SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SCHOOL OF COMPUTING

18CSC205J-Operating Systems

Unit- I

Prepared by

Dr. M. Eliazer

CTECH, SCHOOL OF COMPUTING, SRMIST

UNIT 1 - CONTENT

Session-1

- Operating System Objectives and functions
- Gaining the role of Operating systems

Session-2

- The evolution of operating system
- Major Achievements
- Understanding the evolution of Operating systems from early batch processing systems to modern complex systems

Session-3

- OS Design considerations for Multiprocessor and Multicore
- Understanding the key design issues of Multiprocessor Operating systems and Multicore Operating systems

UNIT 1 - CONTENT

Session-6

- Process Concept - Processes, PCB
- Understanding the Process concept and Maintenance of PCB by OS

Session-7

- Threads – Overview and its Benefits
- Understanding the importance of threads

Session-8

- Process Scheduling - Scheduling Queues, Schedulers, Context switch
- Understanding basics of Process Scheduling

UNIT 1 - CONTENT

Session-11

- Operations on Process - Process creation, Process termination
- Understanding the system calls – fork(),wait(),exit()

Session-12

- Inter Process communication : Shared Memory, Message Passing ,Pipe()
- Understanding the need for IPC

Session-13

- Process synchronization: Background, Critical section Problem
- Understanding the race conditions and the need for the Process synchronization

Session-1

- Operating System Objectives and functions
- Gaining the role of Operating systems

OS OBJECTIVES

What is an Operating System

Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs.

- A program that controls the execution of application programs
- An interface between applications and hardware

Main objectives of an OS

- Convenience
- Efficiency
- Ability to evolve

Operating System Functions

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

Role of Operating System

1. The OS as a User/Computer Interface

- Computer Hardware-Software Structure
  - Layered organization

- OS services to users

2. The Operating System as a Resource Manager

A computer is a set of resources for moving, storing, & processing data

- The OS is responsible for managing these resources
- The OS exercises its control through software

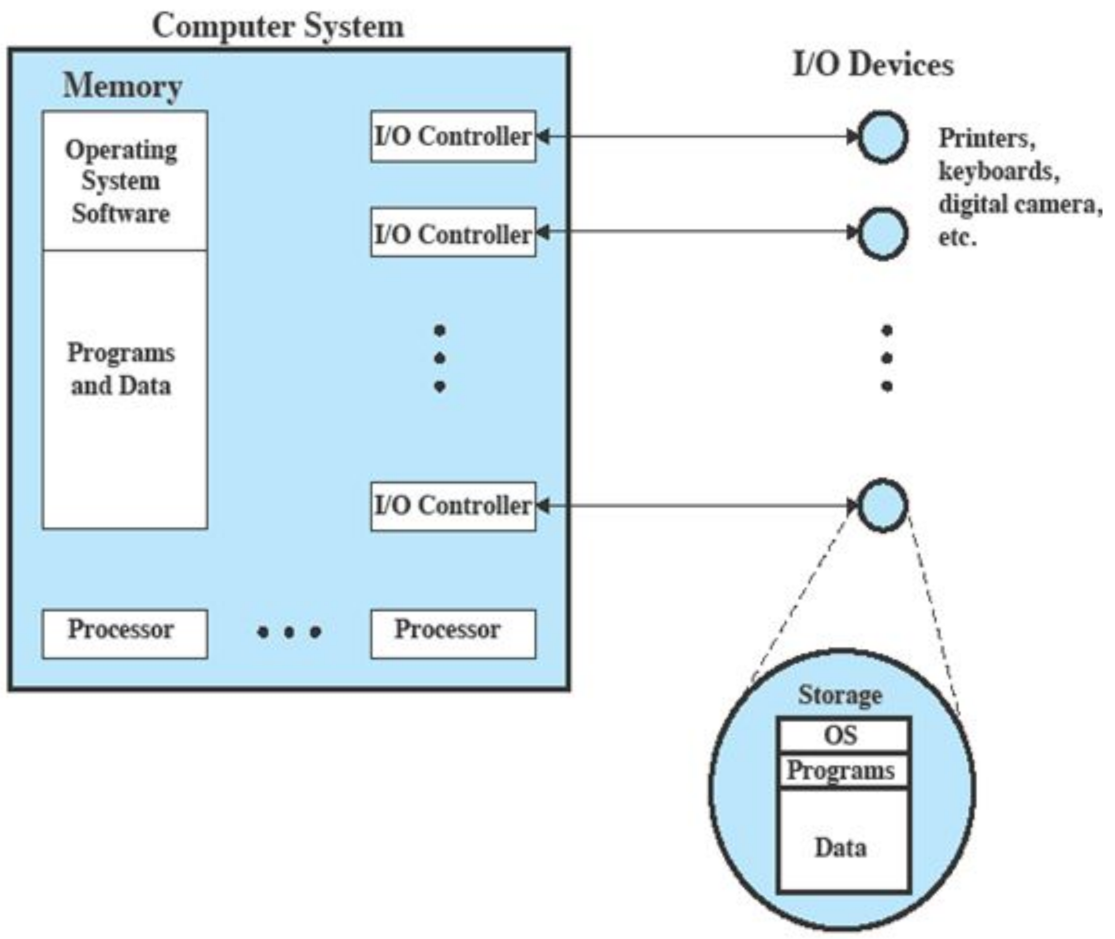
Role of Operating System

3. Operating System as Software

- Functions in the same way as ordinary computer software
- Program, or suite of programs, executed by the processor
- Frequently relinquishes control and must depend on the processor to allow it to regain control

Role of Operating System

4. Operating System as Resource Manager



Session-2

- The evolution of operating system
- Major Achievements
- Understanding the evolution of Operating systems from early batch processing systems to modern complex systems

