent's environment block */
                      NULL, /* use parent's existing directory */
                      &si,
                      &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```





# 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





# Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call . The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

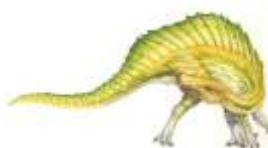
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait()** , process is an **orphan**





# Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.





# Multiprocess Architecture – Chrome Browser

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





# Interprocess Communication

---

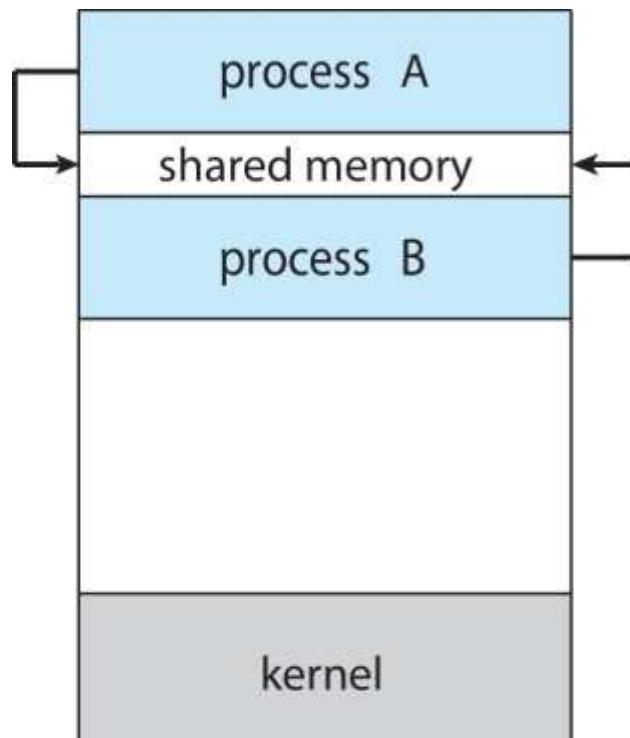
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need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





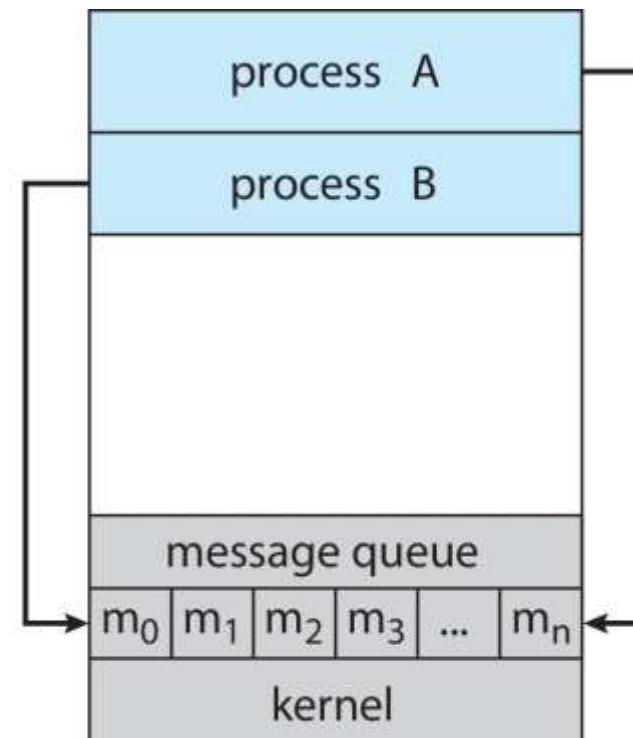
# Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)





# Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - ▶ Producer never waits
    - ▶ Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - ▶ Producer must wait if all buffers are full
    - ▶ Consumer waits if there is no buffer to consume

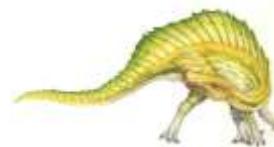




## IPC – Shared Memory

---

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.





# Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use **BUFFER\_SIZE-1** elements





# Producer Process – Shared Memory

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```





# Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```





# What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **counter** that keeps track of the number of full buffers.
- Initially, **counter** is set to 0.
- The integer **counter** is incremented by the producer after it produces a new buffer.
- The integer **counter** is and is decremented by the consumer after it consumes a buffer.





# Producer

---

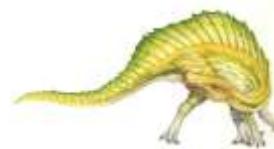
```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```





# Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next_consumed */  
}
```





# 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 <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}

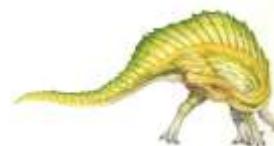




# Race Condition (Cont.)

---

- Question – why was there no race condition in the first solution (where at most  $N - 1$ ) buffers can be filled?
- More in Chapter 6.





# IPC – Message Passing

---

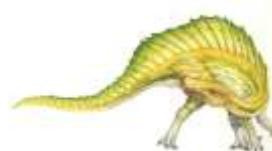
- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send(message)**
  - **receive(message)**
- The *message size* is either fixed or variable





# Message Passing (Cont.)

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?





# Implementation of Communication Link

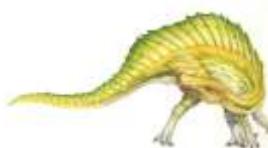
- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering





# Direct Communication

- Processes must name each other explicitly:
  - **send** ( $P$ , message) – send a message to process  $P$
  - **receive**( $Q$ , message) – receive a message from process  $Q$
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional





# Indirect Communication

---

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional





# Indirect Communication (Cont.)

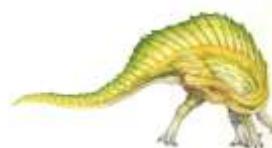
- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **send(A, message)** – send a message to mailbox A
  - **receive(A, message)** – receive a message from mailbox A





# Indirect Communication (Cont.)

- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.  
Sender is notified who the receiver was.

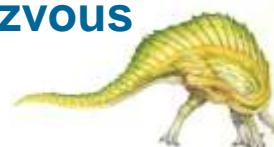




# Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - ▶ A valid message, or
    - ▶ Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**





# Producer-Consumer: Message Passing

- Producer

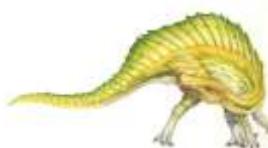
```
message next_produced;
while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
    receive(next_consumed)

    /* consume the item in next_consumed */
}
```

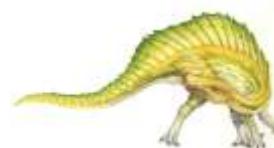




# Buffering

---

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length  
Sender never waits

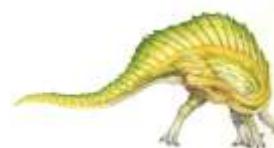




# Examples of IPC Systems - POSIX

- POSIX Shared Memory

- Process first creates shared memory segment  
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object  
`ftruncate(shm_fd, 4096);`
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.





# IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

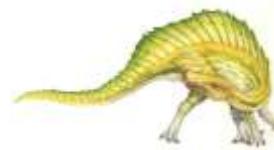
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```





# IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

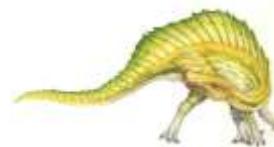
    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

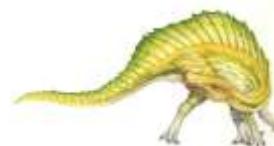
    return 0;
}
```





# Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two ports at creation - Kernel and Notify
  - Messages are sent and received using the `mach_msg()` function
  - Ports needed for communication, created via  
`mach_port_allocate()`
  - Send and receive are flexible; for example four options if mailbox full:
    - ▶ Wait indefinitely
    - ▶ Wait at most n milliseconds
    - ▶ Return immediately
    - ▶ Temporarily cache a message



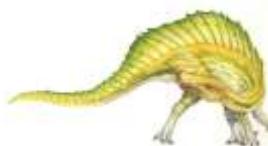


# Mach Messages

```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```





# Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
         MACH_SEND_MSG, // sending a message
         sizeof(message), // size of message sent
         0, // maximum size of received message - unnecessary
         MACH_PORT_NULL, // name of receive port - unnecessary
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```



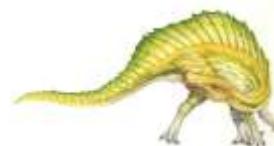


# Mach Message Passing - Server

```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
         MACH_RCV_MSG, // sending a message
         0, // size of message sent
         sizeof(message), // maximum size of received message
         server, // name of receive port
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```

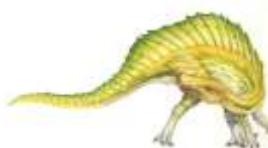




# Examples of IPC Systems – Windows

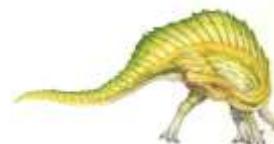
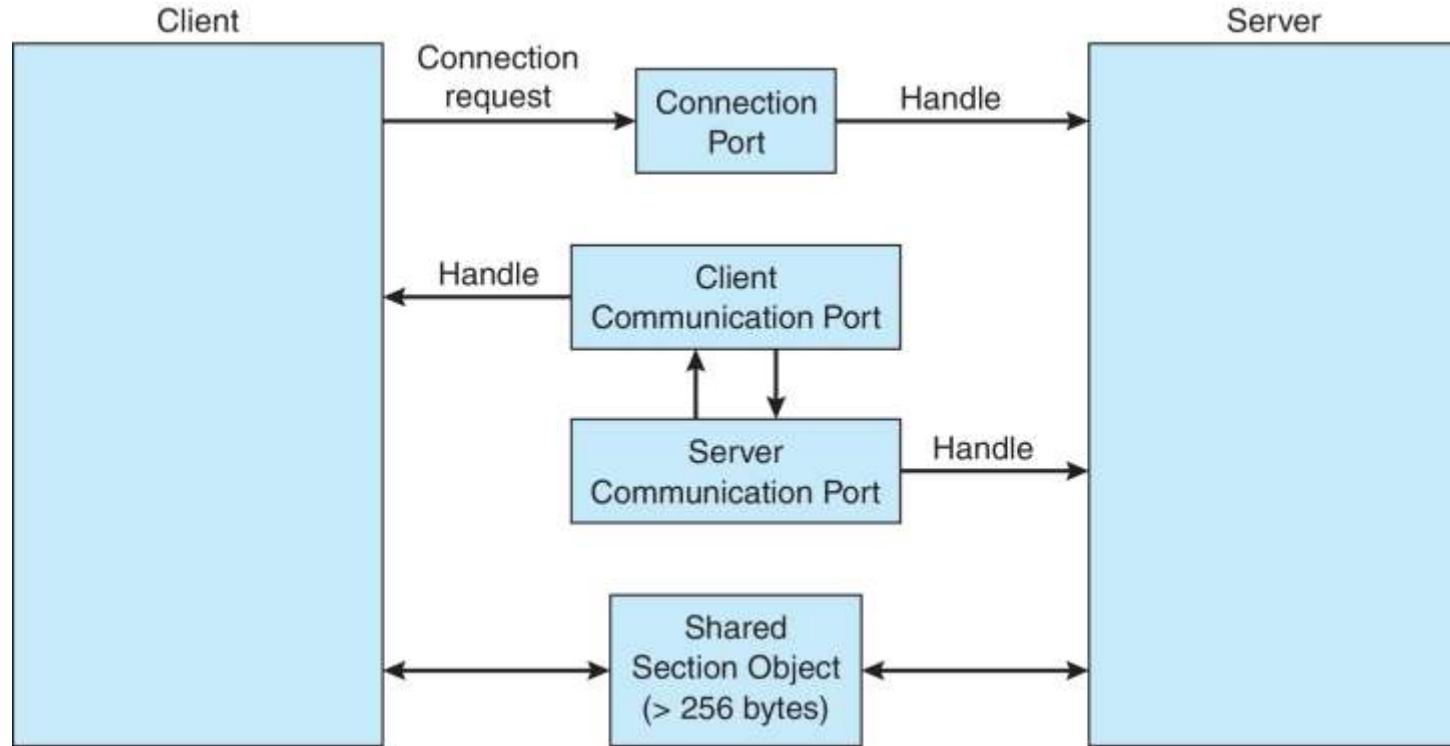
---

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - ▶ The client opens a handle to the subsystem's **connection port** object.
    - ▶ The client sends a connection request.
    - ▶ The server creates two private **communication ports** and returns the handle to one of them to the client.
    - ▶ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.





# Local Procedure Calls in Windows

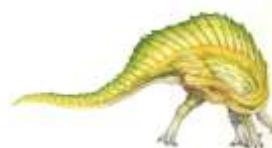




# Pipes

---

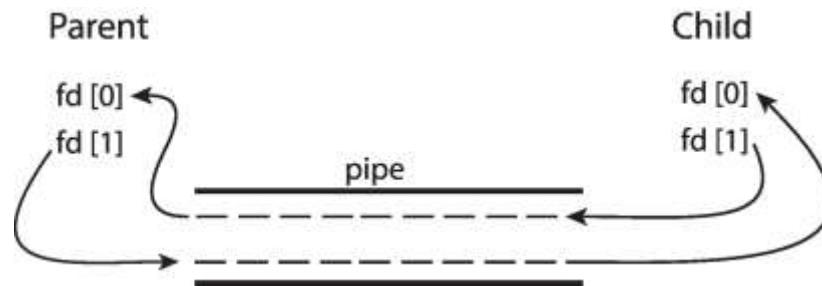
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the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





# Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

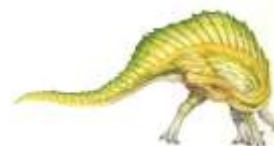




# Named Pipes

---

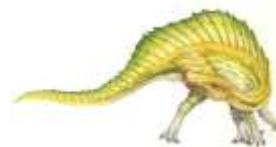
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems





# Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls





# Sockets

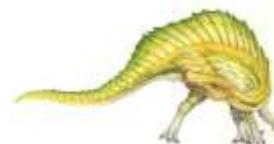
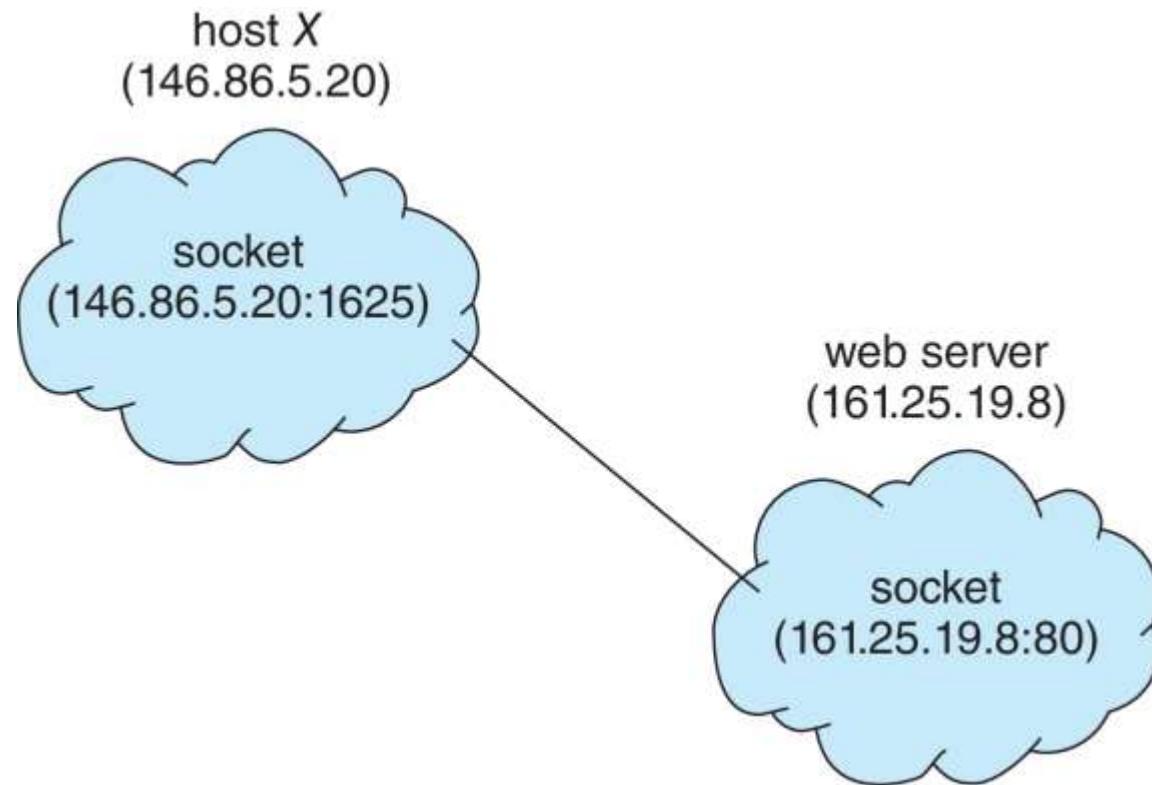
---

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





# Socket Communication





# Sockets in Java

- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **MulticastSocket** class— data can be sent to multiple recipients
- Consider this “Date” server in Java:

```
import java.net.*;
import java.io.*;

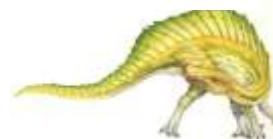
public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```





# Sockets in Java

## The equivalent Date client

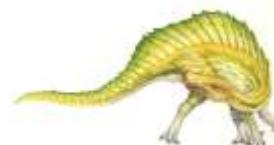
```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```





# Remote Procedure Calls

---

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**





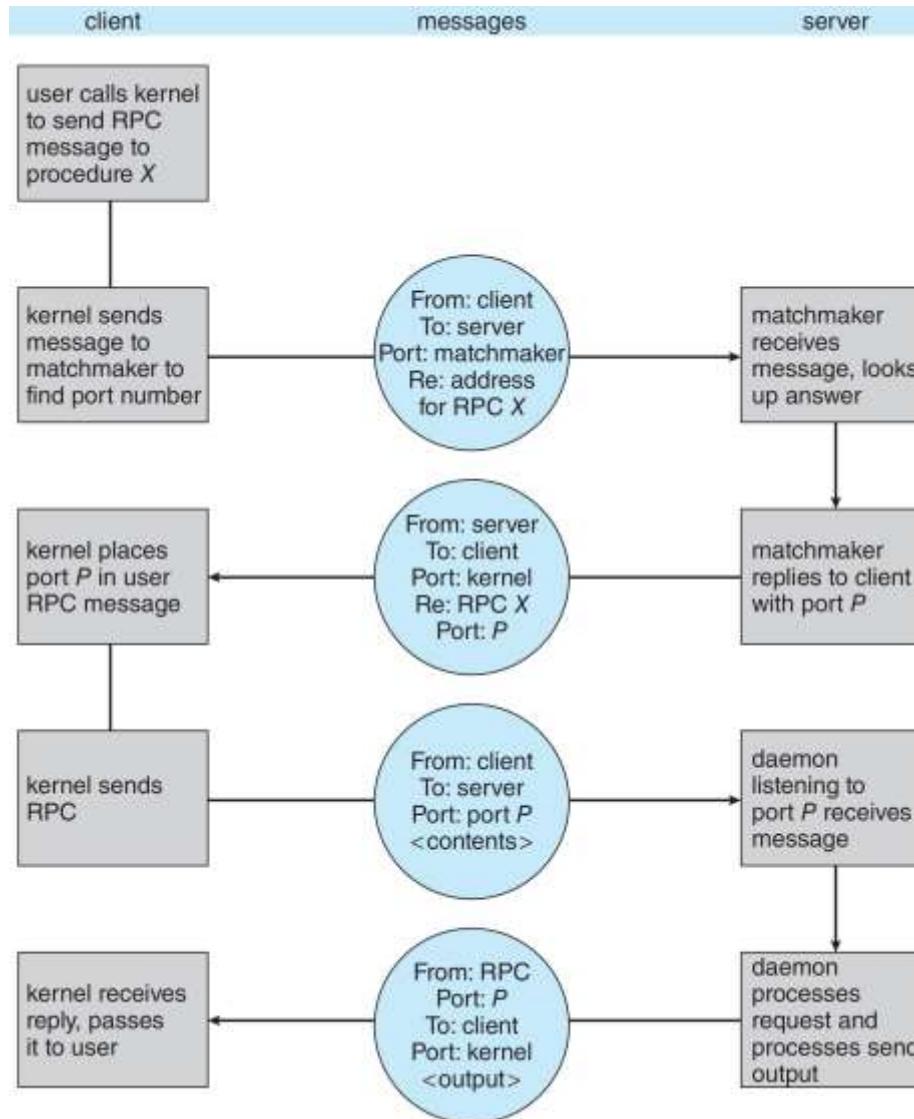
# Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

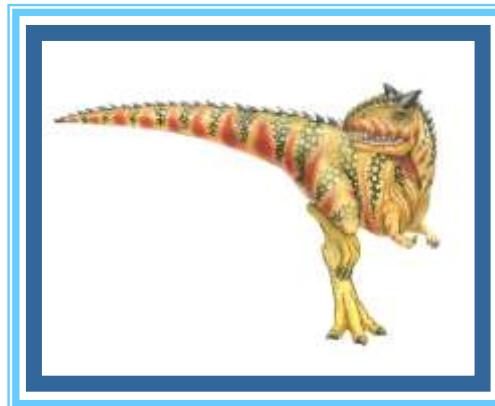




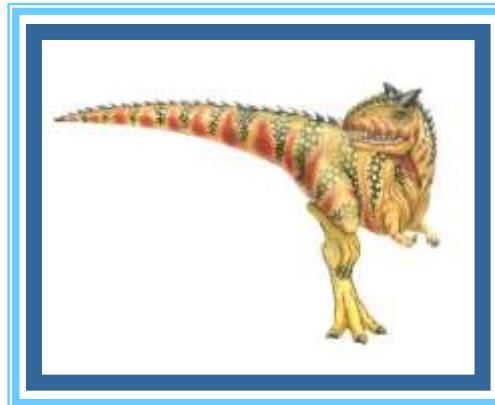
# Execution of RPC



# End of Chapter 3



# Chapter 4: Threads & Concurrency

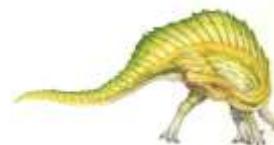




# Outline

---

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

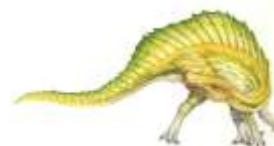




# Objectives

---

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the Pthreads, Java, and Windows threading APIs

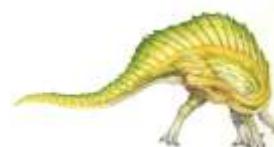




# Motivation

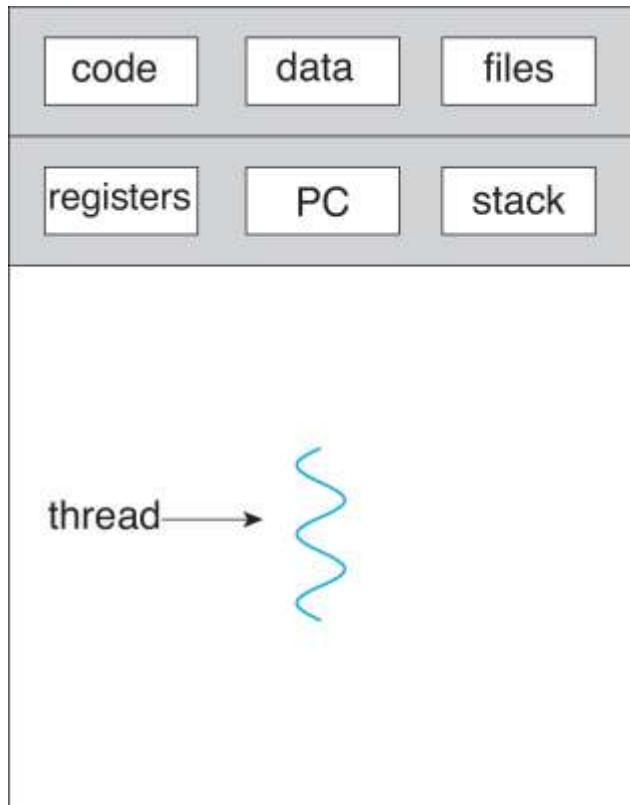
---

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

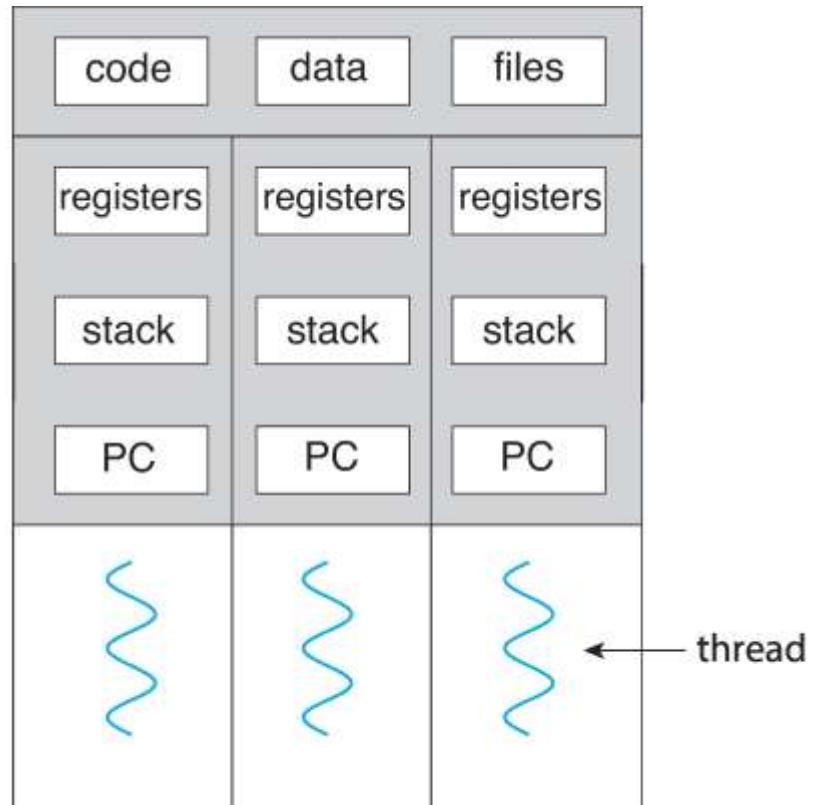




# Single and Multithreaded Processes



single-threaded process

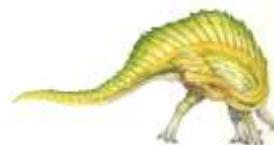
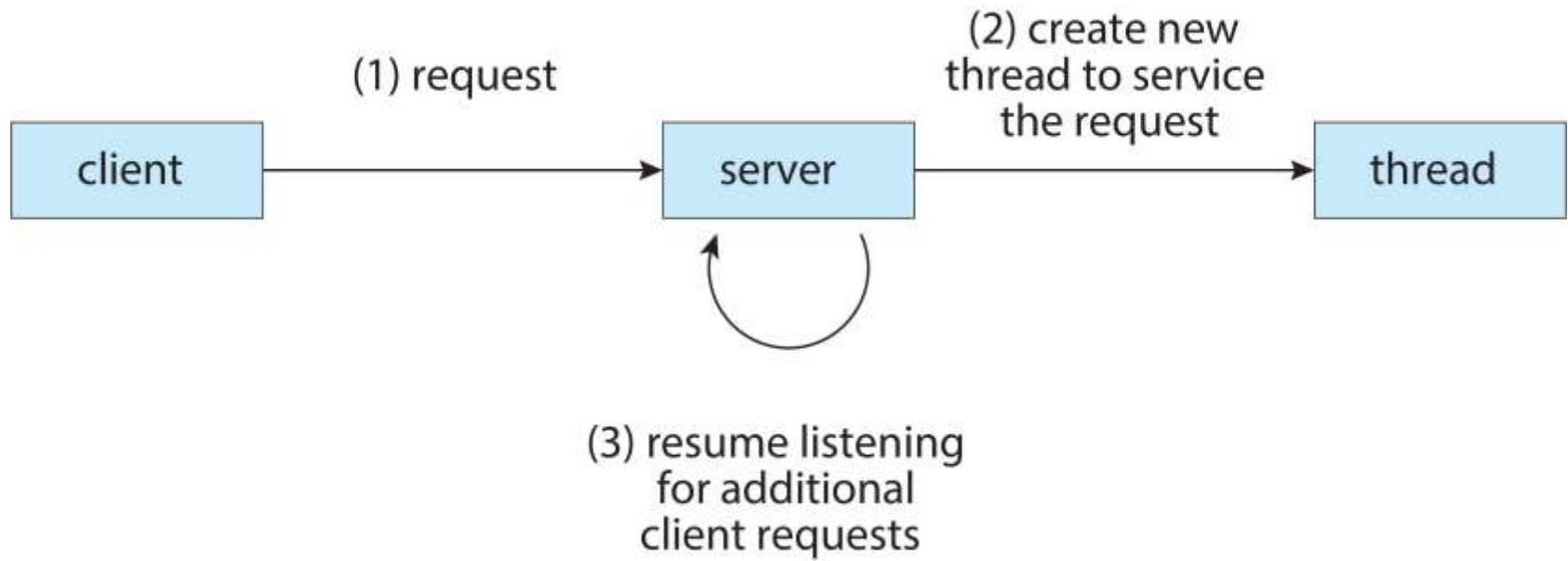


multithreaded process





# Multithreaded Server Architecture

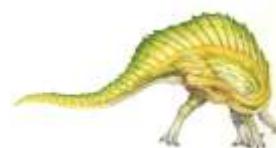




# Benefits

---

- **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, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures





# Multicore Programming

---

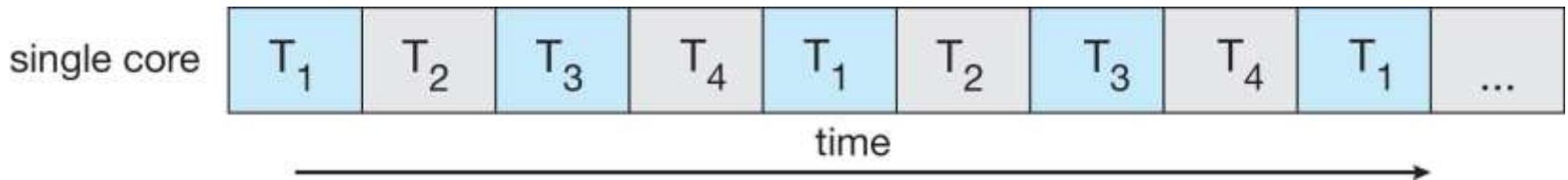
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency



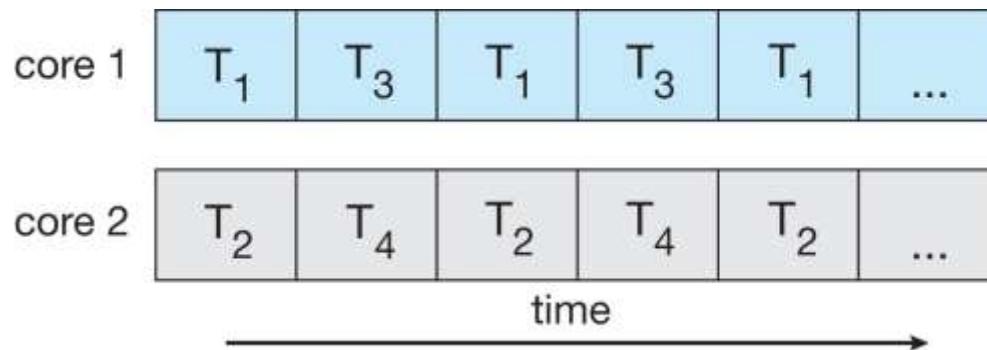


# Concurrency vs. Parallelism

- Concurrent execution on single-core system:



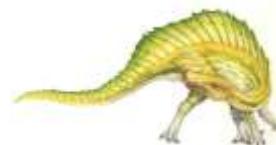
- Parallelism on a multi-core system:





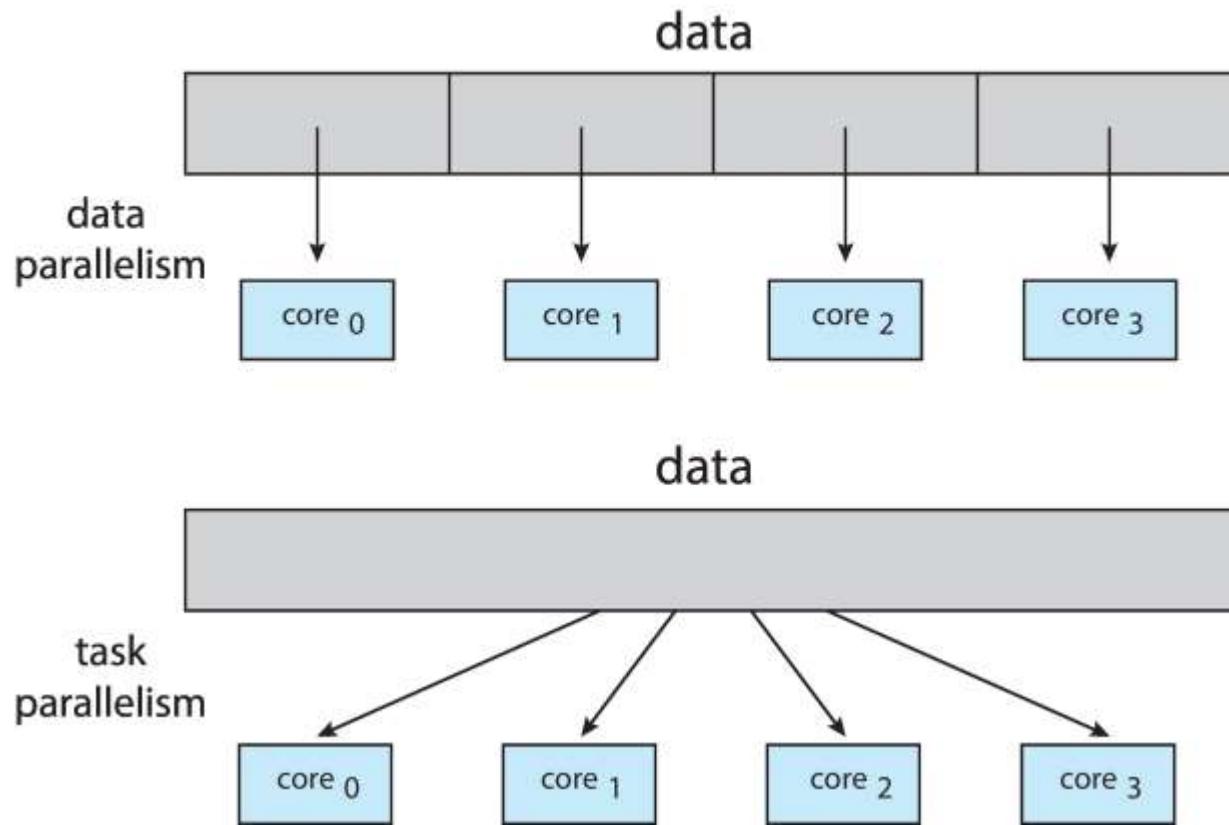
# Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation





# Data and Task Parallelism





# Amdahl's Law

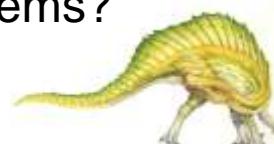
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- $N$  processing cores

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

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As  $N$  approaches infinity, speedup approaches  $1 / S$

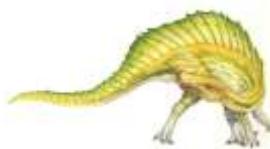
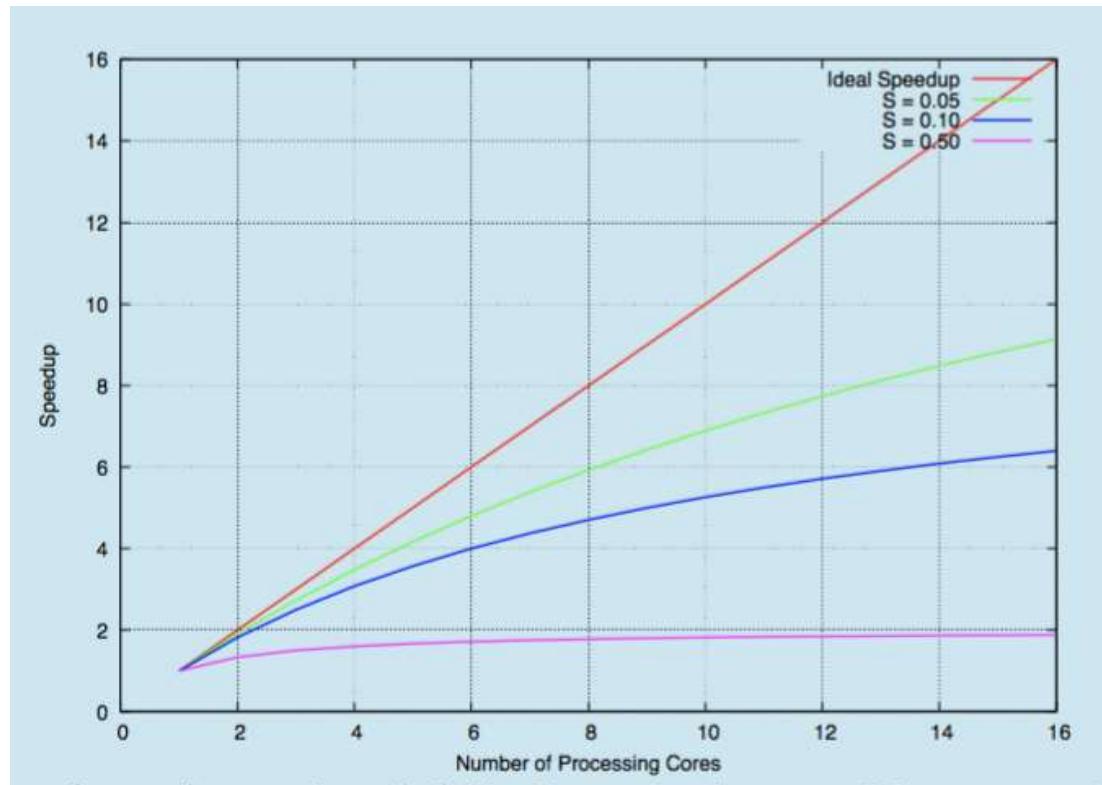
**Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

- But does the law take into account contemporary multicore systems?





# Amdahl's Law





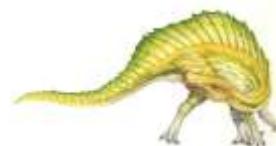
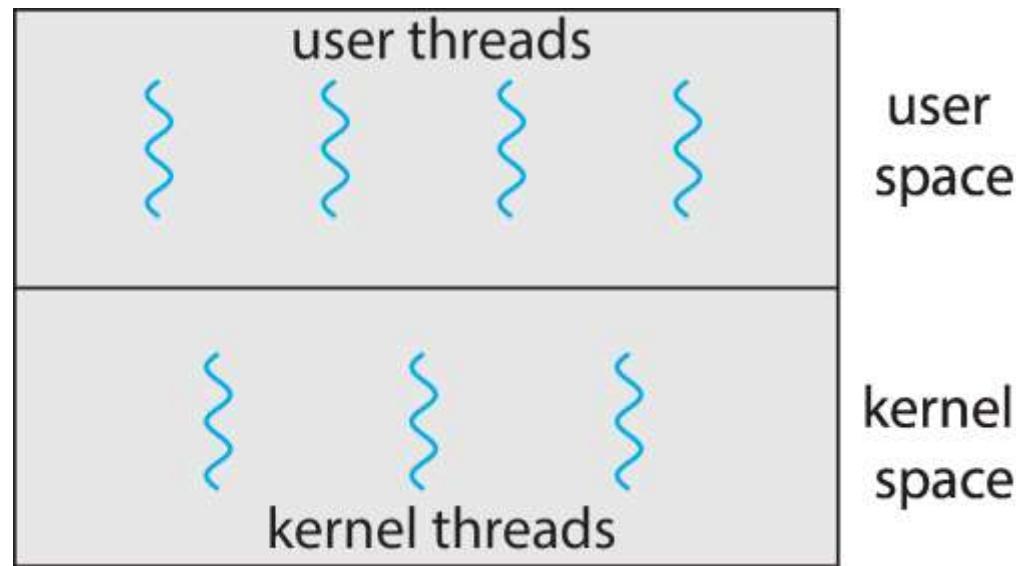
# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android





# User and Kernel Threads

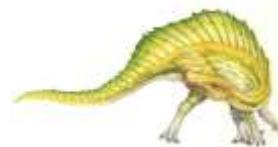




# Multithreading Models

---

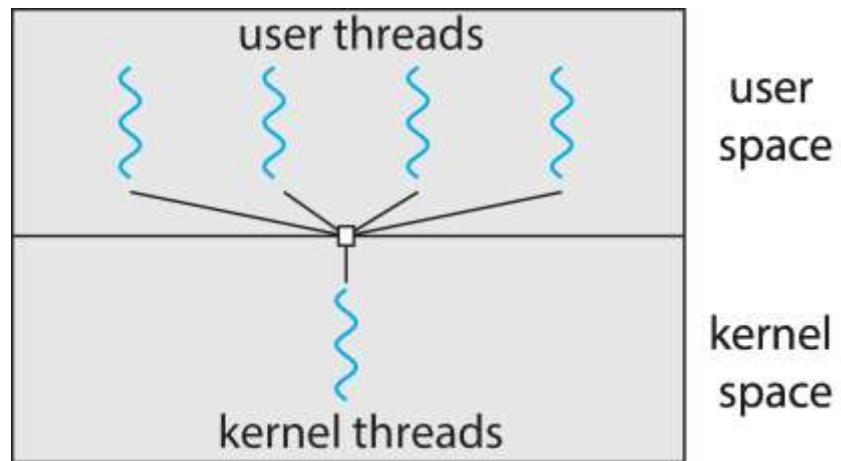
- Many-to-One
- One-to-One
- Many-to-Many





# Many-to-One

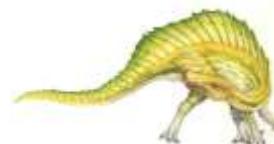
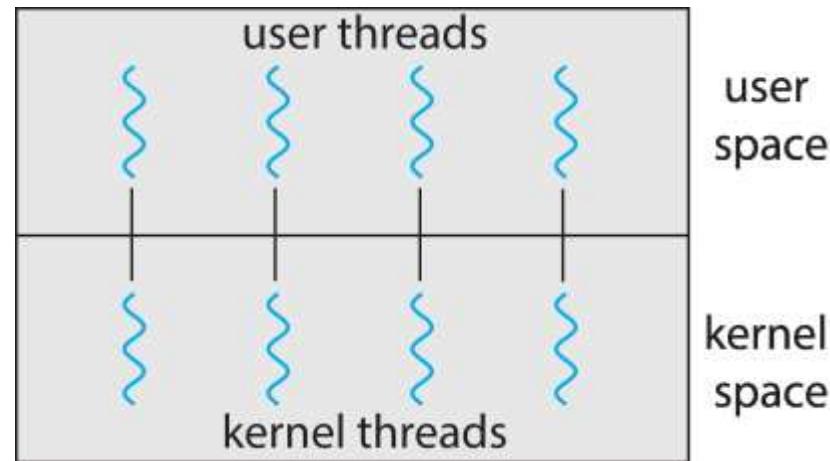
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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





# One-to-One

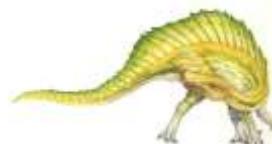
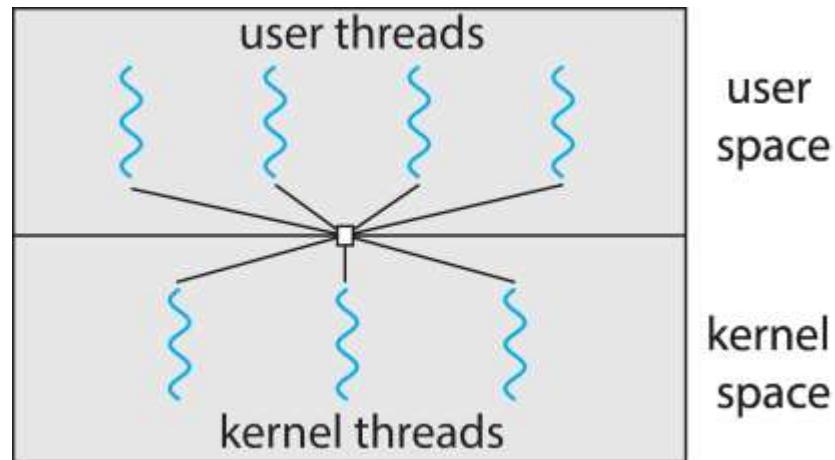
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux





# Many-to-Many Model

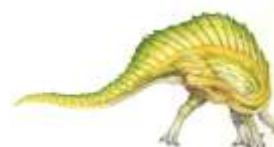
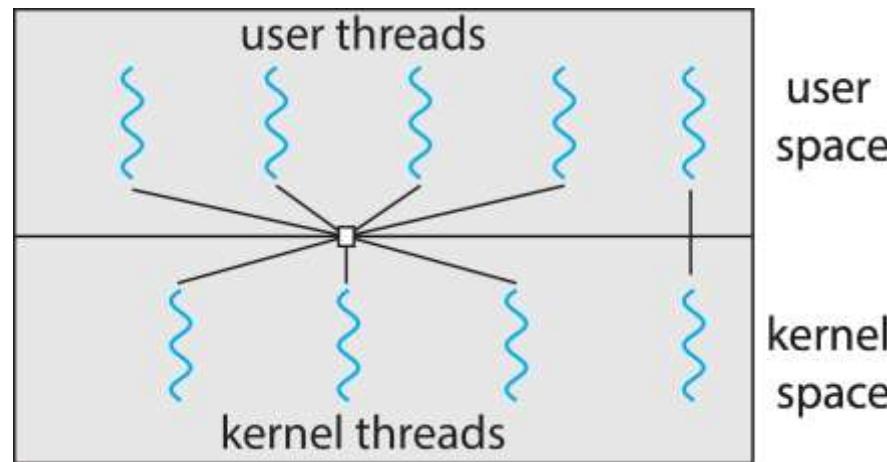
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common





# Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread





# Thread Libraries

---

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS





# 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 (Linux & Mac OS X)





# Pthreads Example

---

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

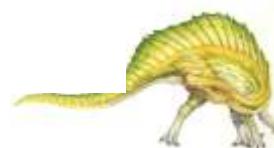
#include <stdlib.h>

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 */

    /* set the default attributes of the thread */
    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);
}
```





# Pthreads Example (Cont.)

---

```
/* The thread will execute 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);
}
```



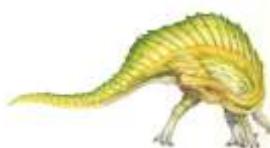


# Pthreads Code for Joining 10 Threads

```
#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);
```

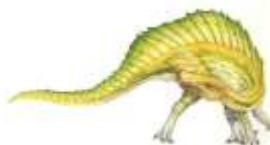




# Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```





# Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

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





# Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

- Standard practice is to implement Runnable interface





# Java Threads

## Implementing Runnable interface:

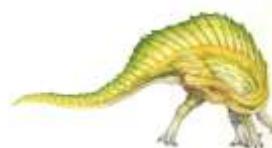
```
class Task implements Runnable  
{  
    public void run() {  
        System.out.println("I am a thread.");  
    }  
}
```

## Creating a thread:

```
Thread worker = new Thread(new Task());  
worker.start();
```

## Waiting on a thread:

```
try {  
    worker.join();  
}  
catch (InterruptedException ie) { }
```





# Java Executor Framework

- Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

```
public interface Executor
{
    void execute(Runnable command);
}
```

- The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```





# Java Executor Framework

---

```
import java.util.concurrent.*;  
  
class Summation implements Callable<Integer>  
{  
    private int upper;  
    public Summation(int upper) {  
        this.upper = upper;  
    }  
  
    /* The thread will execute in this method */  
    public Integer call() {  
        int sum = 0;  
        for (int i = 1; i <= upper; i++)  
            sum += i;  
  
        return new Integer(sum);  
    }  
}
```



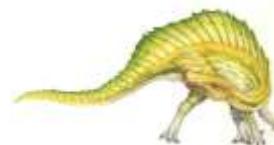


# Java Executor Framework (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```





# Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks



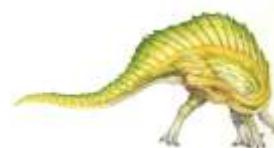


# Thread Pools

---

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - ▶ i.e., Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```





# Java Thread Pools

---

- Three factory methods for creating thread pools in Executors class:
  - static ExecutorService newSingleThreadExecutor()
  - static ExecutorService newFixedThreadPool(int size)
  - static ExecutorService newCachedThreadPool()





# Java Thread Pools (Cont.)

---

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

        /* Run each task using a thread in the pool */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

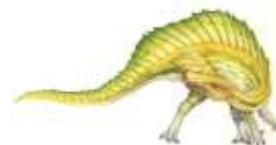
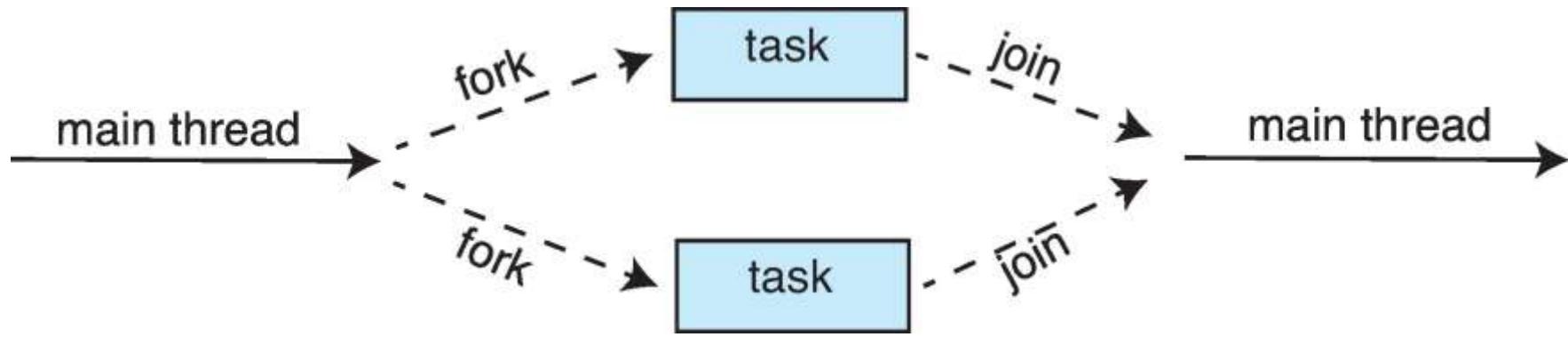
        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```





# Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.