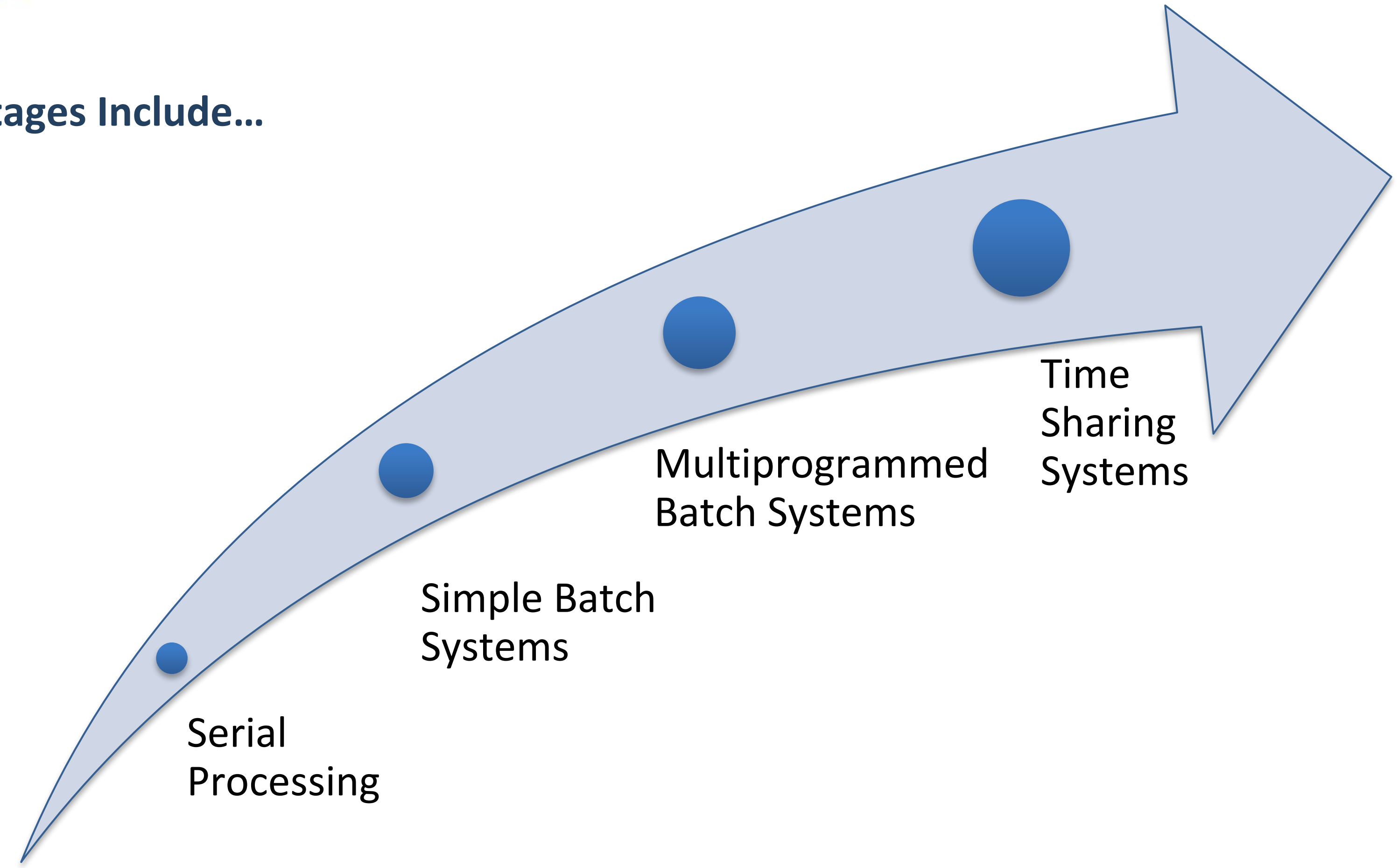
Evolution of Operating Systems

A major OS will evolve over time for a number of reasons:

- Hardware upgrades
- New types of hardware
- New services
- Fixes

Evolution of Operating Systems

Stages Include…



Serial Processing

Earliest Computers:

- No operating system
  Programmers interacted directly with the computer hardware
- Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
- Users have access to the computer in “series”

Problems:

- Scheduling:
  - Most installations used a hardcopy sign-up sheet to reserve computer time.
  - Time allocations could run short or long, resulting in wasted computer time
- Setup time
  - A considerable amount of time was spent just on setting up the program to run

Simple Batch Systems

- Early computers were very expensive
  - Important to maximize processor utilization
- Monitor
  - User no longer has direct access to processor
  - Job is submitted to computer operator who batches them together and places them on an input device
  - Program branches back to the monitor when finished

Monitor Point of View

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

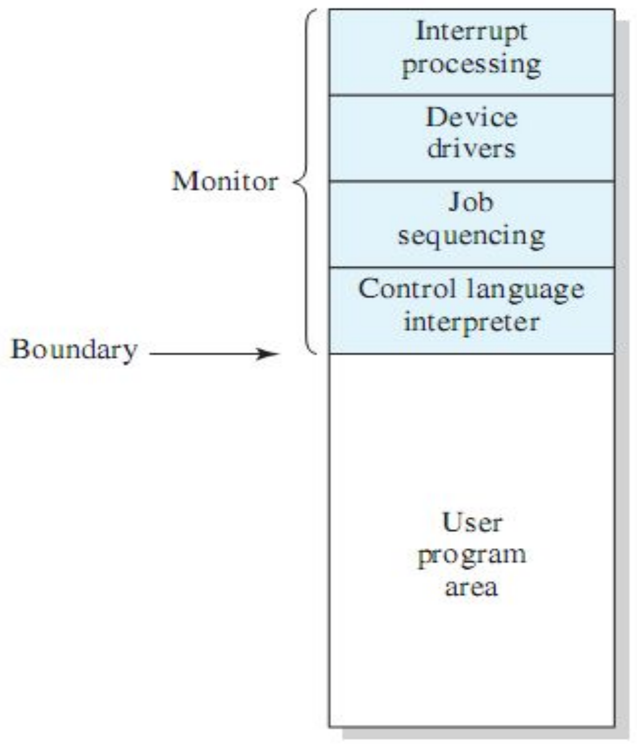


Figure 2.3 Memory Layout for a Resident Monitor

Processor Point of View

- Processor executes instruction from the memory containing the monitor
- Executes the instructions in the user program until it encounters an ending or error condition
- "*control is passed to a job*" means processor is fetching and executing instructions in a user program
- "*control is returned to the monitor*" means that the processor is fetching and executing instructions from the monitor program

Modes of Operation

User Mode

- User program executes in user mode
- Certain areas of memory are protected from user access
- Certain instructions may not be executed

Kernel Mode

- Monitor executes in kernel mode
- Privileged instructions may be executed
- Protected areas of memory may be accessed

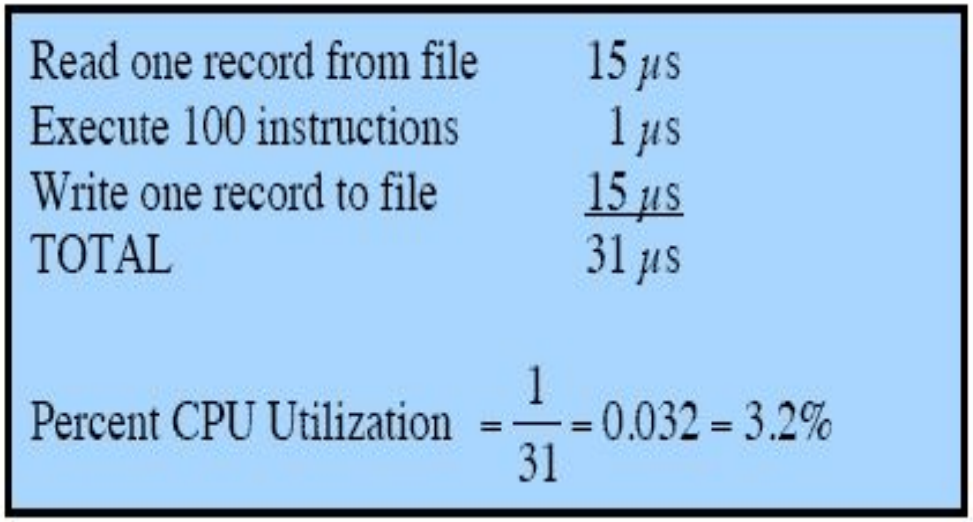
Simple Batch System Overhead

- Processor time alternates between execution of user programs and execution of the monitor

- Sacrifices:
  - some main memory is now given over to the monitor
  - some processor time is consumed by the monitor

- Despite overhead, the simple batch system improves utilization of the computer. (How?)

Multi programmed Batch Systems



Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	$\underline{15\ \mu s}$
TOTAL	31 μs

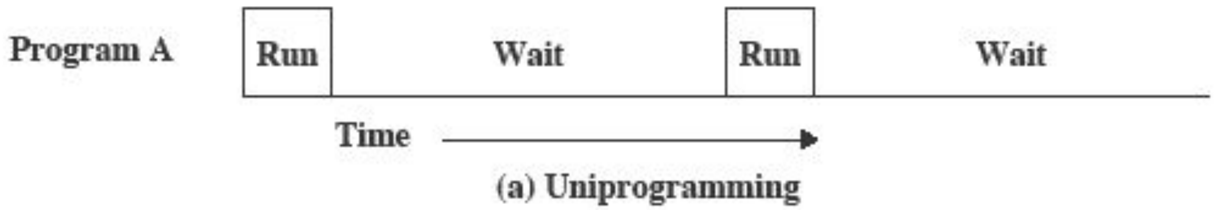
$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 System Utilization Example

Processor is often idle

- Even with automatic job sequencing
- I/O devices are slow compared to processor

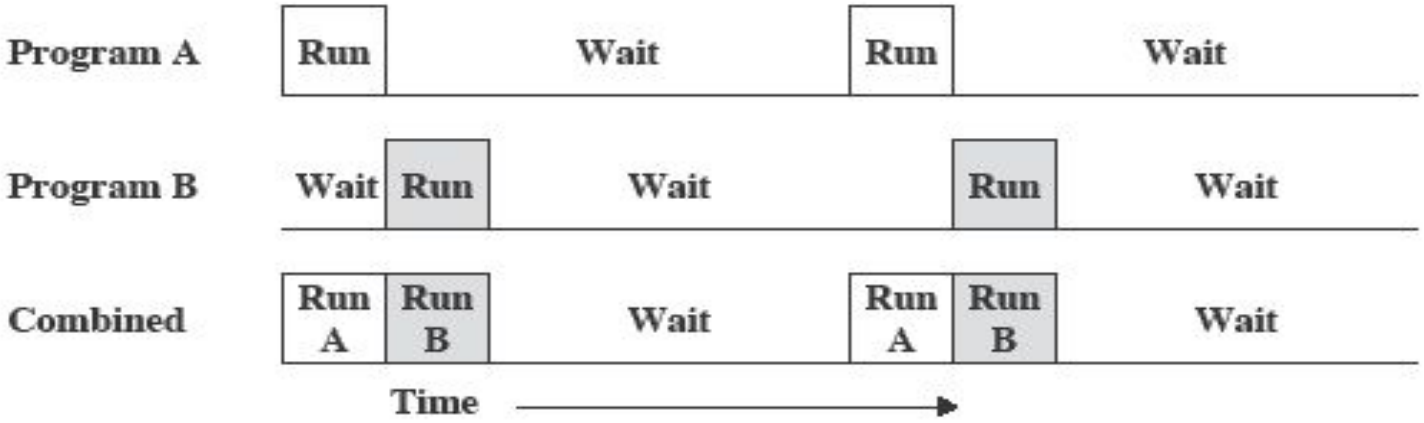
Uniprogramming



(a) Uniprogramming

The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding

Multiprogramming

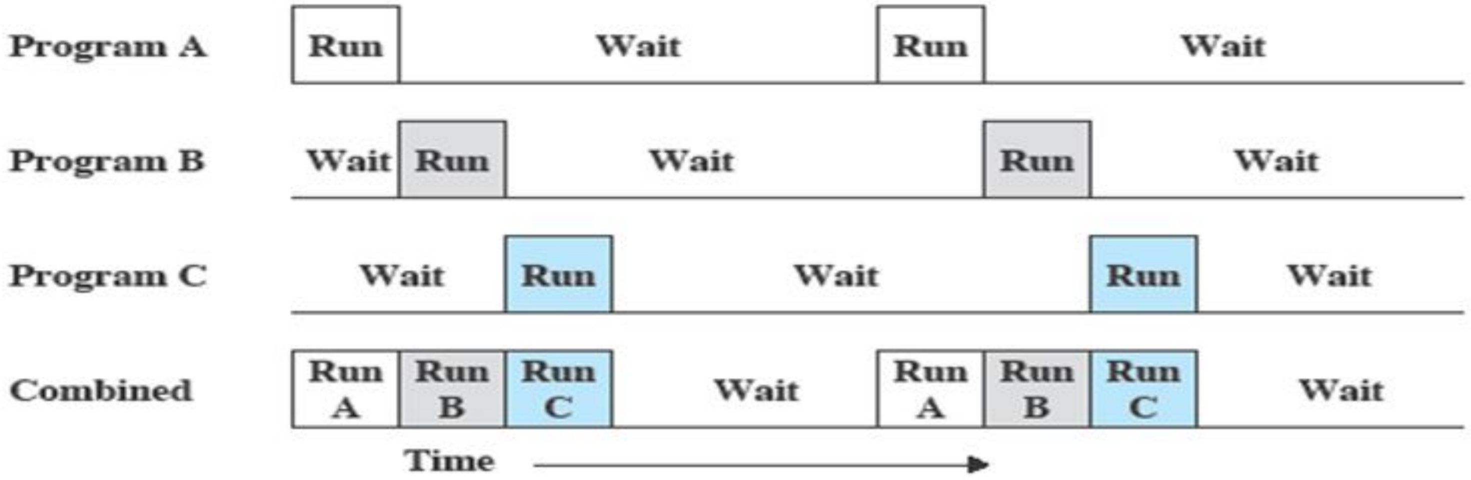


(b) Multiprogramming with two programs

There must be enough memory to hold the OS (resident monitor) and one user program

When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O

Multiprogramming



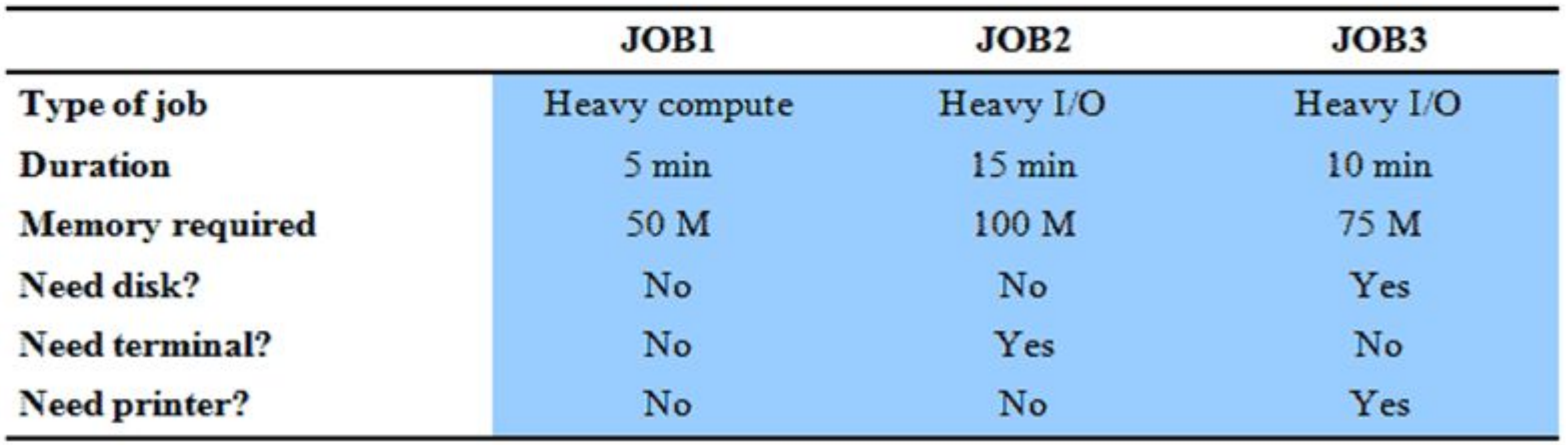
(c) Multiprogramming with three programs

Multiprogramming

- also known as multitasking
- memory is expanded to hold three, four, or more programs and switch among all of them