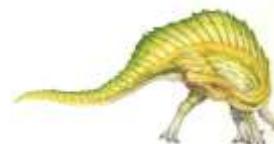
# Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem)
        subtask2 = fork(new Task(subset of problem)

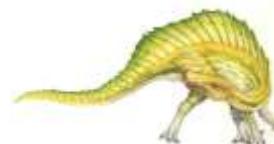
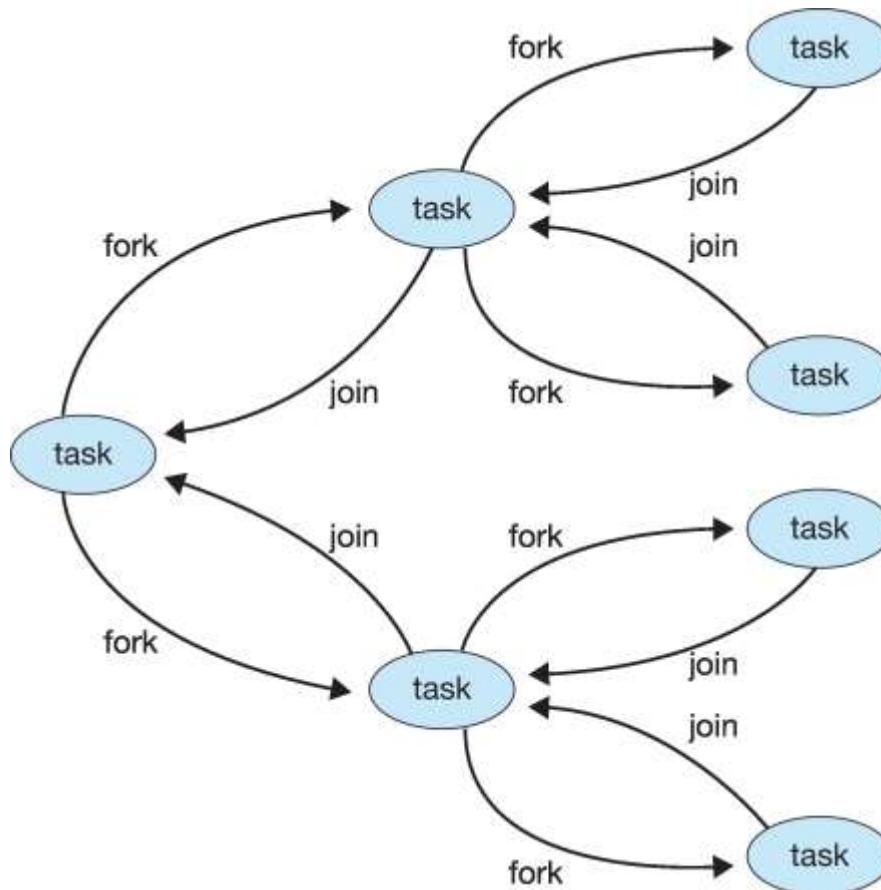
        result1 = join(subtask1)
        result2 = join(subtask2)

    return combined results
```





# Fork-Join Parallelism



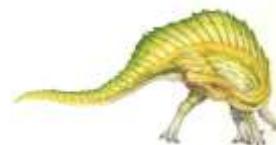


# Fork-Join Parallelism in Java

---

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```





# Fork-Join Parallelism in Java

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

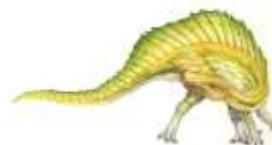
    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];

            return sum;
        }
        else {
            int mid = (begin + end) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

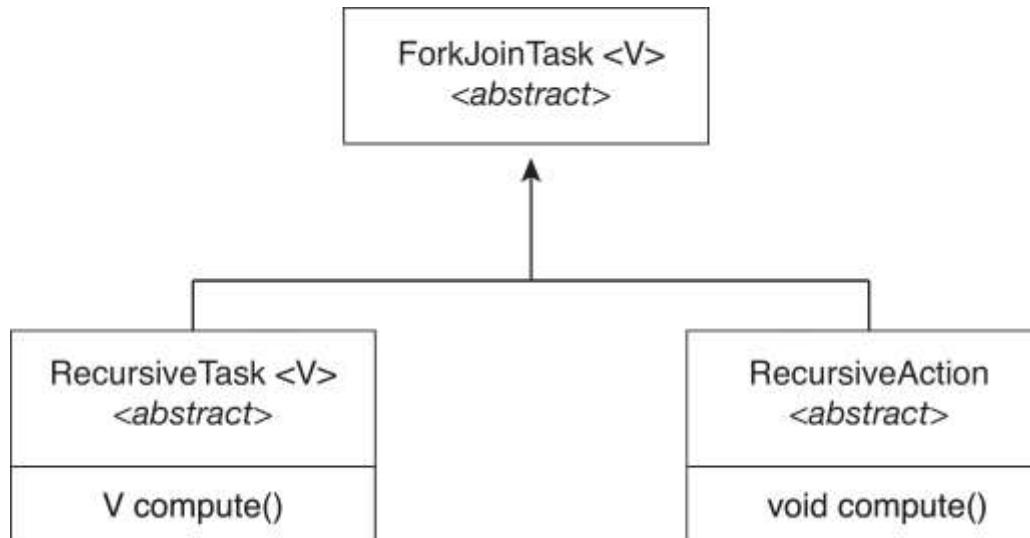
            return rightTask.join() + leftTask.join();
        }
    }
}
```





# Fork-Join Parallelism in Java

- The **ForkJoinTask** is an abstract base class
- **RecursiveTask** and **RecursiveAction** classes extend **ForkJoinTask**
- **RecursiveTask** returns a result (via the return value from the `compute()` method)
- **RecursiveAction** does not return a result





# OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

**#pragma omp parallel**

Create as many threads as there are cores

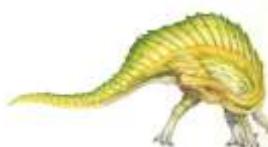
```
#include <omp.h>
#include <stdio.h>

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

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

    /* sequential code */

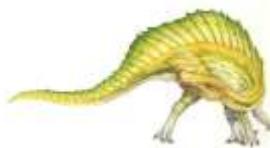
    return 0;
}
```





- 
- Run the for loop in parallel

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





# Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” :

```
^{ printf("I am a block"); }
```

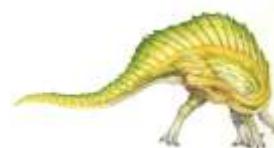
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue





# Grand Central Dispatch

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**
    - ▶ Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
    - ▶ Four system wide queues divided by quality of service:
      - QOS\_CLASS\_USER\_INTERACTIVE
      - QOS\_CLASS\_USER\_INITIATED
      - QOS\_CLASS\_USER.Utility
      - QOS\_CLASS\_USER\_BACKGROUND





# Grand Central Dispatch

- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue  
    (QOS_CLASS_USER_INITIATED, 0)  
  
dispatch_async(queue, { print("I am a closure.") })
```





# Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- The same for loop written using TBB with **parallel\_for** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```





# Threading Issues

---

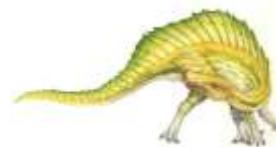
- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





# Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads





# Signal Handling

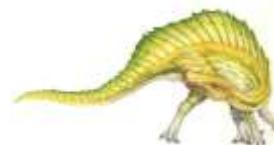
- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
    1. default
    2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process





# Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

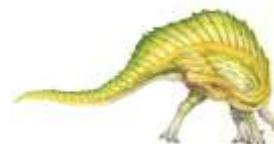




# Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
. . .  
  
/* cancel the thread */  
pthread_cancel(tid);  
  
/* wait for the thread to terminate */  
pthread_join(tid,NULL);
```





# Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals





# Thread Cancellation in Java

- Deferred cancellation uses the `interrupt()` method, which sets the interrupted status of a thread.

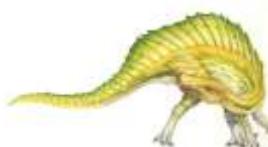
```
Thread worker;
```

```
    . . .
```

```
/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```





# Thread-Local Storage

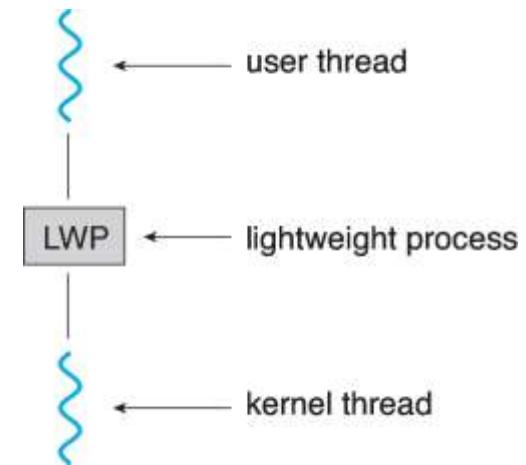
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread





# Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

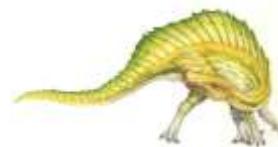




# Operating System Examples

---

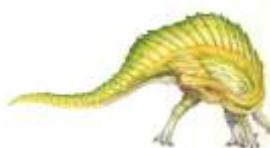
- Windows Threads
- Linux Threads





# Windows Threads

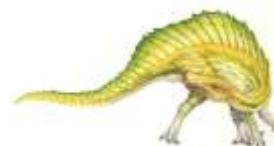
- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread





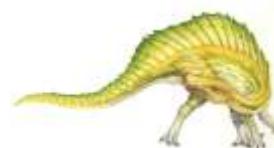
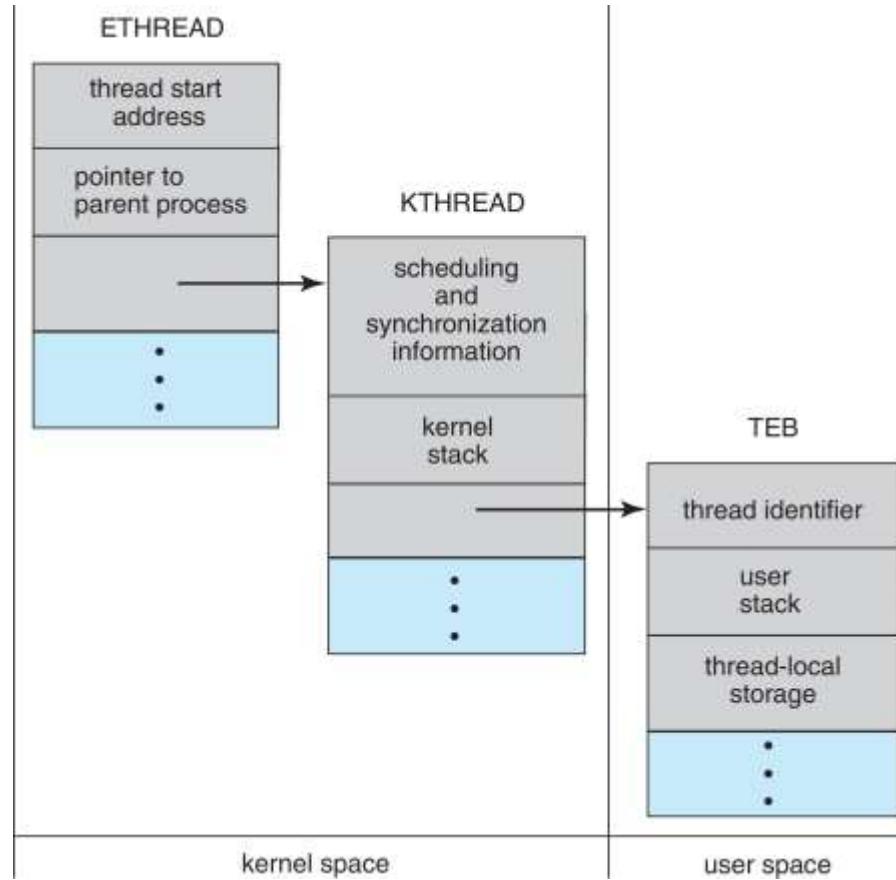
# Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space





# Windows Threads Data Structures





# Linux Threads

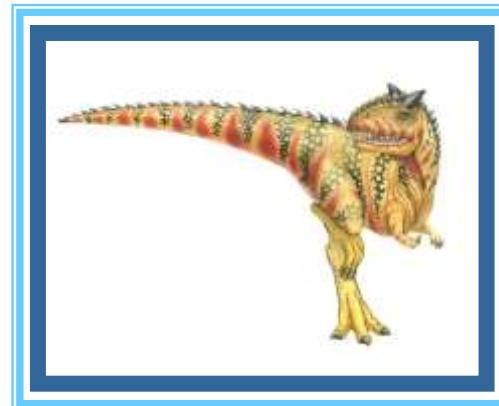
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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

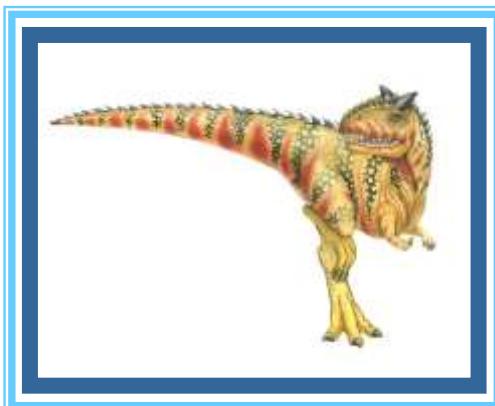
- `struct task_struct` points to process data structures (shared or unique)



# End of Chapter 4



# Chapter 5: CPU Scheduling





# Outline

---

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

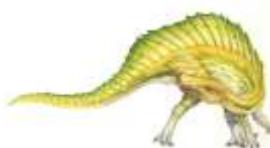




# Objectives

---

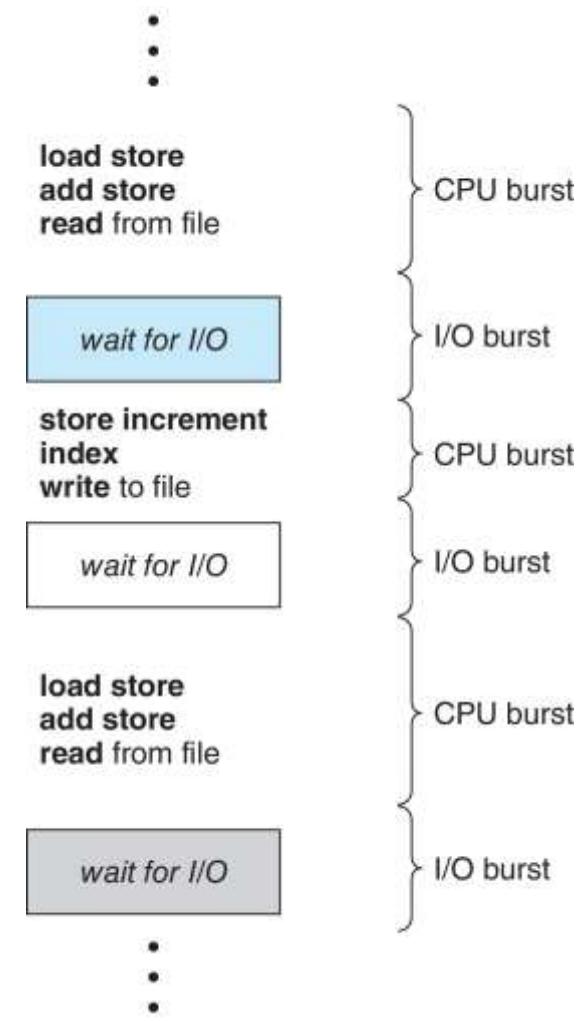
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms





# 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



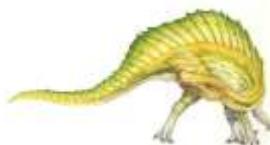
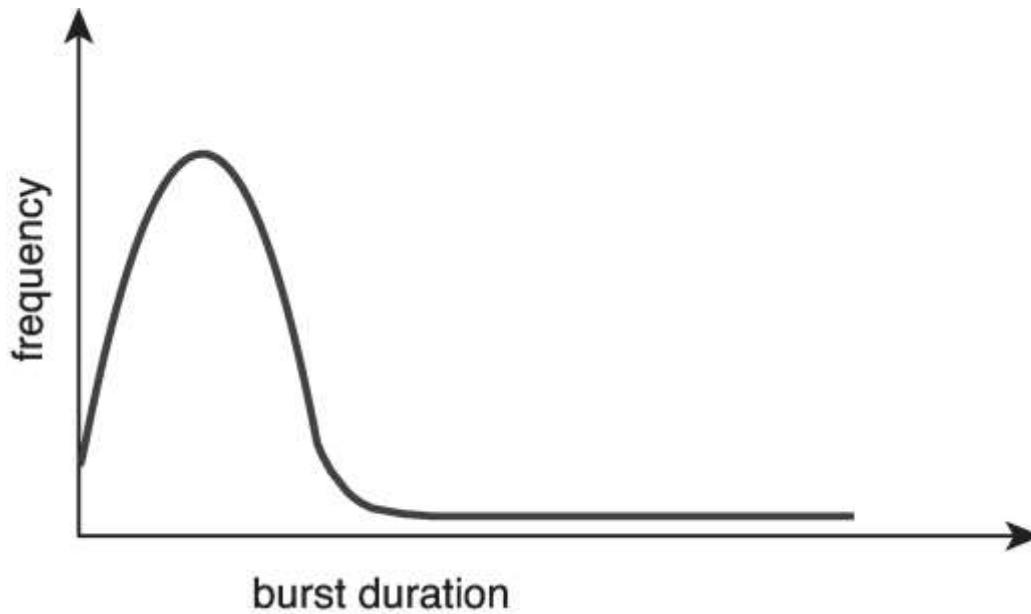


# Histogram of CPU-burst Times

---

Large number of short bursts

Small number of longer bursts





# CPU Scheduler

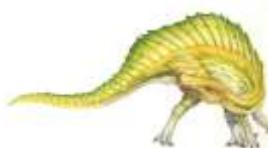
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.





# 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.





# Preemptive Scheduling and Race Conditions

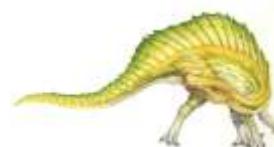
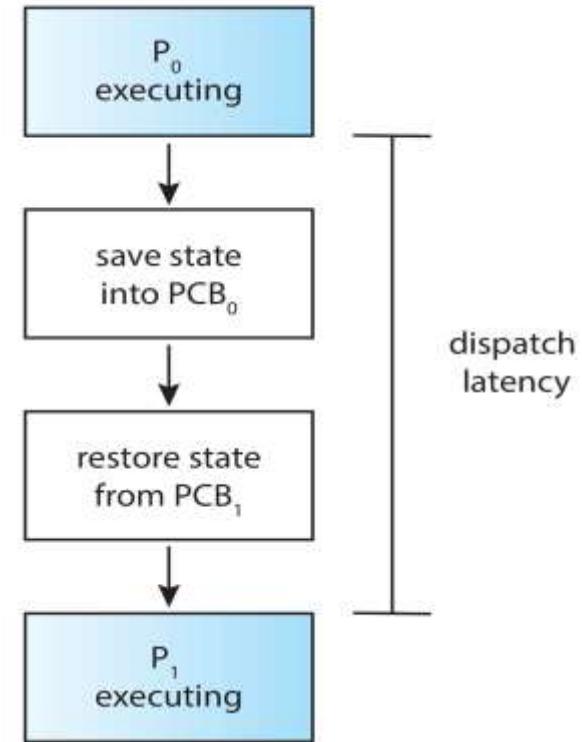
- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.





# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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

---

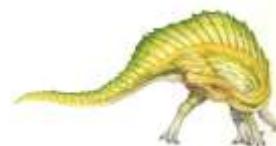
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response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





# Scheduling Algorithm Optimization Criteria

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





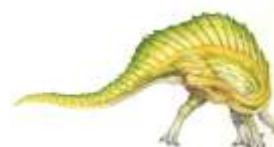
# First- Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	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:  $(0 + 24 + 27)/3 = 17$





# FCFS Scheduling (Cont.)

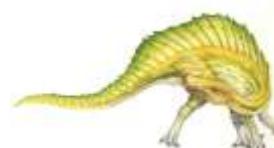
Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ,  $P_3 = 3$
- 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





# 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
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

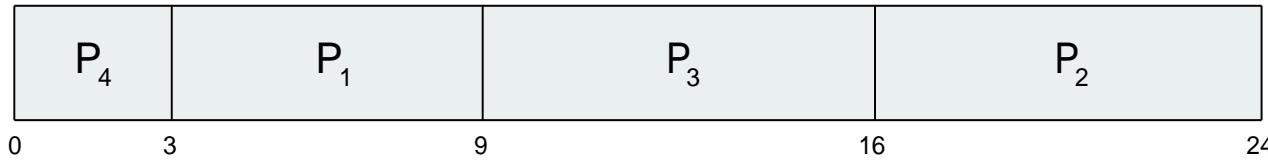




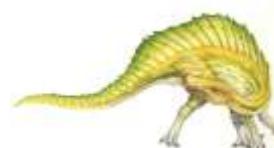
# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



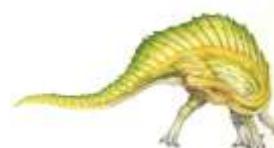
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$





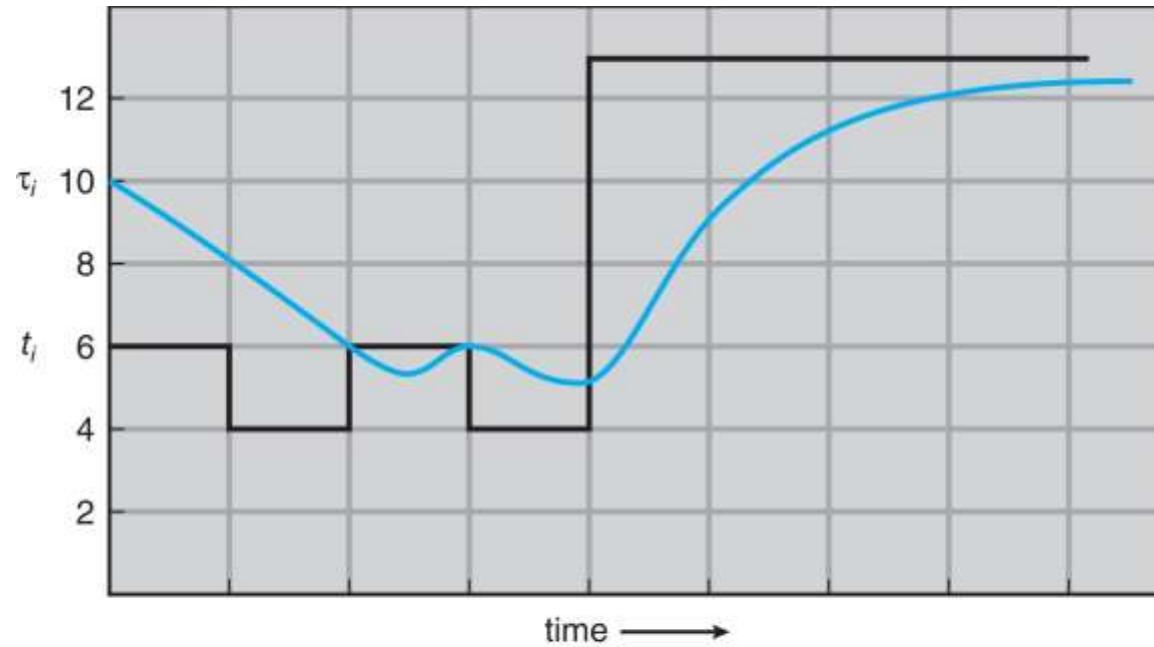
# Determining Length of Next CPU Burst

- 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,  $\alpha$  set to  $\frac{1}{2}$





# Prediction of the Length of the Next CPU Burst



CPU burst ( $t_i$ )	6	4	6	6	4	13	13	13	...
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...



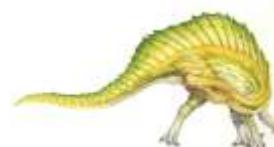


# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$ 
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

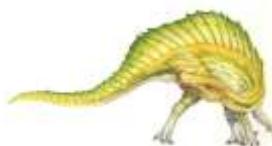
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successor predecessor term has less weight than its predecessor





# Shortest Remaining Time First Scheduling

- Preemptive version of SJN
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN algorithm.
- Is SRT more “optimal” than SJN in terms of the minimum average waiting time for a given set of processes?



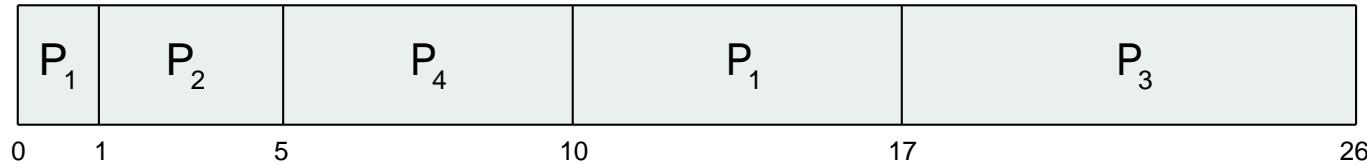


# Example of Shortest-remaining-time-first

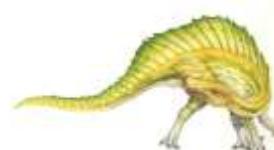
- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive SJF Gantt Chart*



- Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$





# 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  $\Rightarrow$  FIFO (FCFS)
  - $q$  small  $\Rightarrow$  RR
- Note that  $q$  must be large with respect to context switch, otherwise overhead is too high

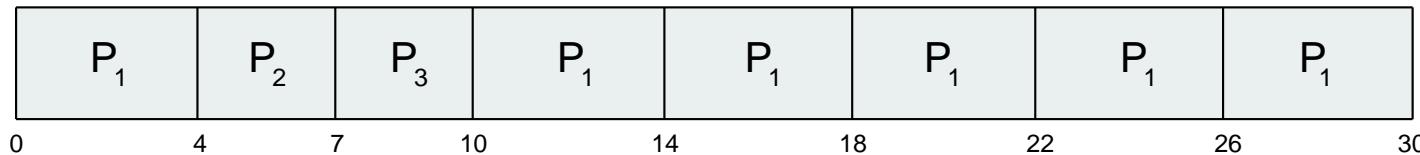




# Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

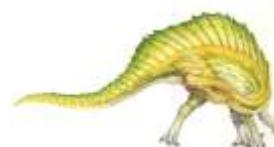
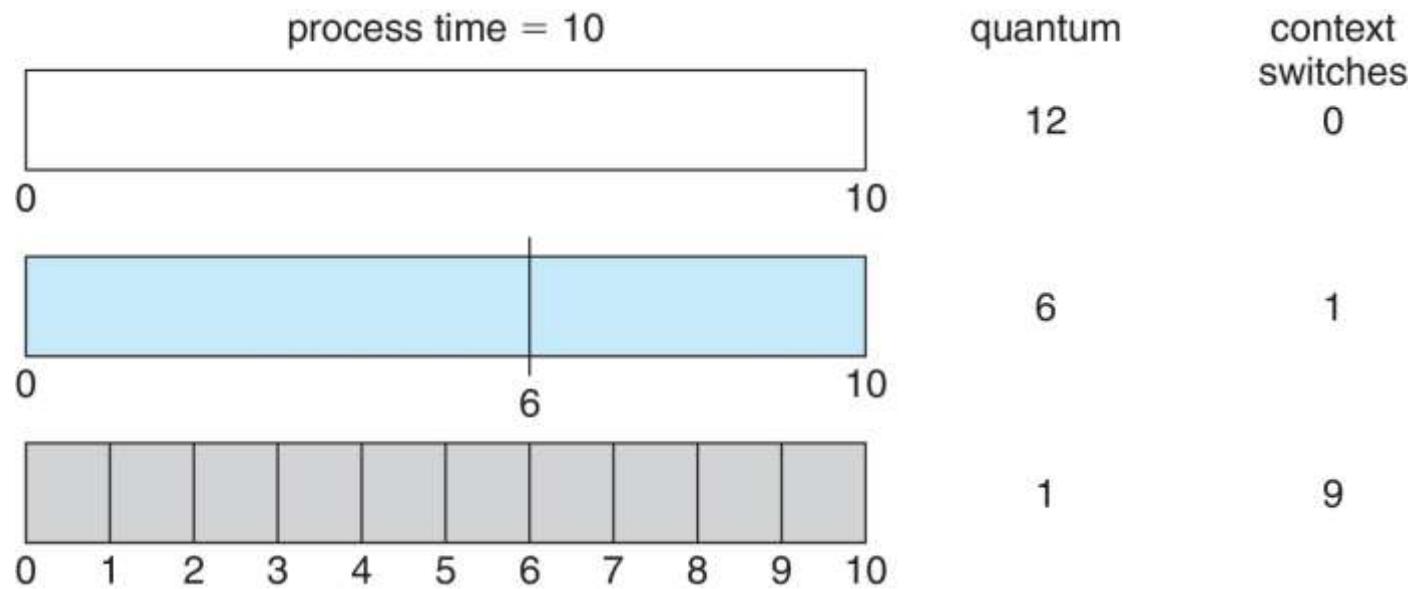


- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds



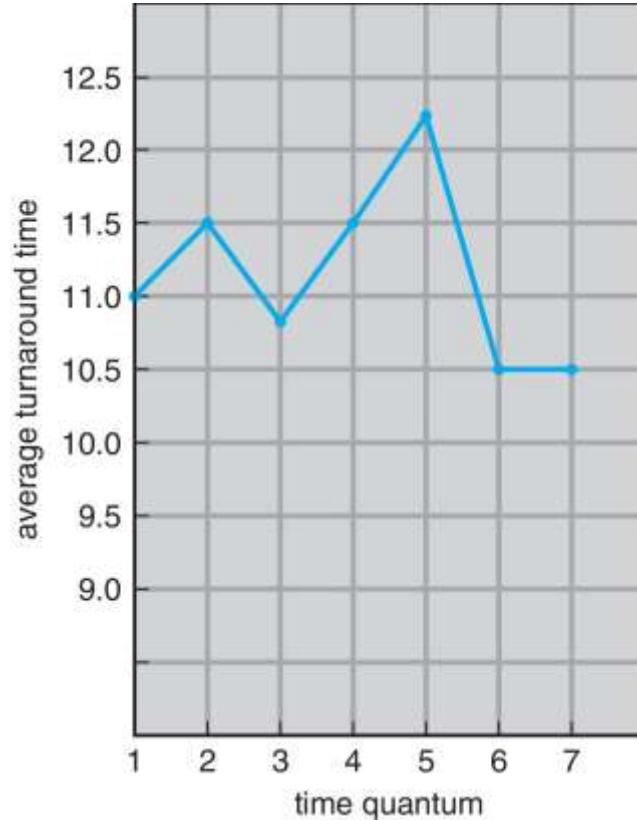


# Time Quantum and Context Switch Time





# Turnaround Time Varies With The Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts  
should be shorter than  $q$





# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process





# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	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 = 8.2



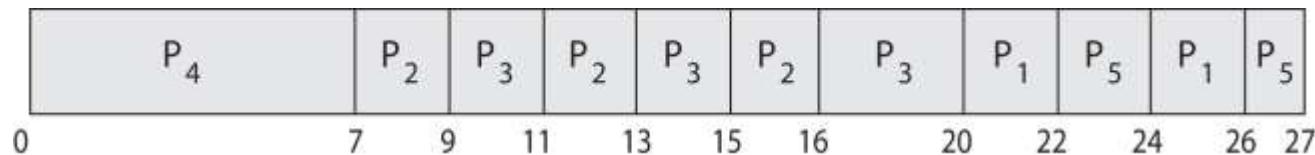


# Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

- Gantt Chart with time quantum = 2

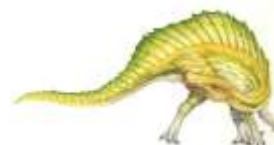




# Multilevel Queue

---

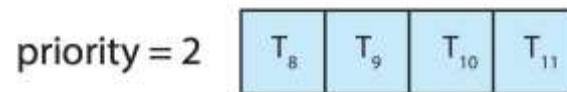
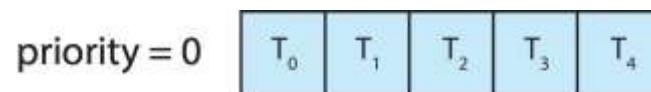
- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues





# Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



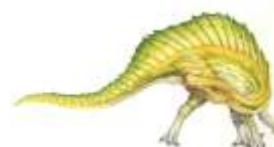
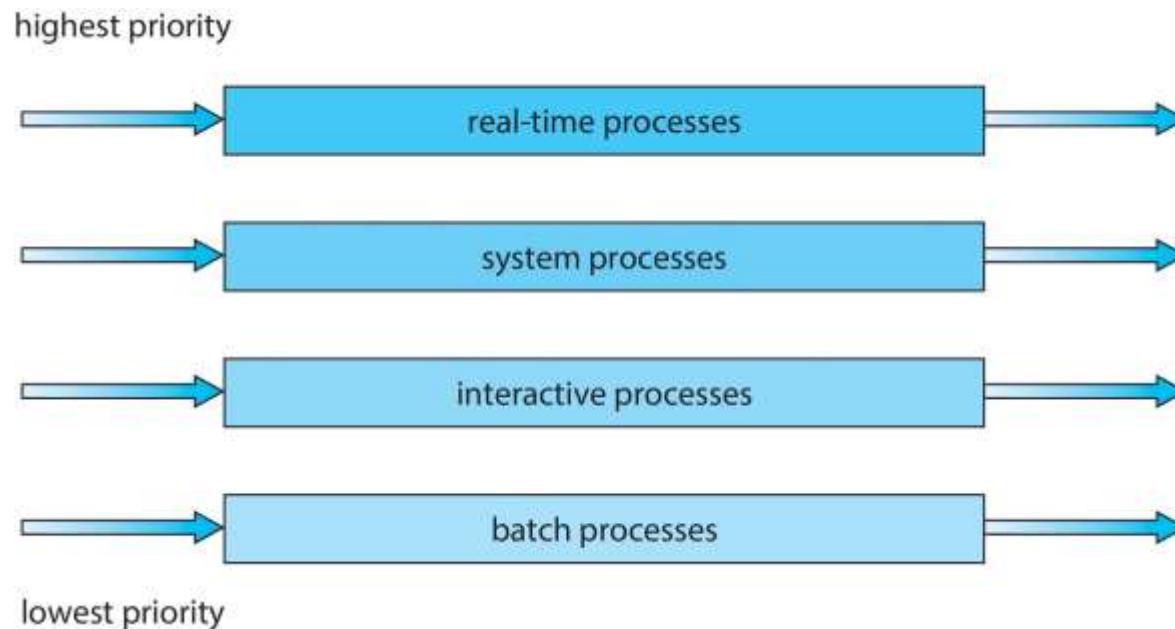
●  
●  
●





# Multilevel Queue

- Prioritization based upon process type





# Multilevel Feedback Queue

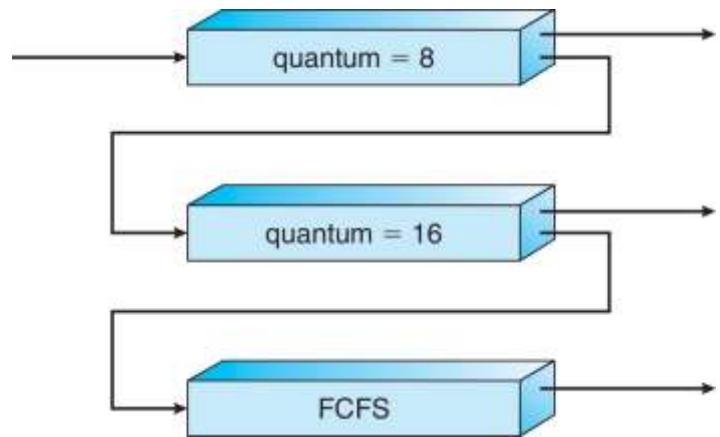
- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method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





# 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 process enters queue  $Q_0$  which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue  $Q_1$
  - At  $Q_1$ , job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue  $Q_2$





# Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
    - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





# Pthread Scheduling

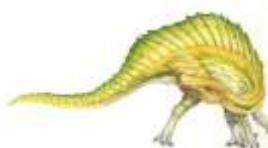
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD\_SCOPE\_PROCESS schedules threads using PCS scheduling
  - PTHREAD\_SCOPE\_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD\_SCOPE\_SYSTEM





# Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```





# Pthread Scheduling API

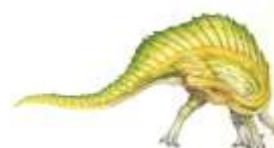
```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





# Multiple-Processor Scheduling

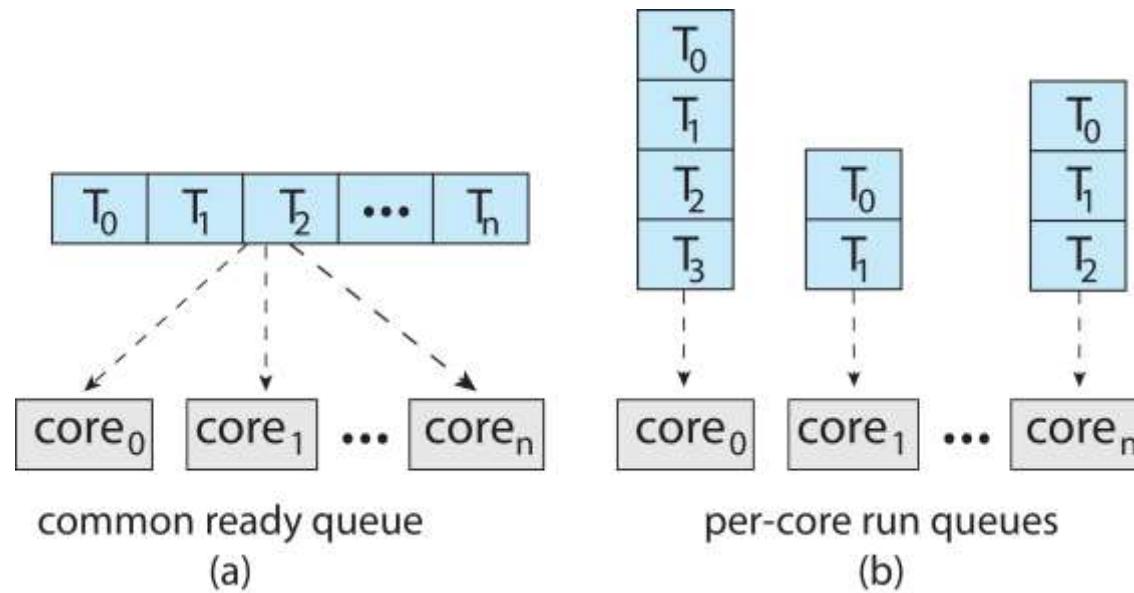
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing





# Multiple-Processor Scheduling

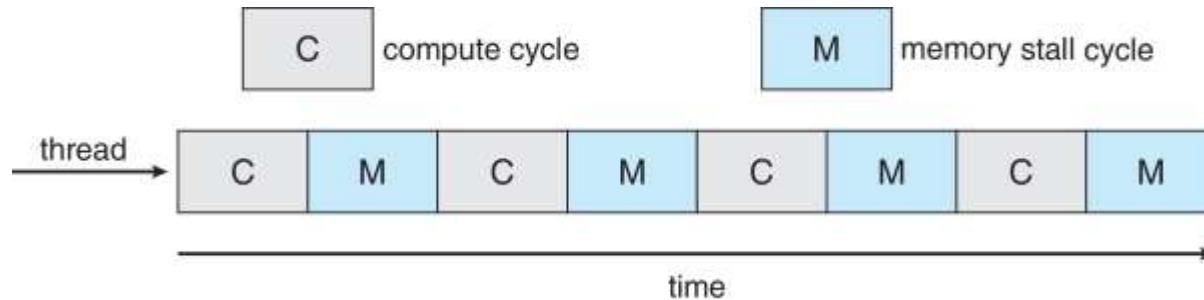
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)





# Multicore Processors

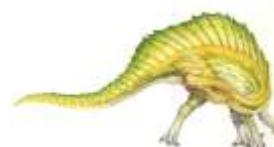
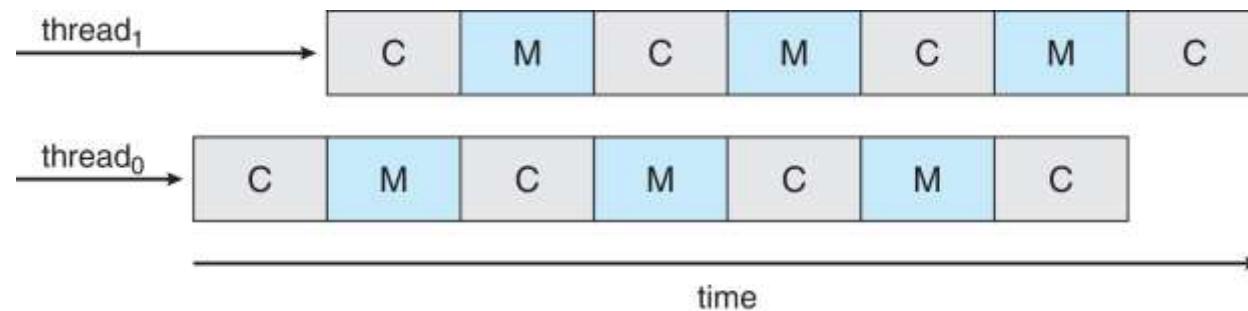
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure





# Multithreaded Multicore System

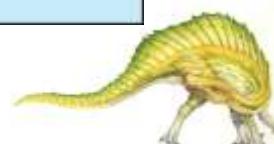
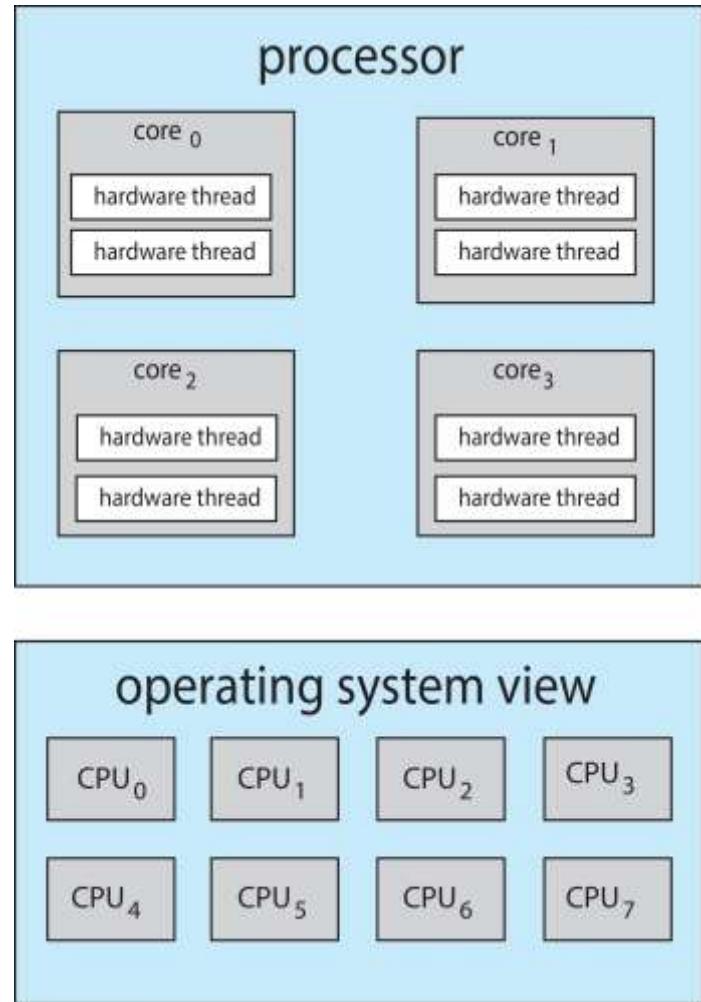
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure





# Multithreaded Multicore System

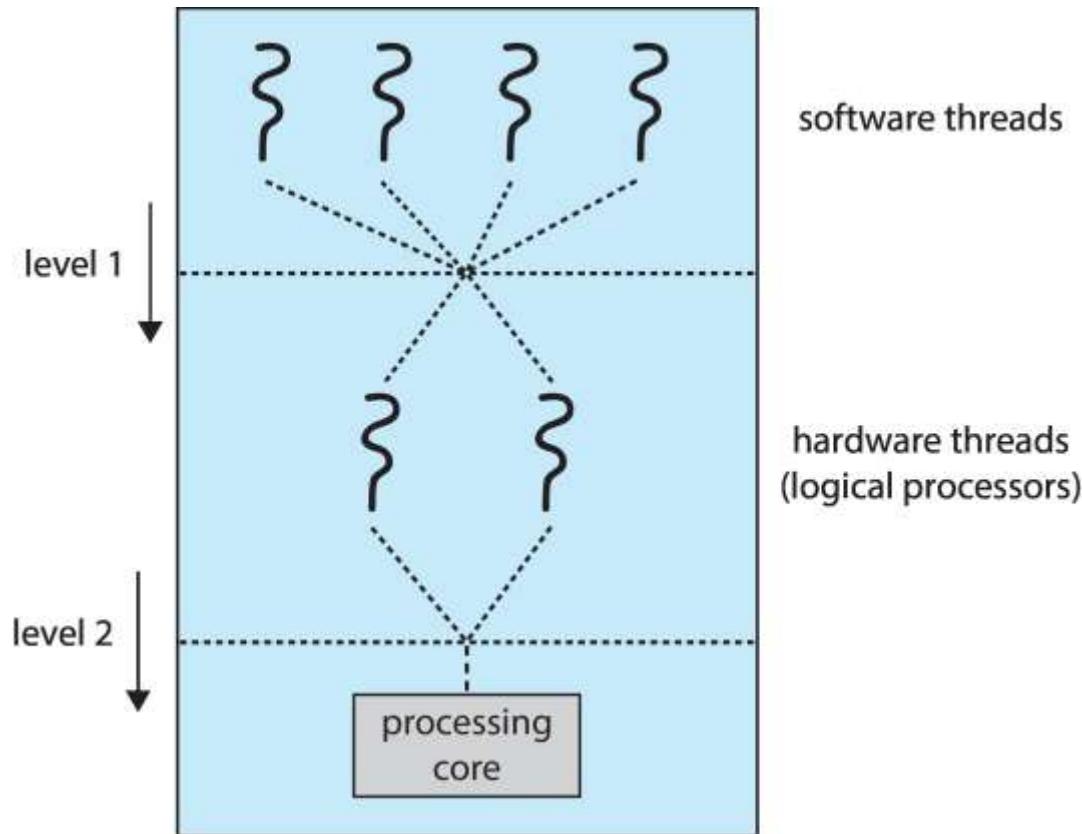
- **Chip-multithreading (CMT)** assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





# Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core.