Multiprogramming Example

Table 2.1 Sample Program Execution Attributes



	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Effects on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

Table 2.2 Effects of Multiprogramming on Resource Utilization

Time-Sharing Systems

- Can be used to handle multiple interactive jobs
- Processor time is shared among multiple users
- Multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation

Batch Multiprogramming vs. Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Table 2.3 Batch Multiprogramming versus Time Sharing

Compatible Time-Sharing Systems

CTSS

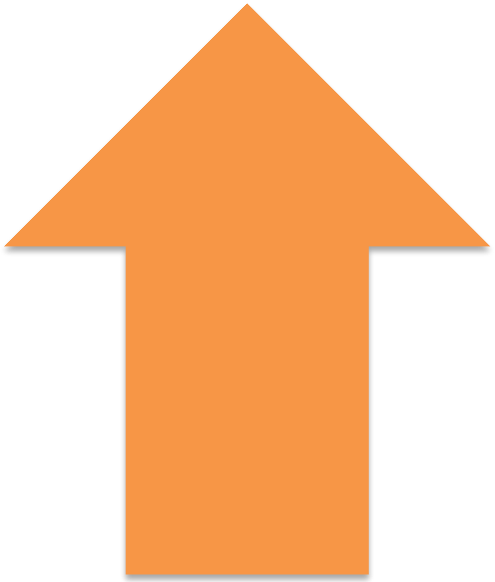
- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC
- Ran on a computer with 32,000 *36*-bit words of main memory, with the resident monitor consuming 5000 of that
- To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word

Time Slicing

- System clock generates interrupts at a rate of approximately one every 0.2 seconds
- At each interrupt OS regained control and could assign processor to another user
- At regular time intervals the current user would be preempted and another user loaded in
- Old user programs and data were written out to disk
- Old user program code and data were restored in main memory when that program was next given a turn

Major Achievements

Operating Systems are among the most complex pieces of software ever developed



Major advances in development include:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management
- System structure

Process

Fundamental to the structure of operating systems

A *process* can be defined as:

- a program in execution
- an instance of a running program
- the entity that can be assigned to, and executed on, a processor
- a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Development of the Process

Three major lines of computer system development created problems in timing and synchronization that contributed to the development:

multiprogramming batch operation

- processor is switched among the various programs residing in main memory

time sharing

- be responsive to the individual user but be able to support many users simultaneously

real-time transaction systems

- a number of users are entering queries or updates against a database

Components of a Process

- A process contains three components:
  - An executable program
  - The associated data needed by the program (variables, work space, buffers, etc.)
  - The execution context (or “process state”) of the program

- The execution context is essential:
  - It is the internal data by which the OS is able to supervise and control the process
  - Includes the contents of the various process registers
  - Includes information such as the priority of the process and whether the process is waiting for the completion of a particular I/O event

Process Management

- The entire state of the process at any instant is contained in its context
- New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

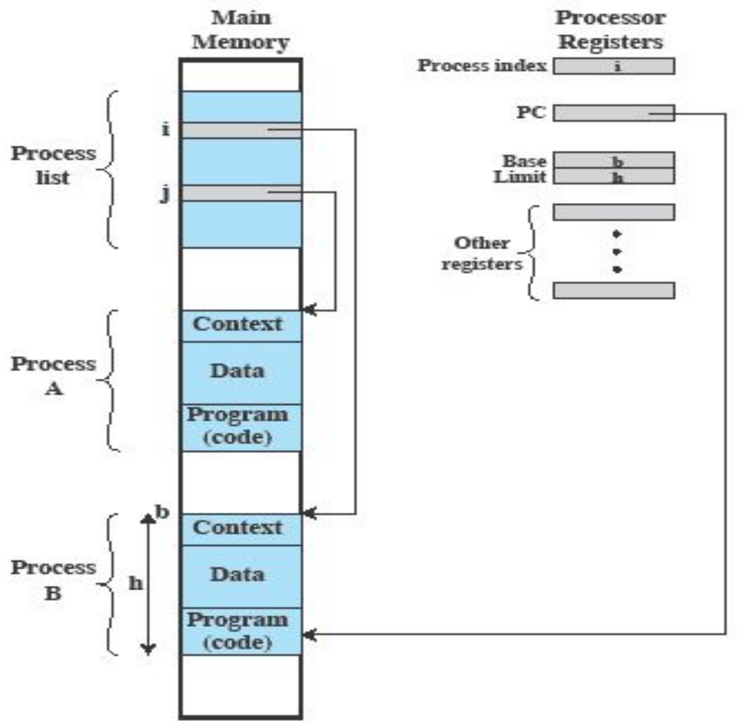


Figure 2.8 Typical Process Implementation

Memory Management

The OS has five principal storage management responsibilities:

process isolation

automatic allocation and management

support of modular programming

protection and access control

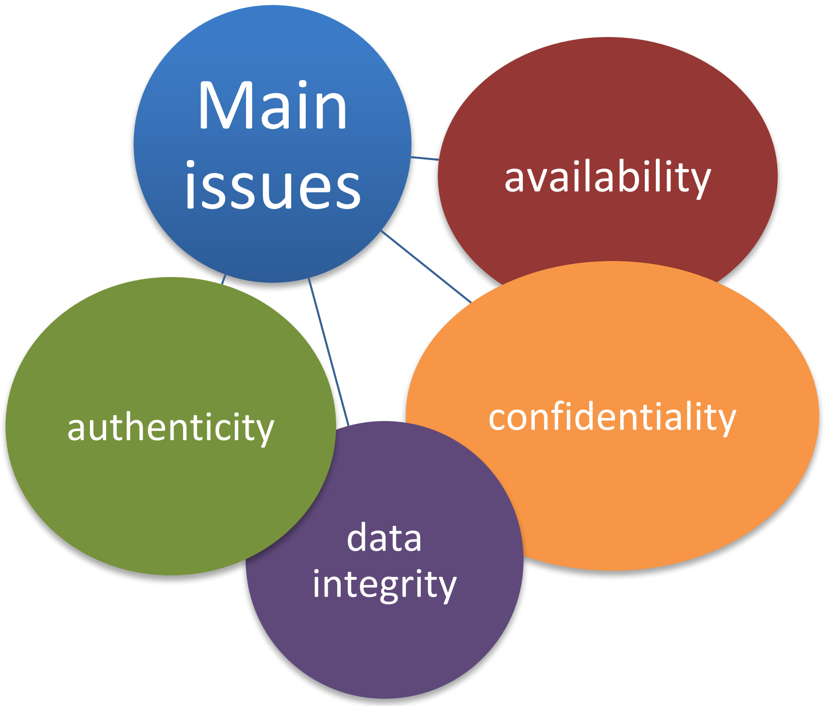
long-term storage

Memory Management (contd..)

- **VIRTUAL MEMORY**
  - A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
  - Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently
- **PAGING**
  - Allows processes to be comprised of a number of fixed-size blocks, called pages
  - Program references a word by means of a virtual address
    - consists of a page number and an offset within the page
    - each page may be located anywhere in main memory
  - Provides for a dynamic mapping between the virtual address used in the program and a real (or physical) address in main memory

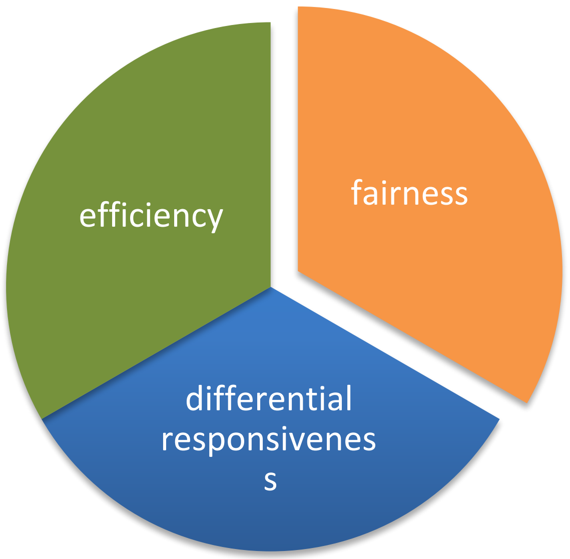
Information Protection and Security

- The nature of the threat that concerns an organization will vary greatly depending on the circumstances
- The problem involves controlling access to computer systems and the information stored in them



Scheduling and Resource Management

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider:



Key Elements of an Operating System

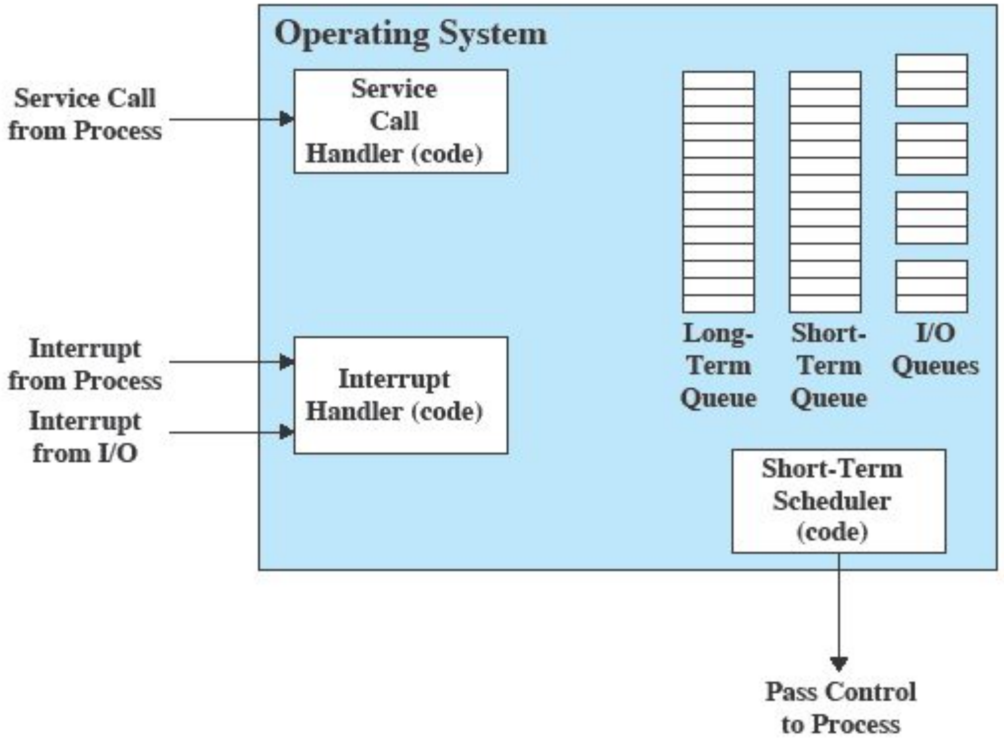


Figure 2.11 Key Elements of an Operating System for Multiprogramming

Session-3

- OS Design considerations for Multiprocessor and Multicore
- Understanding the key design issues of Multiprocessor Operating systems and Multicore Operating systems

OS Design

Distributed Operating System

- Provides the illusion of
  - a single main memory space
  - single secondary memory space
  - unified access facilities
- State of the art for distributed operating systems lags that of uniprocessor and SMP operating systems

Object-Oriented Design

- Used for adding modular extensions to a small kernel
- Enables programmers to customize an operating system without disrupting system integrity
- Eases the development of distributed tools and full-blown distributed operating systems

Symmetric Multiprocessor OS Considerations

- A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors

Key design issues:

Simultaneous concurrent processes or threads
- kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously

Scheduling
- any processor may perform scheduling, which complicates the task of enforcing a scheduling policy

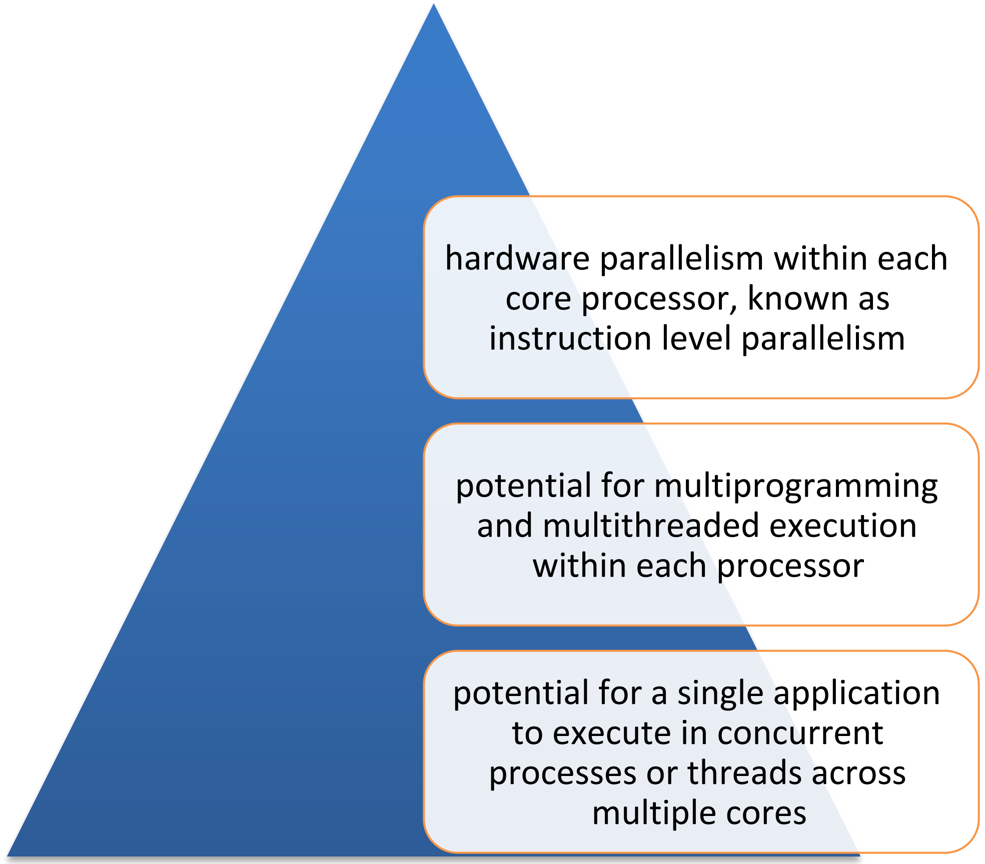
Synchronization
- with multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization

Memory management
- the reuse of physical pages is the biggest problem of concern

Reliability and fault tolerance
- the OS should provide graceful degradation in the face of processor failure

Multicore OS Considerations

- The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently
- Potential for parallelism exists at three levels:


hardware parallelism within each core processor, known as instruction level parallelism
potential for multiprogramming and multithreaded execution within each processor
potential for a single application to execute in concurrent processes or threads across multiple cores

Session-6

- Process Concept - Processes, PCB
- Understanding the Process concept and Maintenance of PCB by OS