# Multiple-Processor Scheduling – Load Balancing

---

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





# Multiple-Processor Scheduling – Processor Affinity

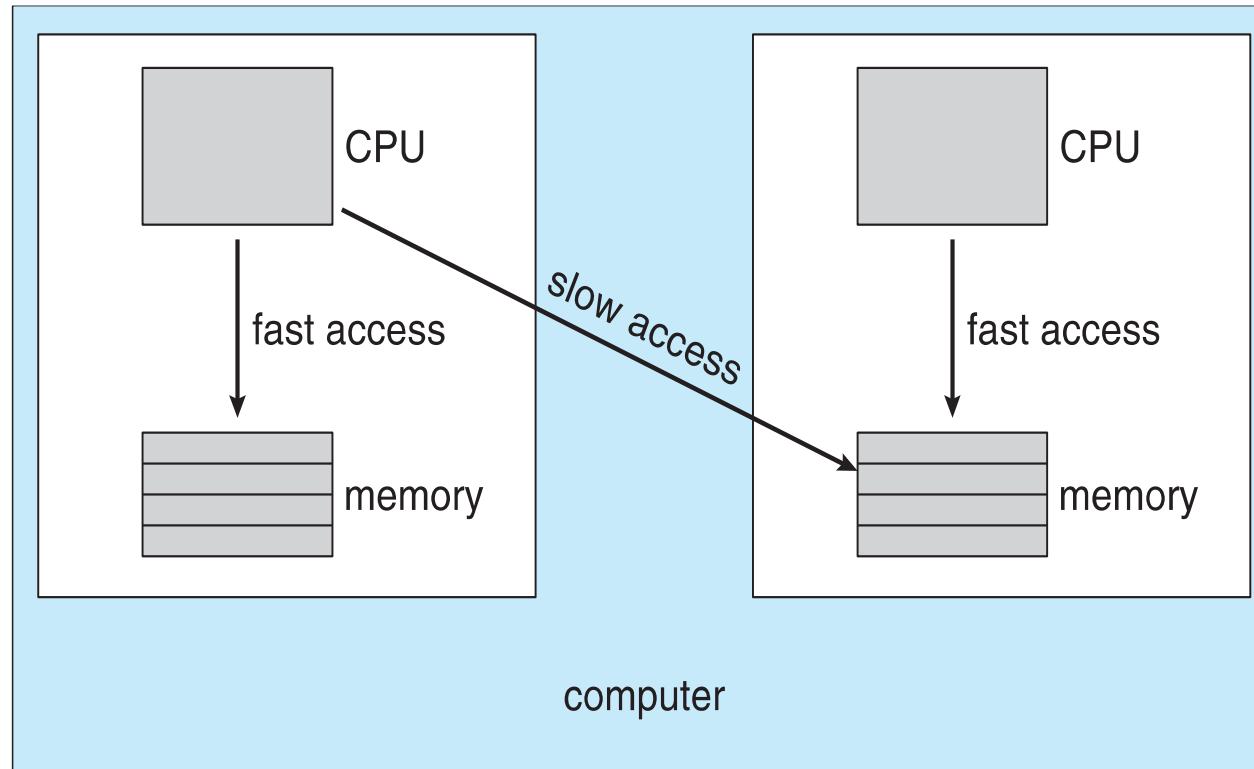
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.





# NUMA and CPU Scheduling

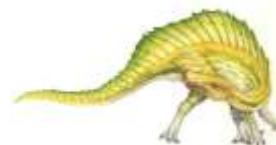
If the operating system is **NUMA-aware**, it will assign memory closer to the CPU the thread is running on.





# Real-Time CPU Scheduling

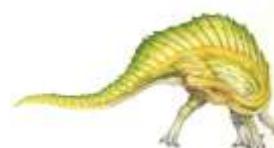
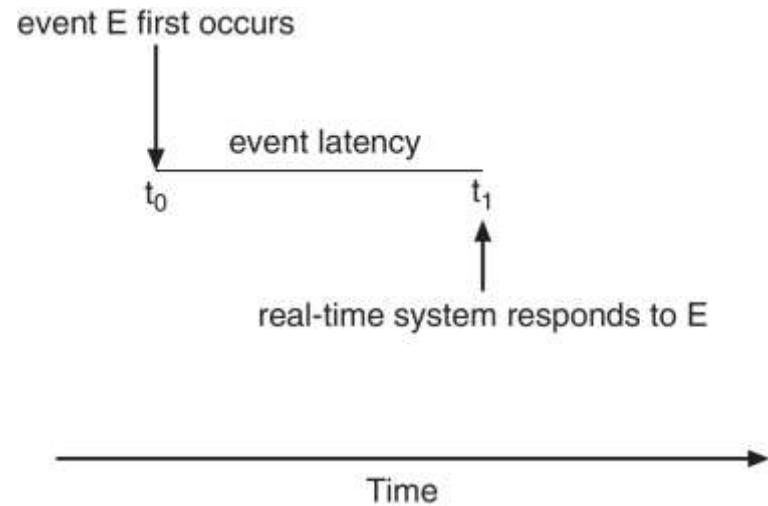
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**





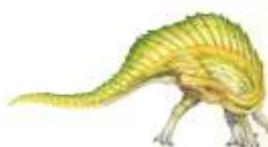
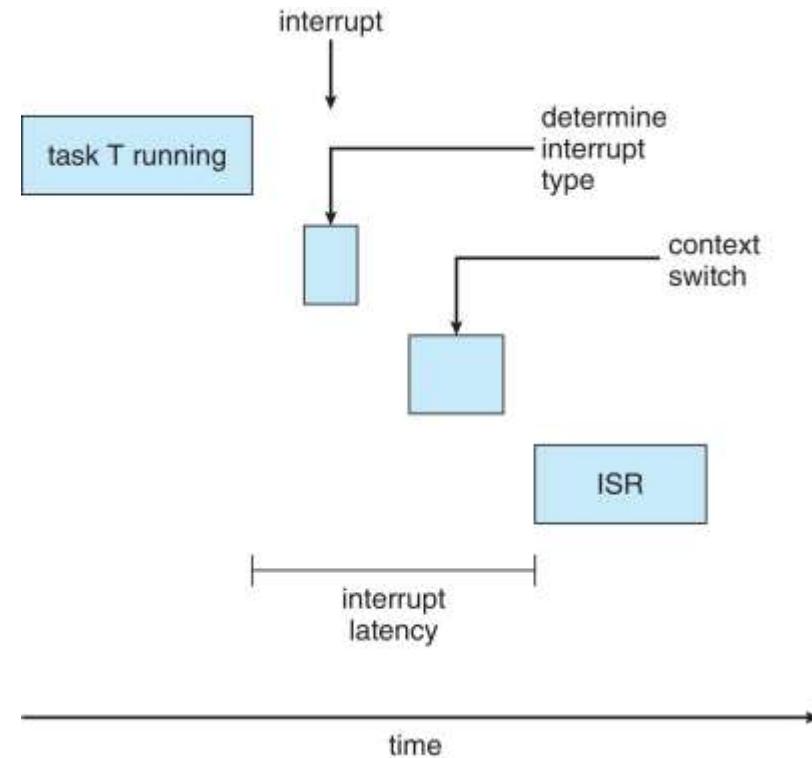
# Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another





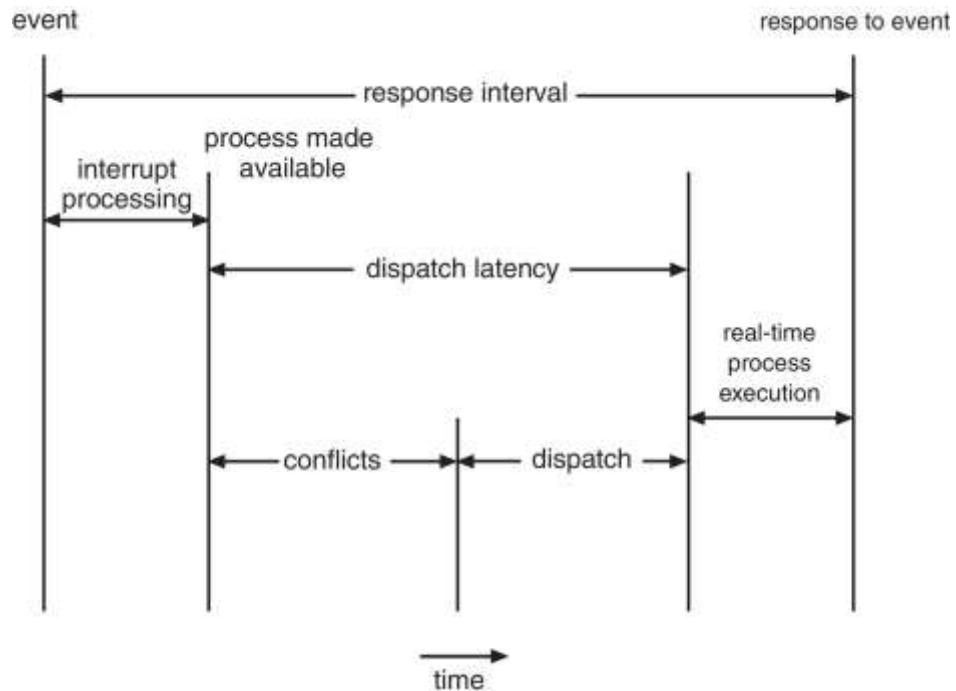
# Interrupt Latency





# Dispatch Latency

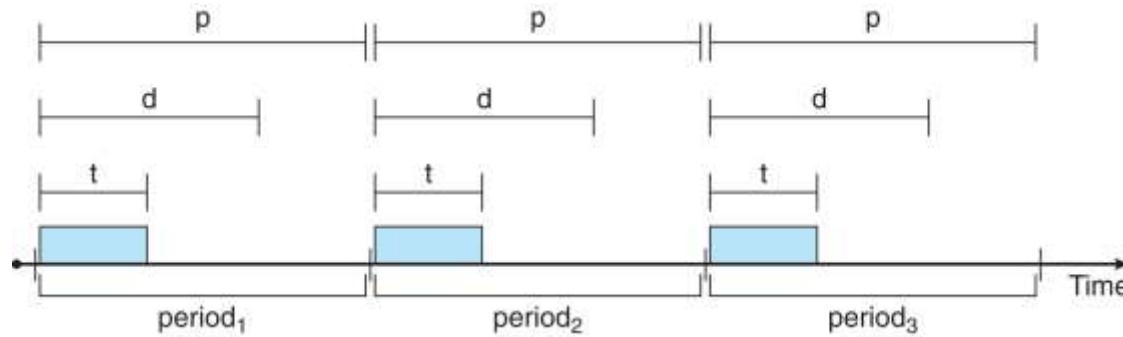
- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes





# Priority-based Scheduling

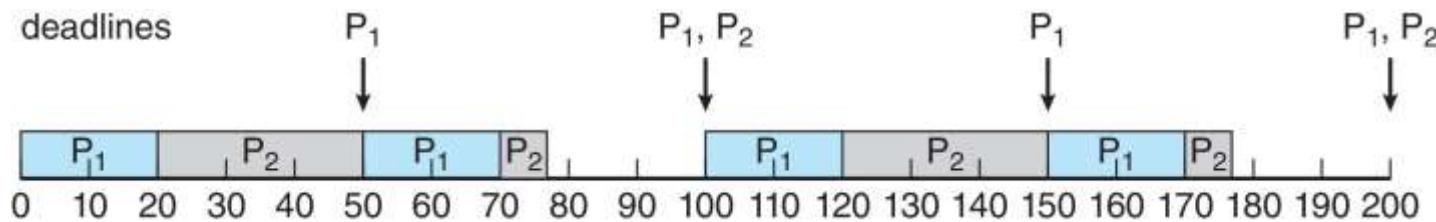
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time  $t$ , deadline  $d$ , period  $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is  $1/p$





# Rate Monotonic Scheduling

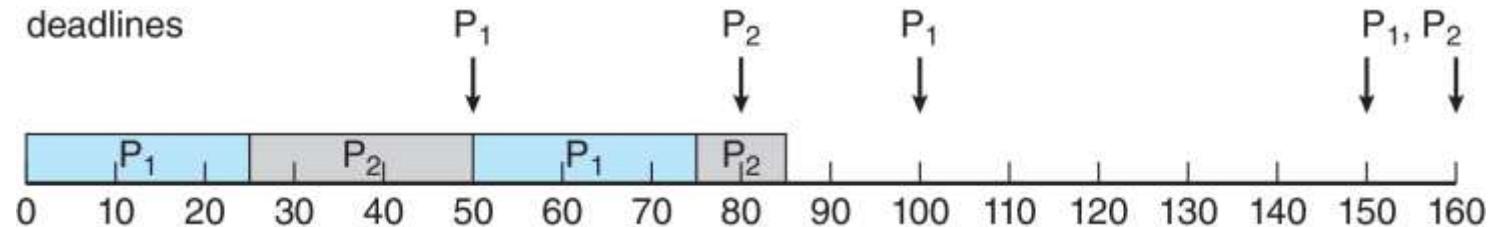
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- $P_1$  is assigned a higher priority than  $P_2$ .





# Missed Deadlines with Rate Monotonic Scheduling

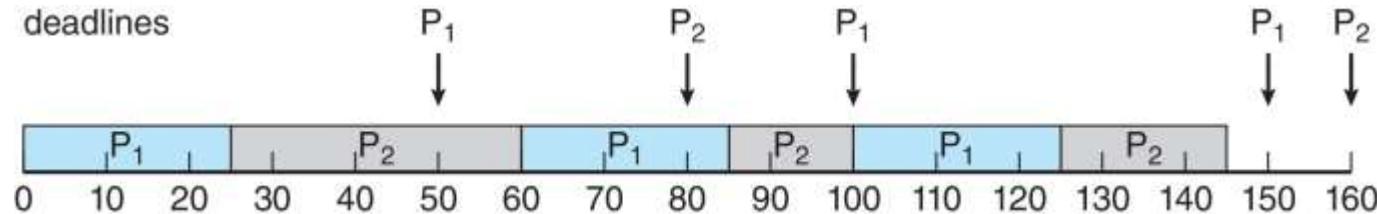
- Process  $P_2$  misses finishing its deadline at time 80
- Figure





# Earliest Deadline First Scheduling (EDF)

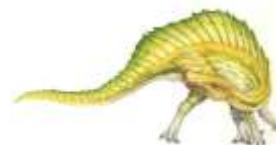
- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Figure





# Proportional Share Scheduling

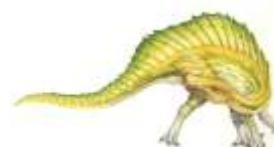
- $T$  shares are allocated among all processes in the system
- An application receives  $N$  shares where  $N < T$
- This ensures each application will receive  $N / T$  of the total processor time





# POSIX Real-Time Scheduling

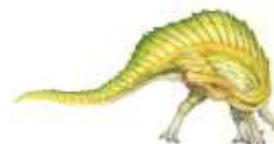
- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED\_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED\_RR - similar to SCHED\_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`





# POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```





# POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

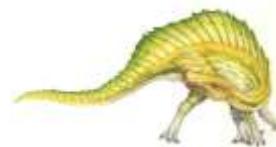




# Operating System Examples

---

- Linux scheduling
- Windows scheduling
- Solaris scheduling





# Linux Scheduling Through Version 2.5

---

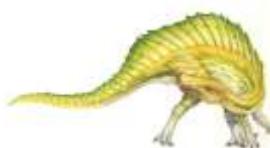
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order  $O(1)$  scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger  $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - ▶ Two priority arrays (active, expired)
    - ▶ Tasks indexed by priority
    - ▶ When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes





# Linux Scheduling in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time





## Linux Scheduling in Version 2.6.23 + (Cont.)

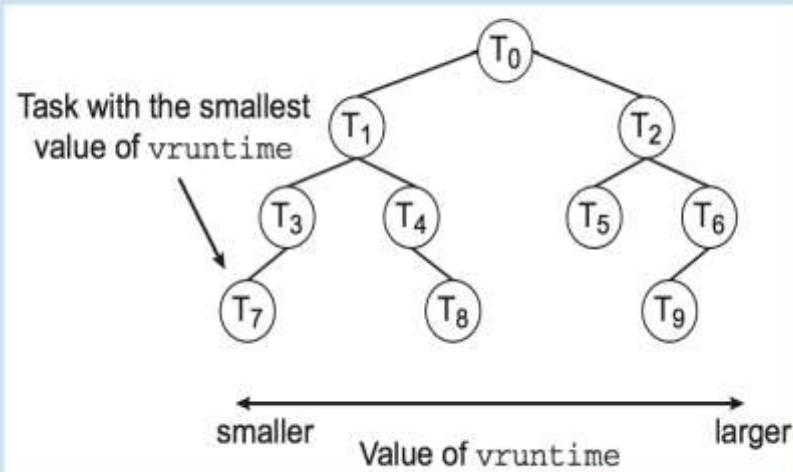
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time



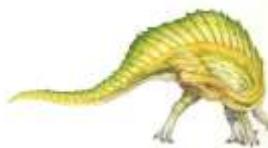


# CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



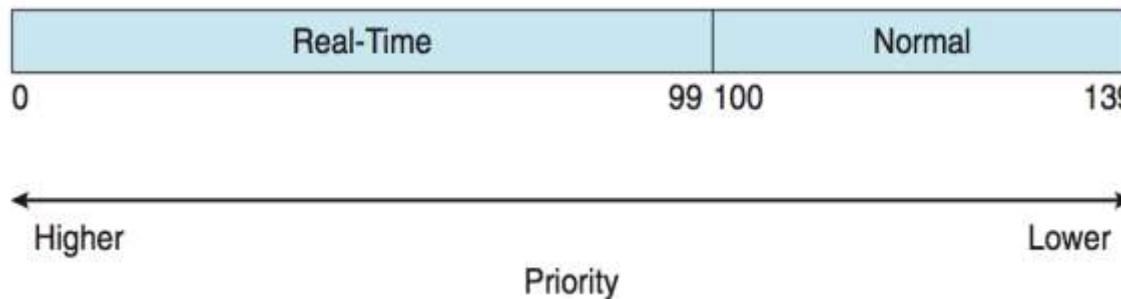
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require  $O(\lg N)$  operations (where  $N$  is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.





# Linux Scheduling (Cont.)

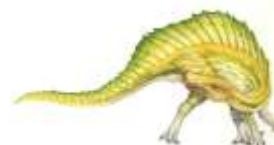
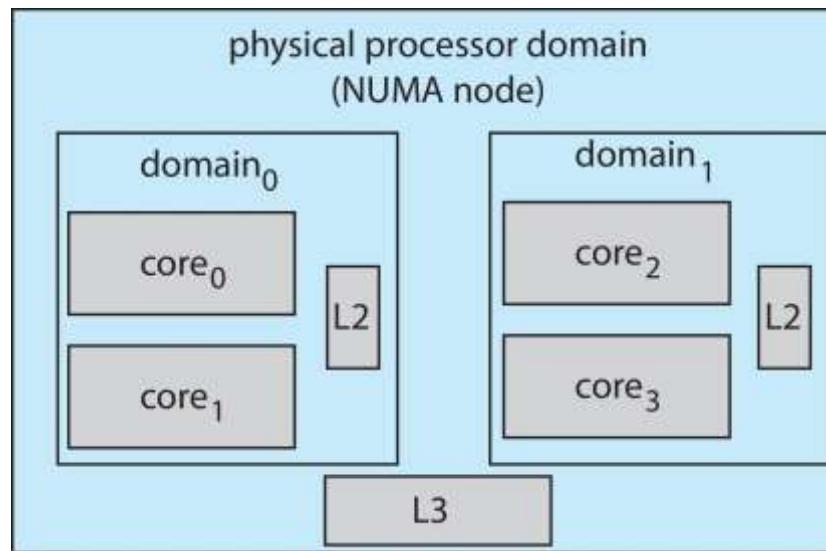
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139





# Linux Scheduling (Cont.)

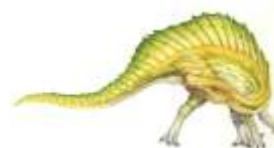
- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.





# Windows Scheduling

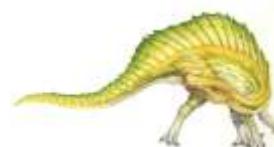
- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**





# Windows Priority Classes

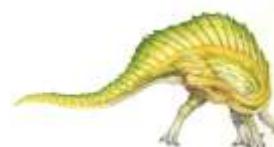
- Win32 API identifies several priority classes to which a process can belong
  - REALTIME\_PRIORITY\_CLASS, HIGH\_PRIORITY\_CLASS, ABOVE\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, IDLE\_PRIORITY\_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME\_CRITICAL, HIGHEST, ABOVE\_NORMAL, NORMAL, BELOW\_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base





# Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework





# Windows Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





# Solaris

---

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

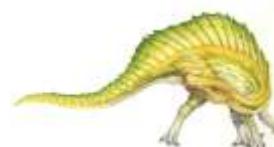




# Solaris Dispatch Table

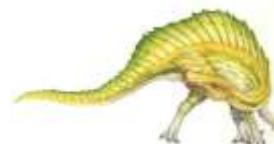
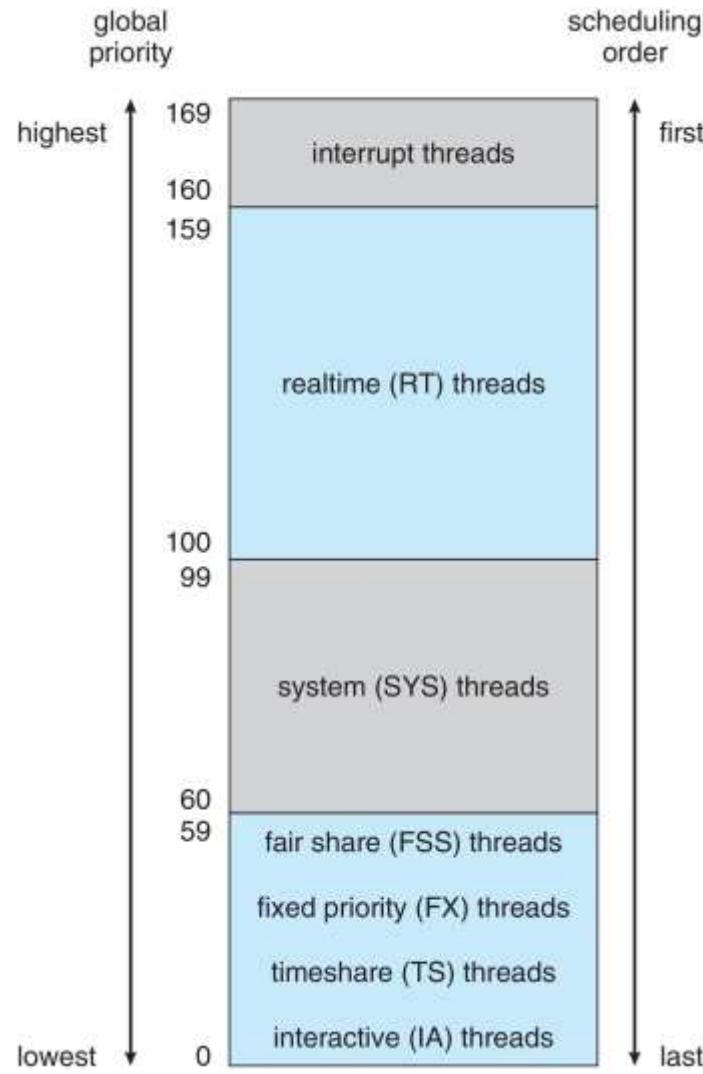
---

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59





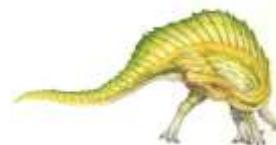
# Solaris Scheduling





# Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR





# Algorithm Evaluation

---

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12



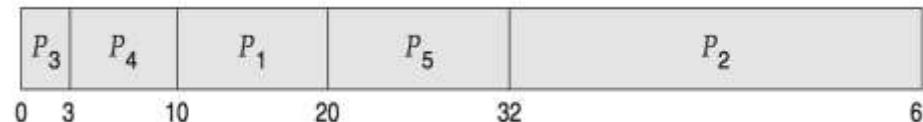


# Deterministic Evaluation

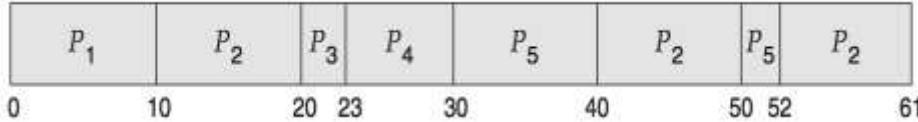
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





# Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.





# Little's Formula

---

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:  
$$n = \lambda \times W$$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds





# Simulations

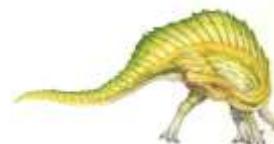
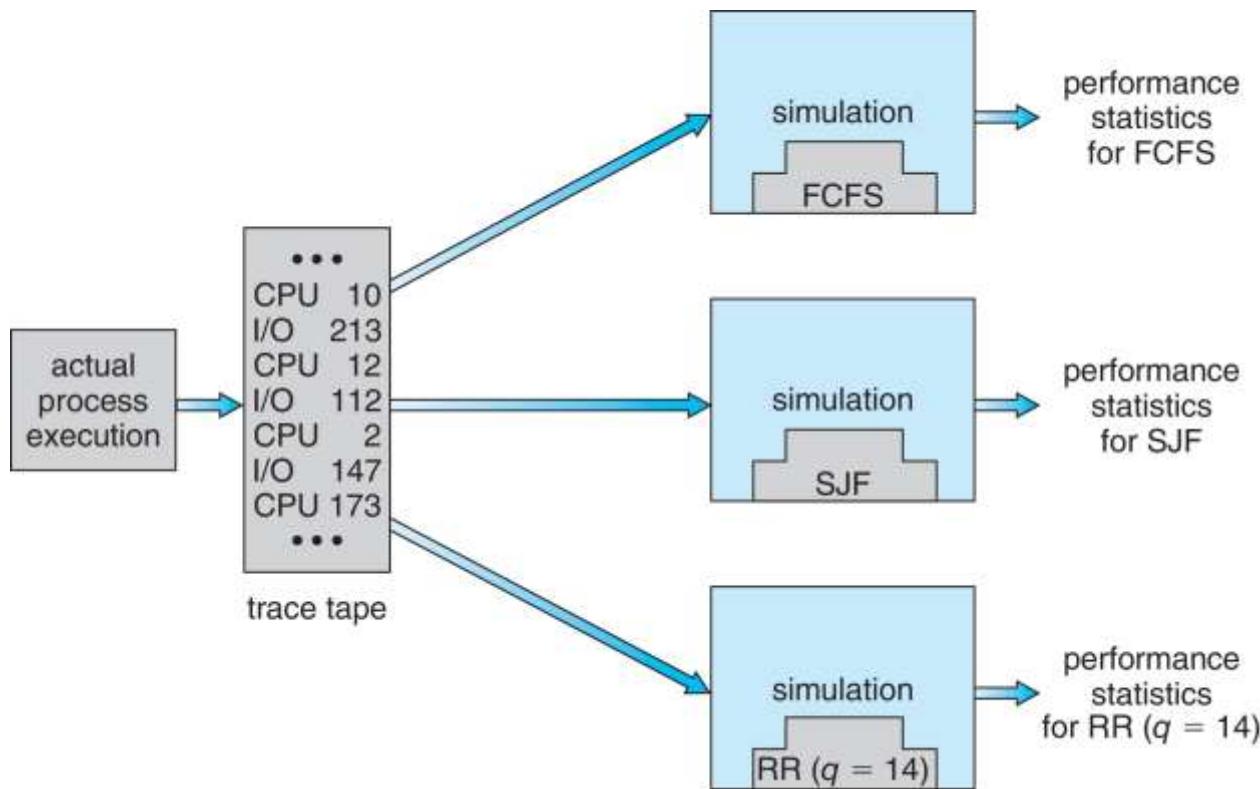
---

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - ▶ Random number generator according to probabilities
    - ▶ Distributions defined mathematically or empirically
    - ▶ Trace tapes record sequences of real events in real systems





# Evaluation of CPU Schedulers by Simulation





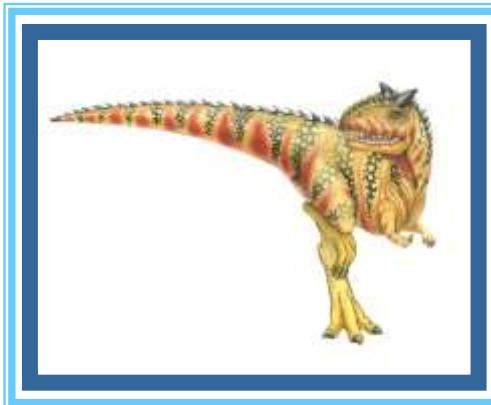
# Implementation

---

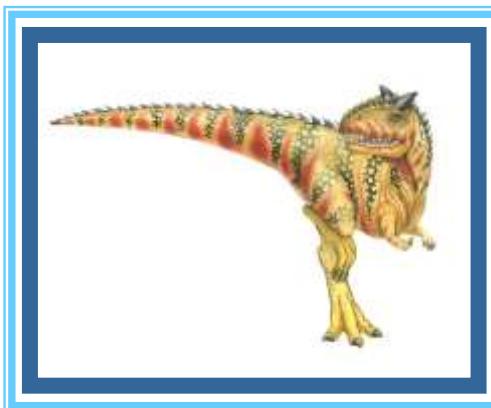
- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



# End of Chapter 5



# Chapter 6: Synchronization Tools

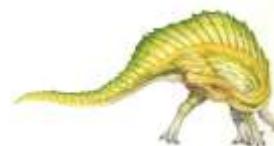




# Outline

---

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation





# Objectives

---

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios





# Background

---

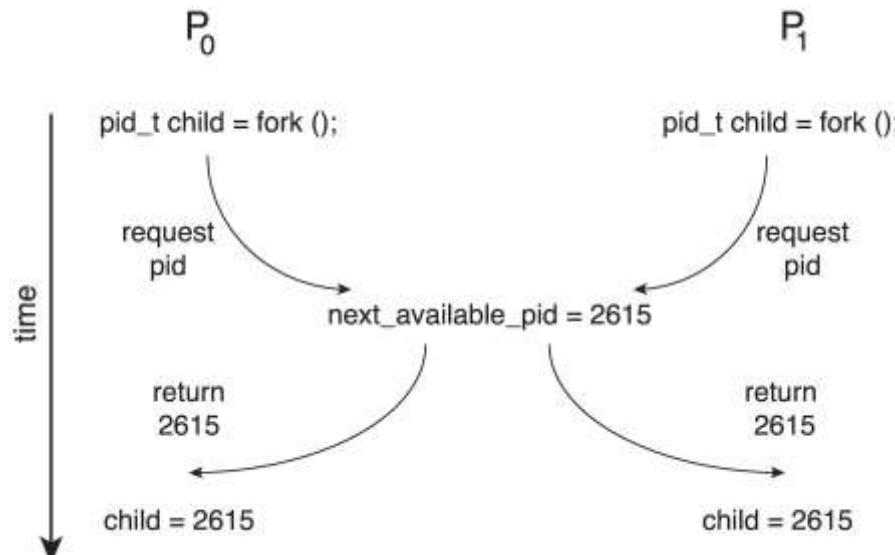
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.



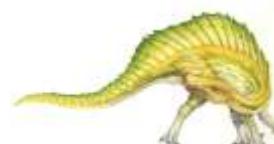


# Race Condition

- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent  $P_0$  and  $P_1$  from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

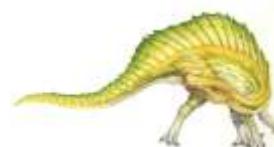




# Critical Section Problem

---

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- 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
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

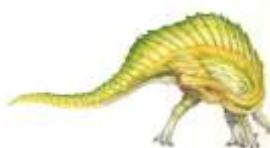




# Critical Section

- General structure of process  $P_i$

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

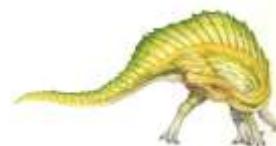




# Critical-Section Problem (Cont.)

Requirements for solution to 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 the selection of the process 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

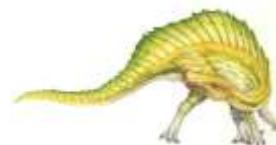




# Interrupt-based Solution

---

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?





# Software Solution 1

---

- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **int turn;**
- The variable **turn** indicates whose turn it is to enter the critical section
- initially, the value of **turn** is set to *i*

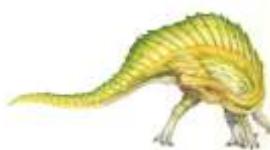




# Algorithm for Process $P_i$

---

```
while (true) {  
  
    while (turn == j);  
  
    /* critical section */  
  
    turn = j;  
  
    /* remainder section */  
  
}
```





# Correctness of the Software Solution

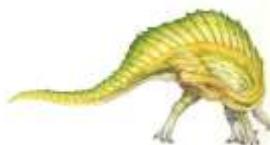
- Mutual exclusion is preserved

$P_i$  enters critical section only if:

**turn = i**

and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?





# 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 $P_i$

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

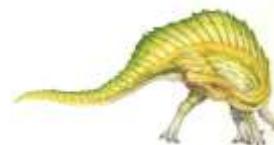




# Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

$P_i$  enters CS only if:  
either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met





# Peterson's Solution and Modern Architecture

---

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!





# Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100





# Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

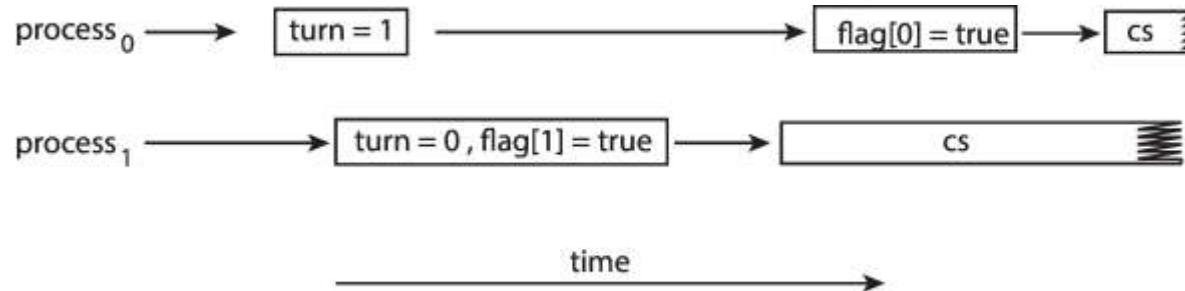
- If this occurs, the output may be 0!





# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



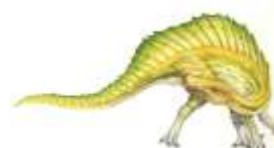
- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.





# Memory Barrier

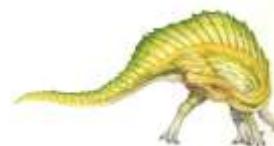
- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.





# Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.





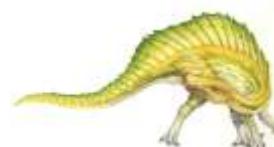
# Memory Barrier Example

---

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1 we are guaranteed that that the value of flag is loaded before the value of x.
- For Thread 2 we ensure that the assignment to x occurs before the assignment flag.





# Synchronization Hardware

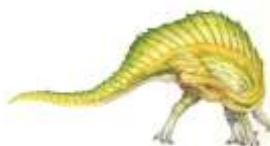
- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - ▶ Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
  1. Hardware instructions
  2. Atomic variables





# Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction





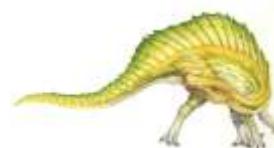
# The test\_and\_set Instruction

- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**





# Solution Using test\_and\_set()

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
    /* remainder section */  
} while (true);
```

- Does it solve the critical-section problem?





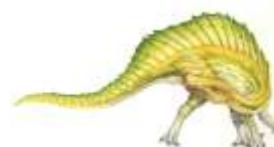
# The compare\_and\_swap Instruction

- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter **value**
- Set the variable **value** the value of the passed parameter **new\_value** but only if **\*value == expected** is true. That is, the swap takes place only under this condition.





# Solution using compare\_and\_swap

- Shared integer `lock` initialized to 0;
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

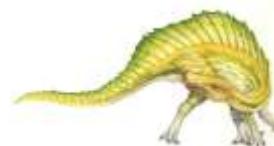




# Bounded-waiting with compare-and-swap

---

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```





# Atomic Variables

---

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **sequence** be an atomic variable
  - Let **increment ()** be operation on the atomic variable **sequence**
  - The Command:  
**increment (&sequence) ;**  
ensures **sequence** is incremented without interruption:





# Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment	atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v, temp, temp+1)));
}
```





# Mutex Locks

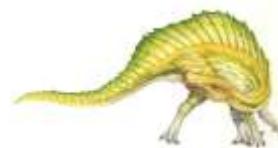
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- Calls to **acquire()** and **release()** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**





# Solution to CS Problem Using Mutex Locks

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





# Semaphore

- 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()**

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

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

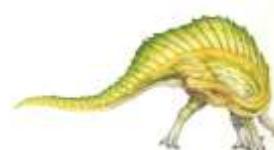
```
signal(S) {  
    S++;  
}
```





# Semaphore (Cont.)

- **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 implement a counting semaphore **S** as a binary semaphore
- With semaphores we can solve various synchronization problems





# Semaphore Usage Example

---

- Solution to the CS Problem

- Create a semaphore “**mutex**” initialized to 1

```
    wait(mutex);
```

CS

```
    signal(mutex);
```

- Consider  $P_1$  and  $P_2$  that with two statements  $S_1$ , and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$

- Create a semaphore “**synch**” initialized to 0

P1:

```
    S1;
```

```
    signal(synch);
```

P2:

```
    wait(synch);
```

```
    S2;
```





# Semaphore Implementation

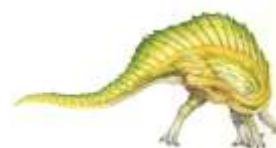
- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution





# Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

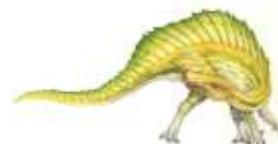




# Implementation with no Busy waiting (Cont.)

- Waiting queue

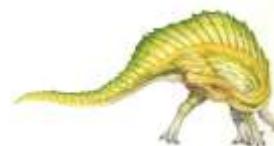
```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```





# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```





# Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) . . . wait(mutex)`
  - `wait(mutex) ... wait(mutex)`
  - Omitting of `wait (mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.





# 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
- Pseudocode syntax of a monitor:

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

    procedure P2 (...) { .... }

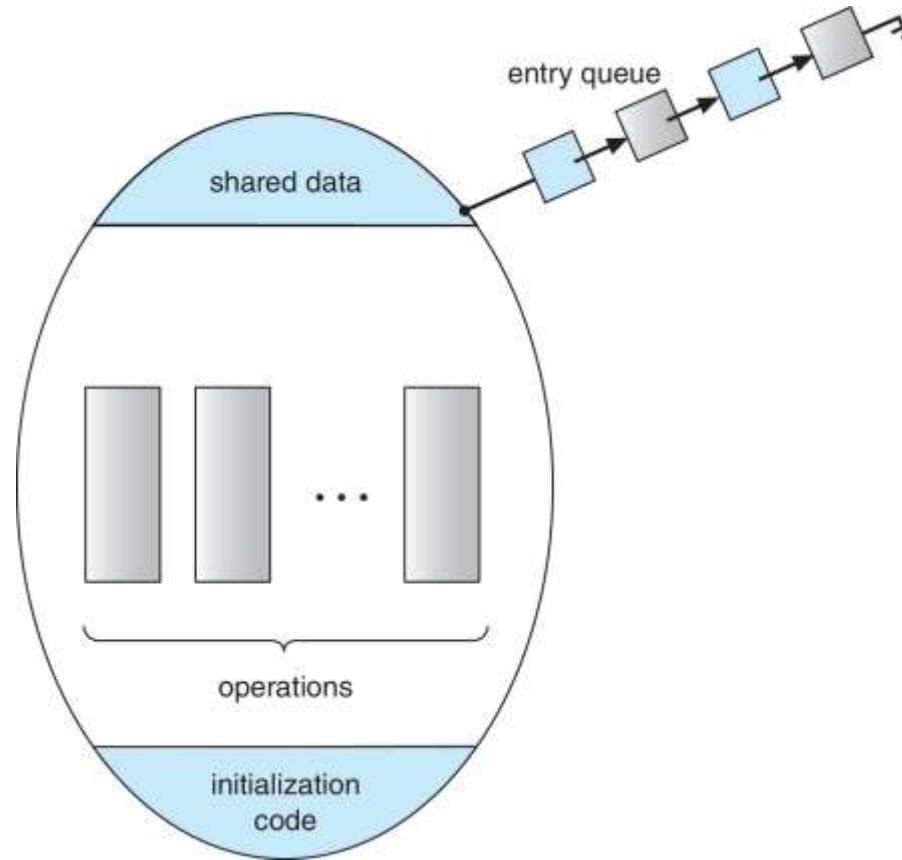
    procedure Pn (...) { ..... }

    initialization code (...) { ... }
}
```





# Schematic view of a Monitor





# Monitor Implementation Using Semaphores

---

- Variables

```
semaphore mutex  
mutex = 1
```

- Each procedure  $P$  is replaced by

```
wait (mutex) ;  
...  
body of P;  
...  
signal (mutex) ;
```

- Mutual exclusion within a monitor is ensured





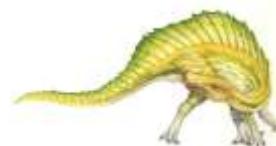
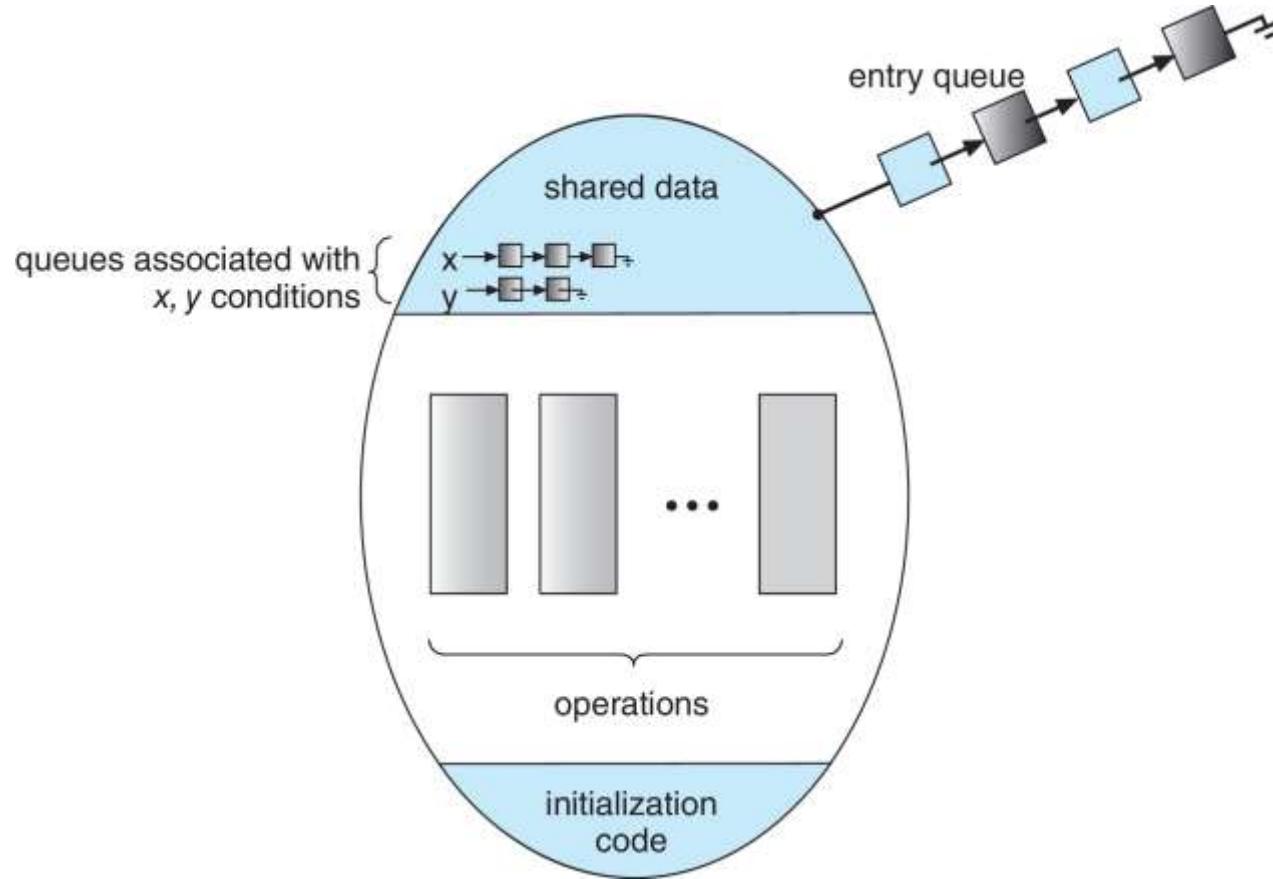
# Condition Variables

- `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





# Usage of Condition Variable Example

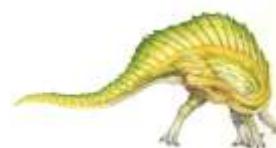
- Consider  $P_1$  and  $P_2$  that need to execute two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$

- Create a monitor with two procedures  $F_1$  and  $F_2$  that are invoked by  $P_1$  and  $P_2$  respectively
  - One condition variable “x” initialized to 0
  - One Boolean variable “done”
  - F1:**

```
s1;  
done = true;  
x.signal();
```

- F2:**

```
if done = false  
    x.wait()  
  
s2;
```





# Monitor Implementation Using Semaphores

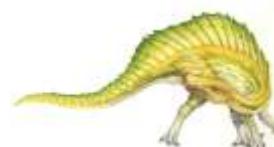
- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0; // number of processes waiting
                     inside the monitor
```

- Each function  $P$  will be replaced by

```
wait(mutex);
...
body of P;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured





# Implementation – Condition Variables

---

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation **x.wait()** can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```





# Implementation (Cont.)

- The operation `x.signal()` can be implemented as:

```
if (x_count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```





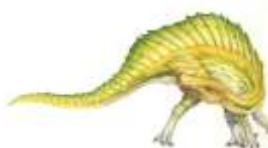
# Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

**x.wait(c)**

where:

- **c** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next





# Single Resource allocation

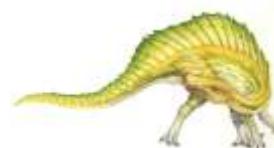
- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource

```
R.acquire(t);  
...  
access the resource;
```

...