Process Concept

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

Process Concept

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

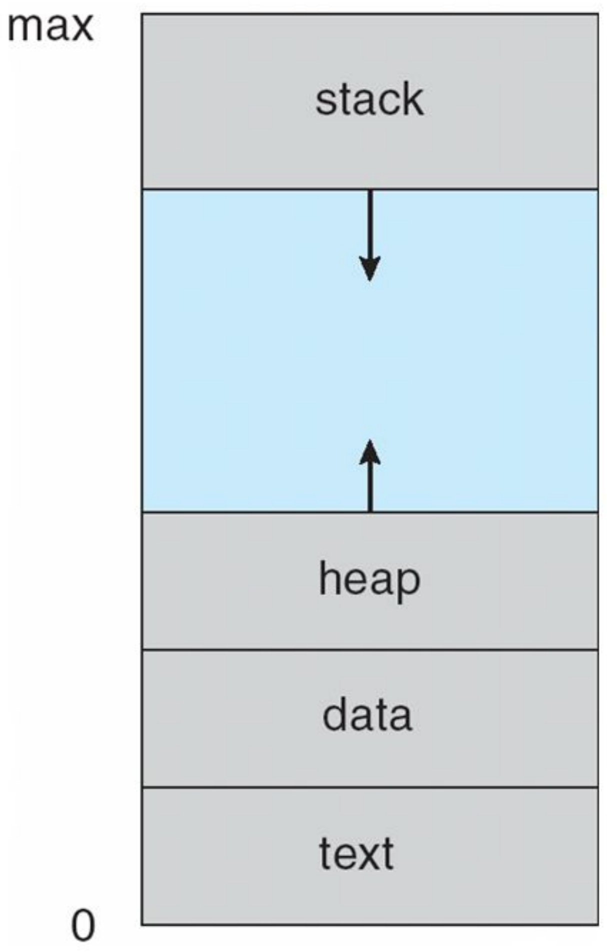
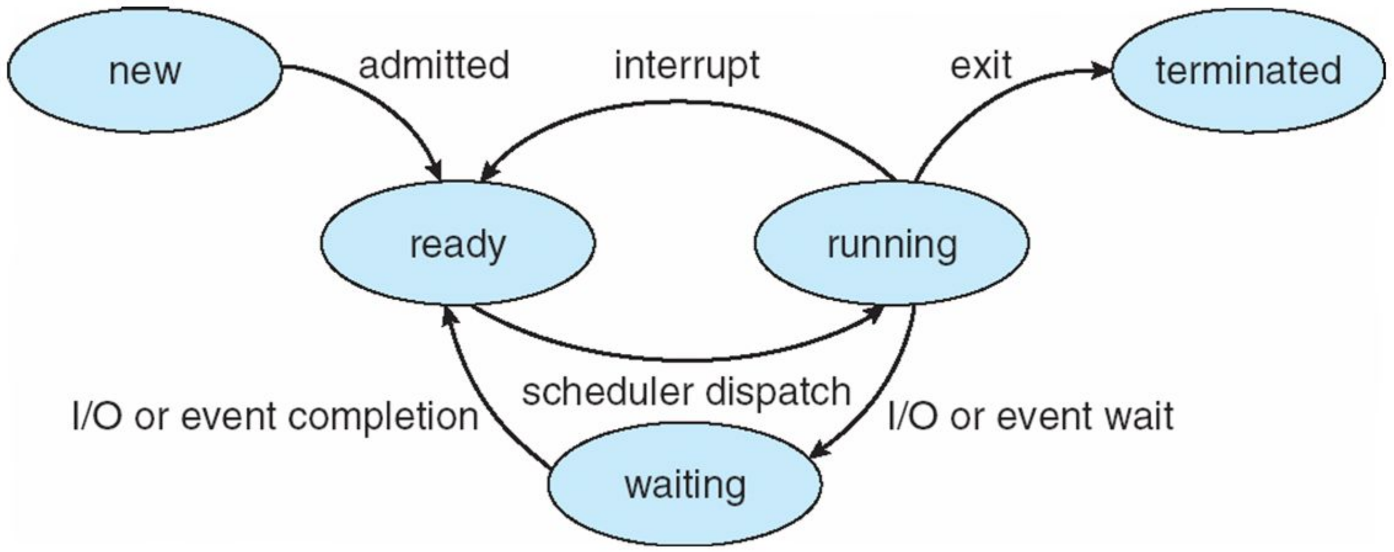