```
R.release;
```

- Where R is an instance of type **ResourceAllocator**





# Single Resource allocation

---

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type **ResourceAllocator** (next slide)
- Access to **ResourceAllocator** is done via:

```
R.acquire(t);  
...  
access the resource;  
...  
R.release;
```

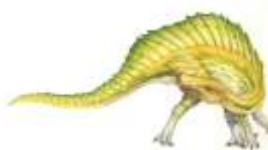
- Where t is the maximum time a process plans to use the resource





# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

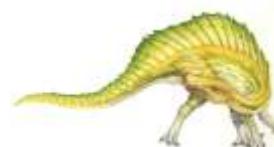




# Single Resource Monitor (Cont.)

- Usage:  
**acquire**  
  
...  
**release**

- Incorrect use of monitor operations
  - **release()** ... **acquire()**
  - **acquire()** ... **acquire()**)
  - Omitting of **acquire()** and/or **release()**





# Liveness

---

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.



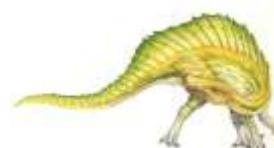


# Liveness

- **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

$P_0$	$P_1$
<code>wait(S) ;</code>	<code>wait(Q) ;</code>
<code>wait(Q) ;</code>	<code>wait(S) ;</code>
...	...
<code>signal(S) ;</code>	<code>signal(Q) ;</code>
<code>signal(Q) ;</code>	<code>signal(S) ;</code>

- Consider if  $P_0$  executes `wait(S)` and  $P_1$  `wait(Q)`. When  $P_0$  executes `wait(Q)`, it must wait until  $P_1$  executes `signal(Q)`
- However,  $P_1$  is waiting until  $P_0$  execute `signal(S)`.
- Since these `signal()` operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.





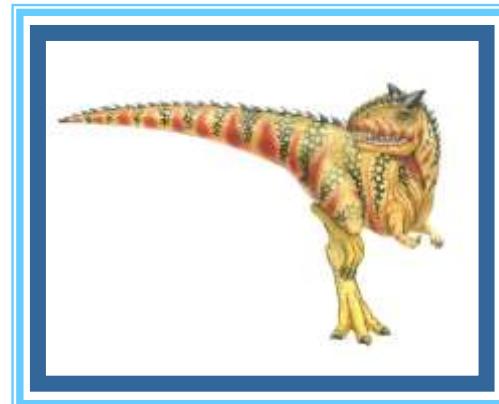
# Liveness

---

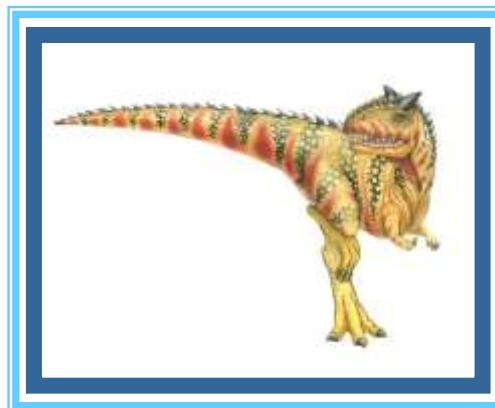
- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance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



# End of Chapter 6



# Chapter 7: Synchronization Examples

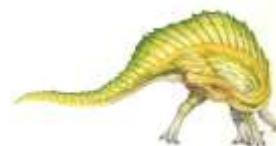




# Outline

---

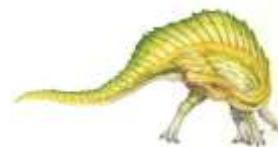
- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers synchronization problems
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems





# Classical Problems of Synchronization

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

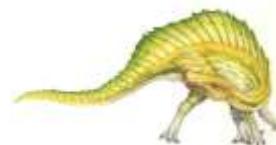




# Bounded-Buffer Problem

---

- $n$  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

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

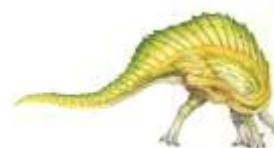




# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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





# Readers-Writers Problem

---

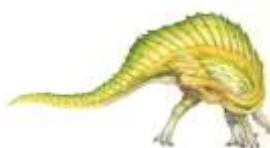
- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do *not* perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the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





# Readers-Writers Problem (Cont.)

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

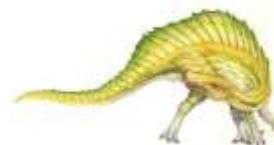




# Readers-Writers Problem (Cont.)

- The structure of a writer process

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





# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
    wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);
    signal(mutex);

    ...
/* reading is performed */

    ...

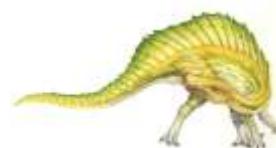
    wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
        signal(rw_mutex);
    signal(mutex);
}
```





# Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation, leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks



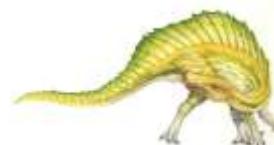


# Dining-Philosophers Problem

- N philosophers sit at a round table with a bowl of rice in the middle.



- They spend their lives alternating thinking and eating.
- They do not 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, the shared data
  - ▶ Bowl of rice (data set)
  - ▶ Semaphore chopstick [5] initialized to 1





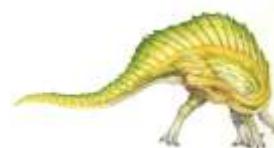
# Dining-Philosophers Problem Algorithm

---

- Semaphore Solution
- The structure of Philosopher  $i$  :

```
while (true) {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
    /* eat for awhile */  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    /* think for awhile */  
  
}
```

- What is the problem with this algorithm?





# 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;
    }

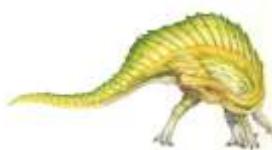
    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```





# 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.)

---

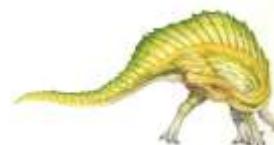
- Each philosopher “i” invokes the operations **pickup ()** and **putdown ()** in the following sequence:

```
DiningPhilosophers.pickup(i) ;
```

```
    /** EAT **/
```

```
DiningPhilosophers.putdown(i) ;
```

- No deadlock, but starvation is possible





# Kernel Synchronization - Windows

---

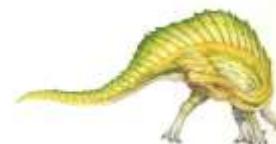
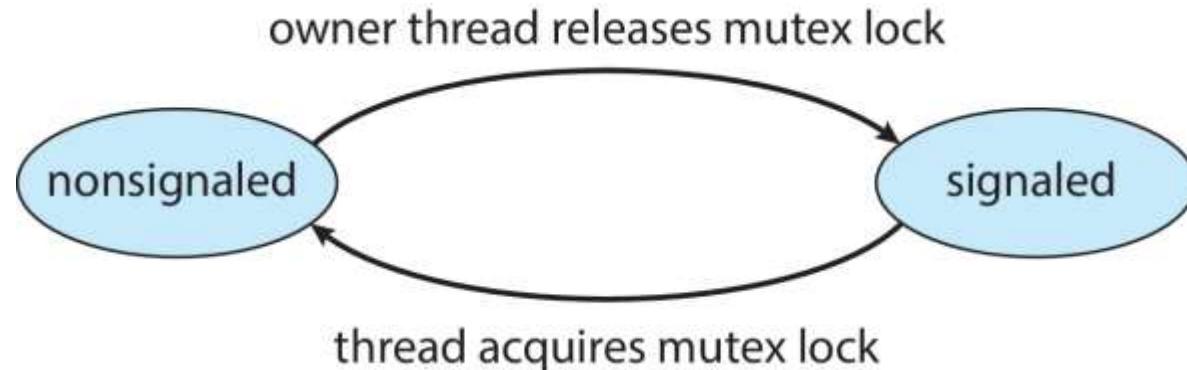
- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - ▶ An event acts much like a condition variable
    - Timers notify one or more thread when time expired
    - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)





# Kernel Synchronization - Windows

- Mutex dispatcher object





# Linux Synchronization

---

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - Atomic integers
  - Spinlocks
  - Reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption





# Linux Synchronization

- Atomic variables

`atomic_t` is the type for atomic integer

- Consider the variables

```
atomic_t counter;  
int value;
```

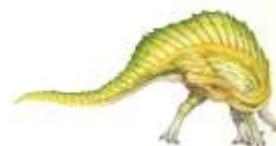
<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&amp;counter,5);</code>	<code>counter = 5</code>
<code>atomic_add(10,&amp;counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4,&amp;counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&amp;counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&amp;counter);</code>	<code>value = 12</code>





# POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS





# POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

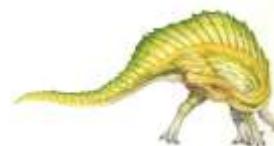
/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

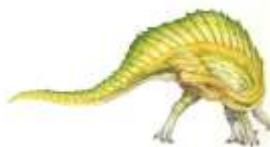




# POSIX Semaphores

---

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.





# POSIX Named Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

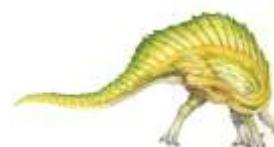
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```





# POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

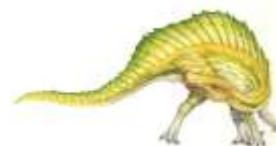




# POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex,NULL);  
pthread_cond_init(&cond_var,NULL);
```





# POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

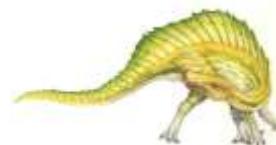




# Java Synchronization

---

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

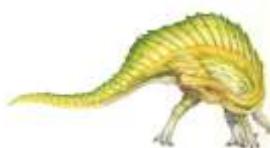




# Java Monitors

---

- Every Java object has associated with it a single lock.
- If a method is declared as **synchronized**, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the **synchronized** method.





# Bounded Buffer – Java Synchronization

---

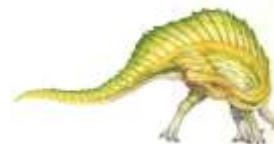
```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

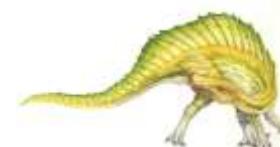
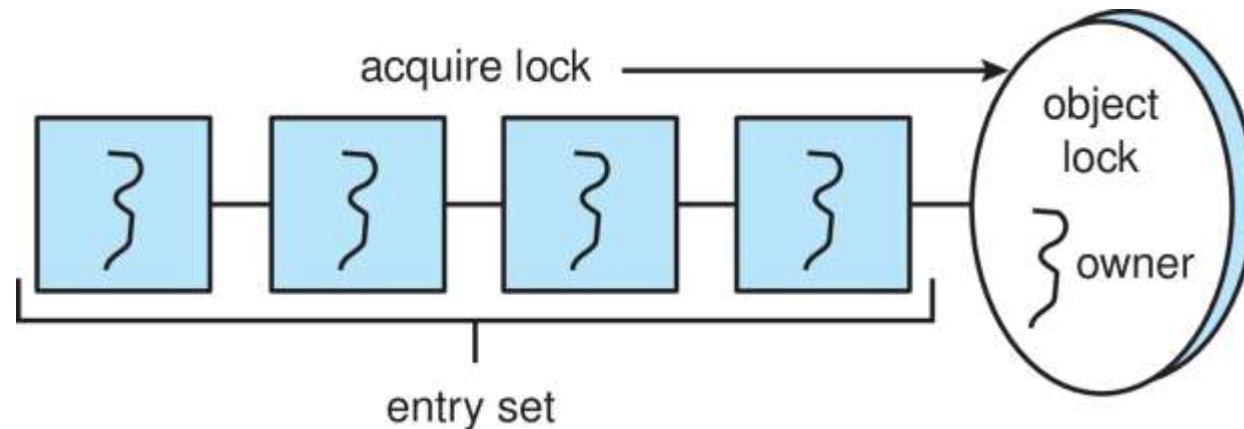
    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```





# Java Synchronization

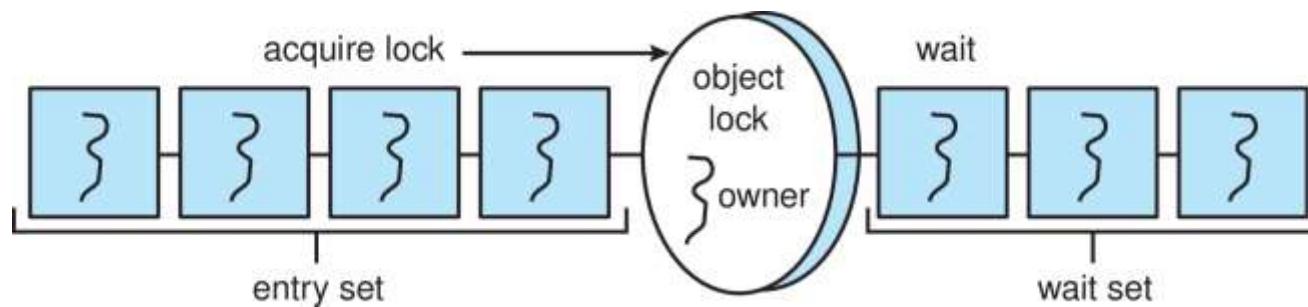
- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:





# Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls **wait()**:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object





# Java Synchronization

- A thread typically calls `wait()` when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls `notify()`:
  1. An arbitrary thread T is selected from the wait set
  2. T is moved from the wait set to the entry set
  3. Set the state of T from blocked to runnable.
- T can now compete for the lock to check if the condition it was waiting for is now true.





# Bounded Buffer – Java Synchronization

---

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```





# Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    notify();

    return item;
}
```





# Java Reentrant Locks

- Similar to mutex locks
- The **finally** clause ensures the lock will be released in case an exception occurs in the **try** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```





# Java Semaphores

---

- Constructor:

```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```





# Java Condition Variables

- Condition variables are associated with an `ReentrantLock`.
- Creating a condition variable using `newCondition()` method of `ReentrantLock`:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the `await()` method, and signals by calling the `signal()` method.





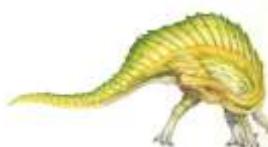
# Java Condition Variables

---

- Example:
- Five threads numbered 0 .. 4
- Shared variable **turn** indicating which thread's turn it is.
- Thread calls **dowork()** when it wishes to do some work. (But it may only do work if it is their turn.)
- If not their turn, wait
- If their turn, do some work for awhile .....
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```





# Java Condition Variables

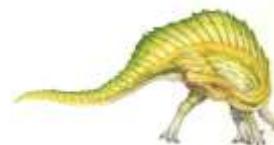
---

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```

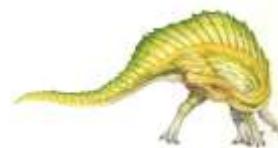




# Alternative Approaches

---

- Transactional Memory
- OpenMP
- Functional Programming Languages





# Transactional Memory

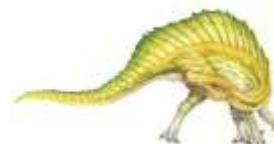
---

- Consider a function update() that must be called atomically. One option is to use mutex locks:

```
void update ()  
{  
    acquire();  
  
    /* modify shared data */  
  
    release();  
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **atomic{S}** which ensure statements in S are executed atomically:

```
void update ()  
{  
    atomic {  
        /* modify shared data */  
    }  
}
```



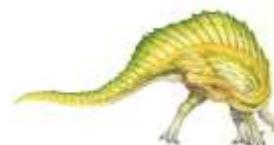


# OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

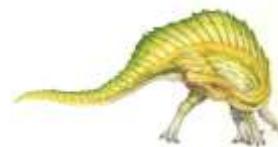




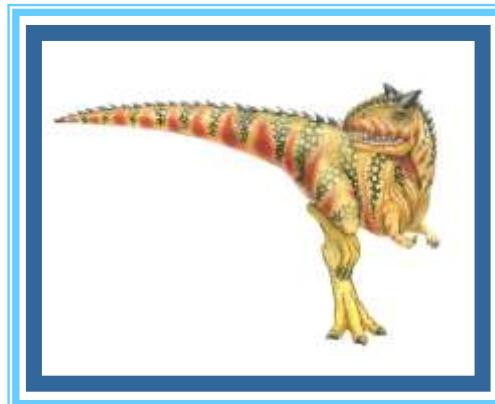
# Functional Programming Languages

---

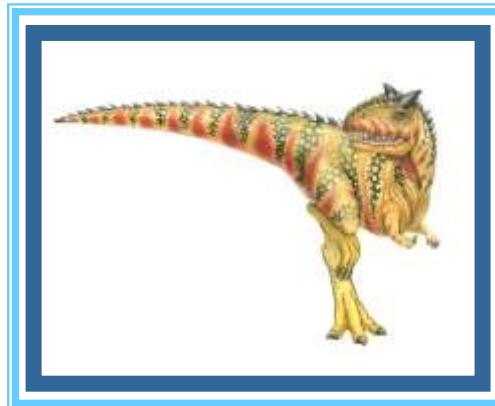
- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.



# End of Chapter 7



# Chapter 9: Main Memory

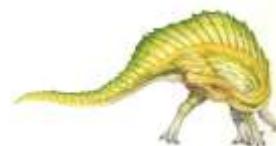




# Chapter 9: Memory Management

---

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture





# Objectives

---

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging





# Background

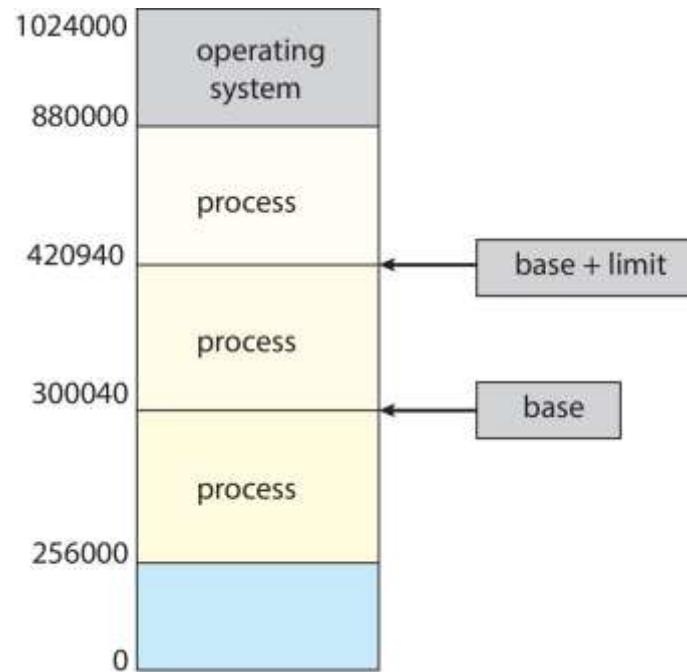
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation





# Protection

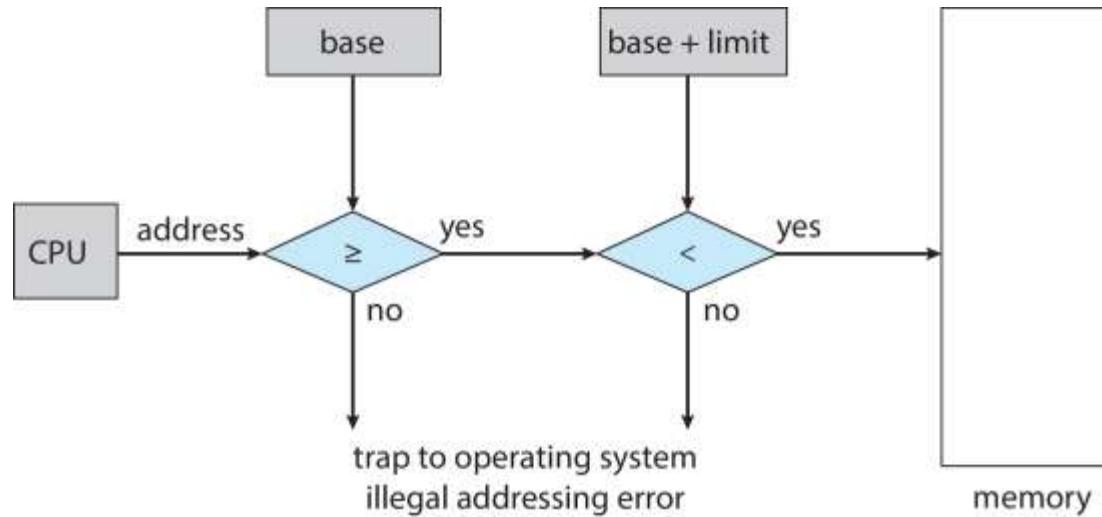
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



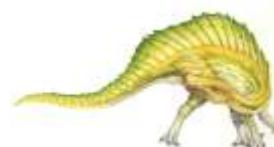


# Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



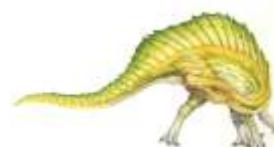
- the instructions to loading the base and limit registers are privileged





# Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - ▶ i.e., “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - ▶ i.e., 74014
  - Each binding maps one address space to another





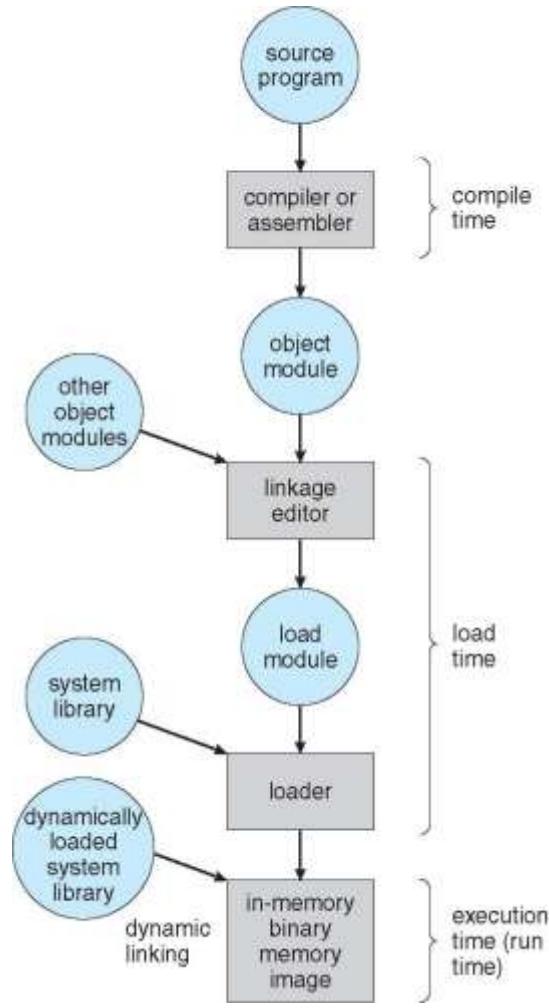
# Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - ▶ Need hardware support for address maps (e.g., base and limit registers)





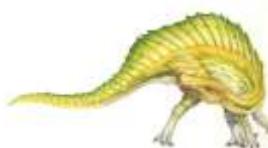
# Multistep Processing of a User Program





# Logical vs. Physical Address Space

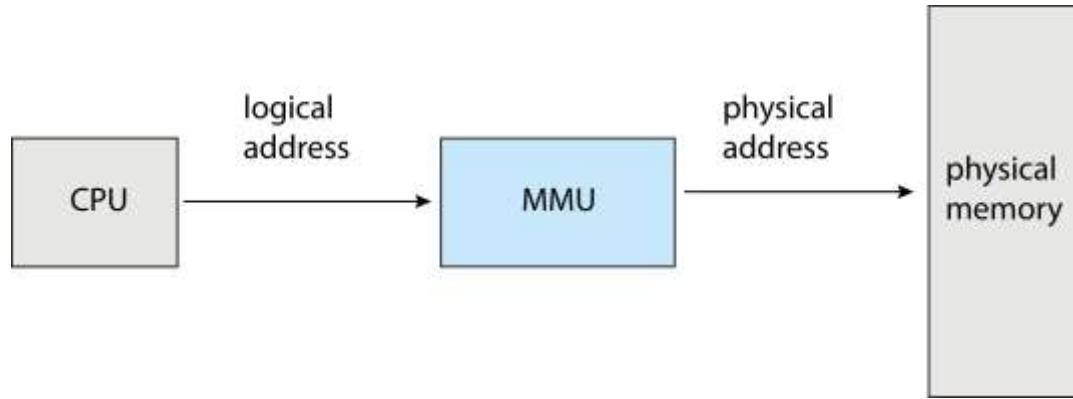
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program





# Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



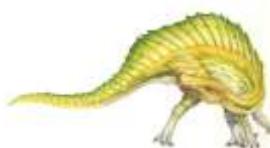
- Many methods possible, covered in the rest of this chapter





# Memory-Management Unit (Cont.)

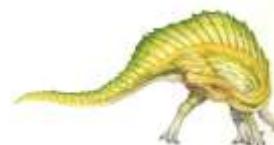
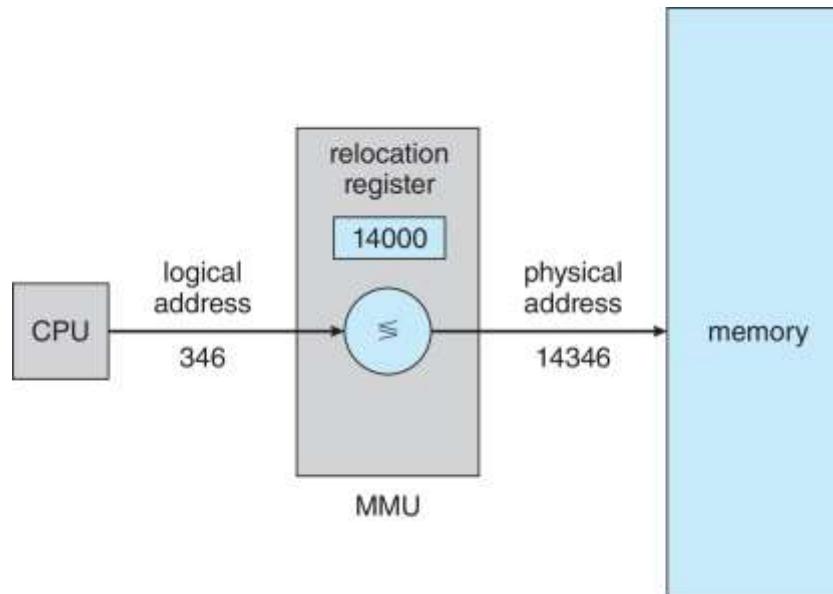
- Consider simple scheme, which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses





# Memory-Management Unit (Cont.)

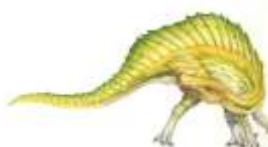
- Consider simple scheme, which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory





# Dynamic Loading

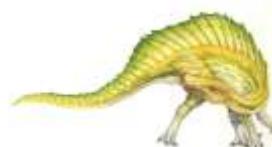
- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading





# Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

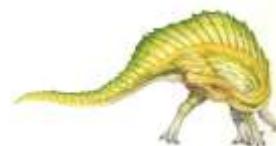




# Contiguous Allocation

---

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory





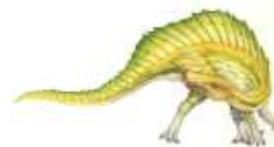
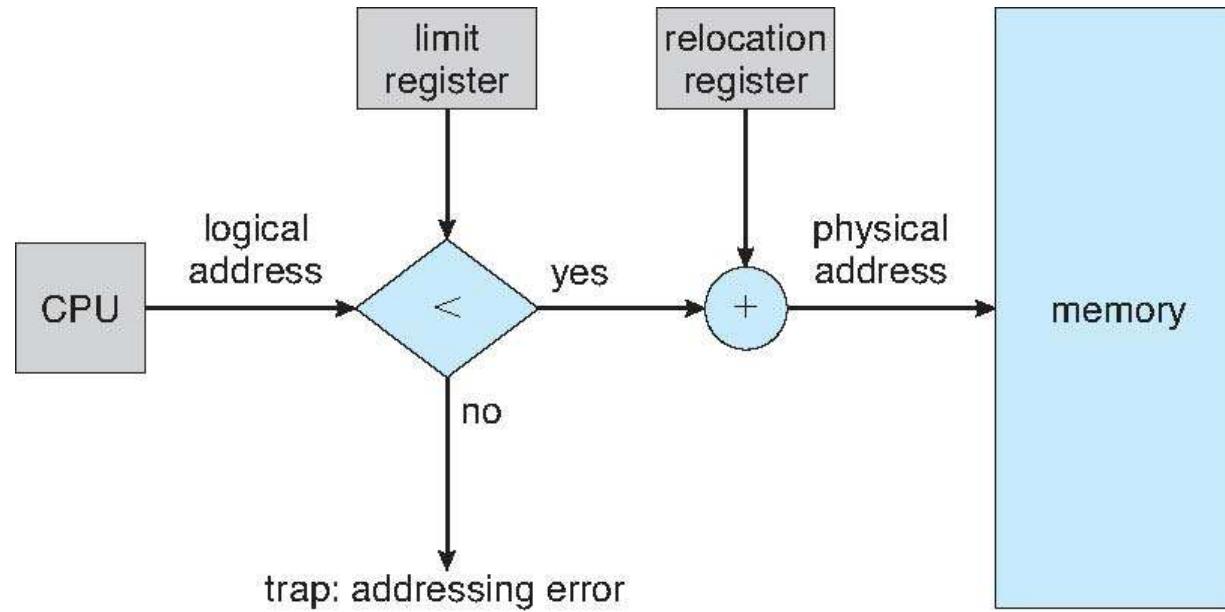
# Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size





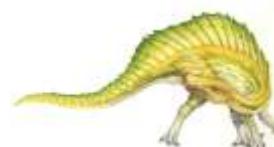
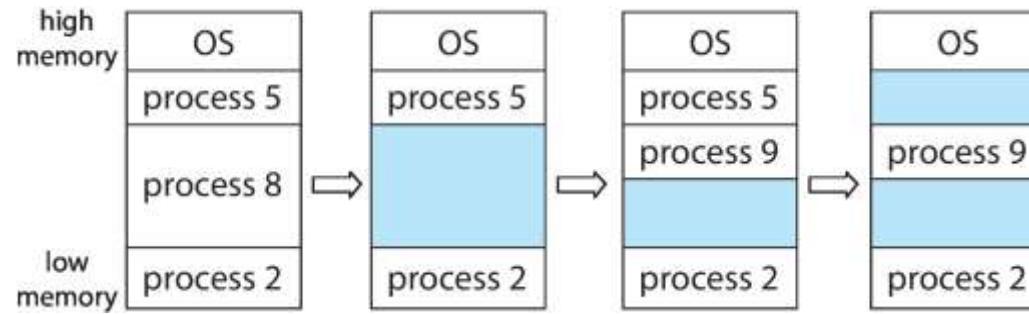
# Hardware Support for Relocation and Limit Registers





# Variable Partition

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    - a) allocated partitions
    - b) free partitions (hole)



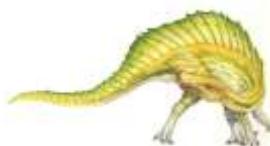


# Dynamic Storage-Allocation Problem

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





# Fragmentation

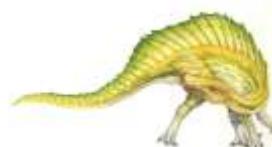
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals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





# Fragmentation (Cont.)

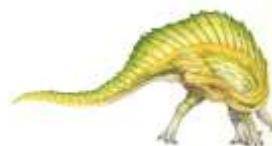
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - ▶ Latch job in memory while it is involved in I/O
    - ▶ Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems





# Paging

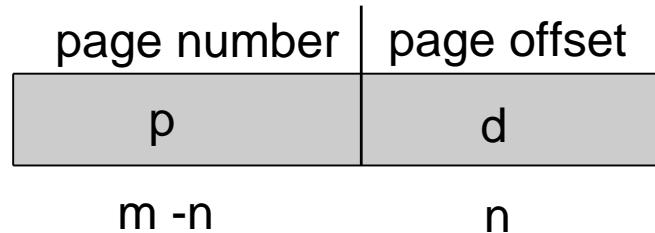
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size  **$N$**  pages, need to find  **$N$**  free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation



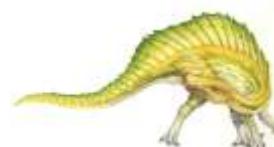


# Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number ( $p$ )** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset ( $d$ )** – combined with base address to define the physical memory address that is sent to the memory unit

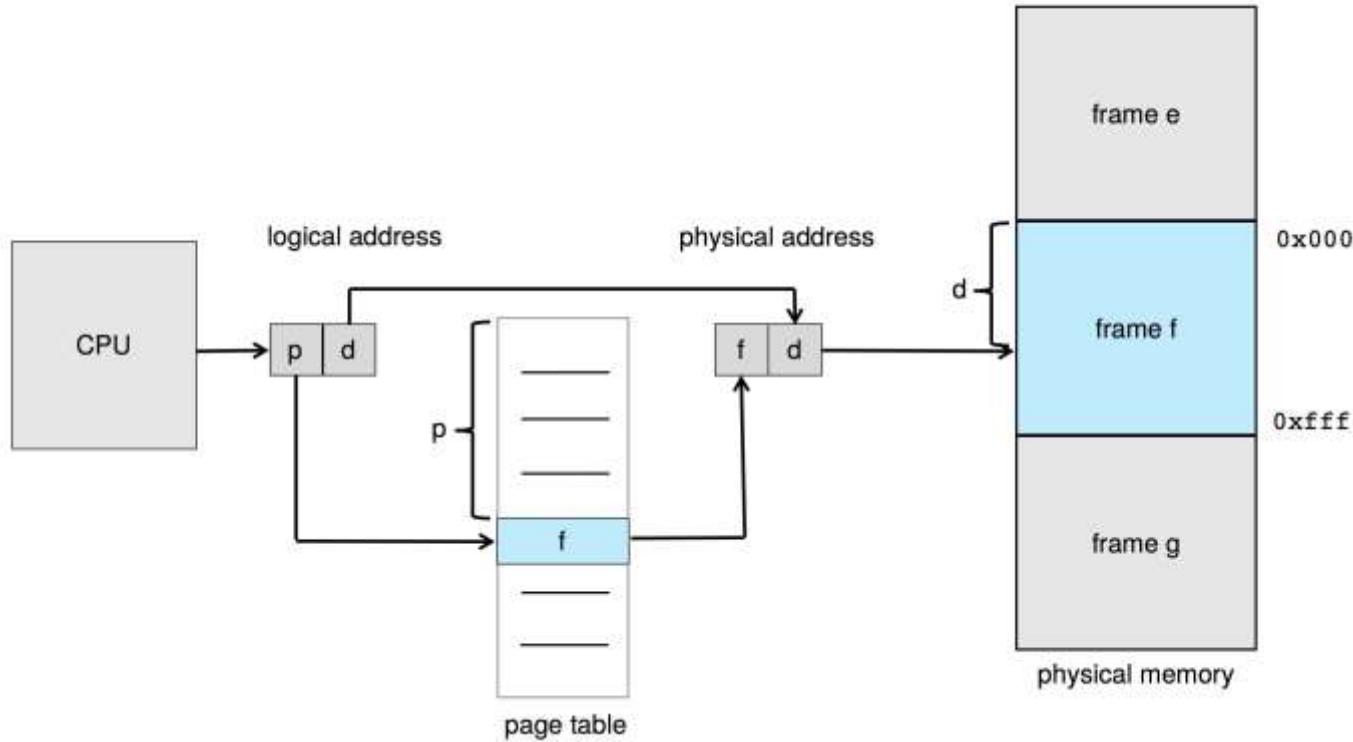


- For given logical address space  $2^m$  and page size  $2^n$



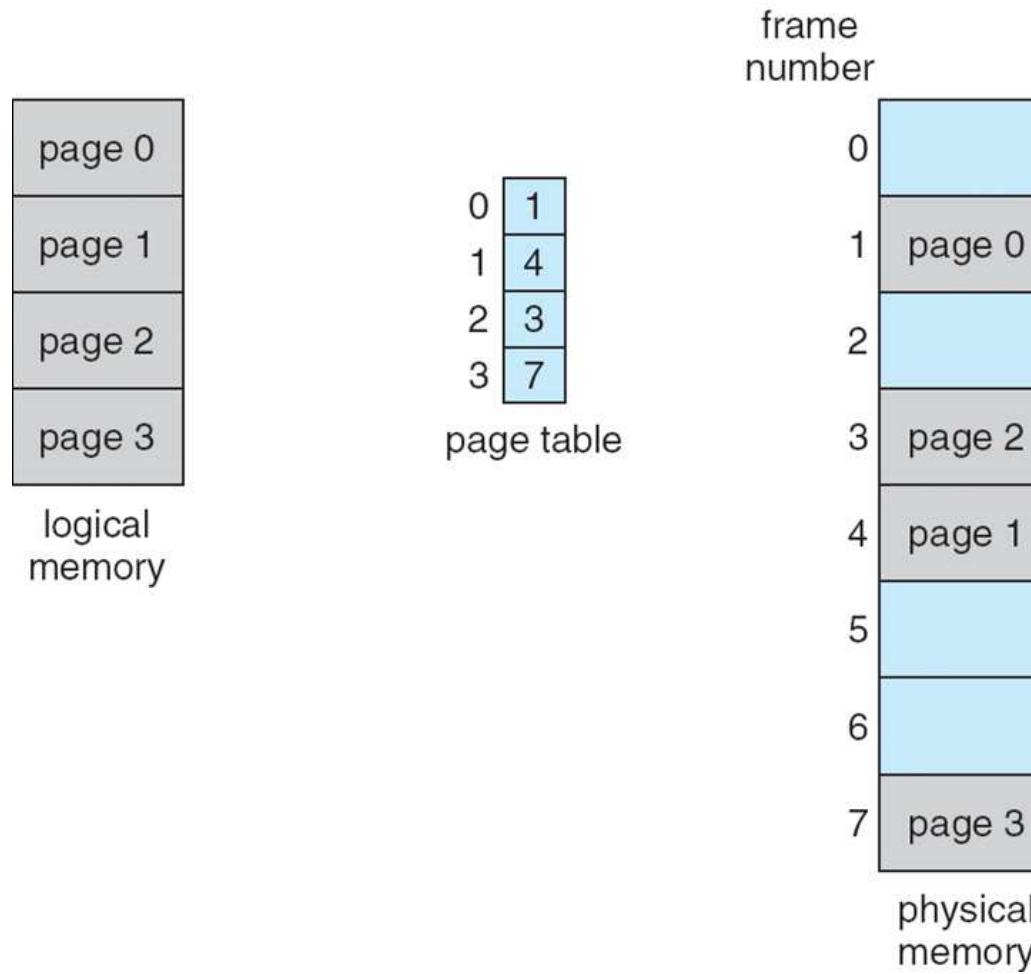


# Paging Hardware





# Paging Model of Logical and Physical Memory





# Paging Example

- Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	
8	
12	
16	
20	a
24	b
28	c

physical memory

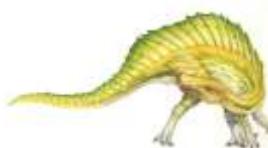




# Paging -- Calculating internal fragmentation

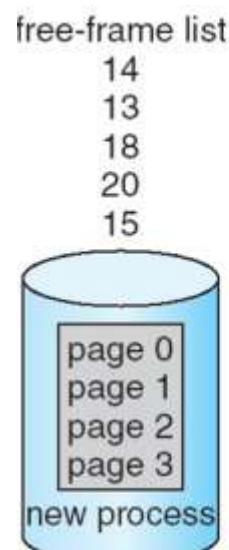
---

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- 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



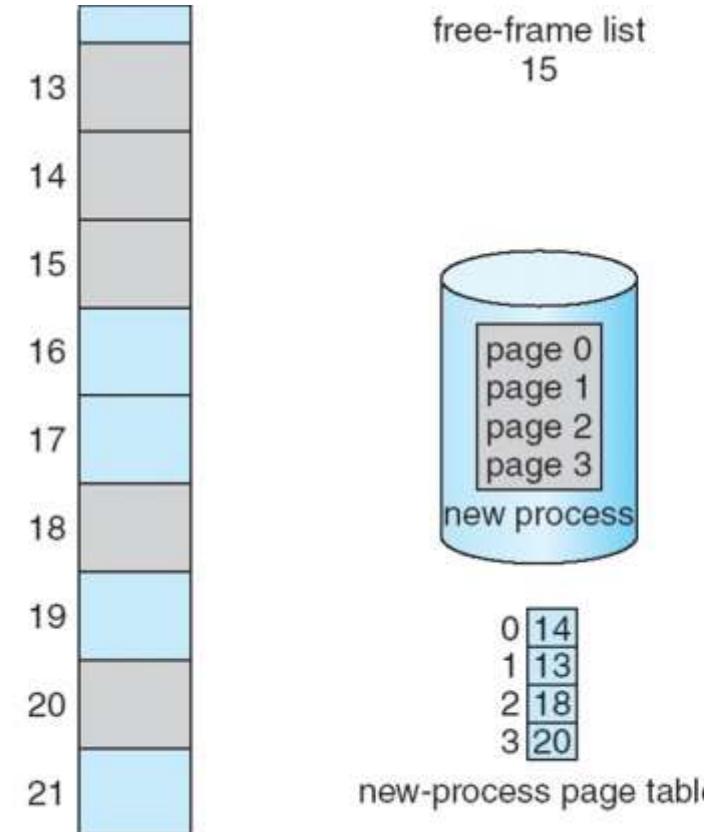


# Free Frames



(a)

Before allocation



(b)

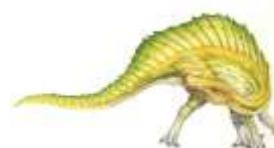
After allocation





# Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register (PTBR)** points to the page table
  - **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** (also called **associative memory**).





# Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access



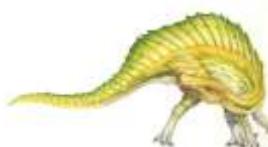


# Hardware

- Associative memory – parallel search

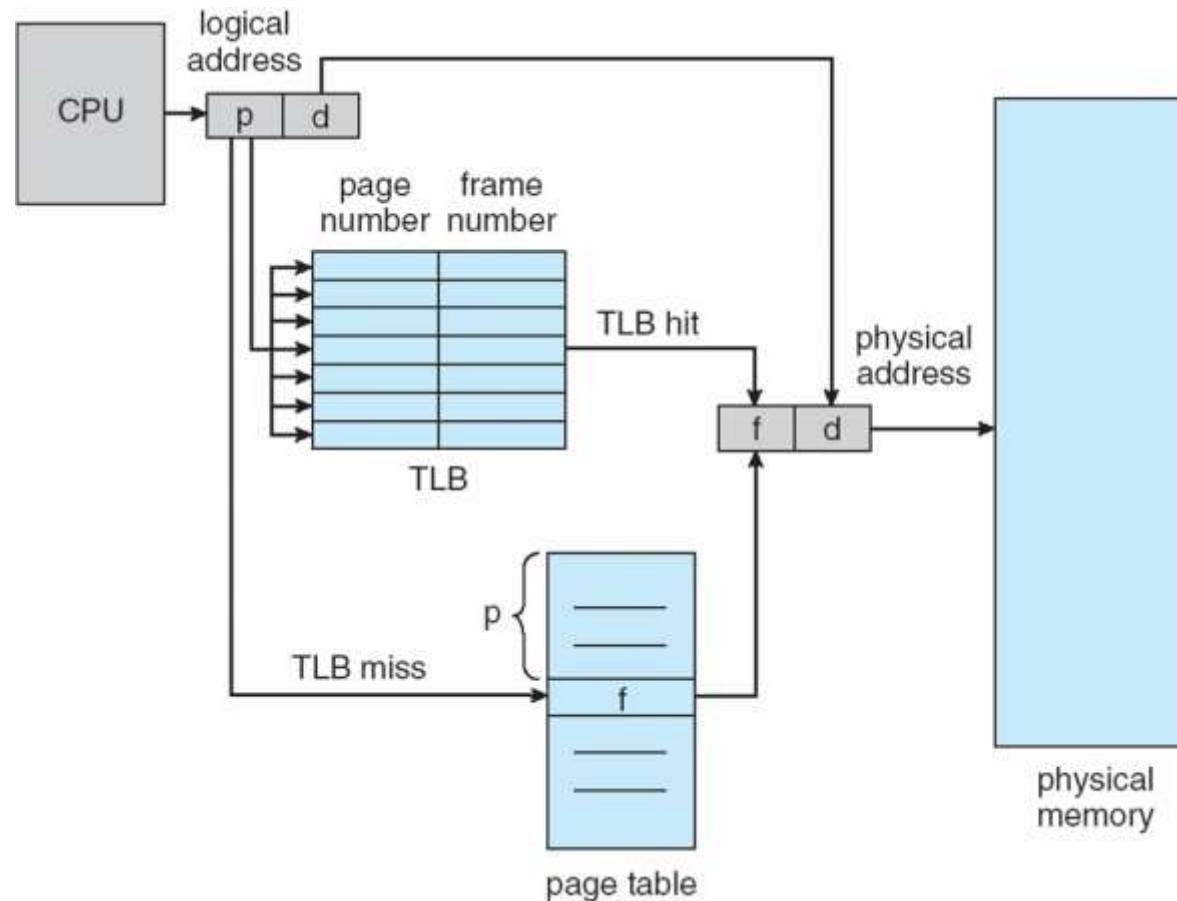
Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory





# Paging Hardware With TLB





# Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time (EAT)**
$$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$
implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,
$$EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$
implying only 1% slowdown in access time.





# Memory Protection

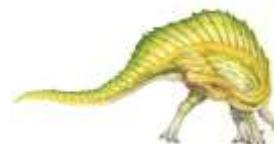
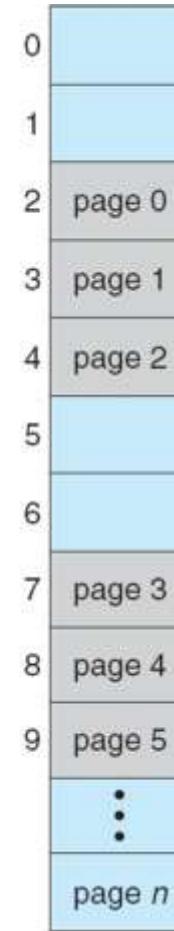
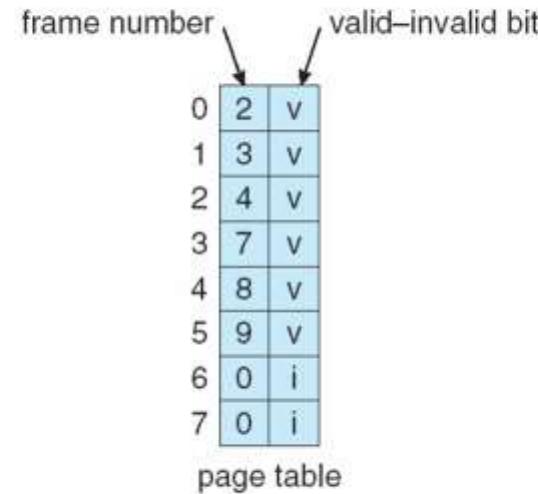
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel





# Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	





# Shared Pages

## ■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

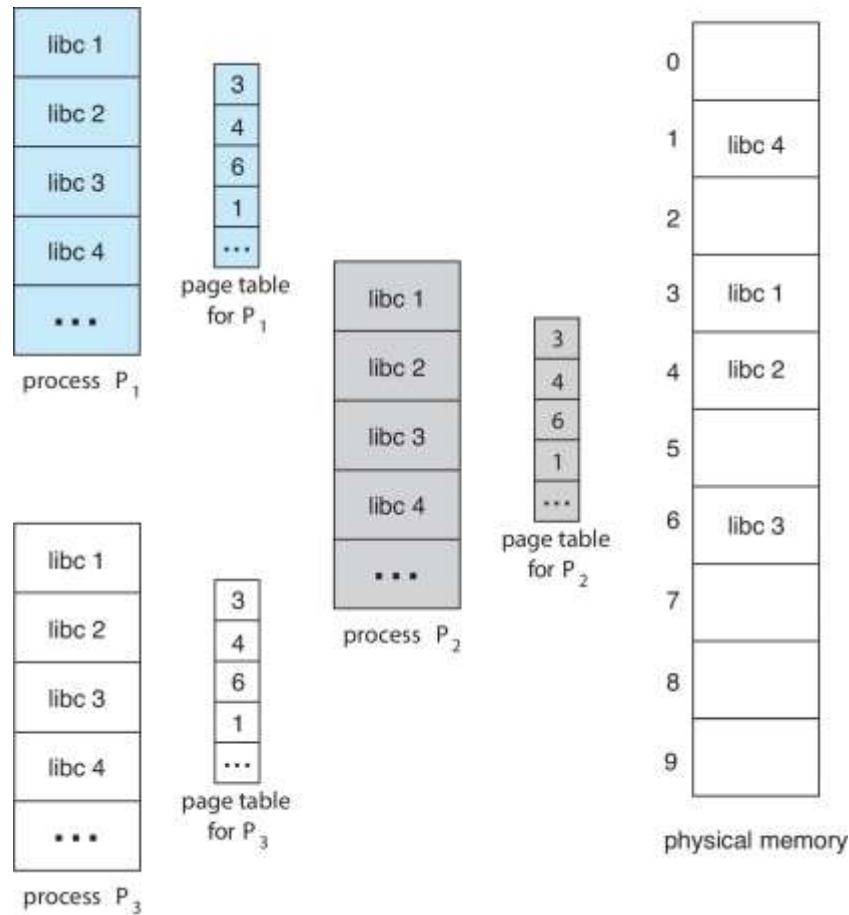
## ■ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space





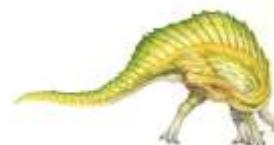
# Shared Pages Example





# Structure of the Page Table

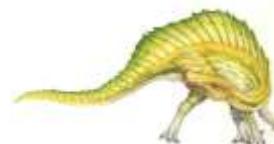
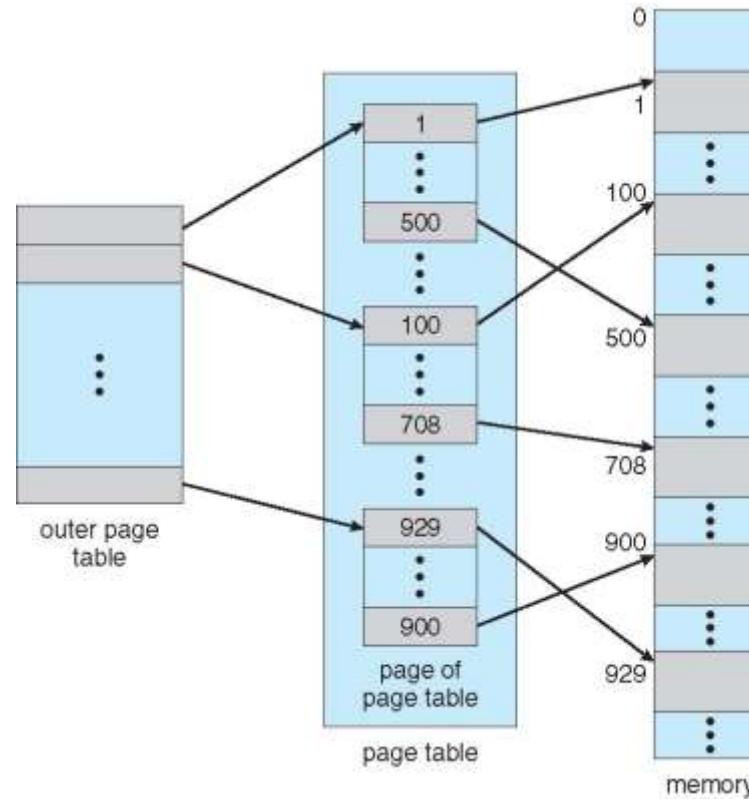
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ( $2^{12}$ )
  - Page table would have 1 million entries ( $2^{32} / 2^{12}$ )
  - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
    - ▶ Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - ▶ Hierarchical Paging
    - ▶ Hashed Page Tables
    - ▶ Inverted Page Tables





# Hierarchical Page Tables

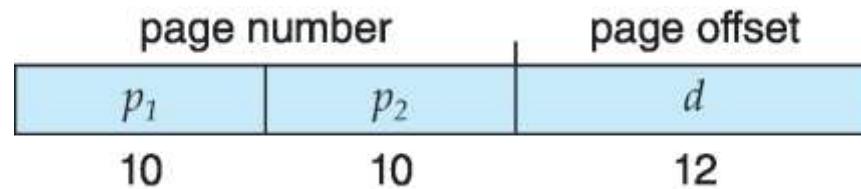
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





# Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

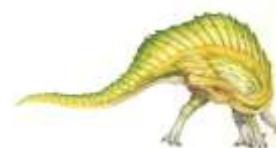
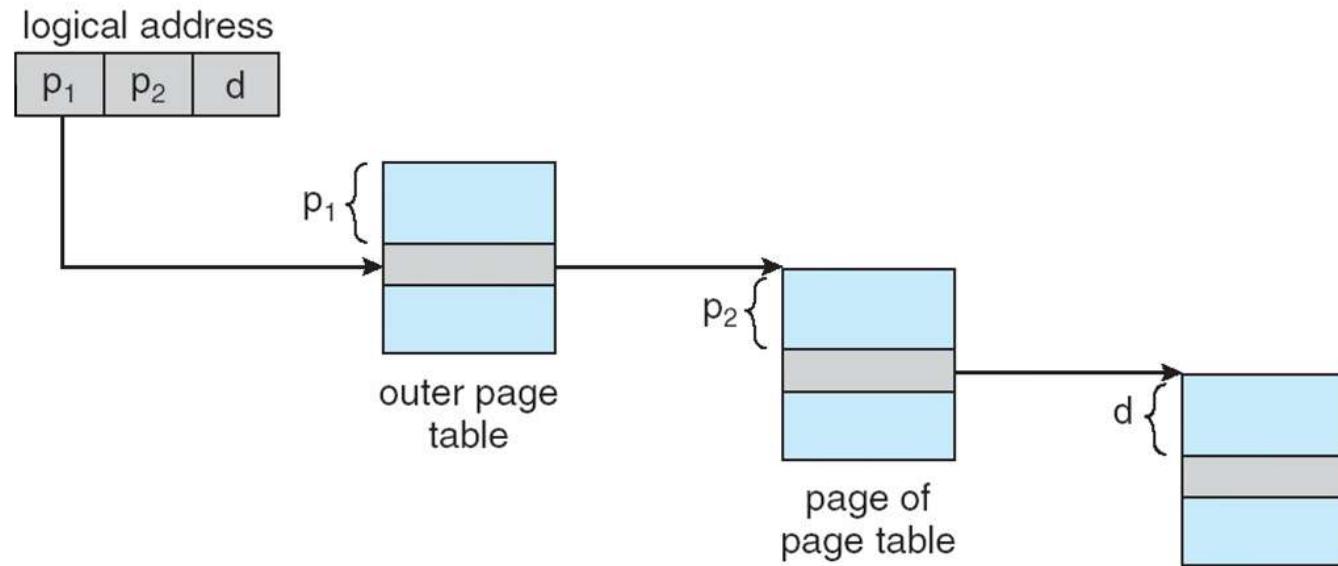


- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





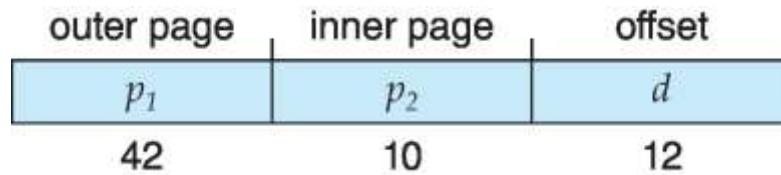
# Address-Translation Scheme





# 64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ( $2^{12}$ )
  - Then page table has  $2^{52}$  entries
  - If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
  - Address would look like

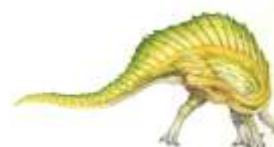
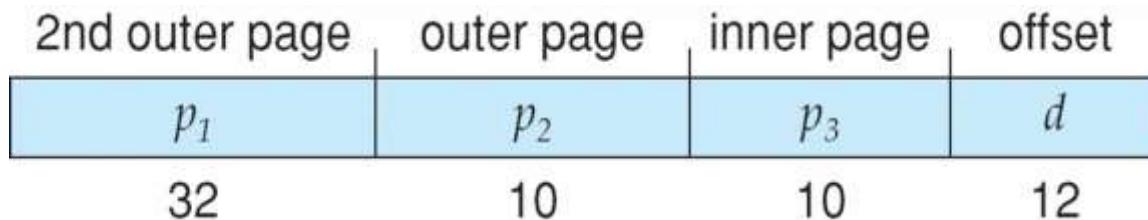
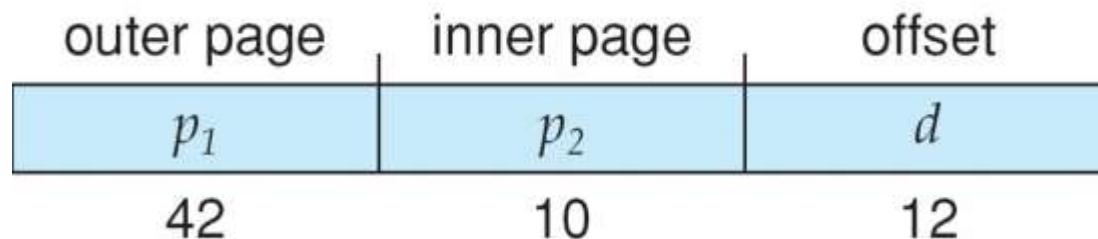


- Outer page table has  $2^{42}$  entries or  $2^{44}$  bytes
- One solution is to add a 2<sup>nd</sup> outer page table
- But in the following example the 2<sup>nd</sup> outer page table is still  $2^{34}$  bytes in size
  - ▶ And possibly 4 memory access to get to one physical memory location





# Three-level Paging Scheme





# Hashed Page Tables

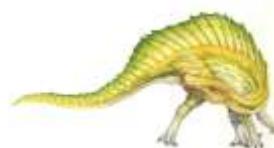
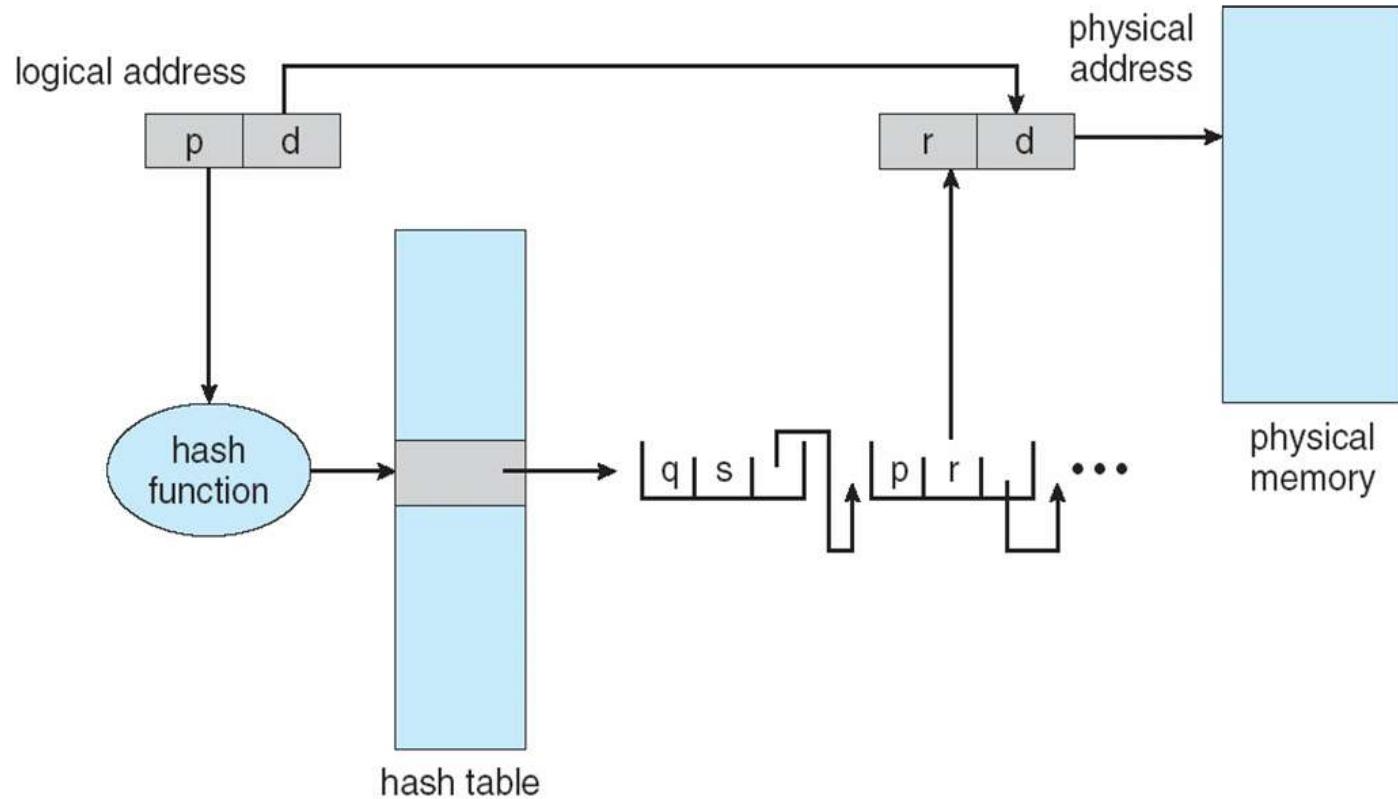
---

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





# Hashed Page Table





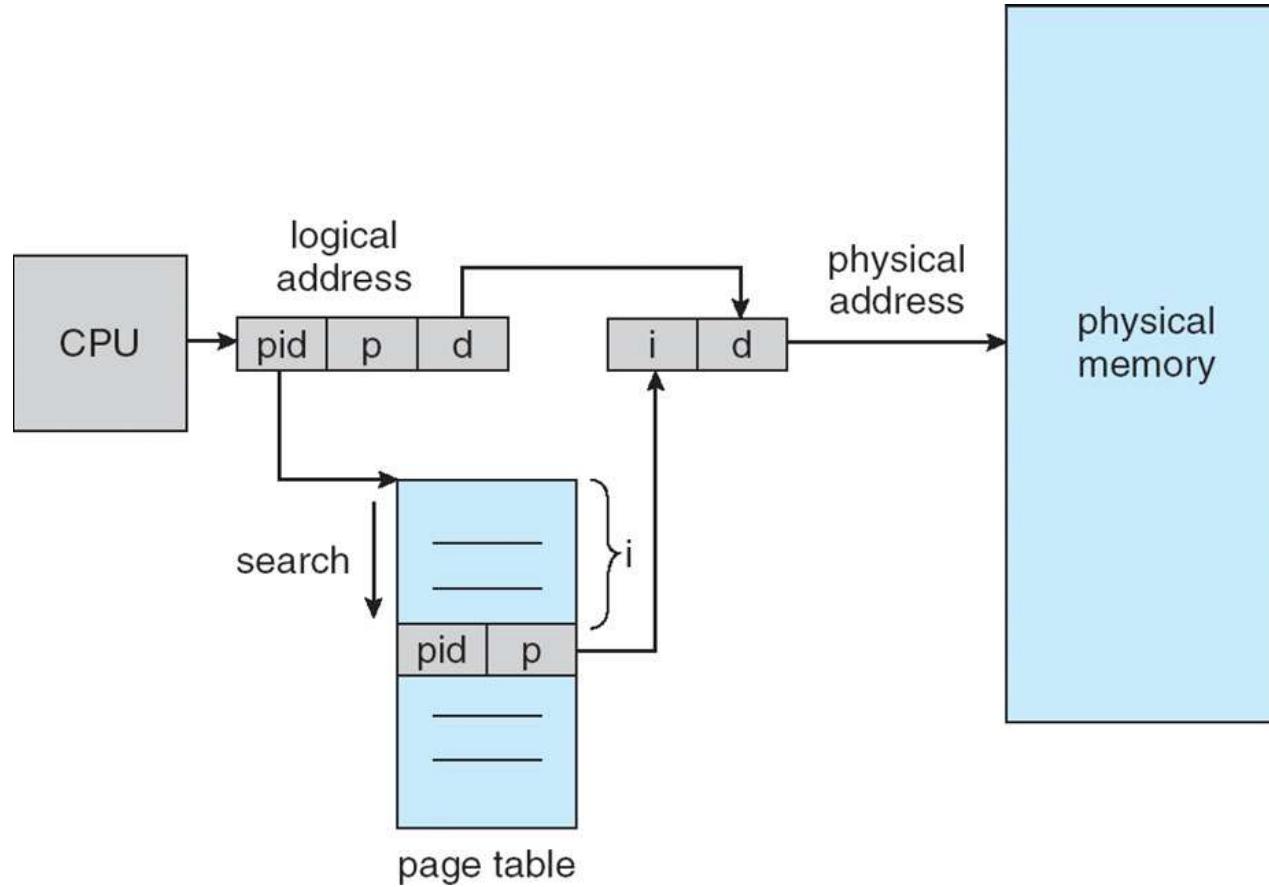
# Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address





# Inverted Page Table Architecture





# Oracle SPARC Solaris

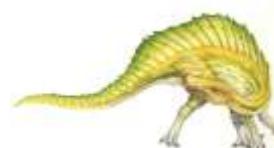
- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - ▶ More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)





# Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - ▶ Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - ▶ If match found, the CPU copies the TSB entry into the TLB and translation completes
    - ▶ If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.





# Swapping

---

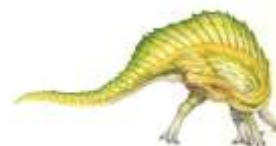
- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk





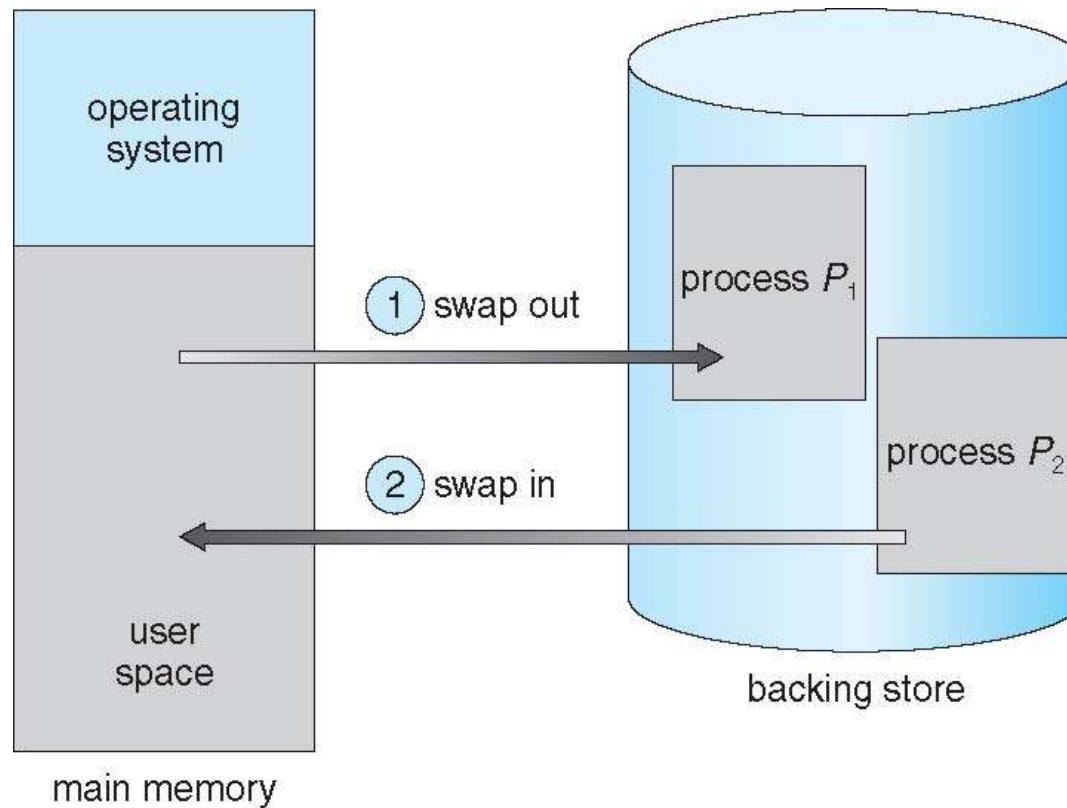
# Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold





# Schematic View of Swapping





# Context Switch Time including Swapping

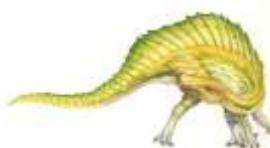
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`





# Context Switch Time and Swapping (Cont.)

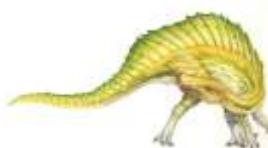
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - ▶ Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - ▶ Swap only when free memory extremely low





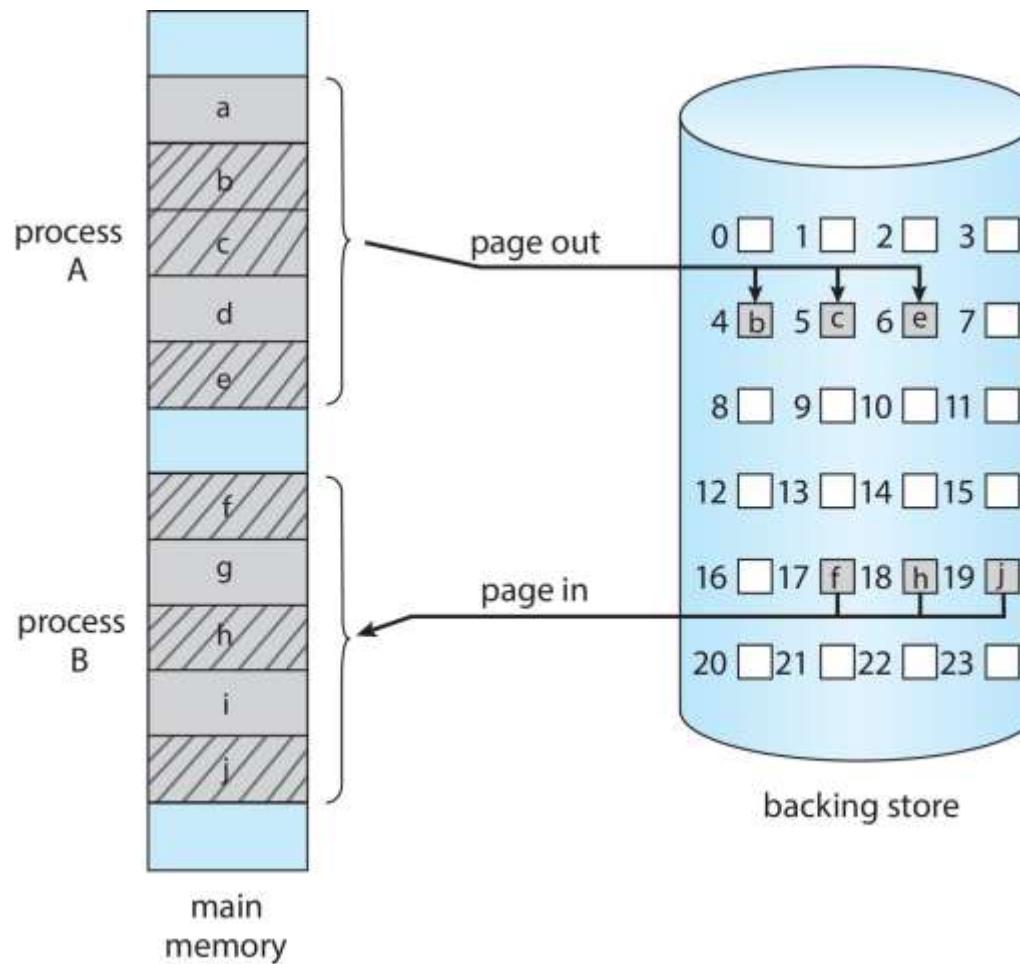
# Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - ▶ Small amount of space
    - ▶ Limited number of write cycles
    - ▶ Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS **asks** apps to voluntarily relinquish allocated memory
    - ▶ Read-only data thrown out and reloaded from flash if needed
    - ▶ Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below





# Swapping with Paging

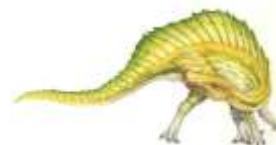




## Example: The Intel 32 and 64-bit Architectures

---

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here





# Example: The Intel IA-32 Architecture

---

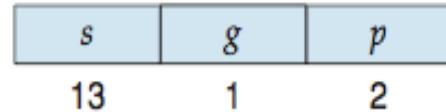
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - ▶ First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
    - ▶ Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)





# Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
  - Selector given to segmentation unit
    - ▶ Which produces linear addresses

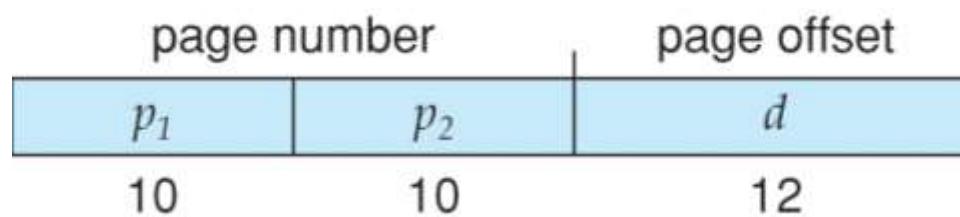
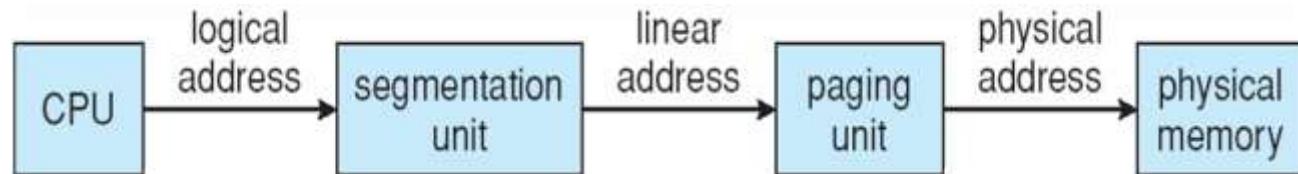


- Linear address given to paging unit
  - ▶ Which generates physical address in main memory
  - ▶ Paging units form equivalent of MMU
  - ▶ Pages sizes can be 4 KB or 4 MB



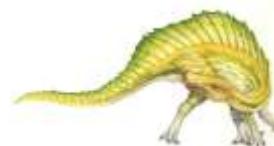
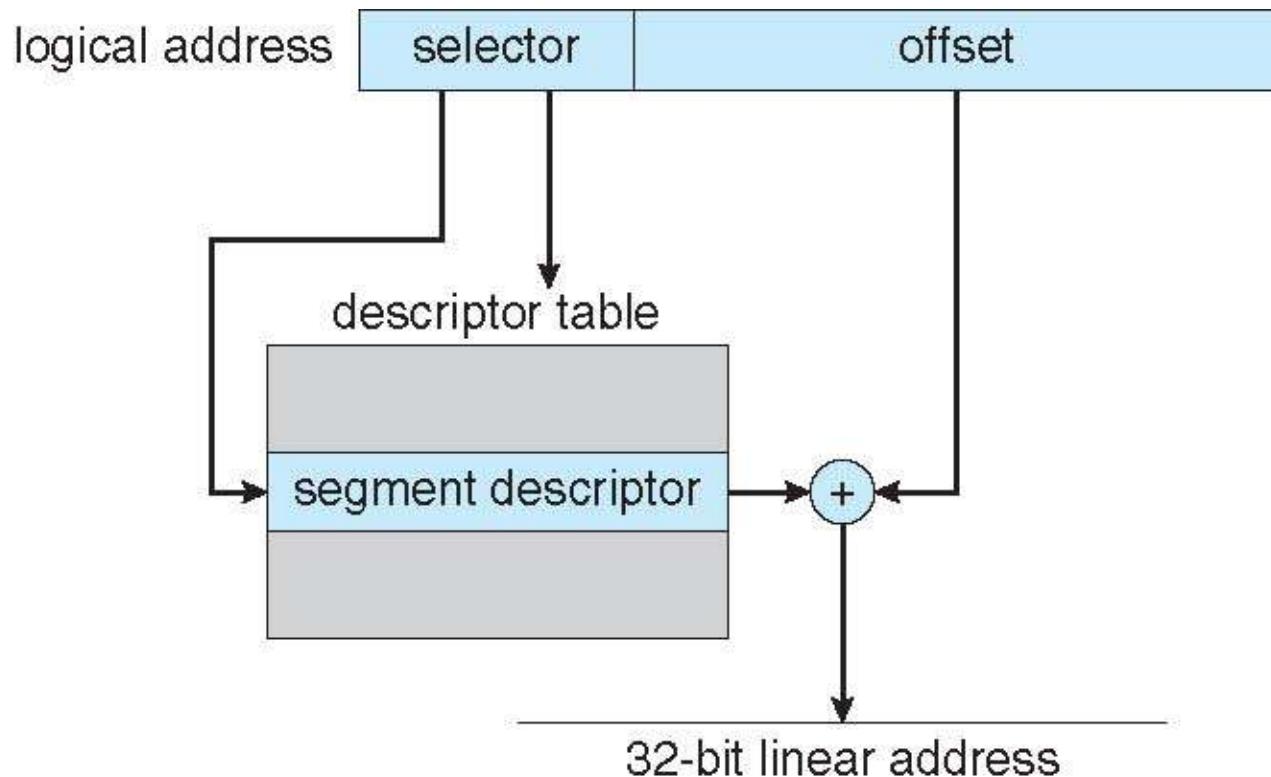


# Logical to Physical Address Translation in IA-32



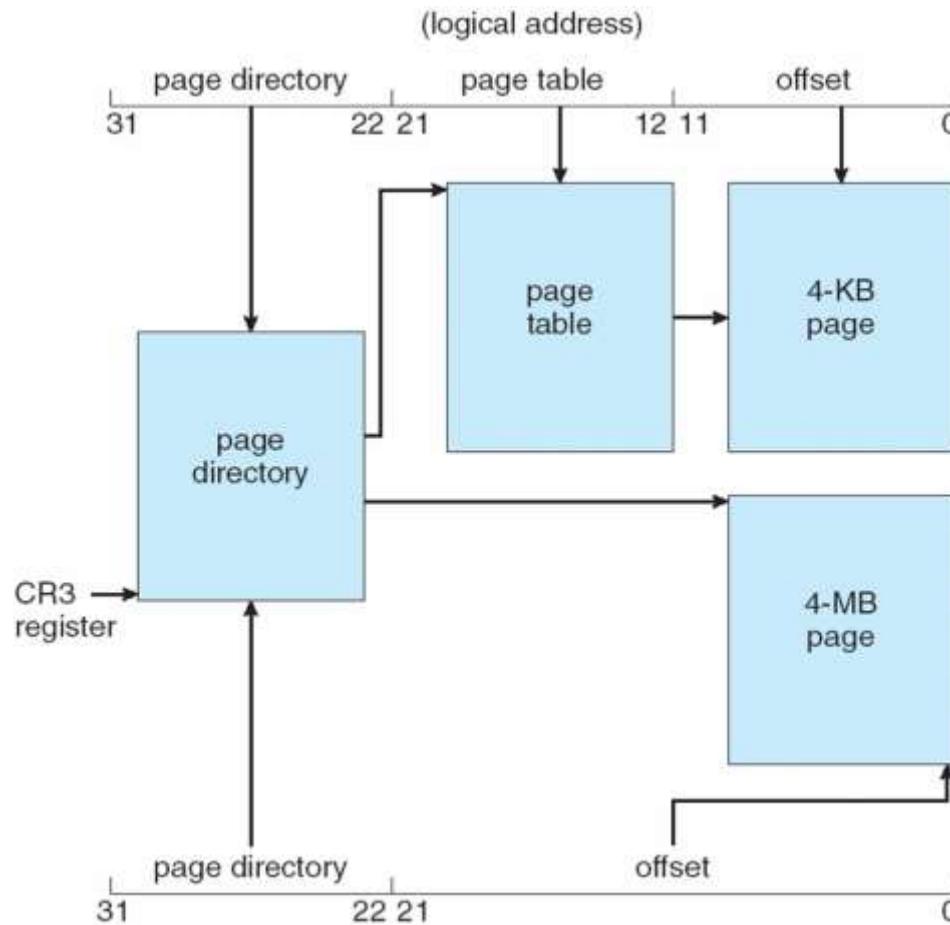


# Intel IA-32 Segmentation





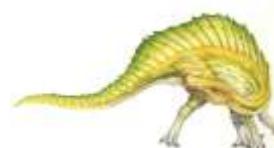
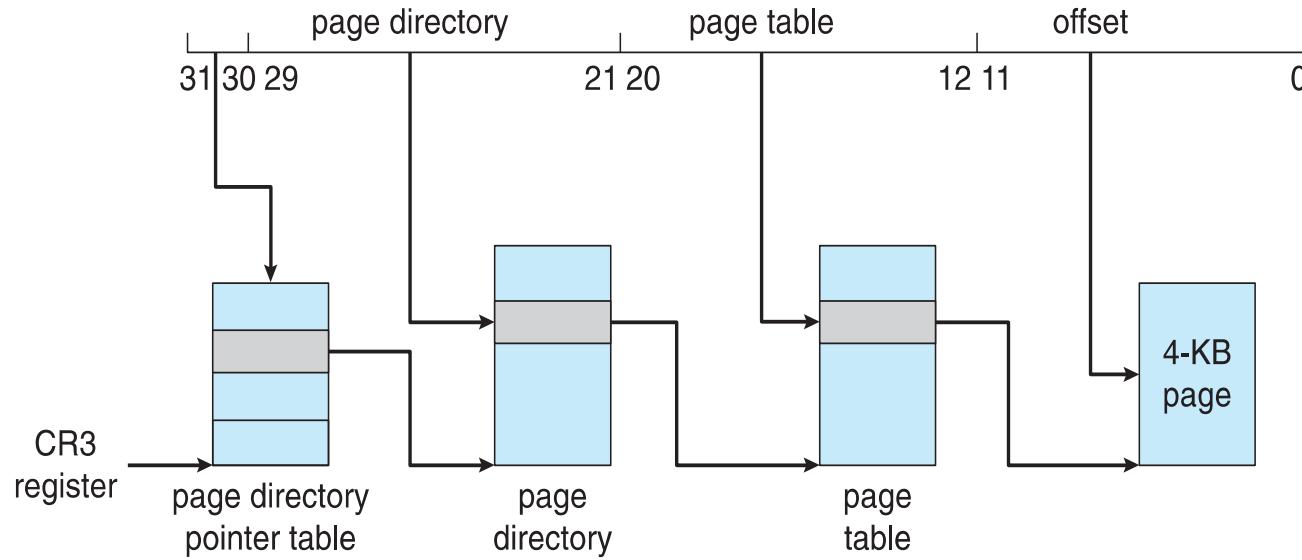
# Intel IA-32 Paging Architecture





# Intel IA-32 Page Address Extensions

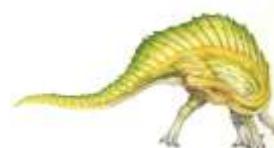
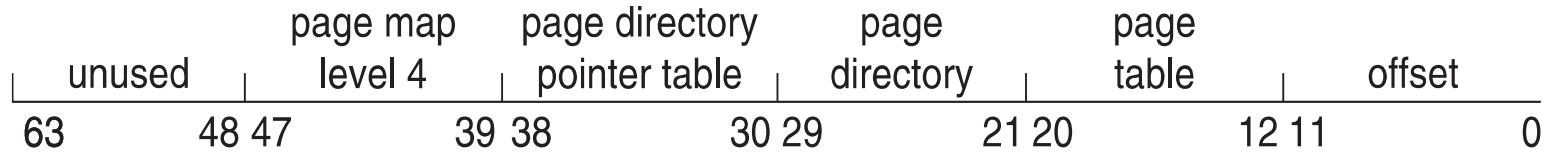
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory





# Intel x86-64

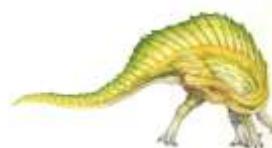