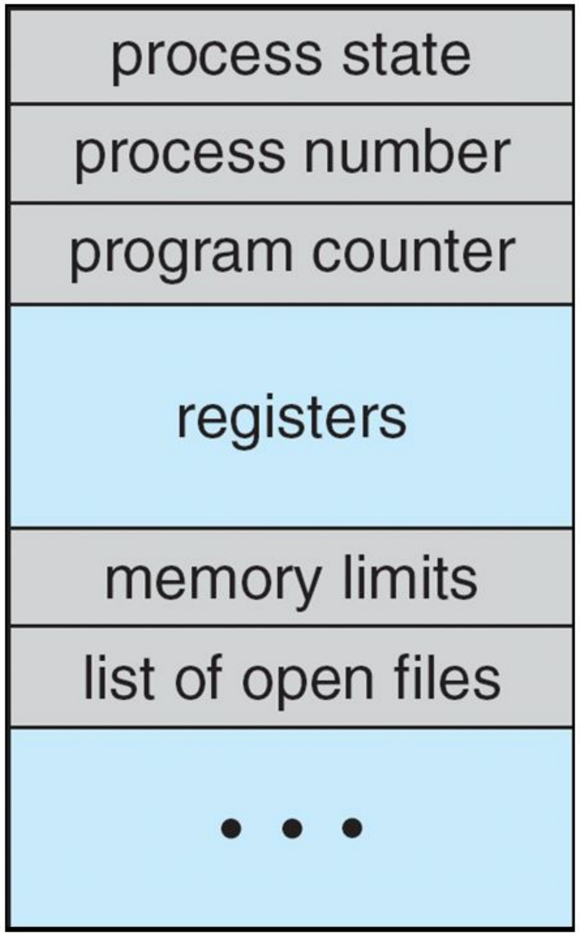
Diagram of Process State



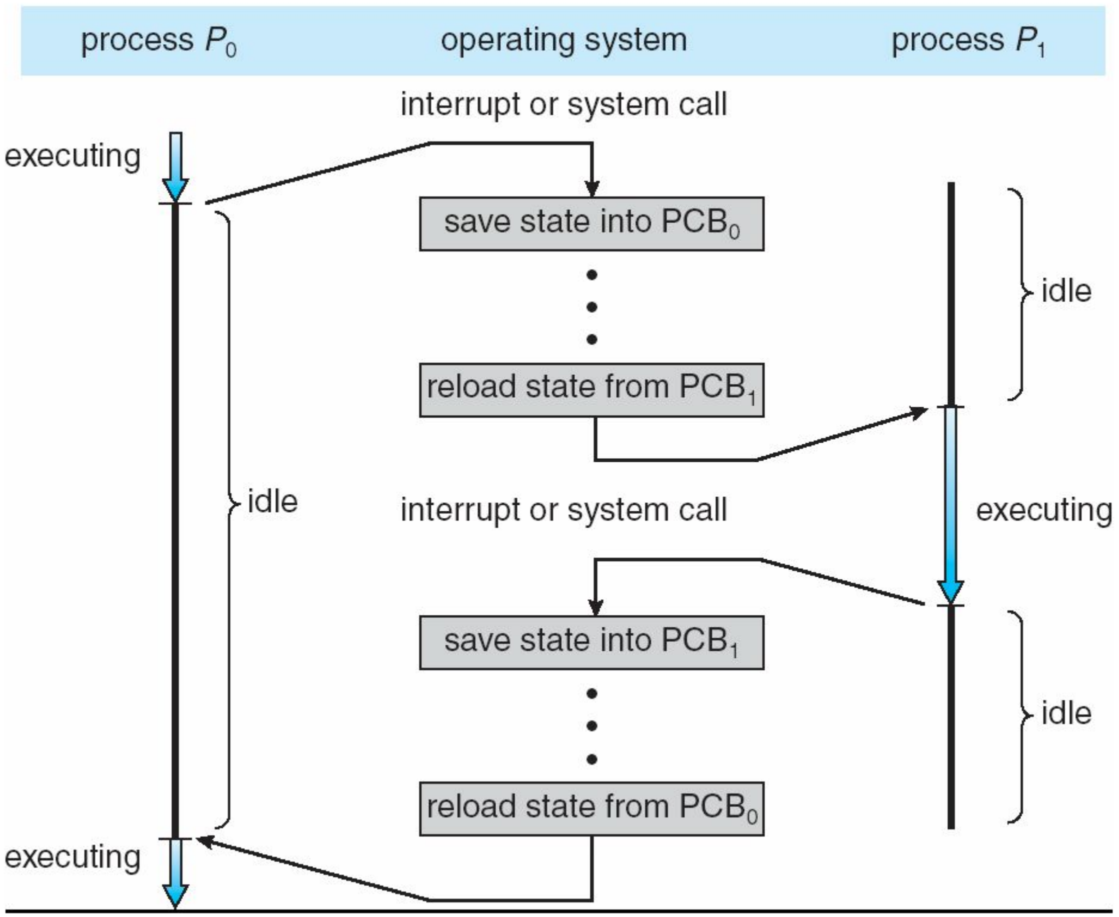
Process Control Block (PCB)