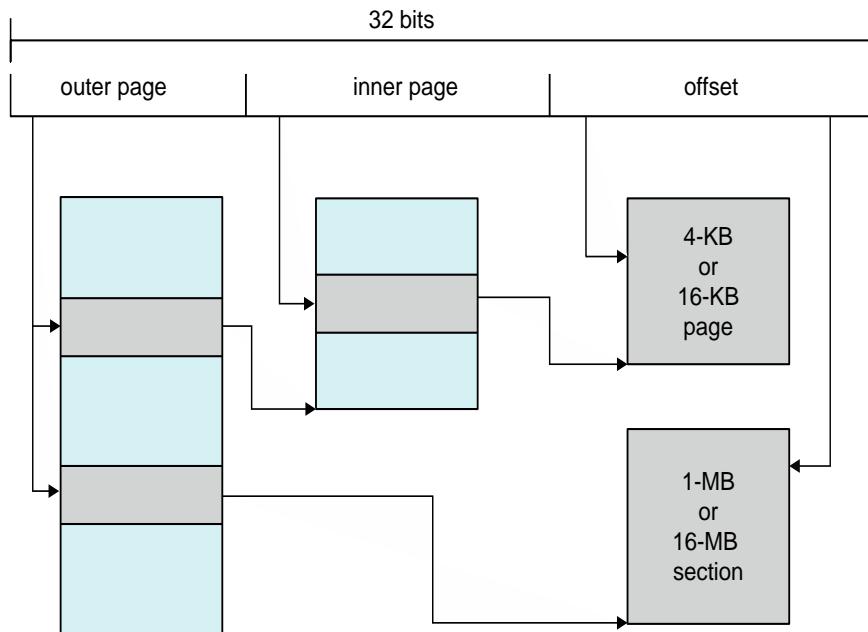
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



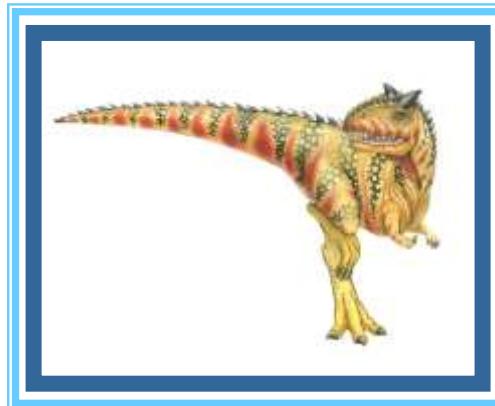


# Example: ARM Architecture

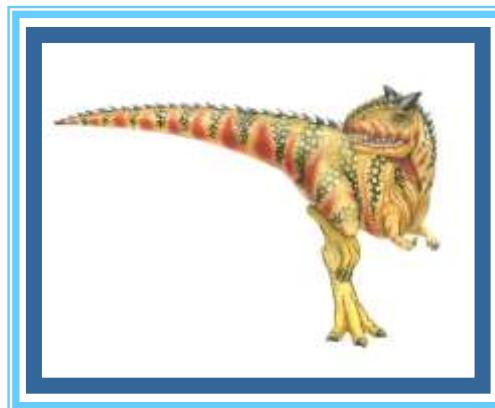
- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



# End of Chapter 9



# Chapter 10: Virtual Memory





# Chapter 10: Virtual Memory

---

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

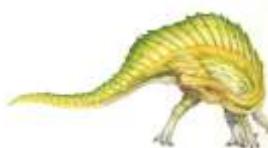




# Objectives

---

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

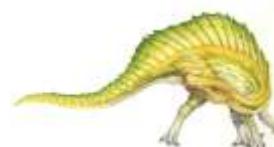




# Background

---

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster





# Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes





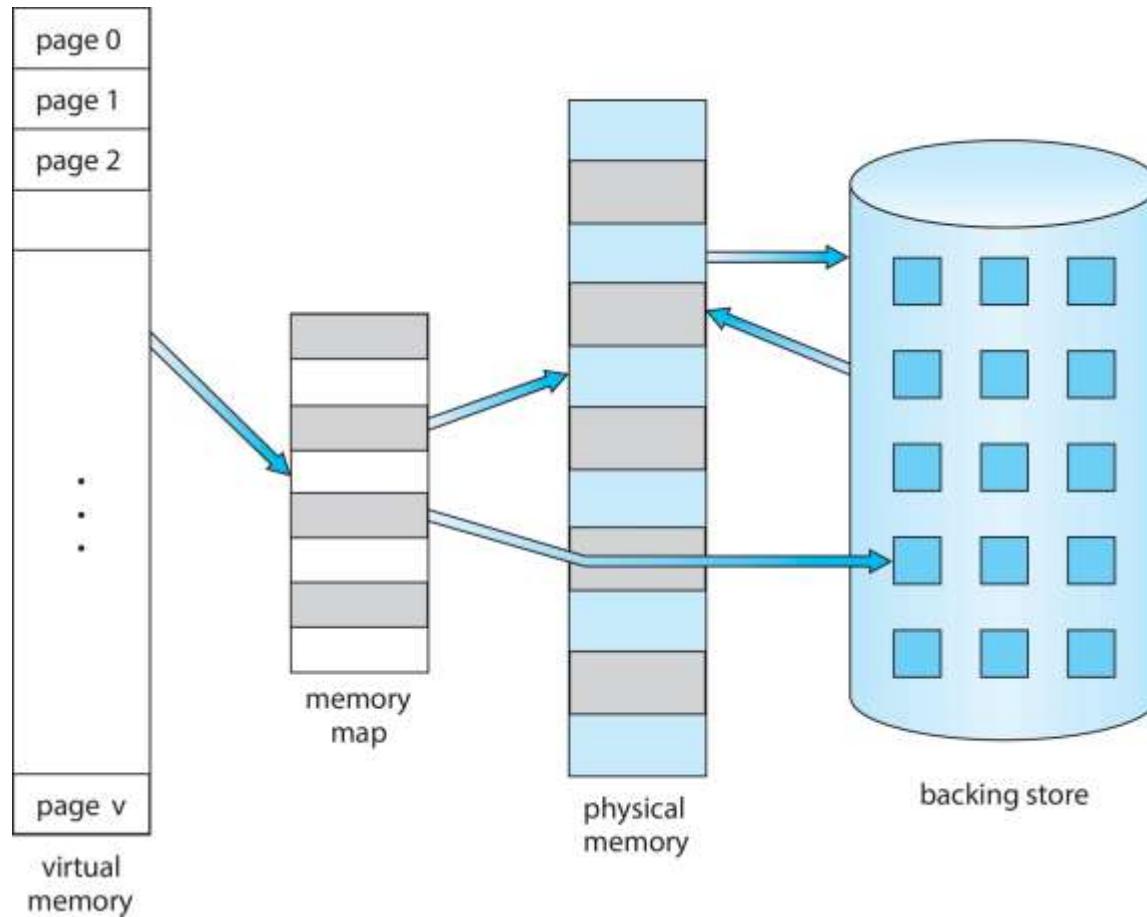
# Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation





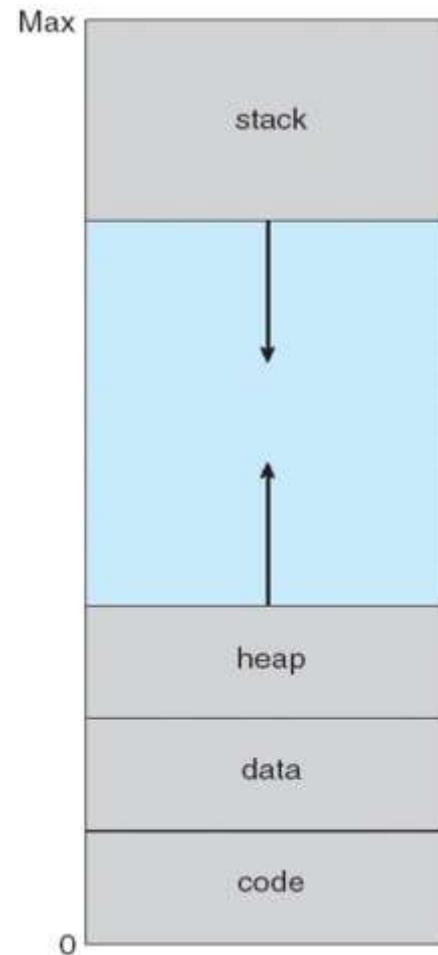
# Virtual Memory That is Larger Than Physical Memory





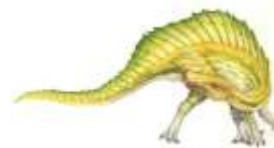
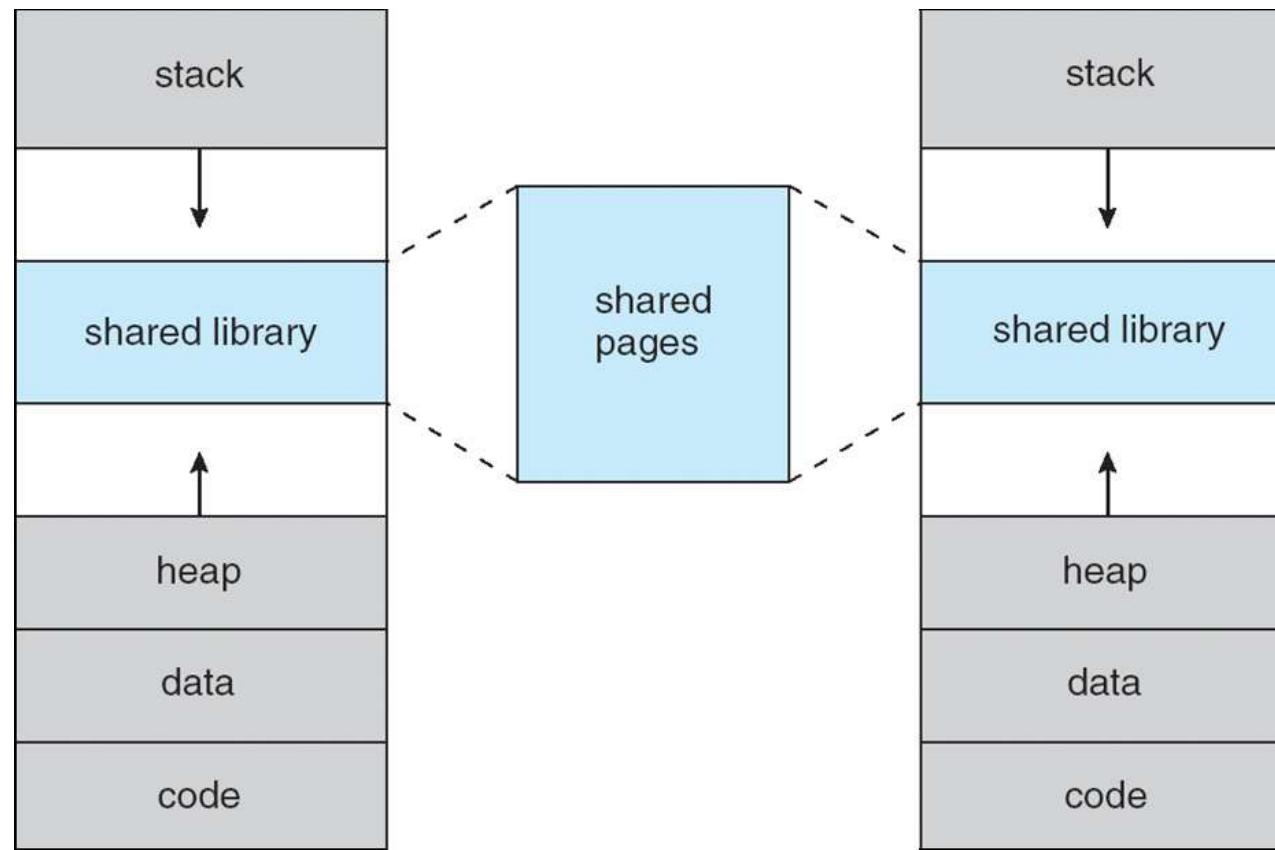
# Virtual-address Space

- 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





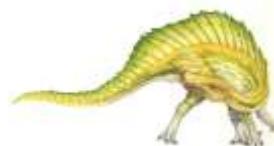
# Shared Library Using Virtual Memory





# Demand Paging

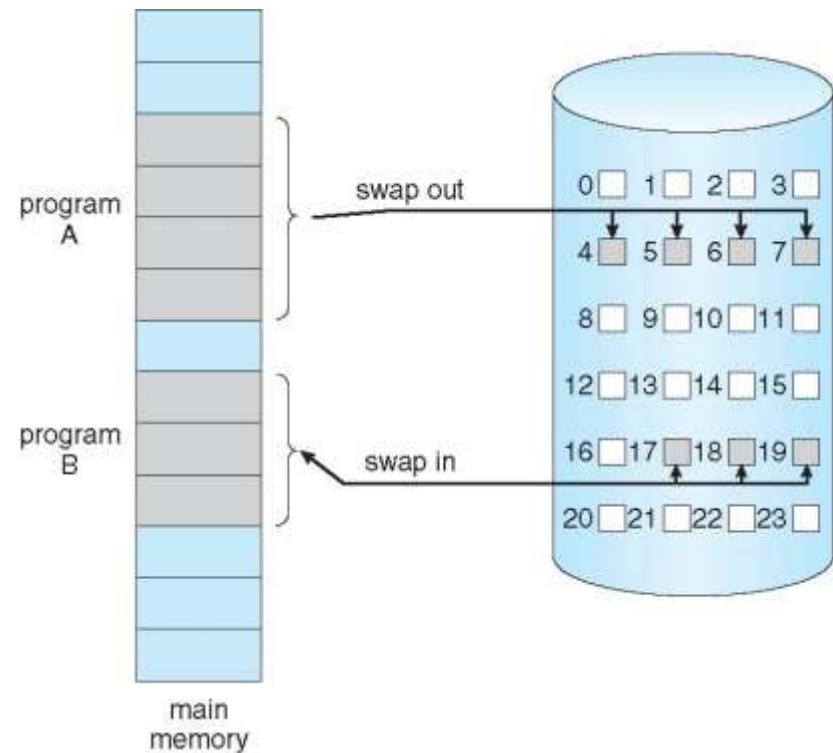
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**





# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)





# Basic Concepts

---

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - ▶ Without changing program behavior
    - ▶ Without programmer needing to change code





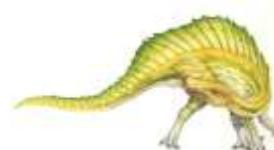
# Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

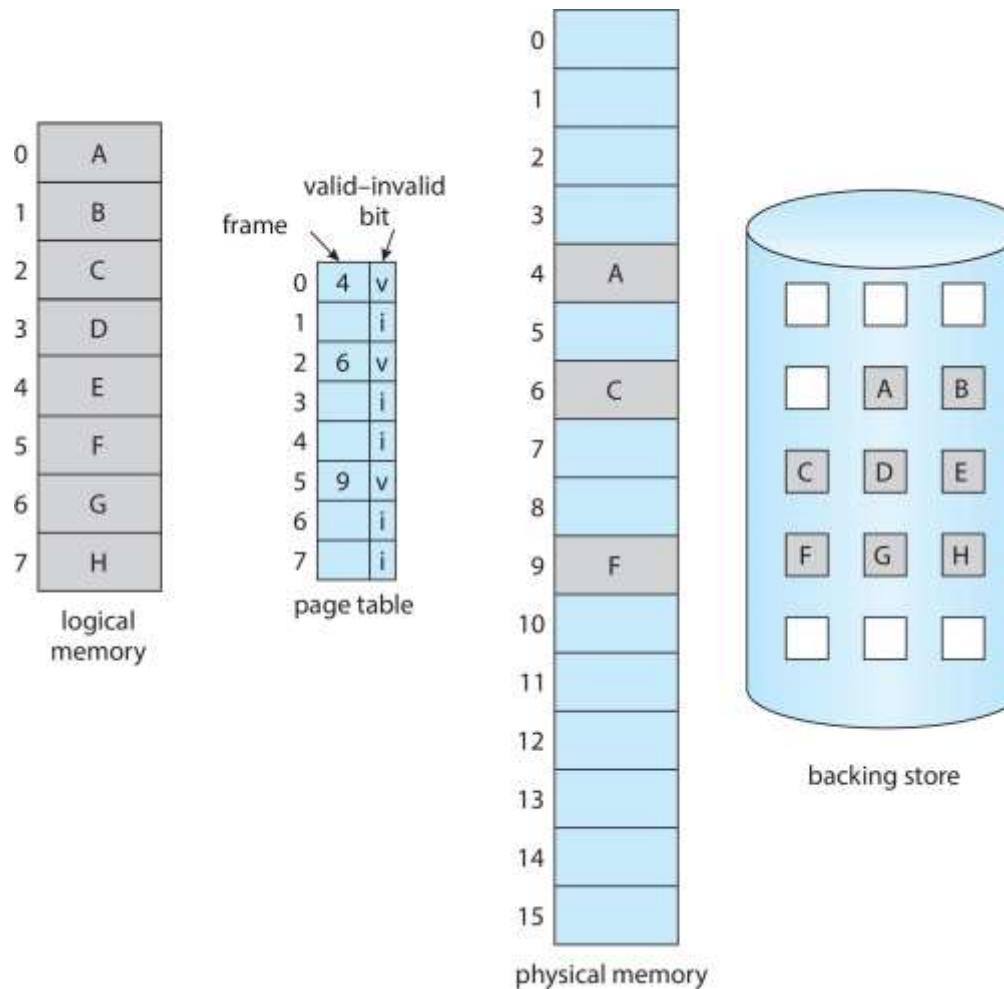
page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault





# Page Table When Some Pages Are Not in Main Memory





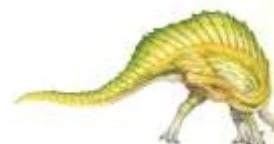
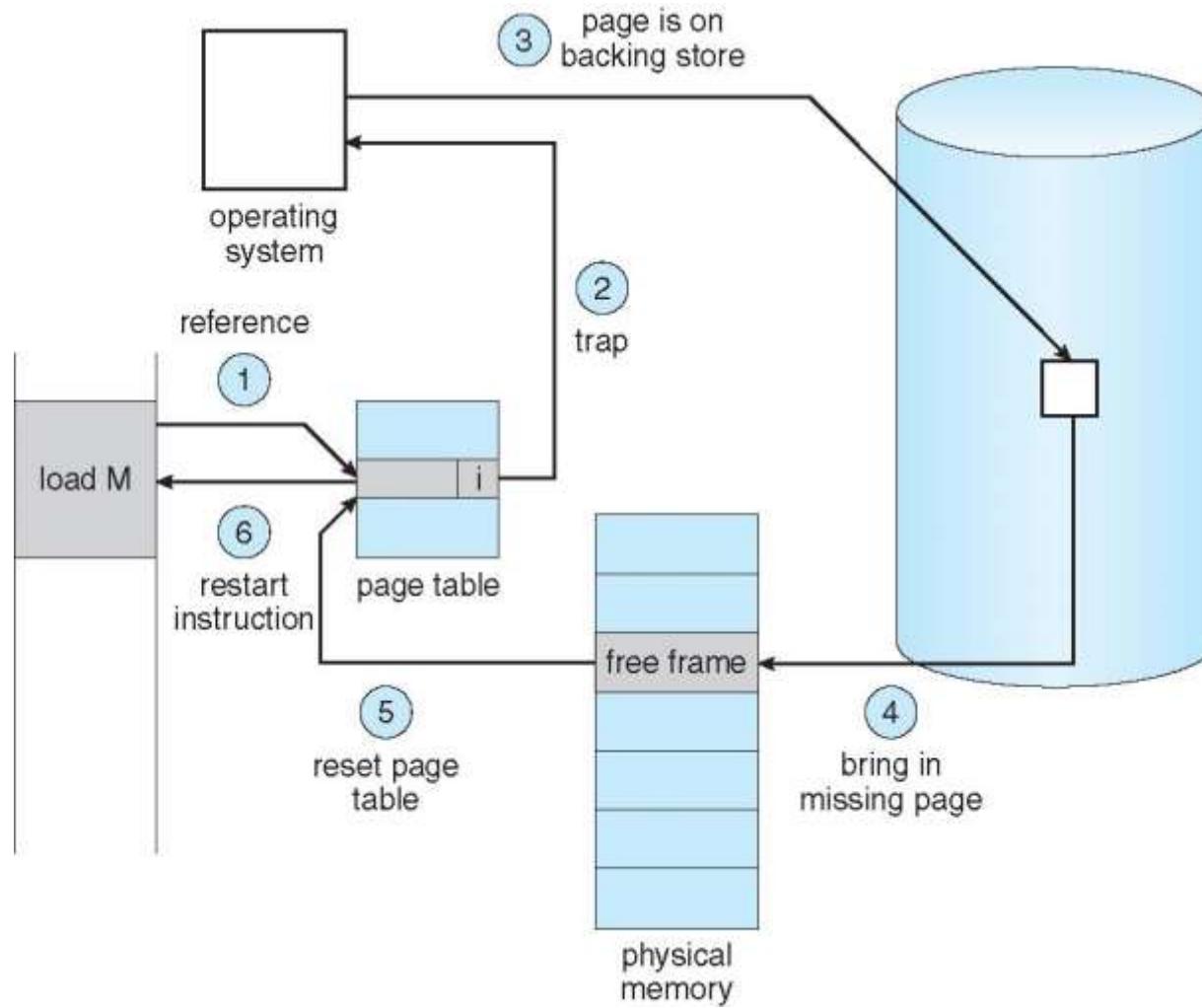
# Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
  - Page fault
2. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory  
Set validation bit = **V**
6. Restart the instruction that caused the page fault





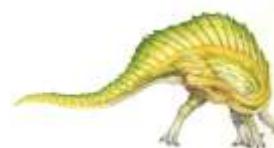
# Steps in Handling a Page Fault (Cont.)





# Aspects of Demand Paging

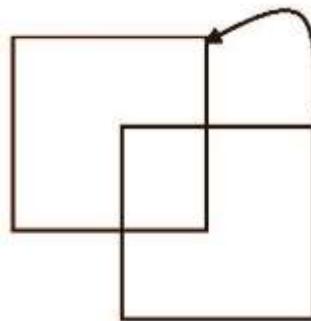
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart



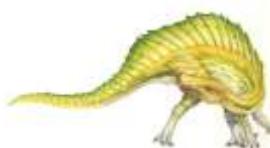


# Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



- Auto increment/decrement location
- Restart the whole operation?
  - What if source and destination overlap?





# Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



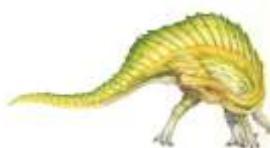
- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.





# Stages in Demand Paging – Worse Case

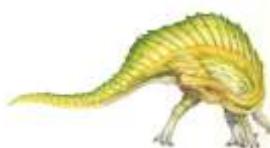
1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
  - a) Wait in a queue for this device until the read request is serviced
  - b) Wait for the device seek and/or latency time
  - c) Begin the transfer of the page to a free frame





# Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

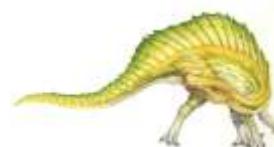




# Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + \text{swap page out} \\ & + \text{swap page in}) \end{aligned}$$





# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then  
 $\text{EAT} = 8.2 \text{ microseconds.}$   
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses





# Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
    - ▶ Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)





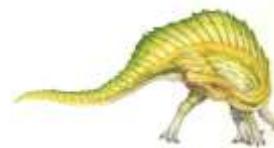
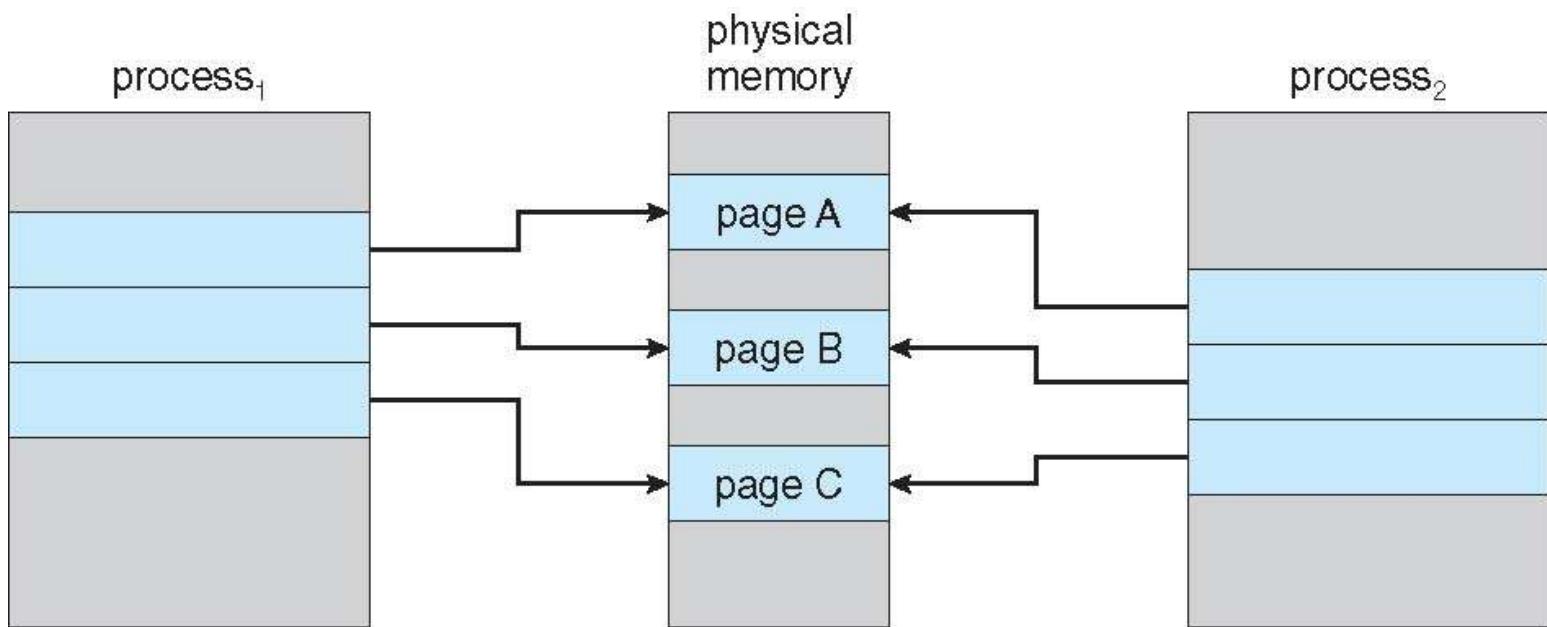
# Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - ▶ Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient



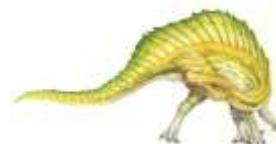
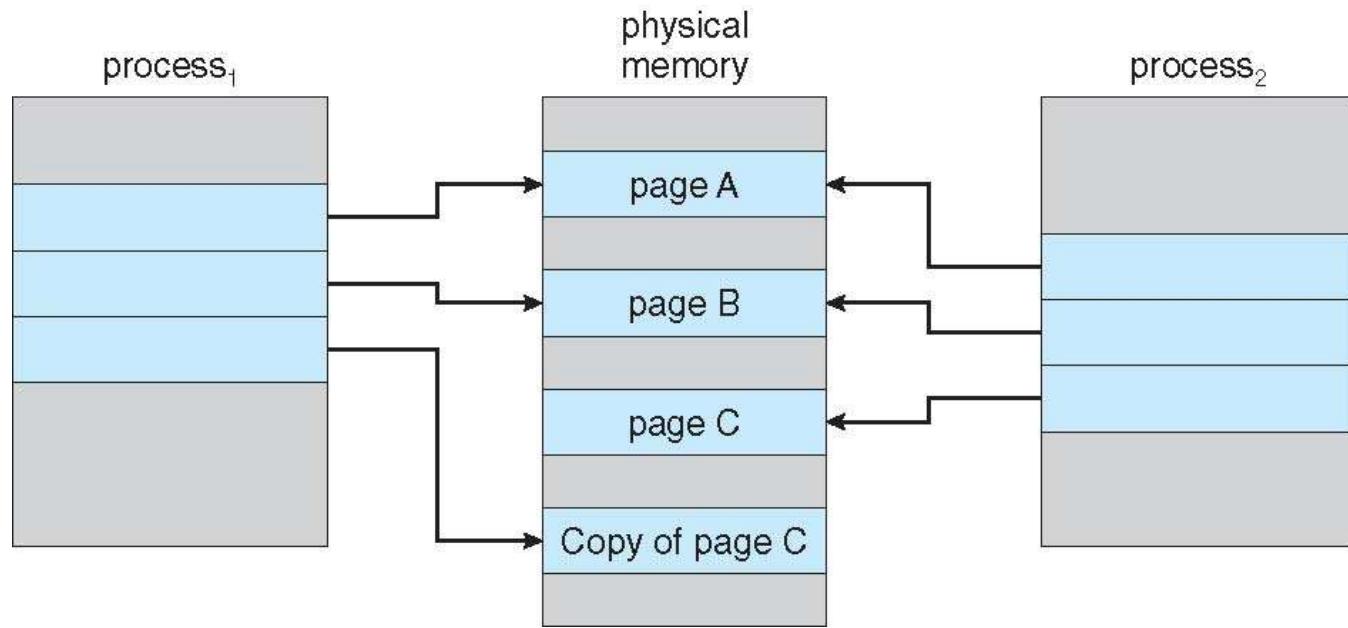


# Before Process 1 Modifies Page C





# After Process 1 Modifies Page C





# What Happens if There is no Free Frame?

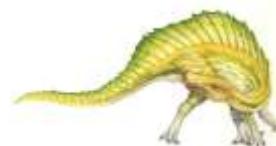
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times





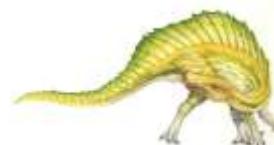
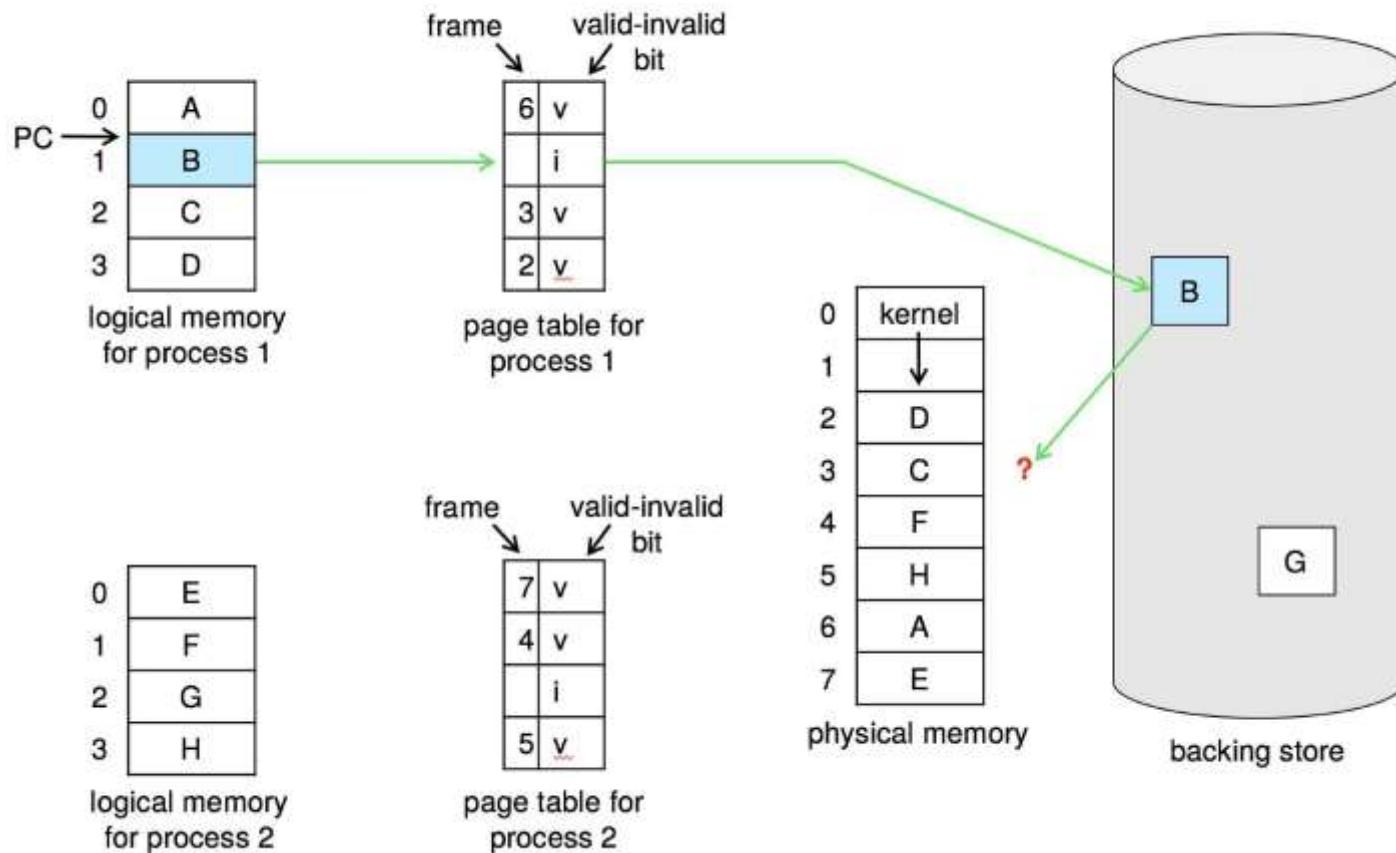
# Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory





# Need For Page Replacement





# Basic Page Replacement

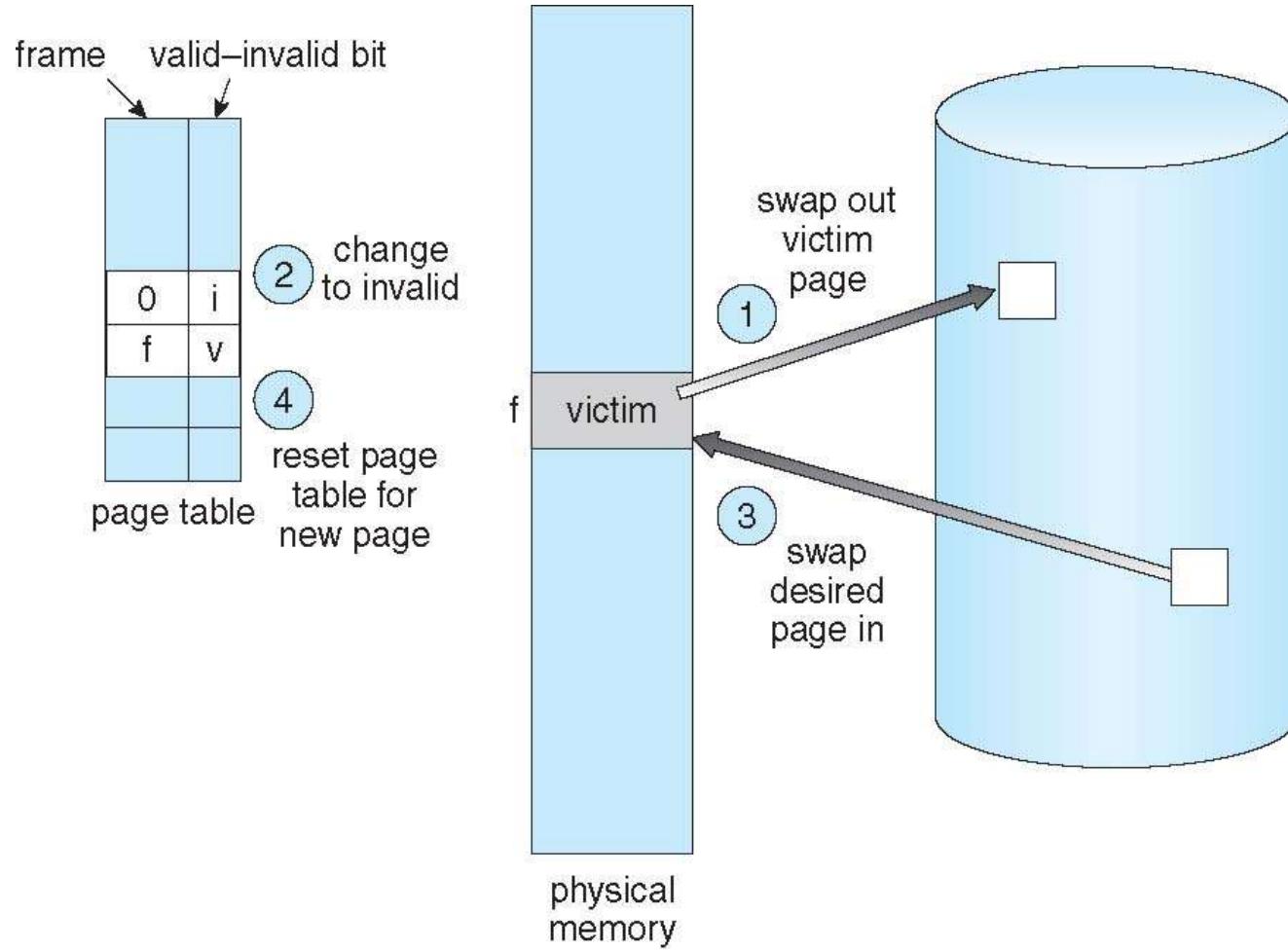
1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT





# Page Replacement





# Page and Frame Replacement Algorithms

---

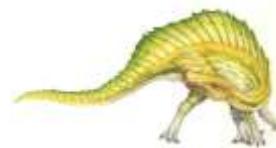
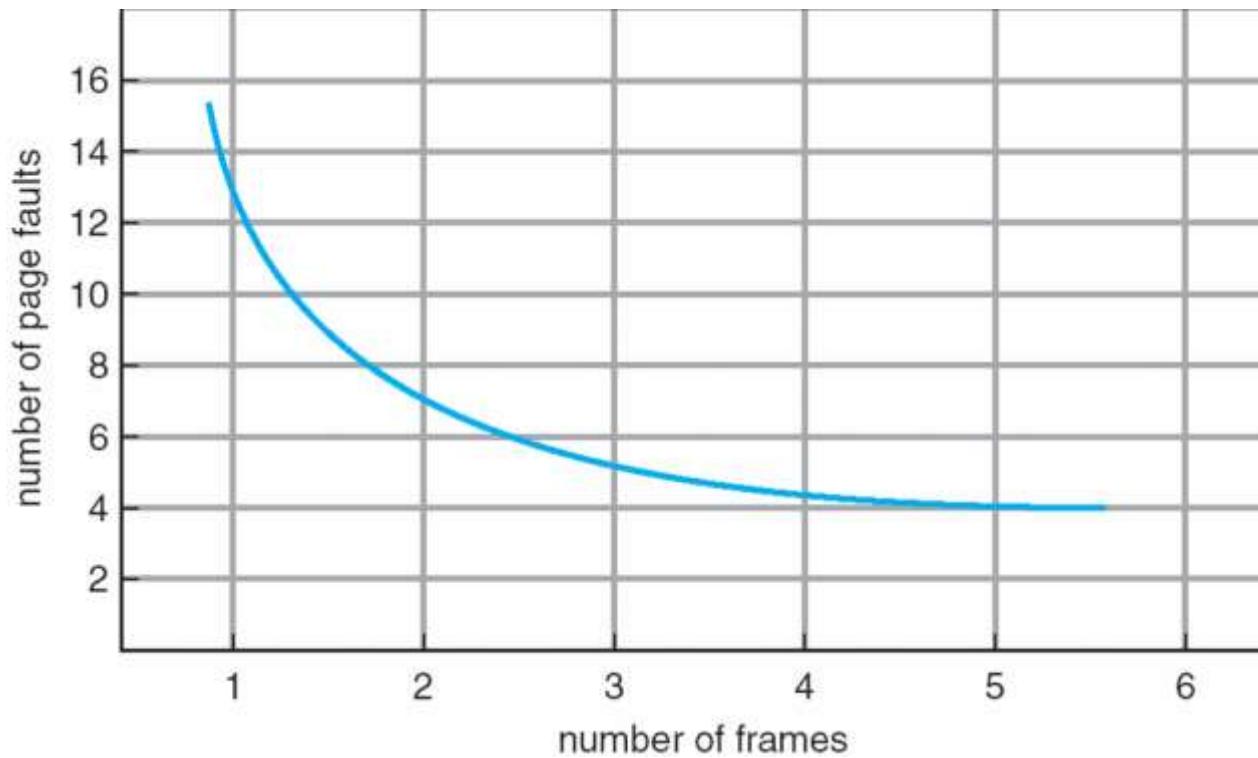
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1





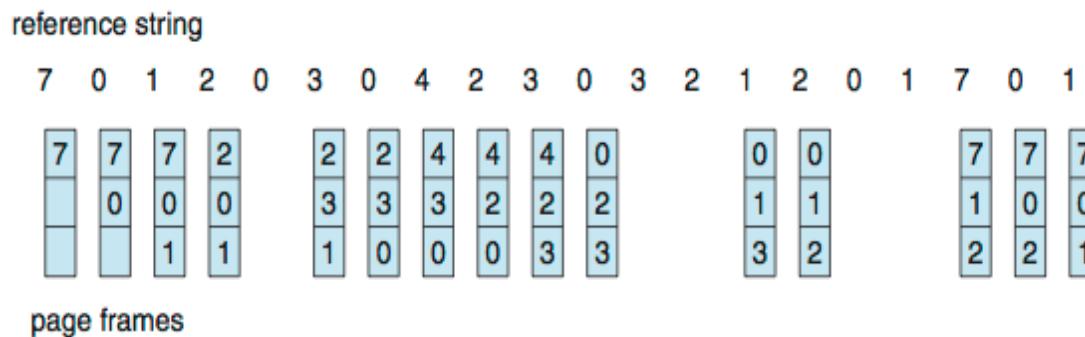
# Graph of Page Faults Versus the Number of Frames





# First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



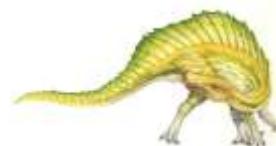
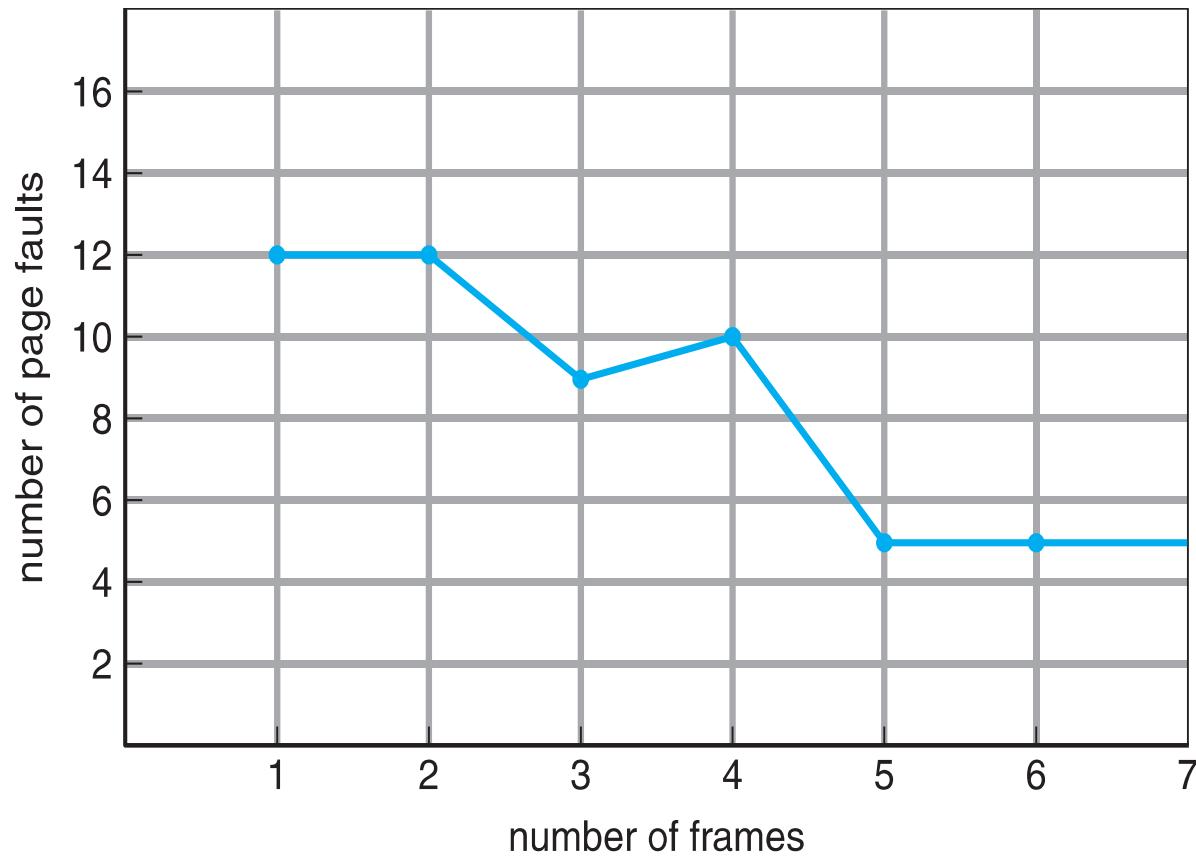
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - ▶ **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue





# FIFO Illustrating Belady' s Anomaly



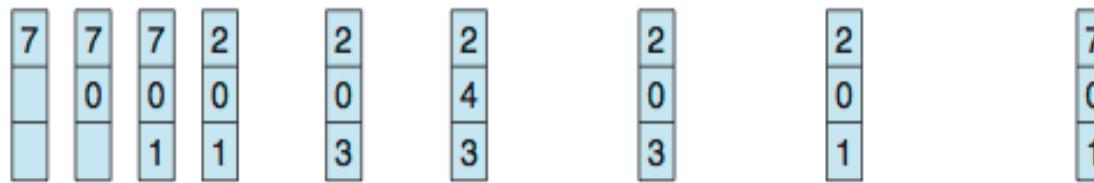


# Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



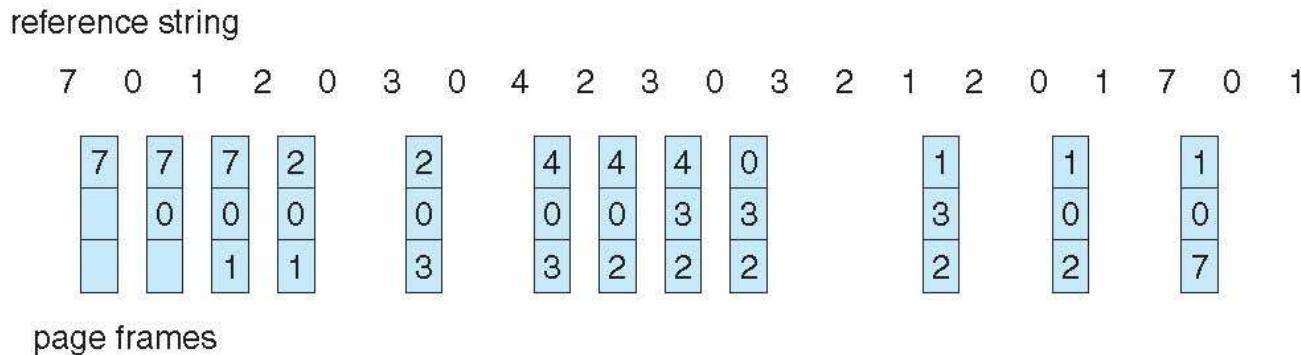
page frames





# Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



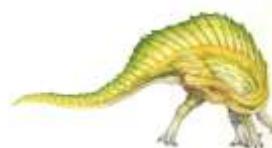
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





# LRU Algorithm (Cont.)

- 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

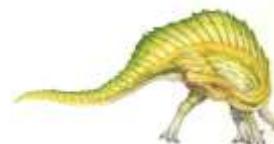
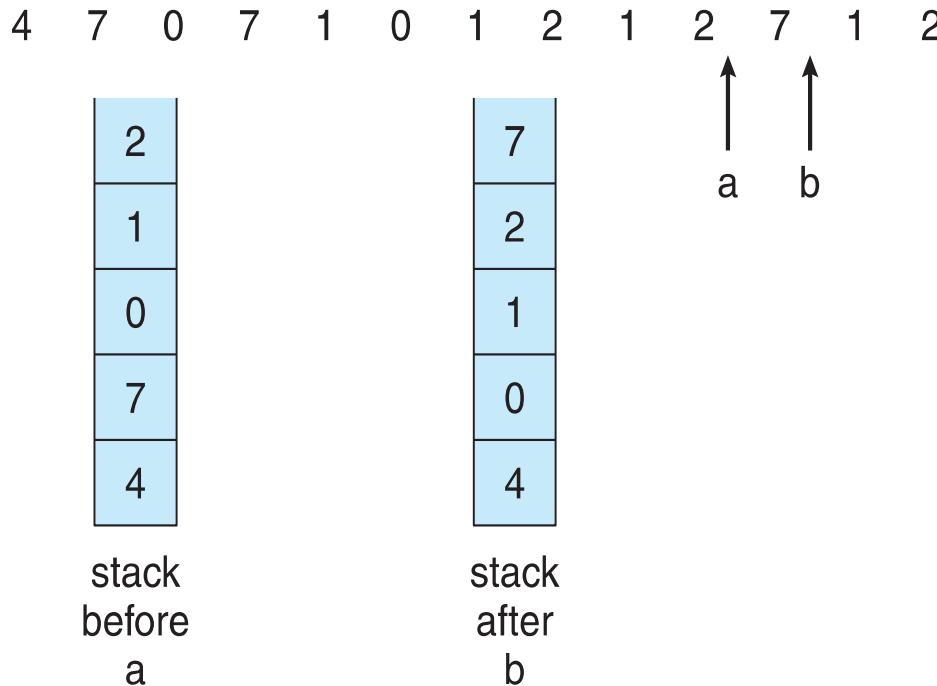




# LRU Algorithm (Cont.)

- 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





# 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





# LRU Approximation Algorithms (cont.)

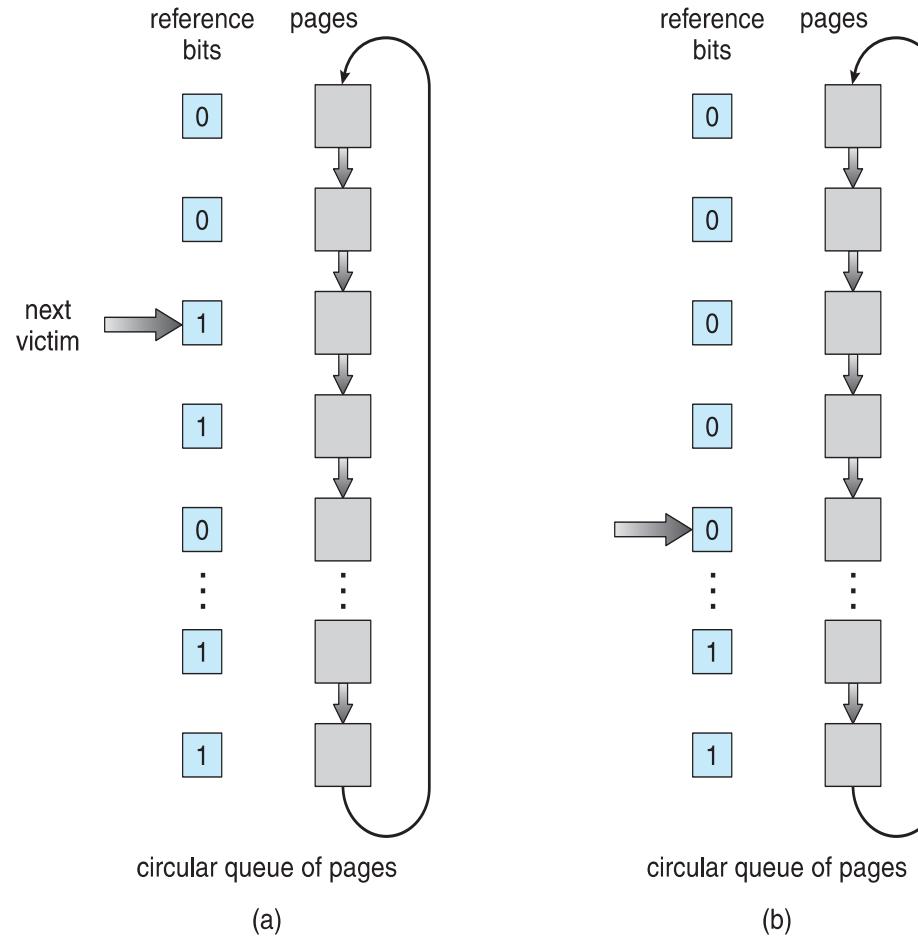
## ■ Second-chance algorithm

- 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 Algorithm





# Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used nor modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times





# Counting Algorithms

---

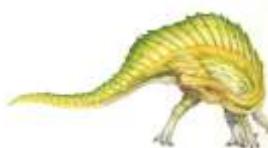
- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used (LFU) Algorithm:**
  - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used





# Page-Buffering Algorithms

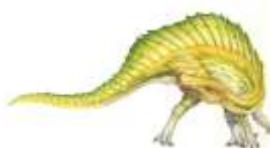
- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected





# Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.





# Allocation of Frames

---

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***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  $p_i = \frac{s_i}{S} \times m$

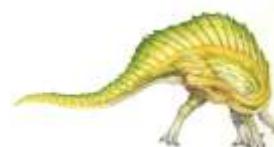
$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

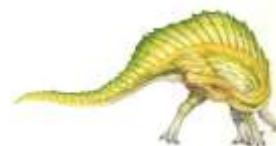
$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$





# 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





# Reclaiming Pages

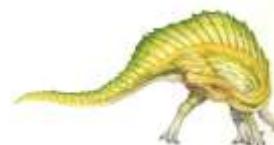
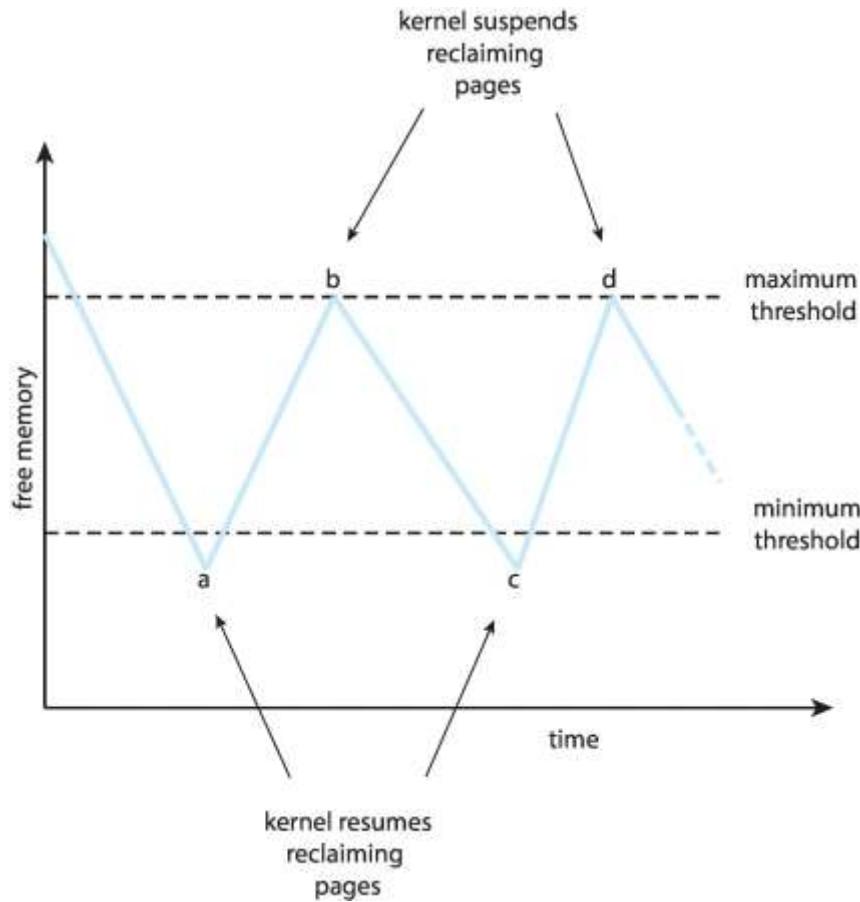
---

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.





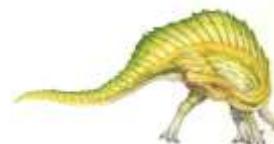
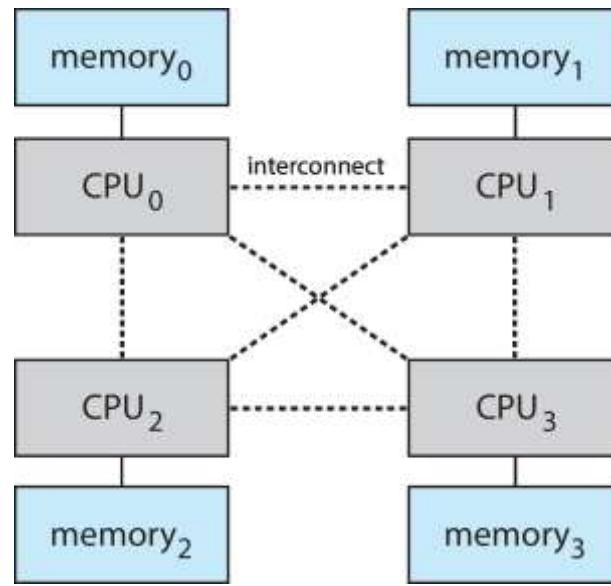
# Reclaiming Pages Example





# Non-Uniform Memory Access

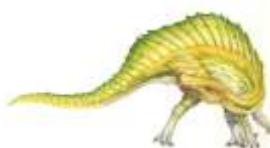
- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture





# Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **Igroups**
    - ▶ Structure to track CPU / Memory low latency groups
    - ▶ Used my schedule and pager
    - ▶ When possible schedule all threads of a process and allocate all memory for that process within the Igroup





# Thrashing

---

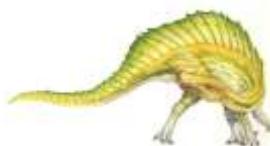
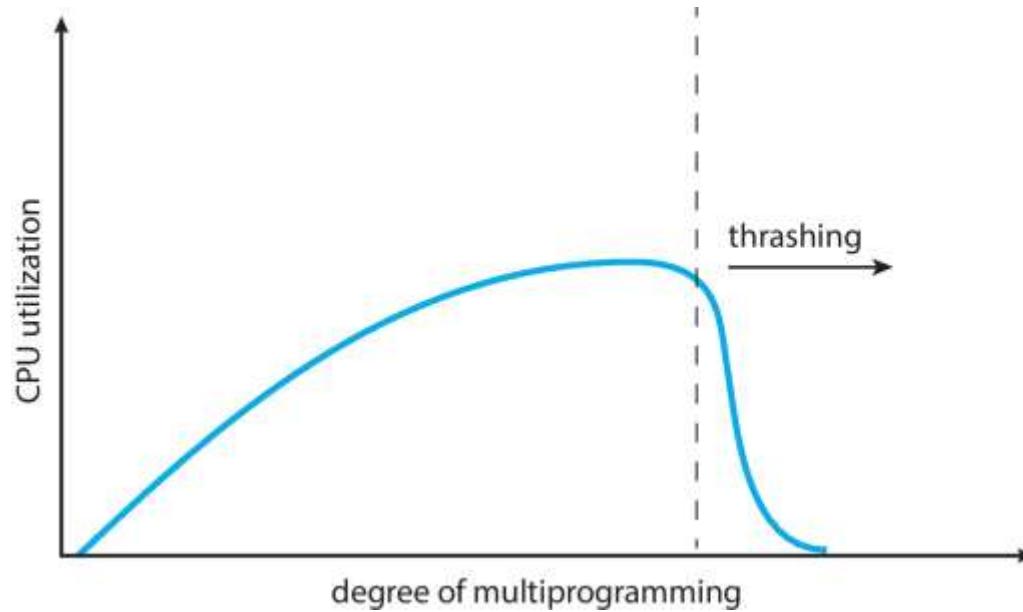
- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - ▶ Low CPU utilization
    - ▶ Operating system thinking that it needs to increase the degree of multiprogramming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





# Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out





# Demand Paging and Thrashing

- Why does demand paging work?

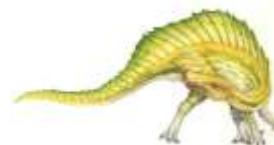
## Locality model

- Process migrates from one locality to another
- Localities may overlap

- Why does thrashing occur?

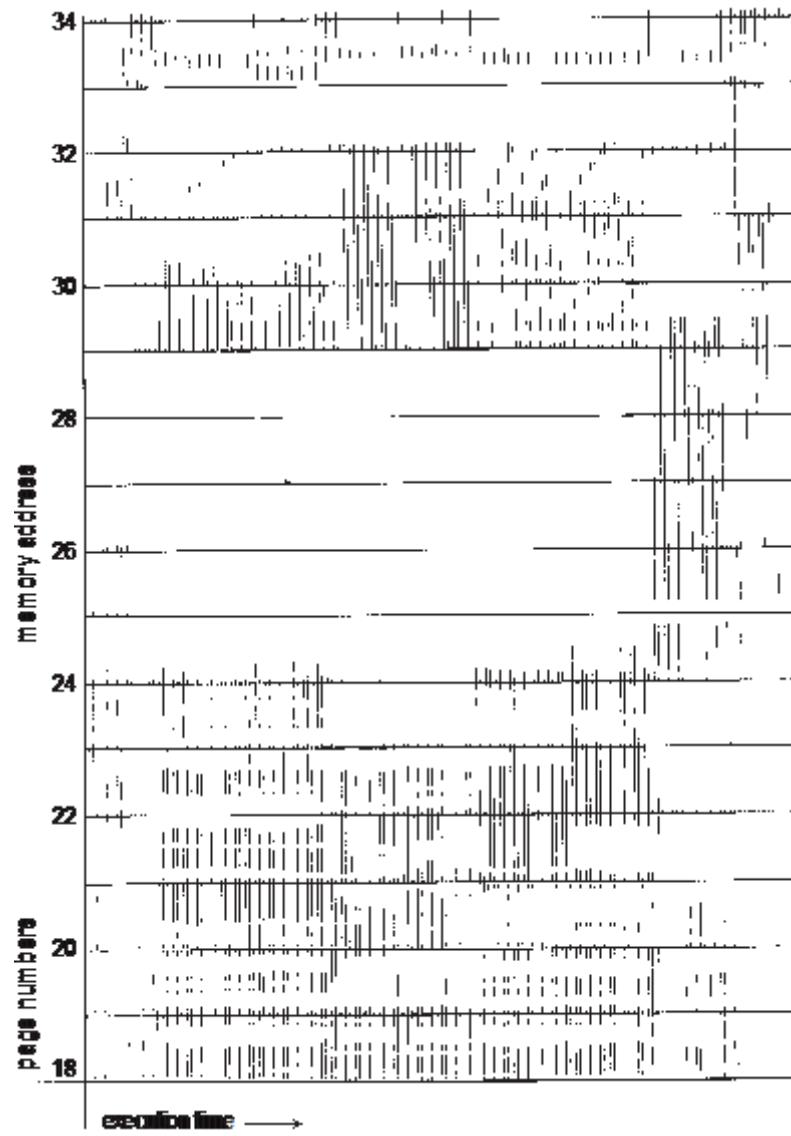
$\Sigma$  size of locality > total memory size

- Limit effects by using local or priority page replacement





# Locality In A Memory-Reference Pattern





# 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  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality



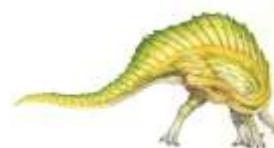
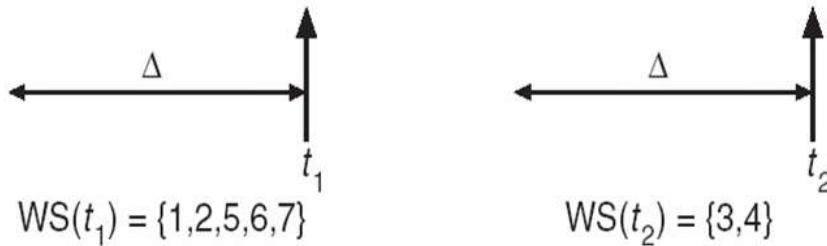


# Working-Set Model (Cont.)

- if  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...





# Keeping Track of the Working Set

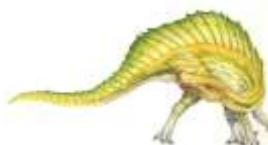
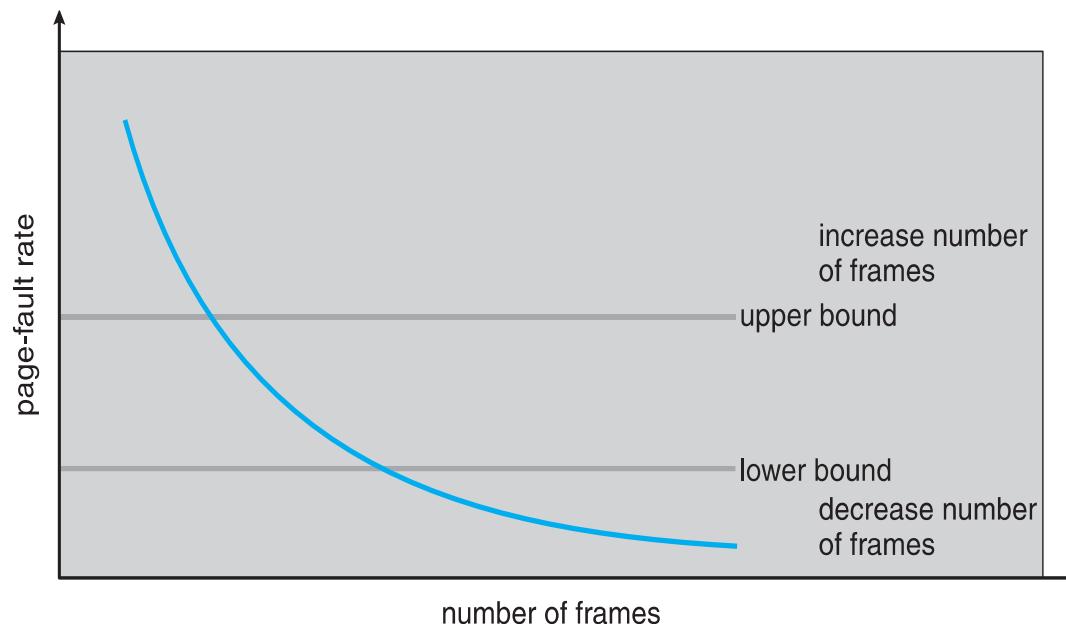
- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units





# Page-Fault Frequency

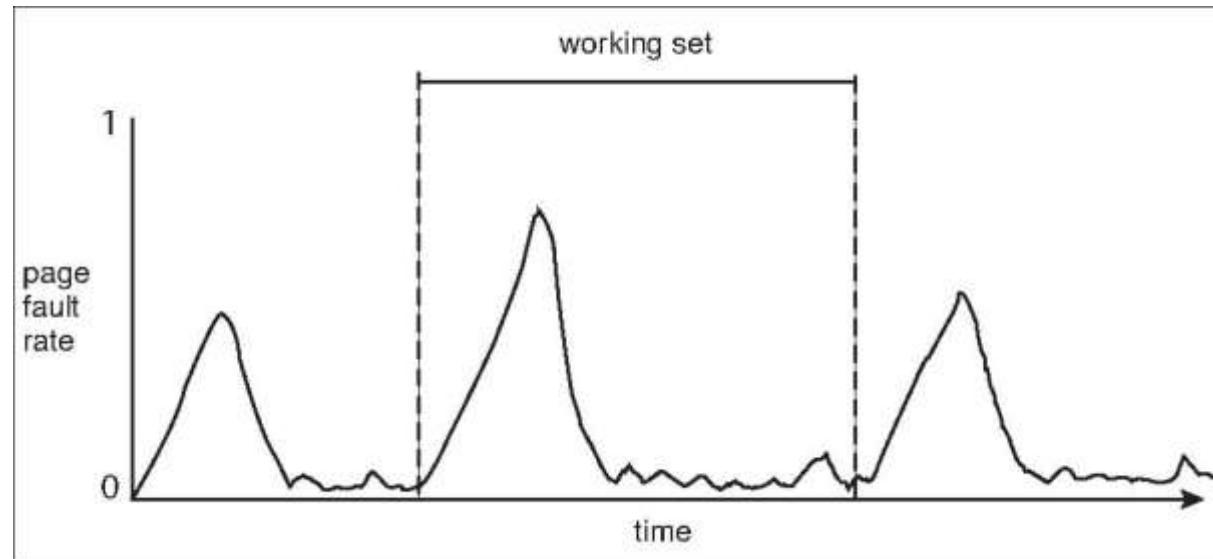
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame





# Working Sets and Page Fault Rates

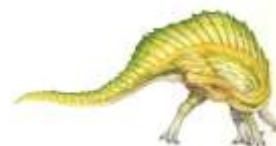
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time





# Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - ▶ i.e., for device I/O





# Buddy System

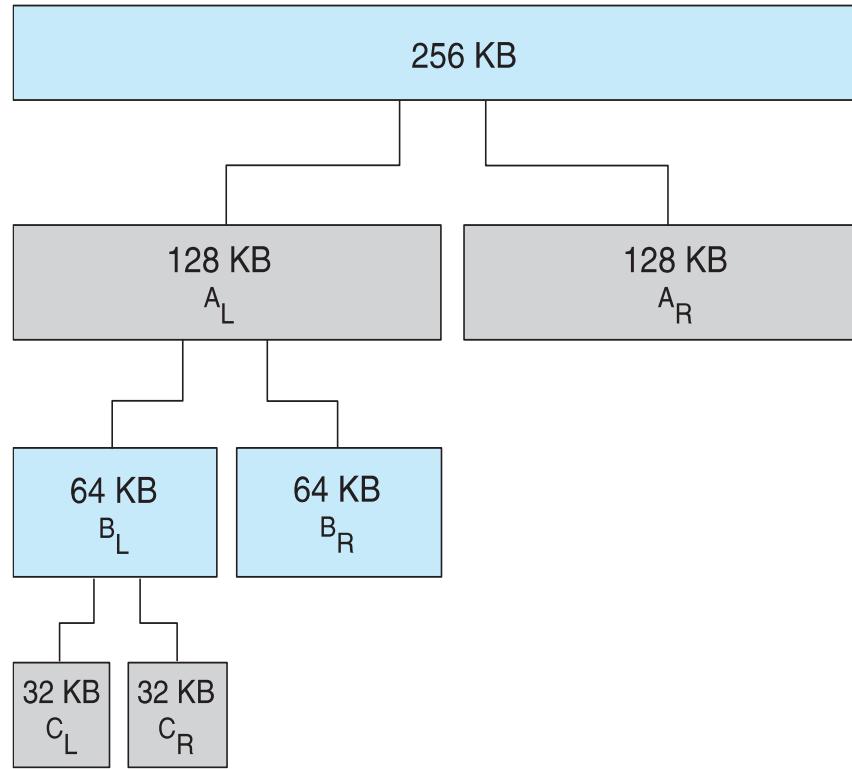
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - ▶ Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into  $A_L$  and  $A_R$  of 128KB each
    - ▶ One further divided into  $B_L$  and  $B_R$  of 64KB
      - One further into  $C_L$  and  $C_R$  of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation





# Buddy System Allocator

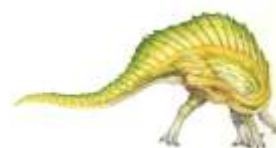
physically contiguous pages





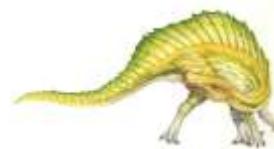
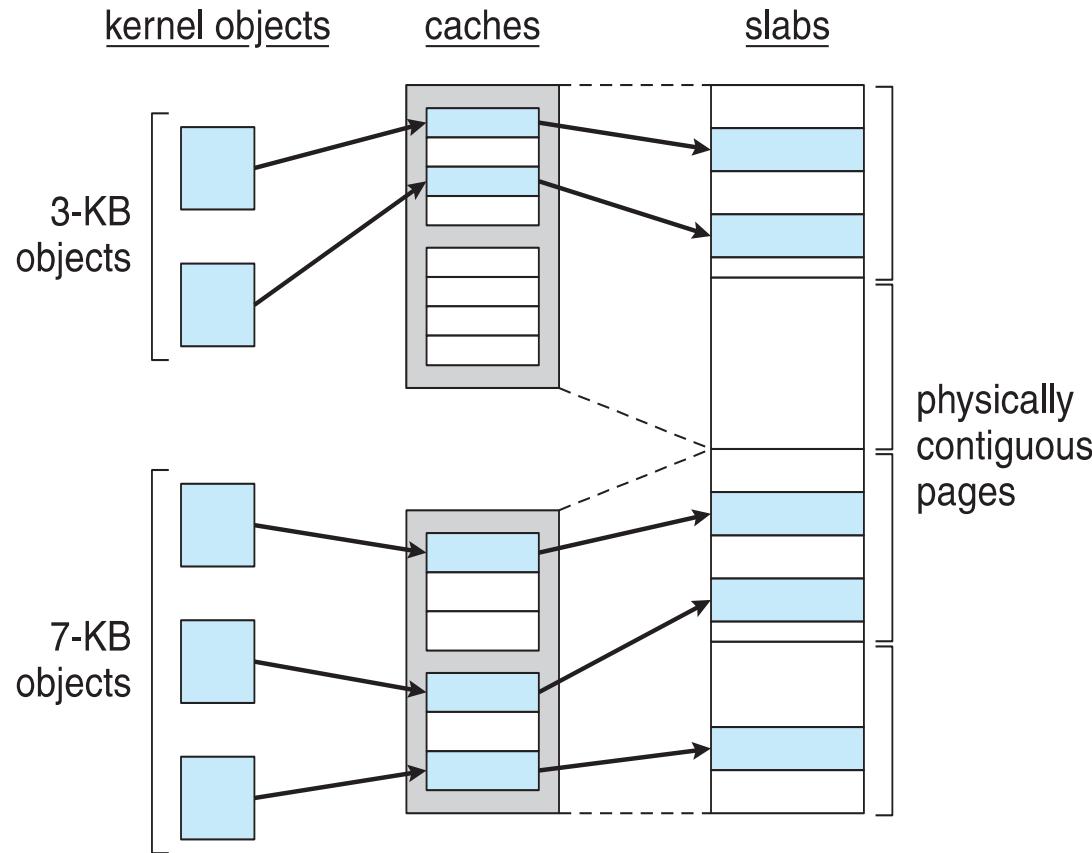
# Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction





# Slab Allocation





# Slab Allocator in Linux

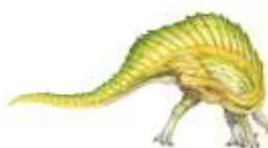
- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty





# Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - ▶ Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

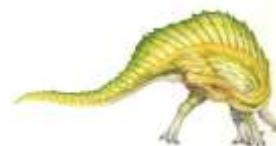




# Other Considerations

---

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking





# Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume  $s$  pages are prepaged and  $\alpha$  of the pages is used
  - Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  than the cost of prepaging  $s * (1 - \alpha)$  unnecessary pages?
  - $\alpha$  near zero  $\Rightarrow$  prepaging loses





# Page Size

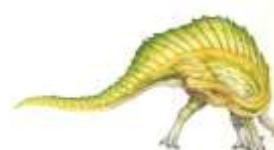
- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range  $2^{12}$  (4,096 bytes) to  $2^{22}$  (4,194,304 bytes)
- On average, growing over time





# TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





# Program Structure

- Program structure

- int[128,128] data;
- Each row is stored in one page
- Program 1

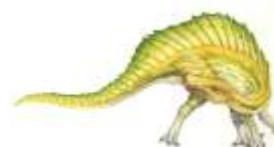
```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

$128 \times 128 = 16,384$  page faults

- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

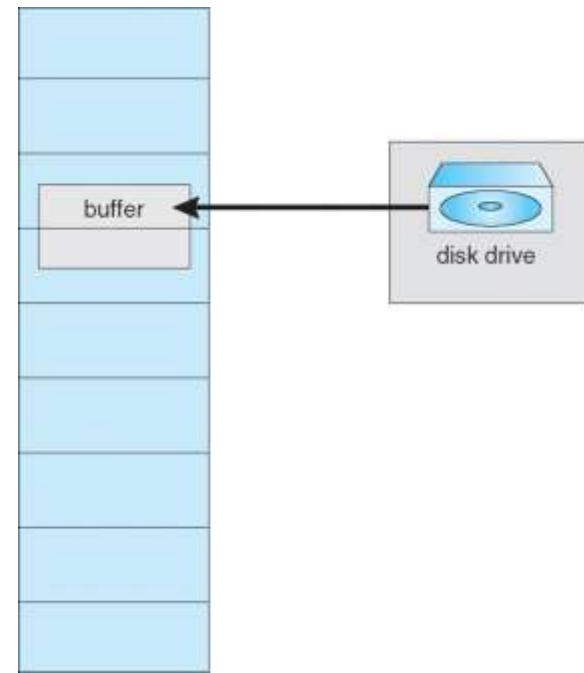
128 page faults





# I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory

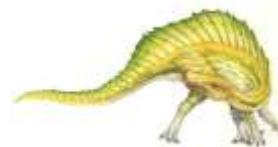




# Operating System Examples

---

- Windows
- Solaris

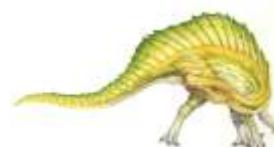




# Windows

---

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum





# Solaris

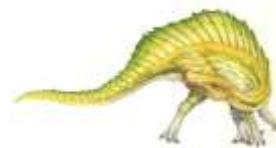
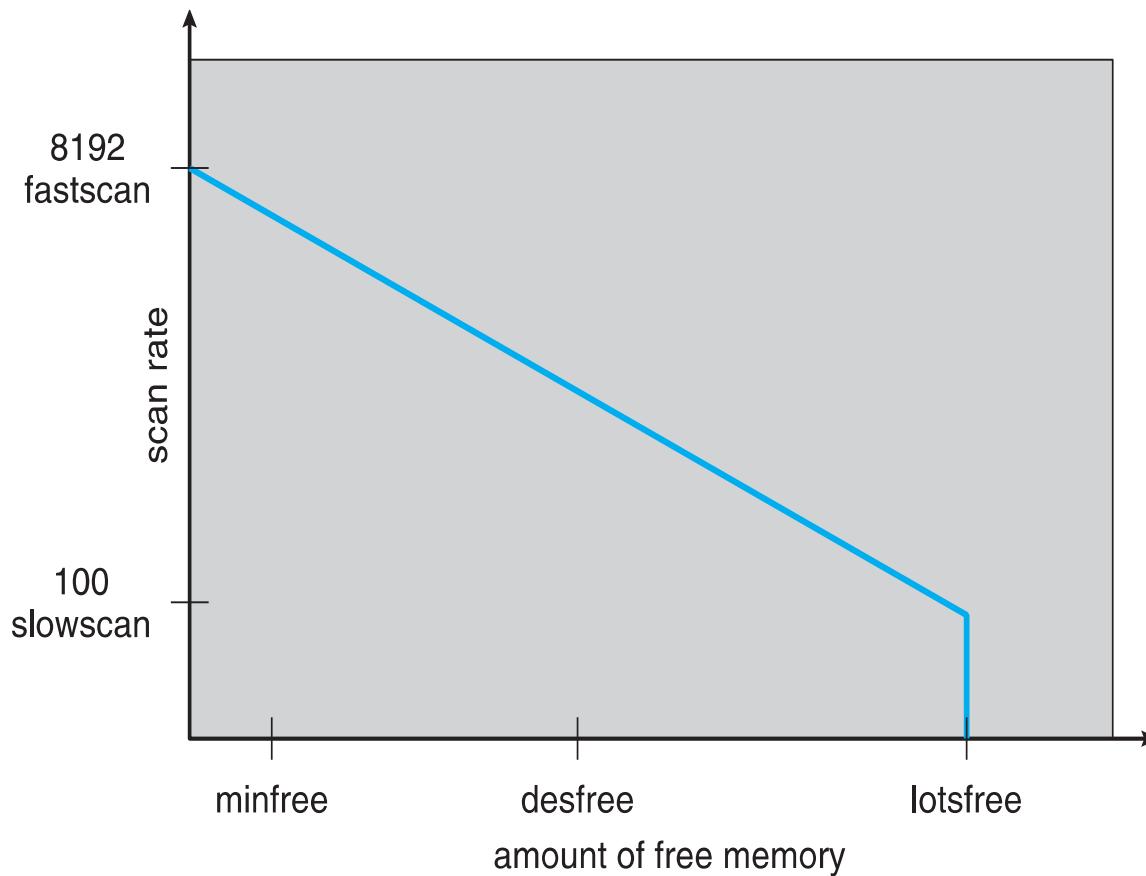
---

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to begin swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

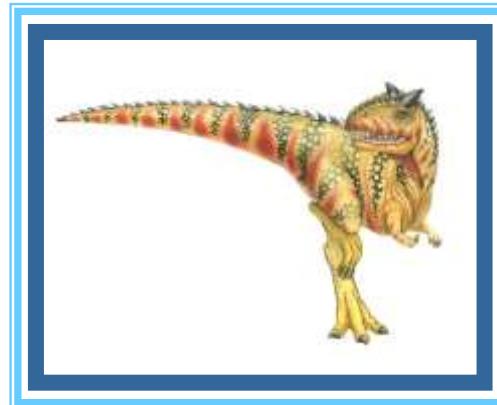




# Solaris 2 Page Scanner



# End of Chapter 10





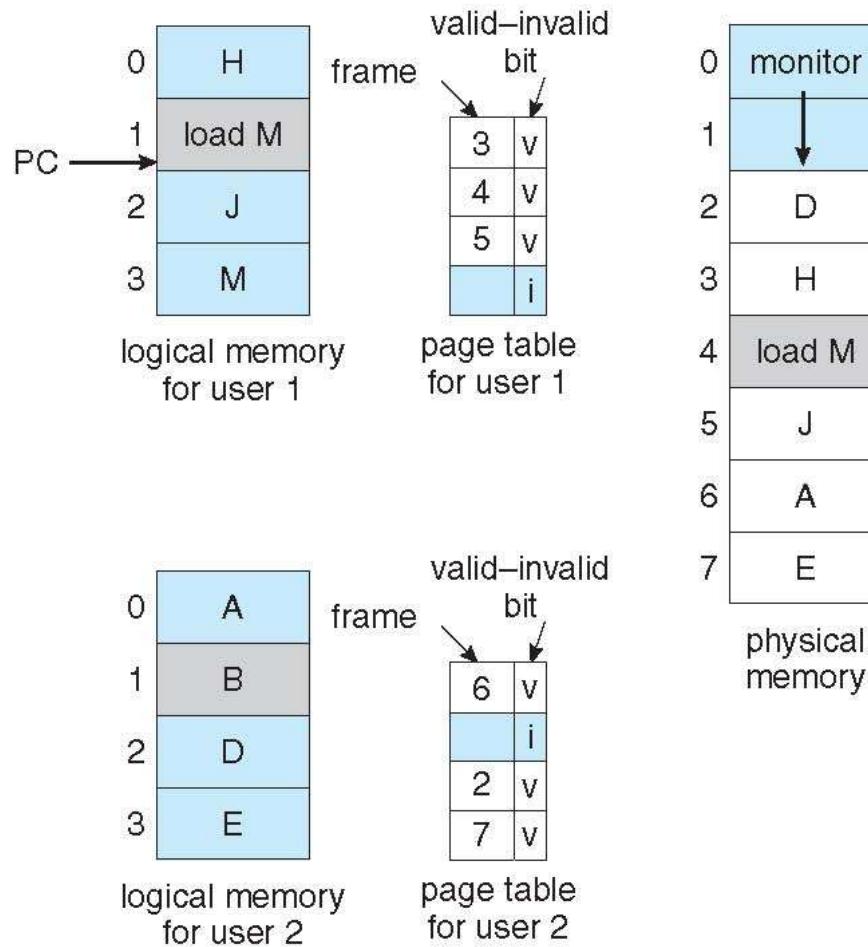
# Performance of Demand Paging

- Stages in Demand Paging (worse case)
  1. Trap to the operating system
  2. Save the user registers and process state
  3. Determine that the interrupt was a page fault
  4. Check that the page reference was legal and determine the location of the page on the disk
  5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
  6. While waiting, allocate the CPU to some other user
  7. Receive an interrupt from the disk I/O subsystem (I/O completed)
  8. Save the registers and process state for the other user
  9. Determine that the interrupt was from the disk
  10. Correct the page table and other tables to show page is now in memory
  11. Wait for the CPU to be allocated to this process again
  12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





# Need For Page Replacement





# 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





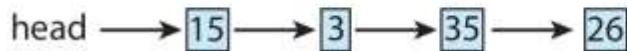
# Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames

free-frame list



modified frame list



- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames—say, three—and store their compressed versions in a single page frame.

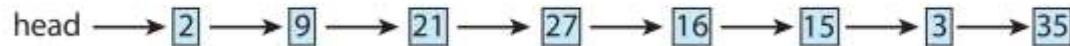




# Memory Compression (Cont.)

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

free-frame list



modified frame list

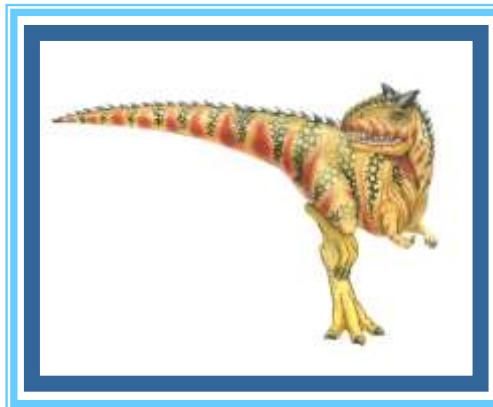


compressed frame list



# Chapter 13:

# File-System Interface

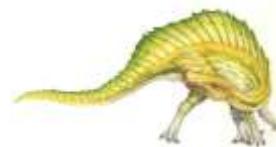




# Outline

---

- File Concept
- Access Methods
- Disk and Directory Structure
- Protection
- Memory-Mapped Files

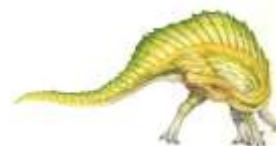




# Objectives

---

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection





# File Concept

- Contiguous logical address space
- Types:
  - Data
    - ▶ Numeric
    - ▶ Character
    - ▶ Binary
  - Program
- Contents defined by file's creator
  - Many types
    - ▶ **text file**,
    - ▶ **source file**,
    - ▶ **executable file**

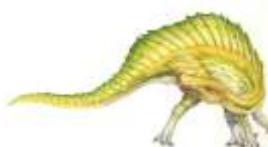




# File Attributes

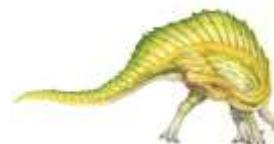
---

- **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 device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure





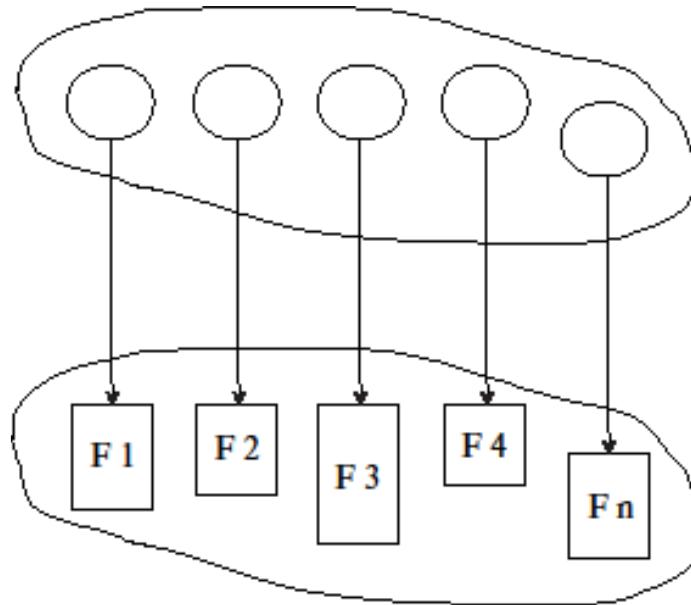
# File info Window on Mac OS X



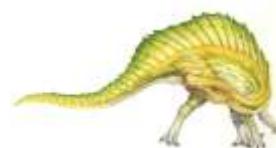


# Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk





# 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 entry  $F_i$ , and move the content of entry to memory
- Close ( $F_i$ ) – move the content of entry  $F_i$  in memory to directory structure on disk





# Open Files

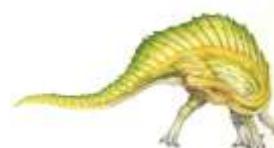
- Several pieces of data are needed to manage open files:
  - **Open-file table:** tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count:** counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information





# File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

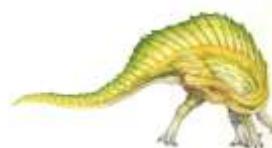




# File Locking Example – Java API

---

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
        }
    }
}
```





# File Locking Example – Java API (Cont.)

```
// this locks the second half of the file - shared  
sharedLock = ch.lock(raf.length()/2+1, raf.length(),  
                      SHARED);  
/** Now read the data . . . */  
// release the lock  
sharedLock.release();  
} catch (java.io.IOException ioe) {  
    System.err.println(ioe);  
}finally {  
    if (exclusiveLock != null)  
        exclusiveLock.release();  
    if (sharedLock != null)  
        sharedLock.release();  
}  
}  
}
```





# File Types – Name, Extension

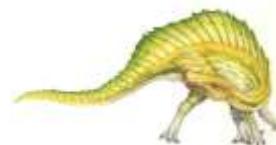
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information





# File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

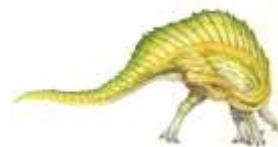




# Access Methods

---

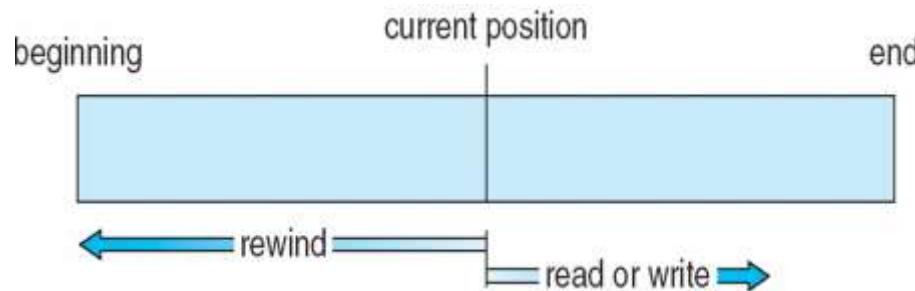
- A file is fixed length **logical records**
- **Sequential Access**
- **Direct Access**
- **Other Access Methods**





# Sequential Access

- Operations
  - **read next**
  - **write next**
  - **Reset**
  - no read after last write (rewrite)
- Figure





# Direct Access

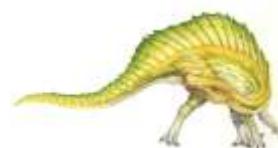
- Operations
  - **read *n***
  - **write *n***
  - **position to *n***
    - ▶ **read next**
    - ▶ **write next**
    - ▶ **rewrite *n***
- $n$  = **relative block number**
- Relative block numbers allow OS to decide where file should be placed





# Simulation of Sequential Access on Direct-access File

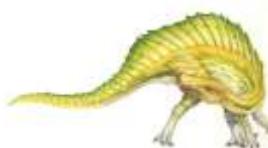
sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$





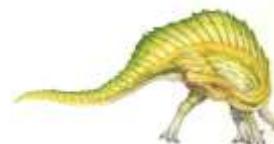
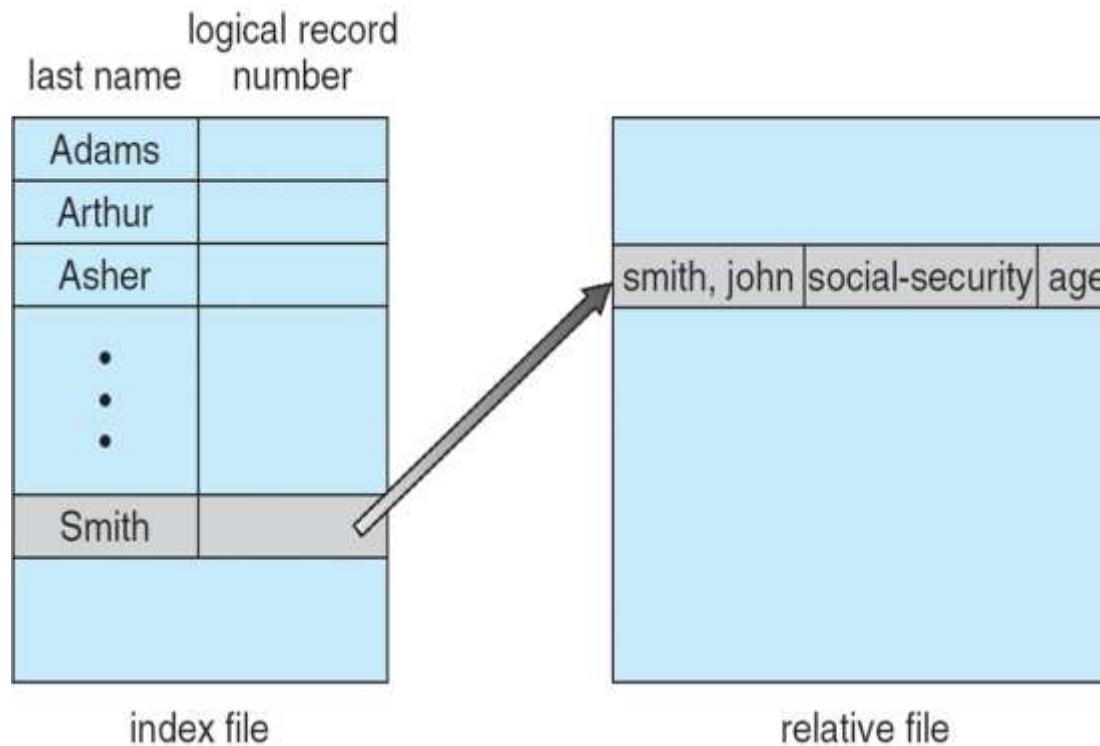
# Other Access Methods

- Can be other access methods built on top of base methods
- General involve creation of an **index** for the file
- Keep index in memory for fast determination of location of data to be operated on (consider Universal Produce Code (UPC code) plus record of data about that item)
- If the index is too large, create an in-memory index, which is an index of a disk index
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)





# Example of Index and Relative Files

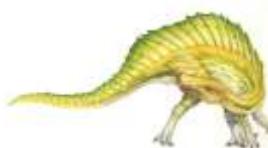




# Disk Structure

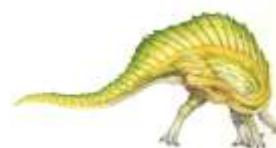
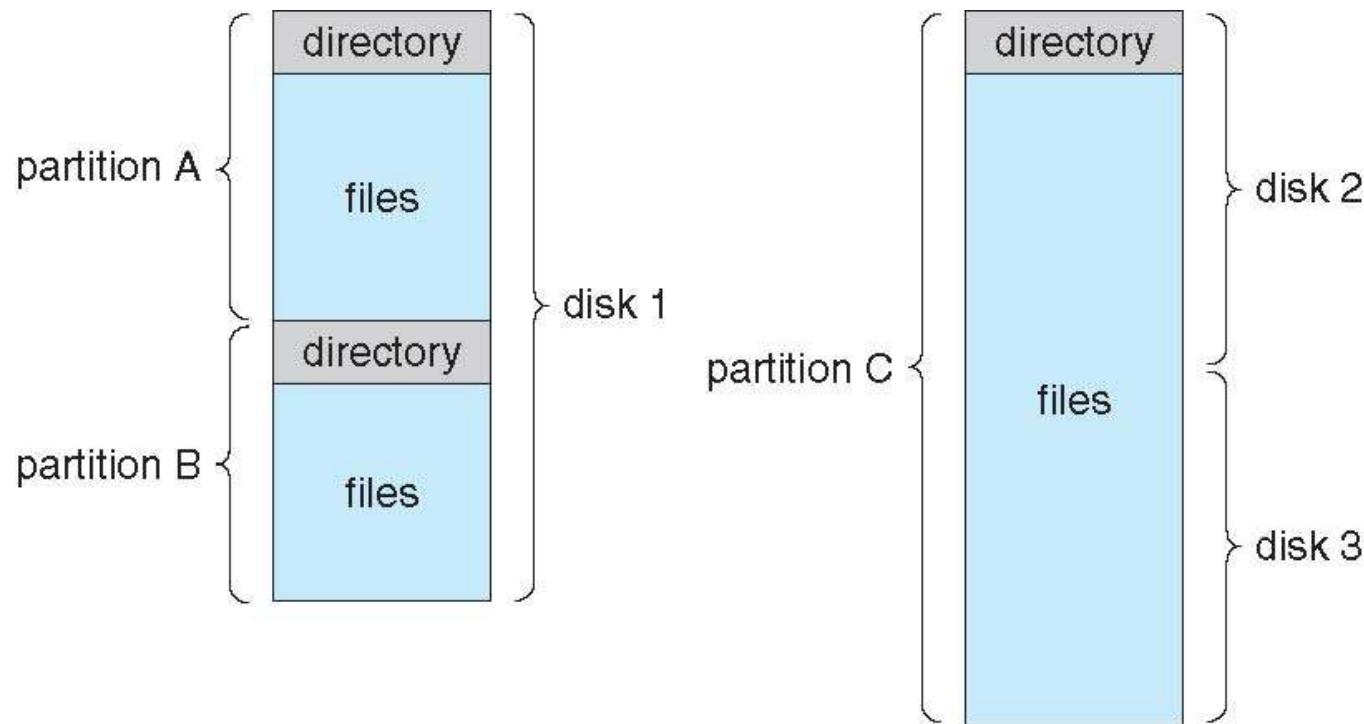
---

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer





# A Typical File-system Organization





# Types of File Systems

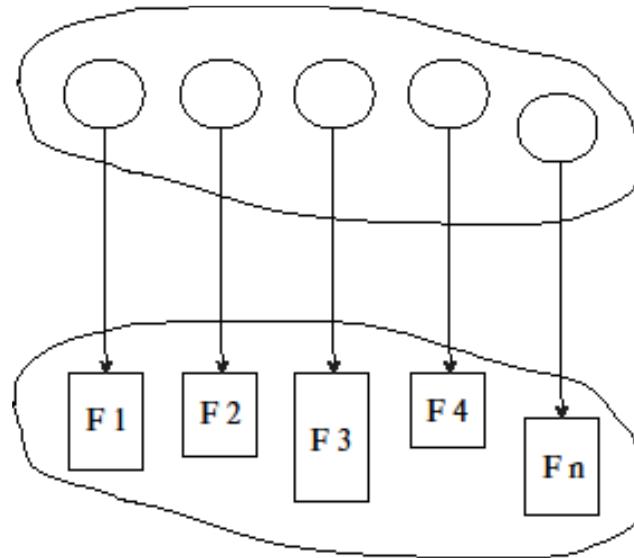
- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems



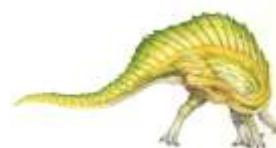


# Directory Structure

- A collection of nodes containing information about all files



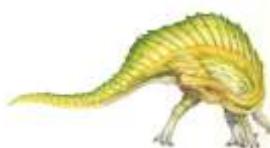
- Both the directory structure and the files reside on disk





# Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system





# Directory Organization

The directory is organized logically to obtain

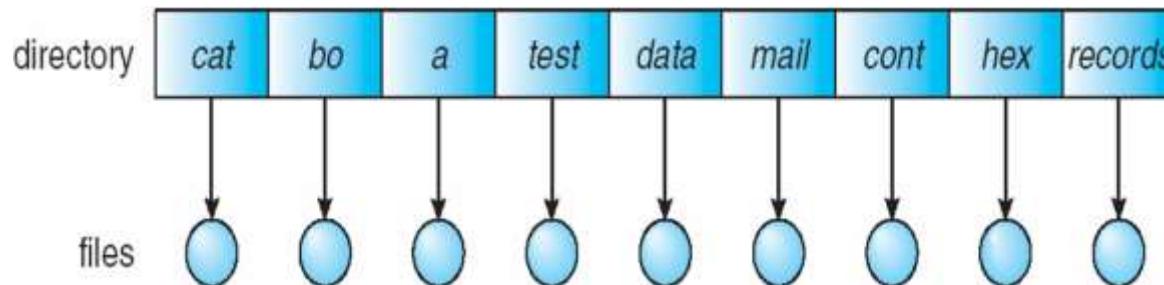
- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





# Single-Level Directory

- A single directory for all users



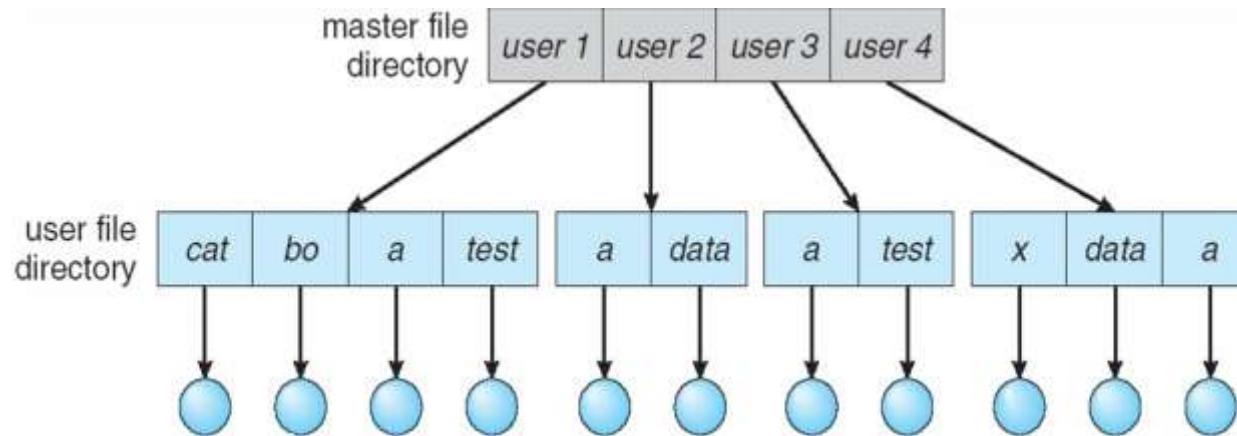
- Naming problem
- Grouping problem



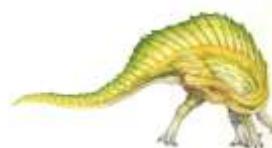


# Two-Level Directory

- Separate directory for each user

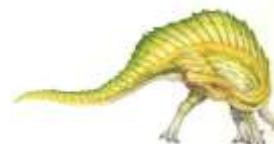
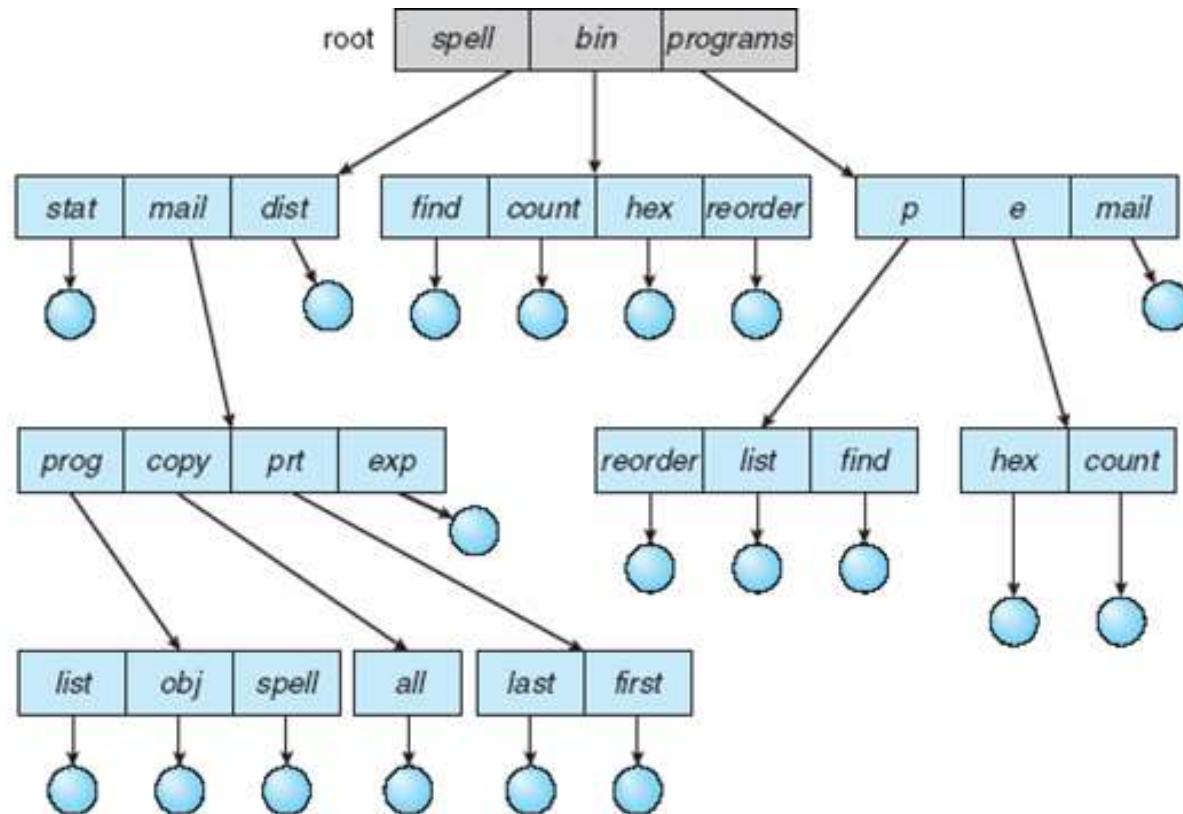


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





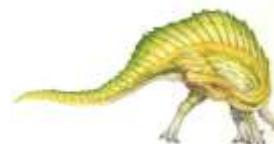
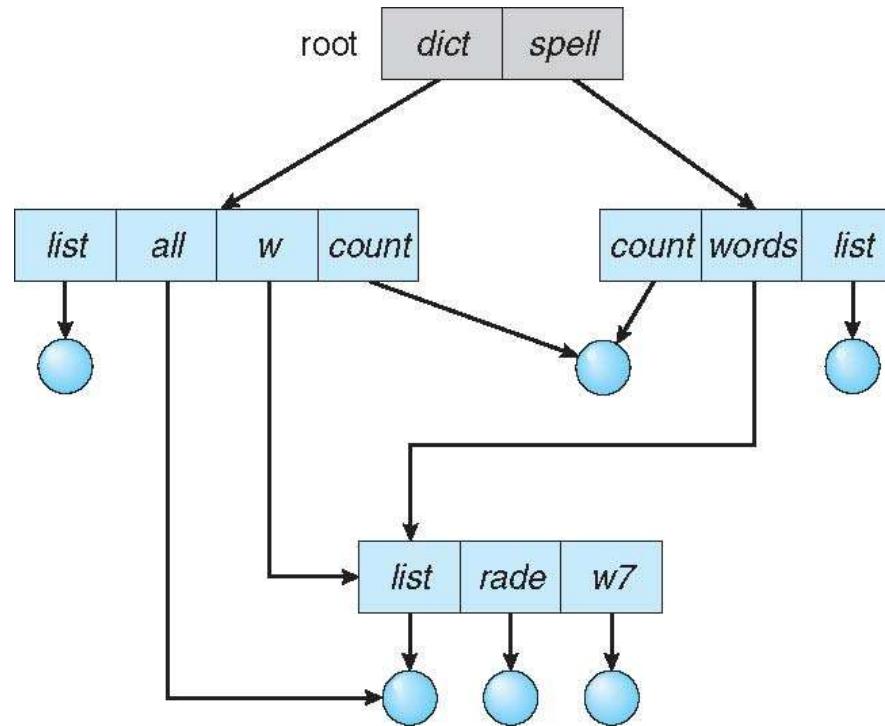
# Tree-Structured Directories





# Acyclic-Graph Directories

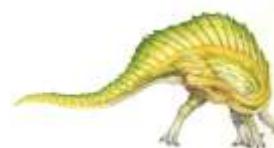
- Have shared subdirectories and files
- Example





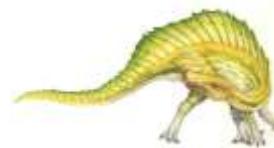
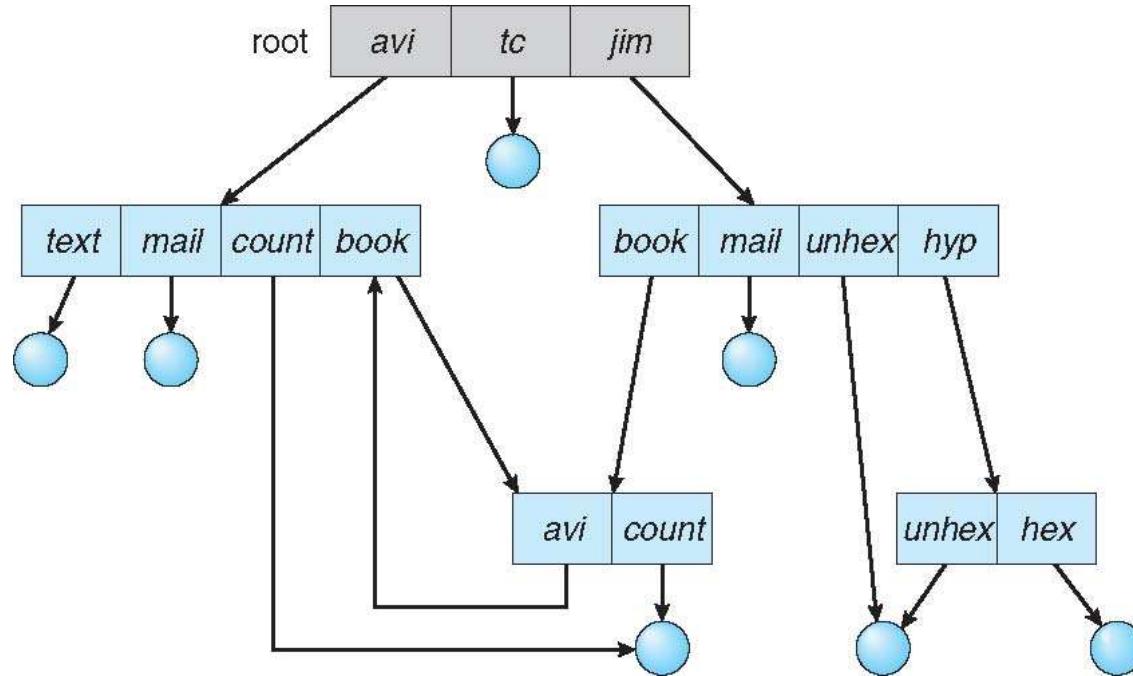
# Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If **dict** deletes **w/list**  $\Rightarrow$  dangling pointer
  - Solutions:
    - Backpointers, so we can delete all pointers.
      - ▶ Variable size records a problem
    - Backpointers using a daisy chain organization
    - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file





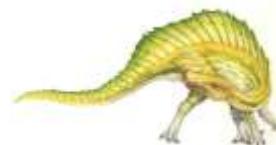
# General Graph Directory





# General Graph Directory (Cont.)

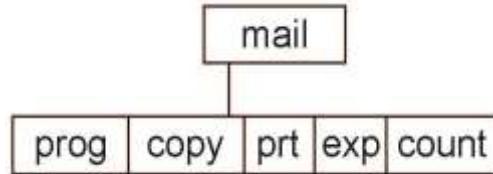
- How do we guarantee no cycles?
  - Allow only links to files not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK





# Current Directory

- Can designate one of the directories as the current (working) directory
  - `cd /spell/mail/prog`
  - `type list`
- Creating and deleting a file is done in current directory
- Example of creating a new file
  - If in current directory is `/mail`
  - The command  
**`mkdir <dir-name>`**
  - Results in:



- Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

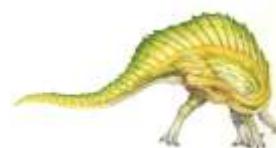




# Protection

---

- File owner/creator should be able to control:
  - What can be done
  - By whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**



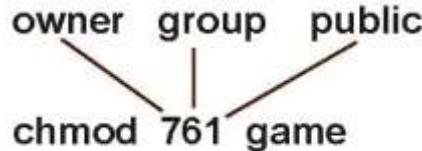


# Access Lists and Groups in Unix

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) <b>owner access</b>	7	⇒	1 1 1
b) <b>group access</b>	6	⇒	1 1 0
c) <b>public access</b>	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a file (say *game*) or subdirectory, define an appropriate access.



- Attach a group to a file

**chgrp      G      game**





# A Sample UNIX Directory Listing

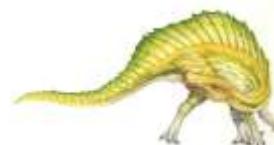
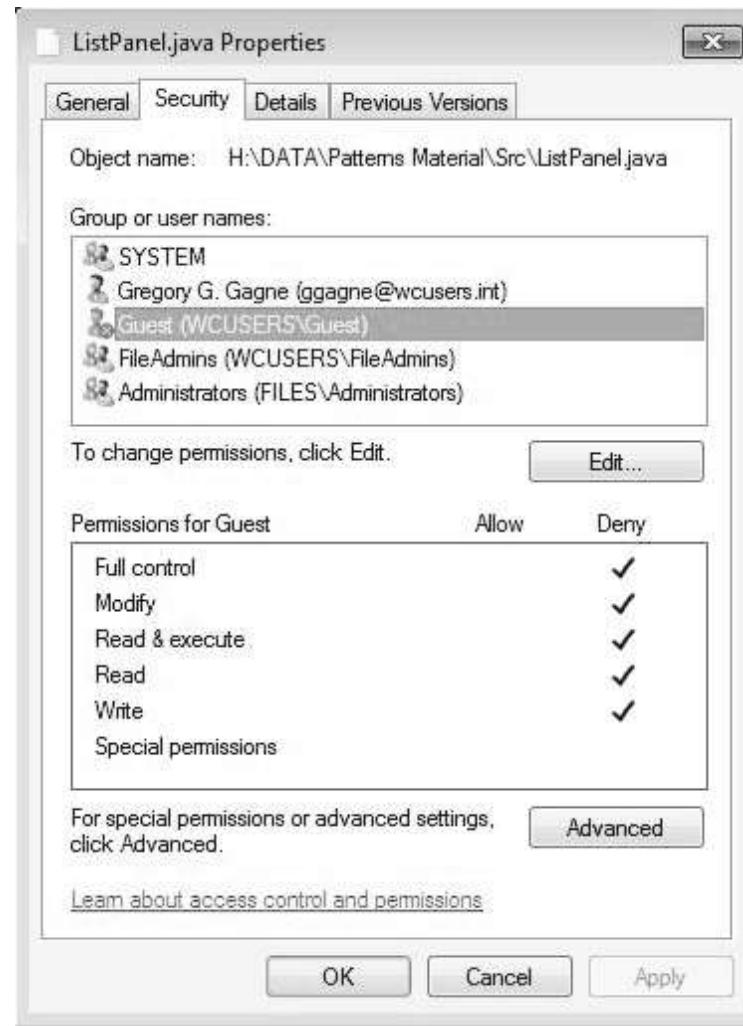
---

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/



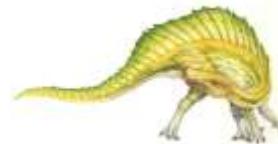


# Windows 7 Access-Control List Management

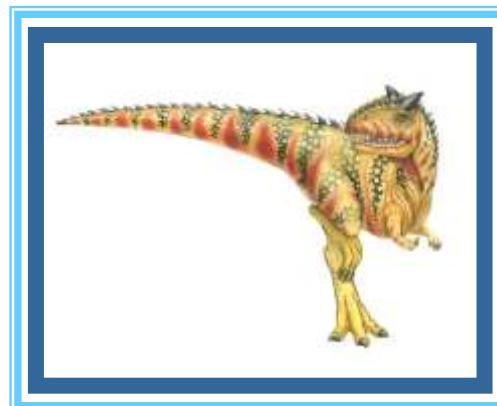




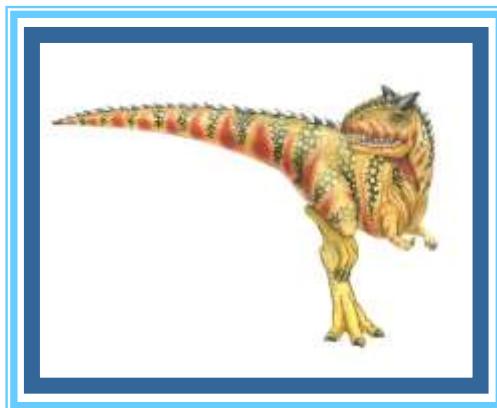
# Memory-Mapped Files



# End of Chapter 13



# Chapter 14: File System Implementation





# Outline

---

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

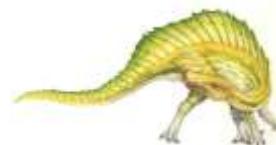




# Objectives

---

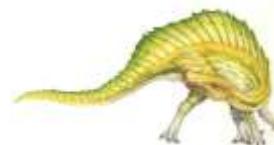
- Describe the details of implementing local file systems and directory structures
- Discuss block allocation and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Look at recovery from file system failures
- Describe the WAFL file system as a concrete example





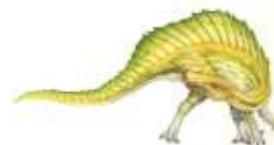
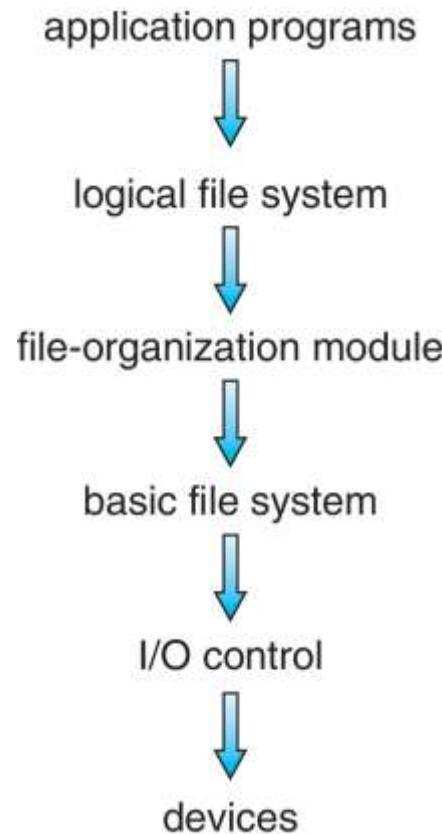
# File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block (FCB)** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers





# Layered File System

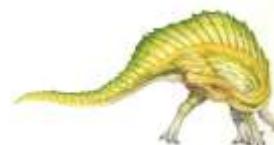




# File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like

read drive1, cylinder 72, track 2, sector 10, into memory location 1060
  - Outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation





# File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer





# File System Layers (Cont.)

- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE





# File-System Operations

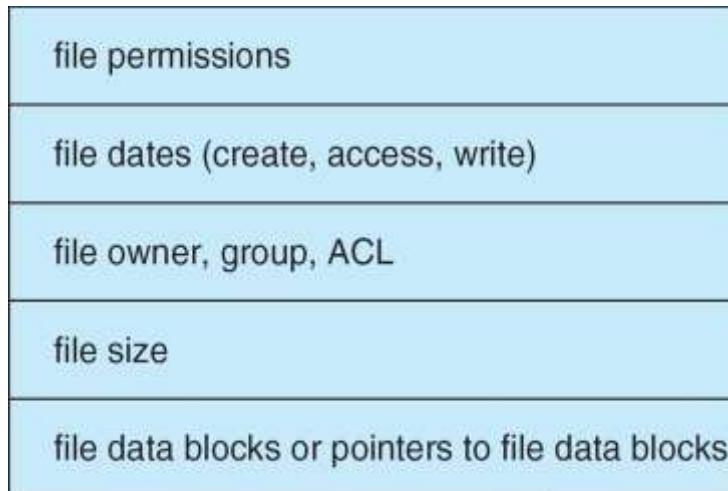
- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table





# File Control Block (FCB)

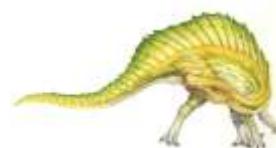
- OS maintains **FCB** per file, which contains many details about the file
  - Typically, inode number, permissions, size, dates
  - Example





# In-Memory File System Structures

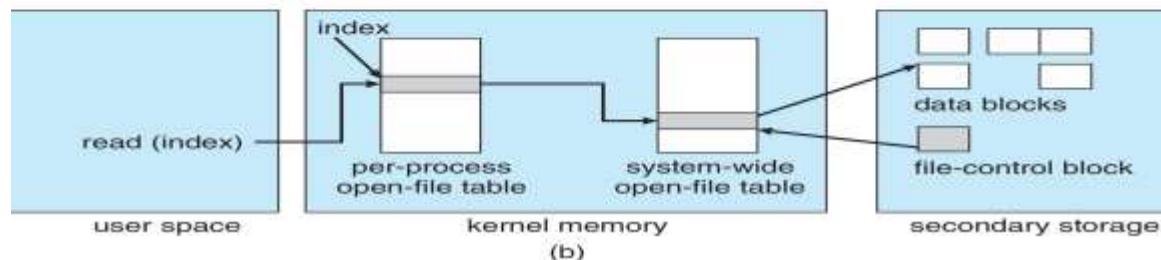
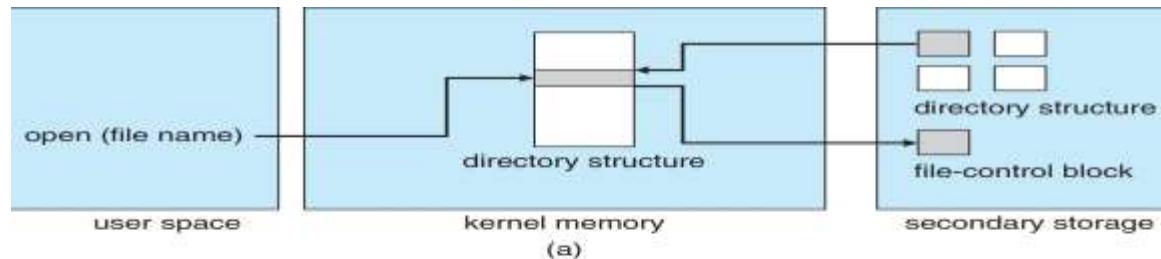
- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info





# In-Memory File System Structures (Cont.)

- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file

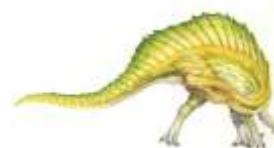




# Directory Implementation

---

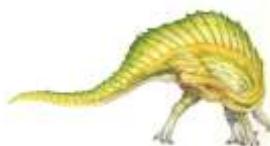
- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - ▶ Linear search time
    - ▶ Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method





# Allocation Method

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - File Allocation Table (FAT)





# Contiguous Allocation Method

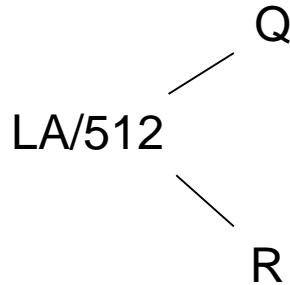
- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - Finding space on the disk for a file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line (downtime)** or **on-line**



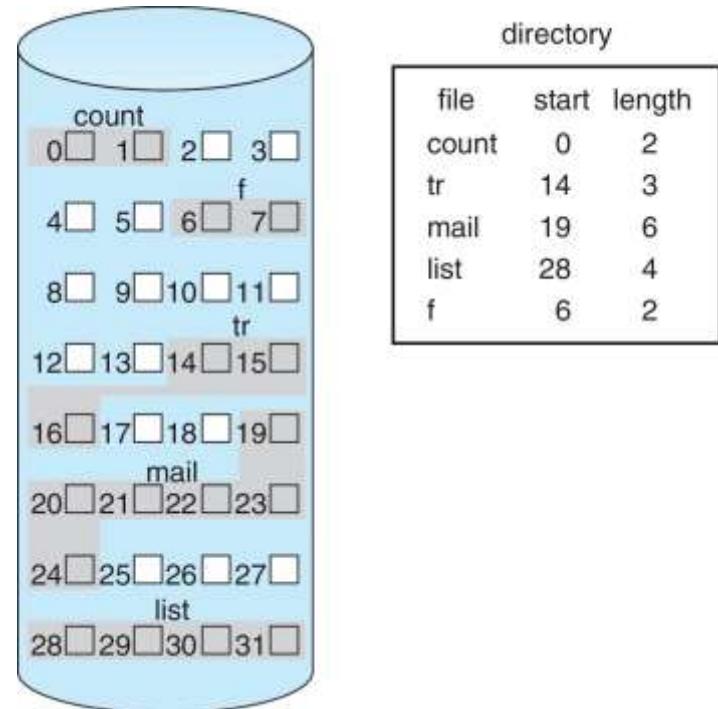


# Contiguous Allocation (Cont.)

- Mapping from logical to physical  
(block size = 512 bytes)



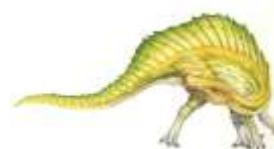
- Block to be accessed = starting address +  $Q$
- Displacement into block =  $R$





# Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents





# Linked Allocation

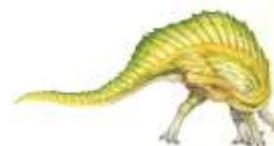
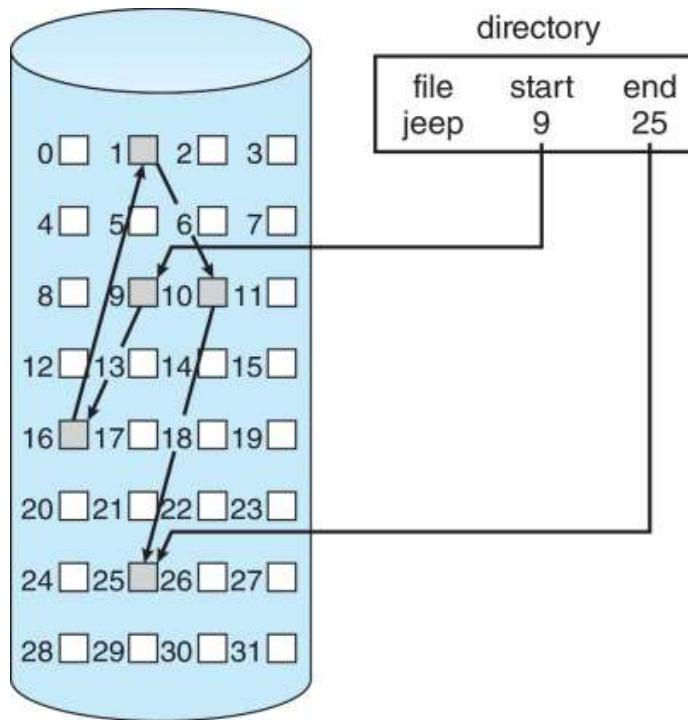
- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks





# Linked Allocation Example

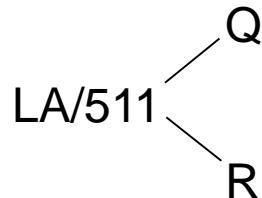
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme





# Linked Allocation (Cont.)

- Mapping



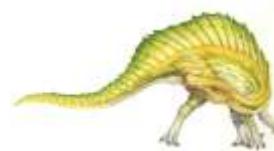
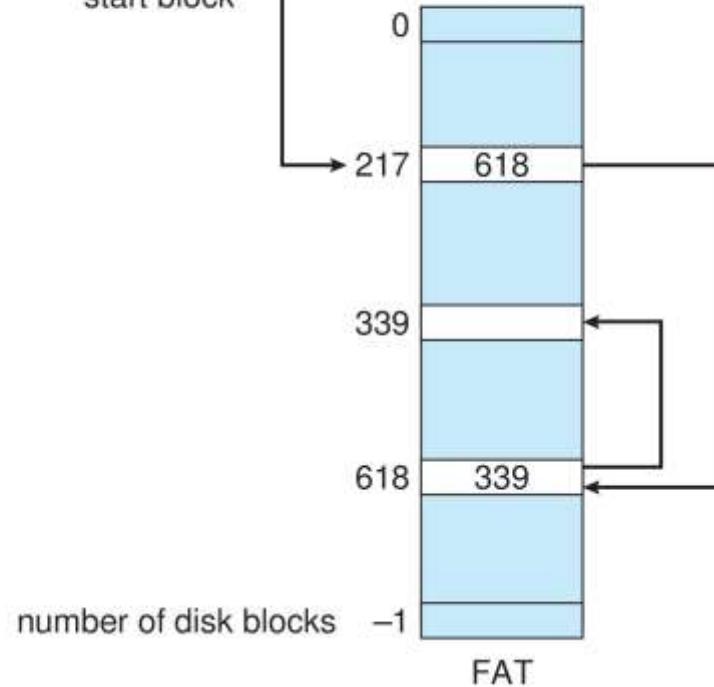
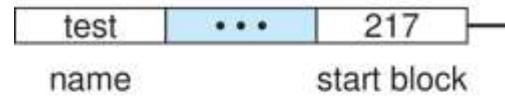
- Block to be accessed is the  $Q^{\text{th}}$  block in the linked chain of blocks representing the file.
- Displacement into block =  $R + 1$





# File-Allocation Table

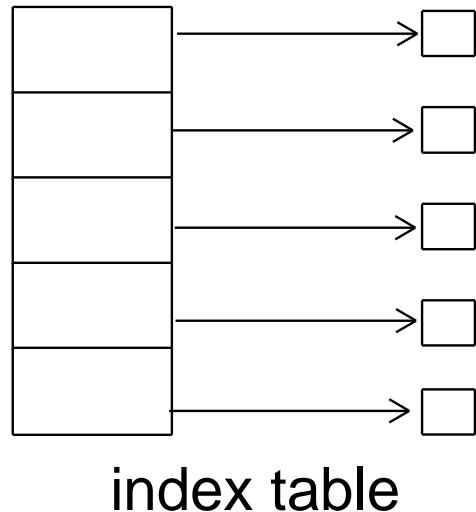
directory entry





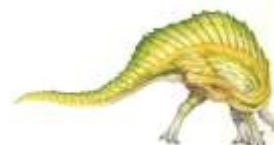
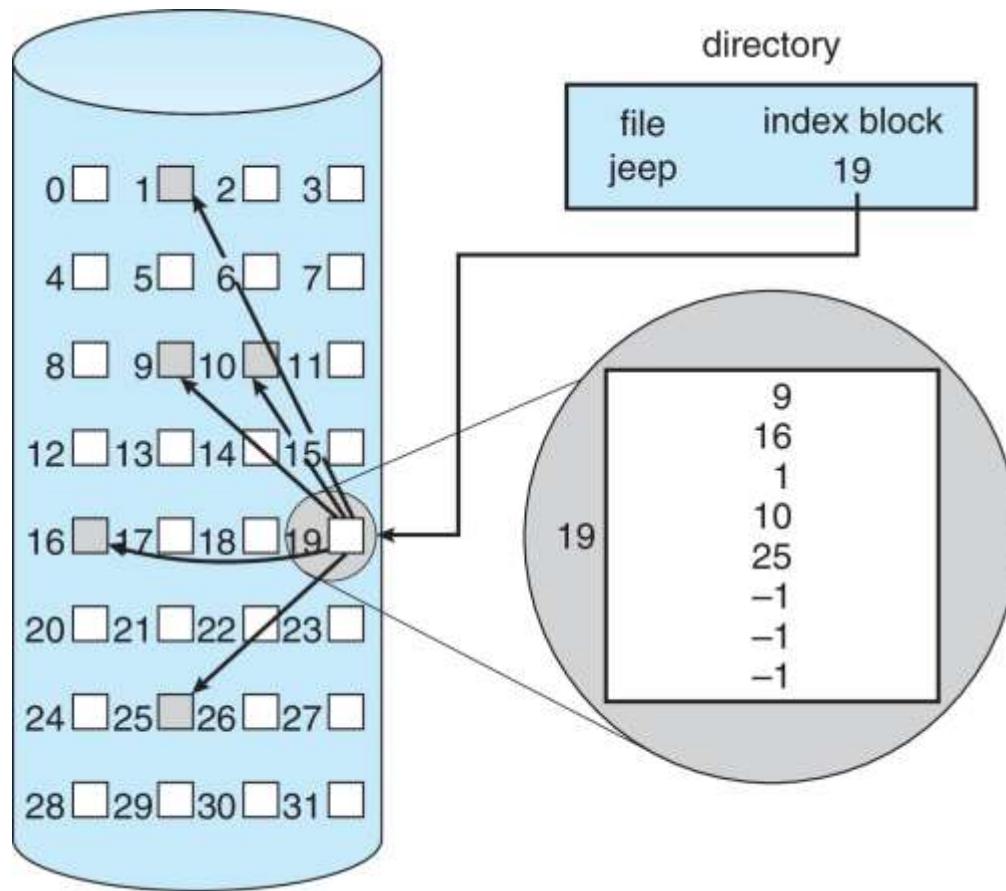
# Indexed Allocation Method

- Each file has its own **index block**(s) of pointers to its data blocks
- Logical view





# Example of Indexed Allocation





# Performance

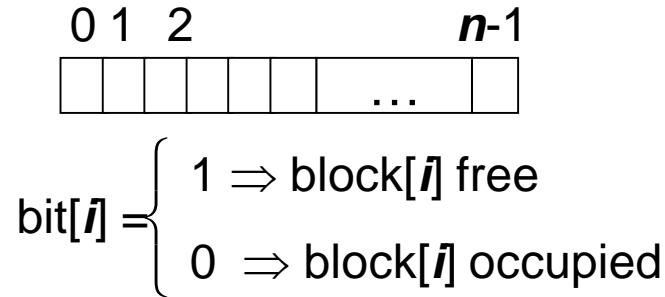
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
  - Select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - Goal is to reduce CPU cycles and overall path needed for I/O





# Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term “block” for simplicity)
- Bit vector** or **bit map** ( $n$  blocks)



Block number calculation

$$\begin{aligned} & (\text{number of bits per word}) * \\ & (\text{number of 0-value words}) + \\ & \text{offset of first 1 bit} \end{aligned}$$

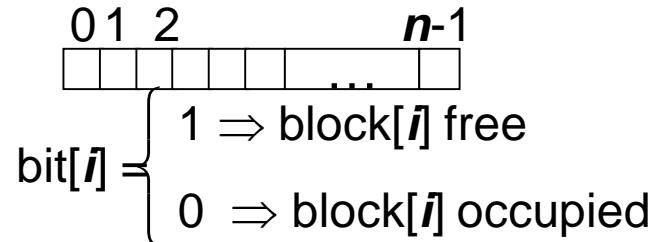
CPUs have instructions to return offset within word of first “1” bit





# Free-Space Management

- File system maintains **free-space list** to track available blocks
- Bit vector** or **bit map** ( $n$  blocks)



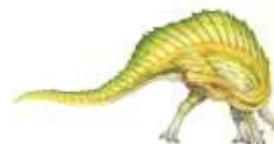
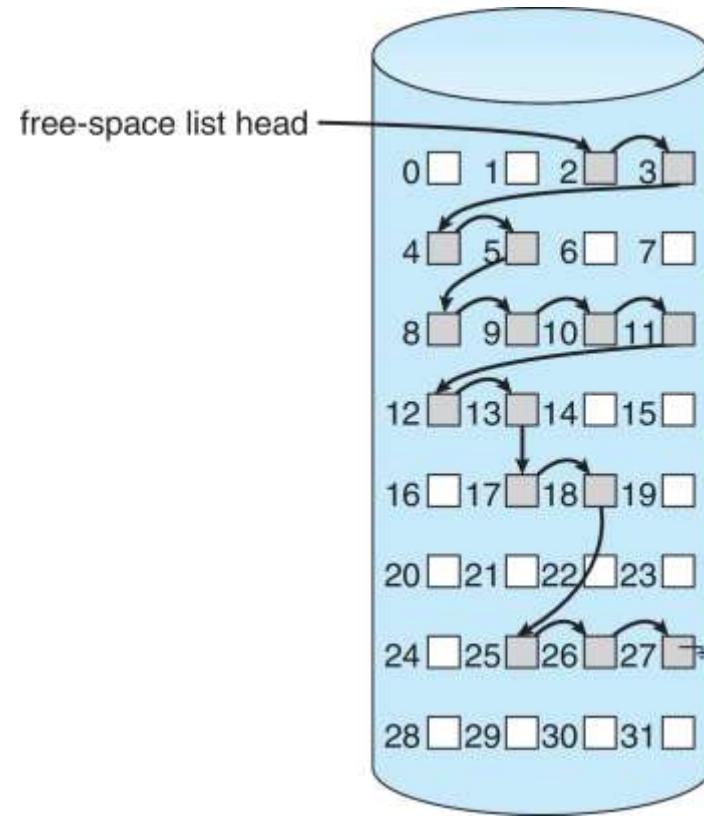
- Bit map requires extra space
  - Example:
    - block size = 4KB =  $2^{12}$  bytes
    - disk size =  $2^{40}$  bytes (1 terabyte)
    - $n = 2^{40}/2^{12} = 2^{28}$  bits (or 32MB)
    - if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files





# Linked Free Space List on Disk

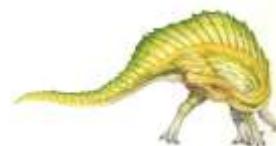
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste. Linked Free Space List on Disk of space
  - No need to traverse the entire list (if # free blocks recorded)





# Free-Space Management (Cont.)

- Grouping
  - Modify linked list to store address of next  $n-1$  free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - ▶ Keep address of first free block and count of following free blocks
    - ▶ Free space list then has entries containing addresses and counts





# Free-Space Management (Cont.)

- Space Maps
  - Used in **ZFS**
  - Consider meta-data I/O on very large file systems
    - ▶ Full data structures like bit maps cannot fit in memory → thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - ▶ Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - ▶ Uses counting algorithm
  - But records to log file rather than file system
    - ▶ Log of all block activity, in time order, in counting format
  - Metaslab activity → load space map into memory in balanced-tree structure, indexed by offset
    - ▶ Replay log into that structure
    - ▶ Combine contiguous free blocks into single entry

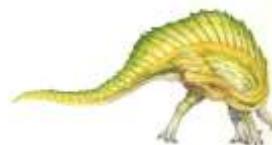




# TRIMing Unused Blocks

---

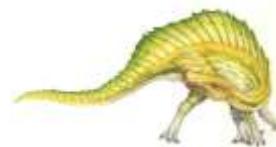
- HDDS overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (like NVM) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - ▶ Can be garbage collected or if block is free, now block can be erased





# Efficiency and Performance

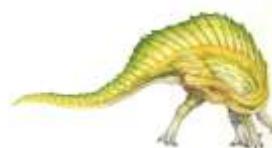
- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures





# Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - ▶ No buffering / caching – writes must hit disk before acknowledgement
    - ▶ **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

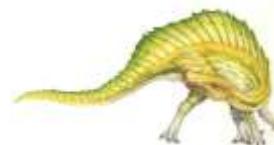




# Page Cache

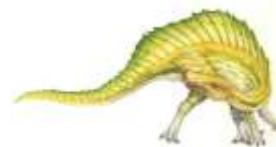
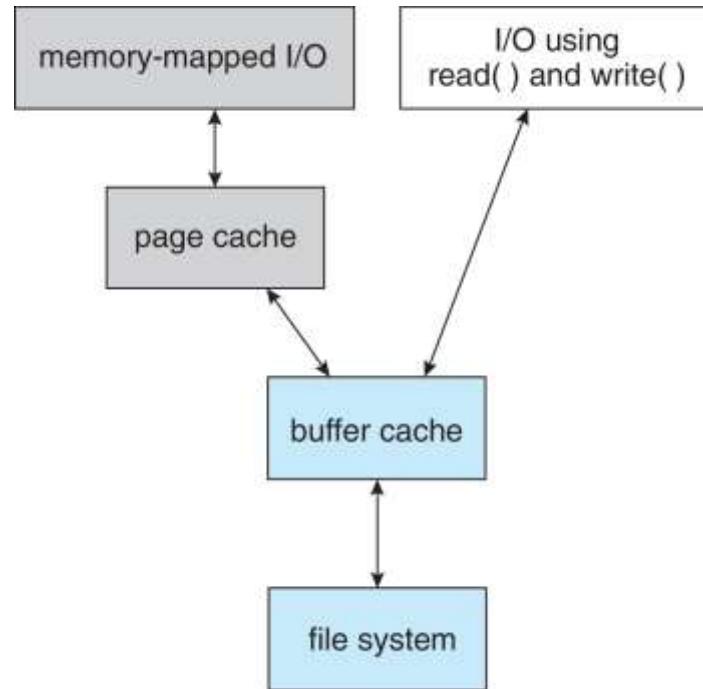
---

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure





# I/O Without a Unified Buffer Cache





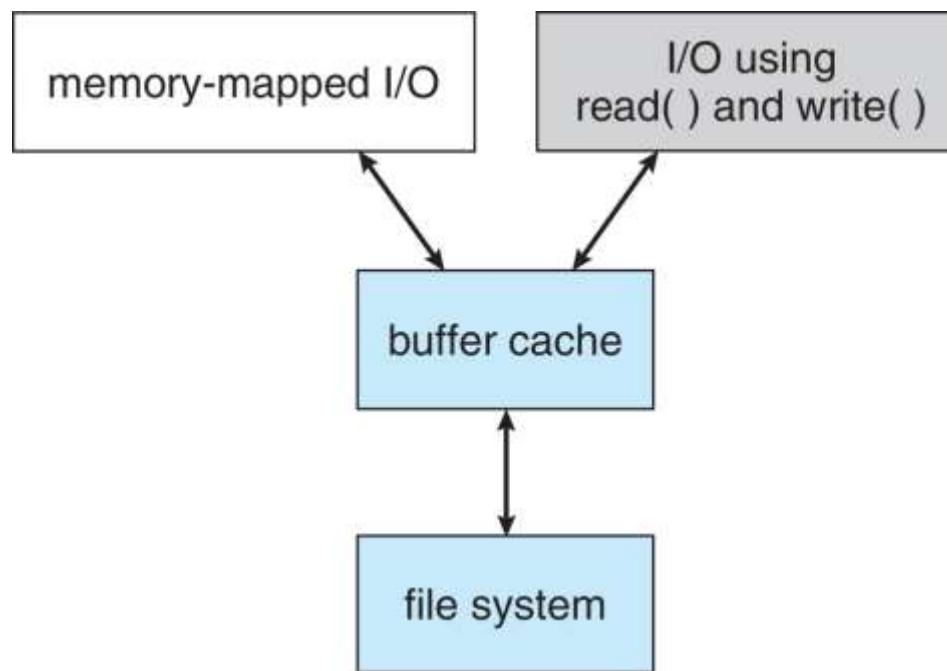
# Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?





# I/O Using a Unified Buffer Cache

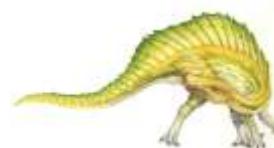




# Recovery

---

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup





# Log Structured File Systems

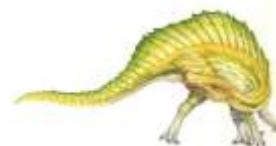
- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata





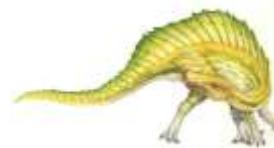
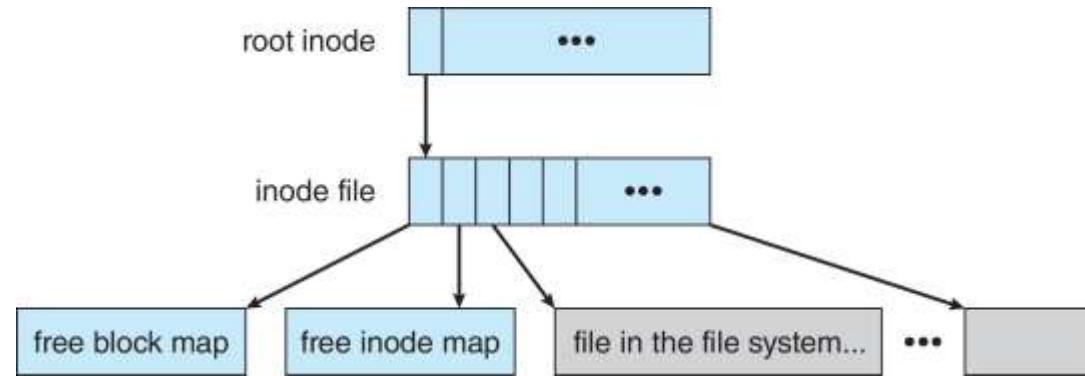
# Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications



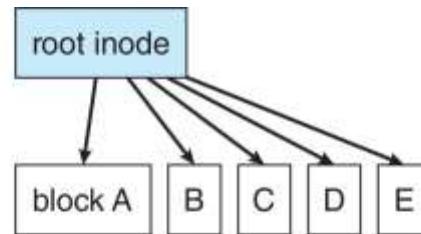


# The WAFL File Layout

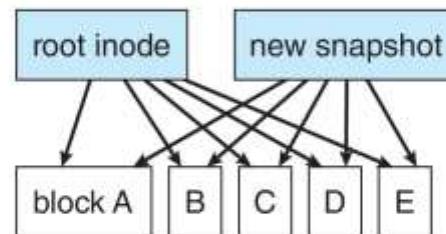




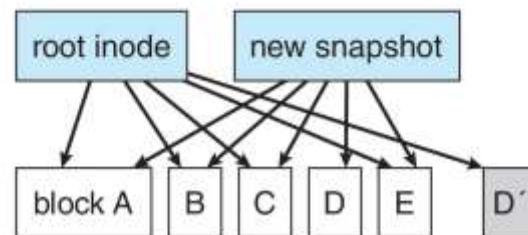
# Snapshots in WAFL



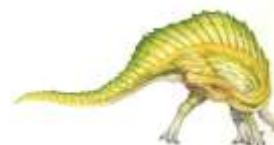
(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.





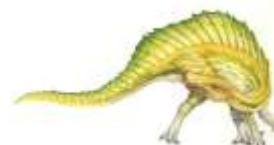
# The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

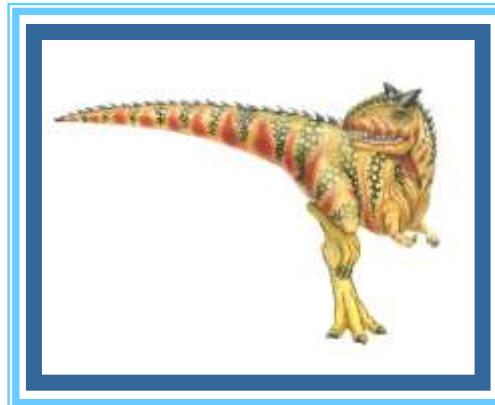
**Apple File System (APFS)** is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, iOS, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

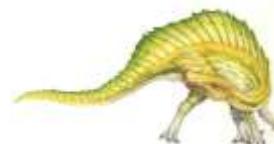
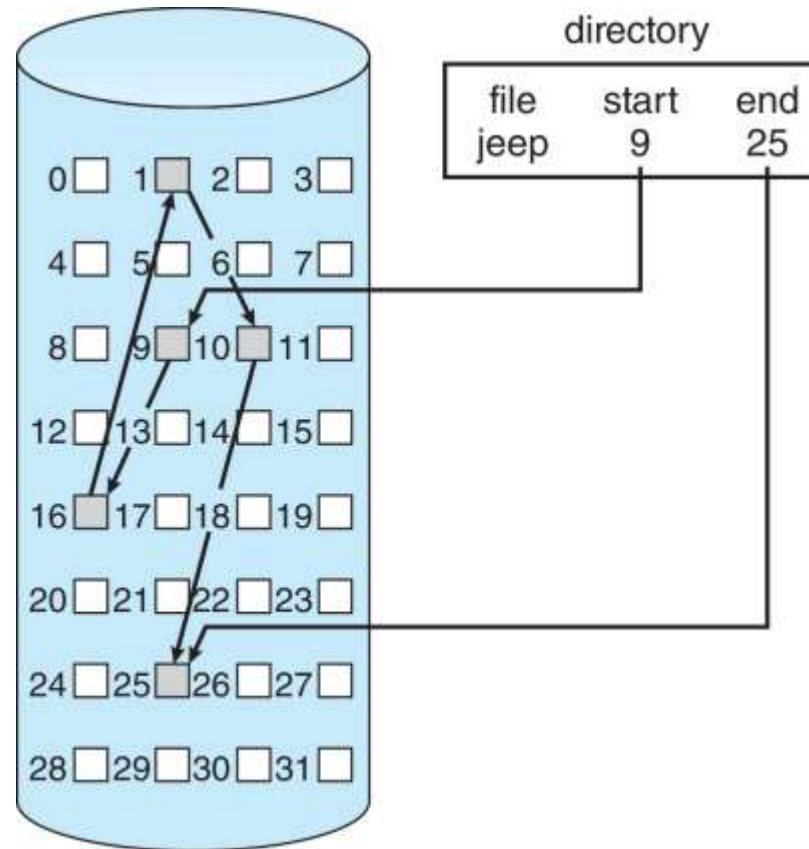


# End of Chapter 14





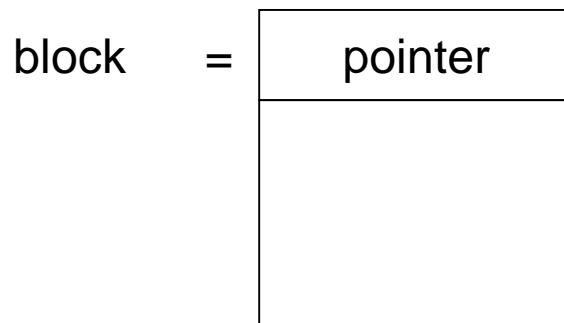
# Linked Allocation



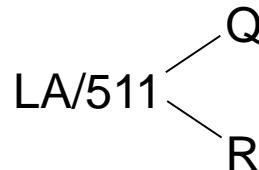


# Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk

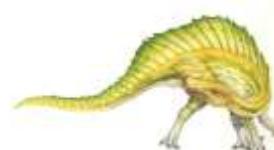


- Mapping

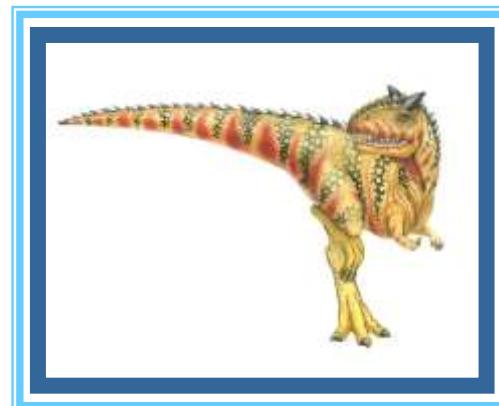


Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1



# End of Chapter 14





# In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