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



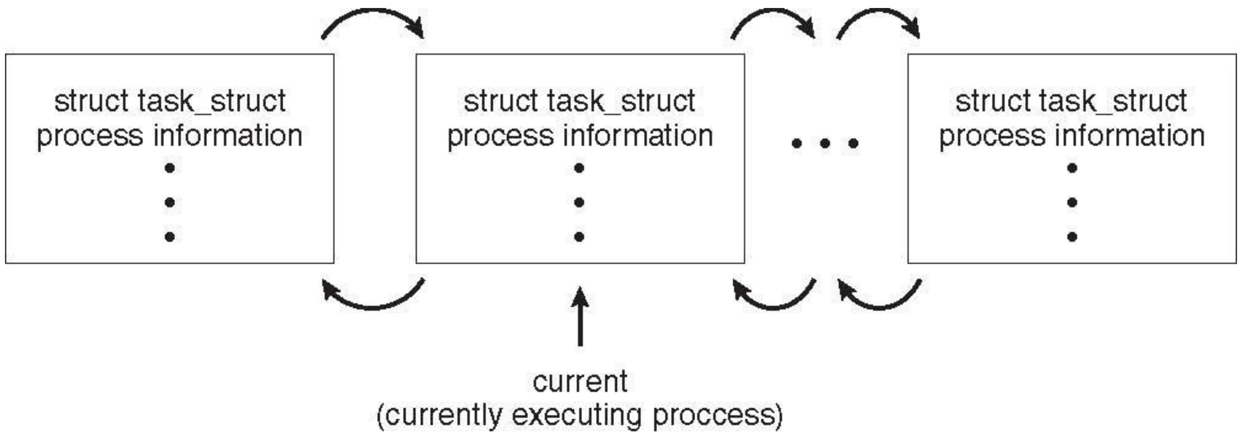
CPU Switch From Process to Process



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 */
```



Session-7

- Threads – Overview and its Benefits
- Understanding the importance of threads

Threads

- Processes have two characteristics:
  - **Resource ownership** - process includes a virtual address space to hold the process image
  - **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

- The unit of dispatching is referred to as a ***thread*** or lightweight process
- The unit of resource ownership is referred to as a process or ***task***

Multithreading – Types of Threading

The ability of an OS to support multiple, concurrent paths of execution within a single process.

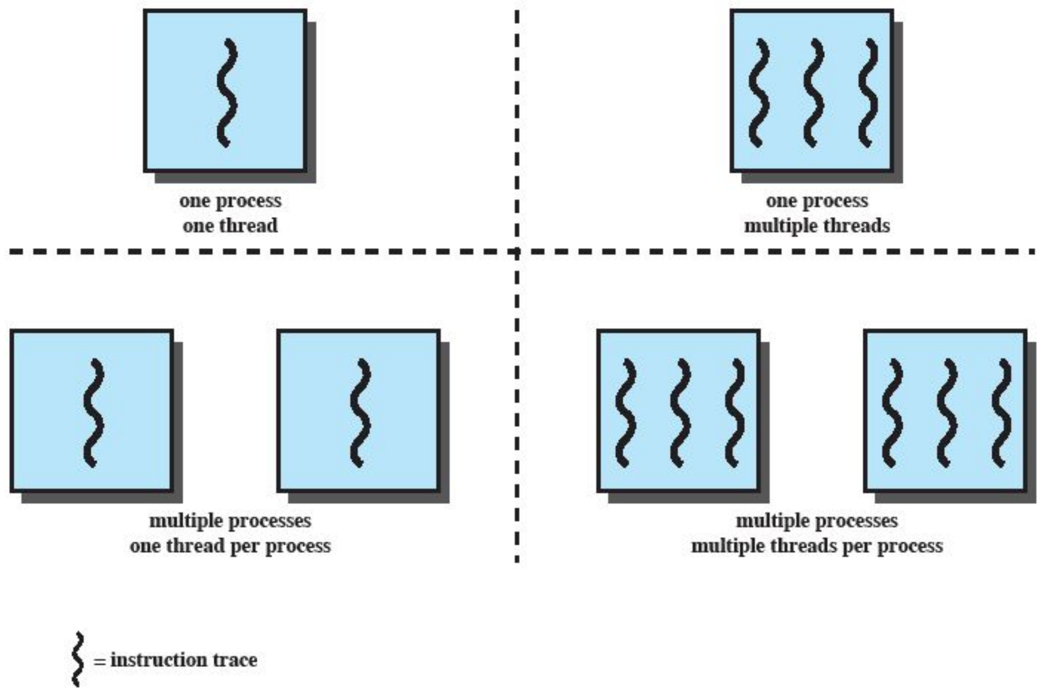


Figure 4.1 Threads and Processes [ANDE97]

Single Thread Approaches

- MS-DOS supports a single user process and a single thread.
- Some UNIX, support multiple user processes but only support one thread per process

Threads vs. processes

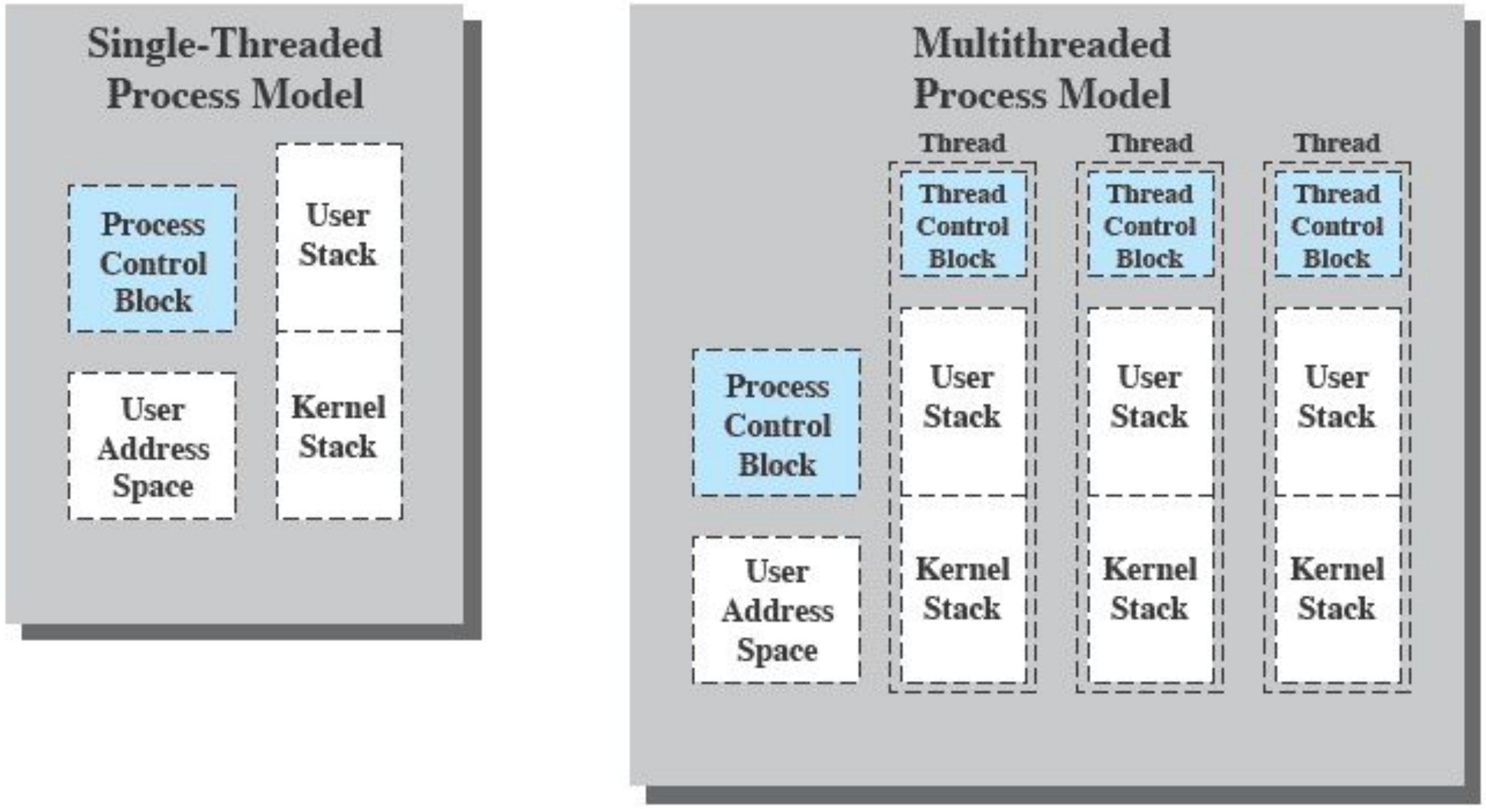


Figure 4.2 Single Threaded and Multithreaded Process Models

Importance of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time that switching processes
- Threads can communicate with each other without invoking the kernel

One or More Threads in Process

- Each thread has
  - An execution state (running, ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process (all threads of a process share this)

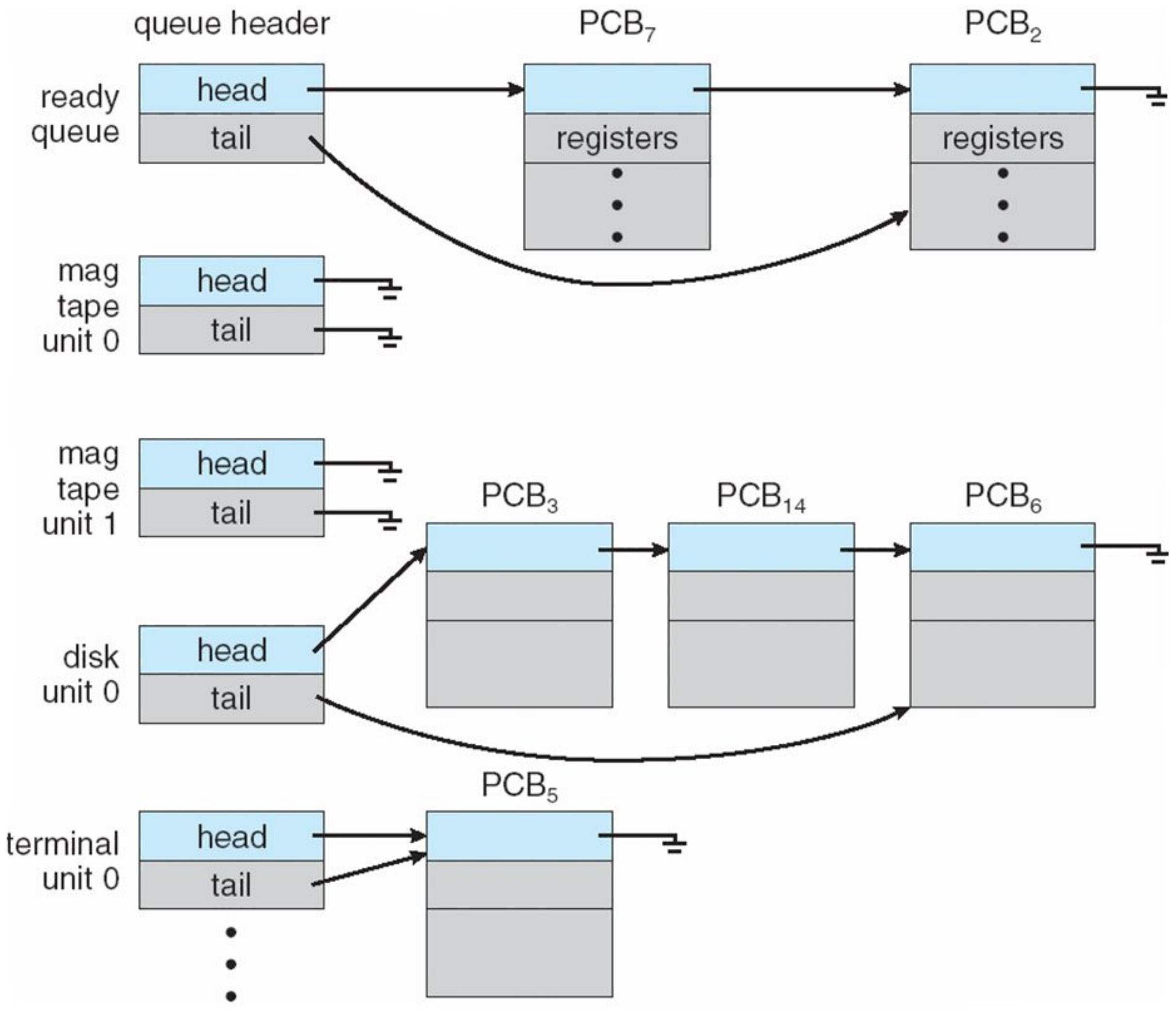
Session-8

- Process Scheduling - Scheduling Queues, Schedulers, Context switch
- Understanding basics of Process Scheduling

Process Scheduling

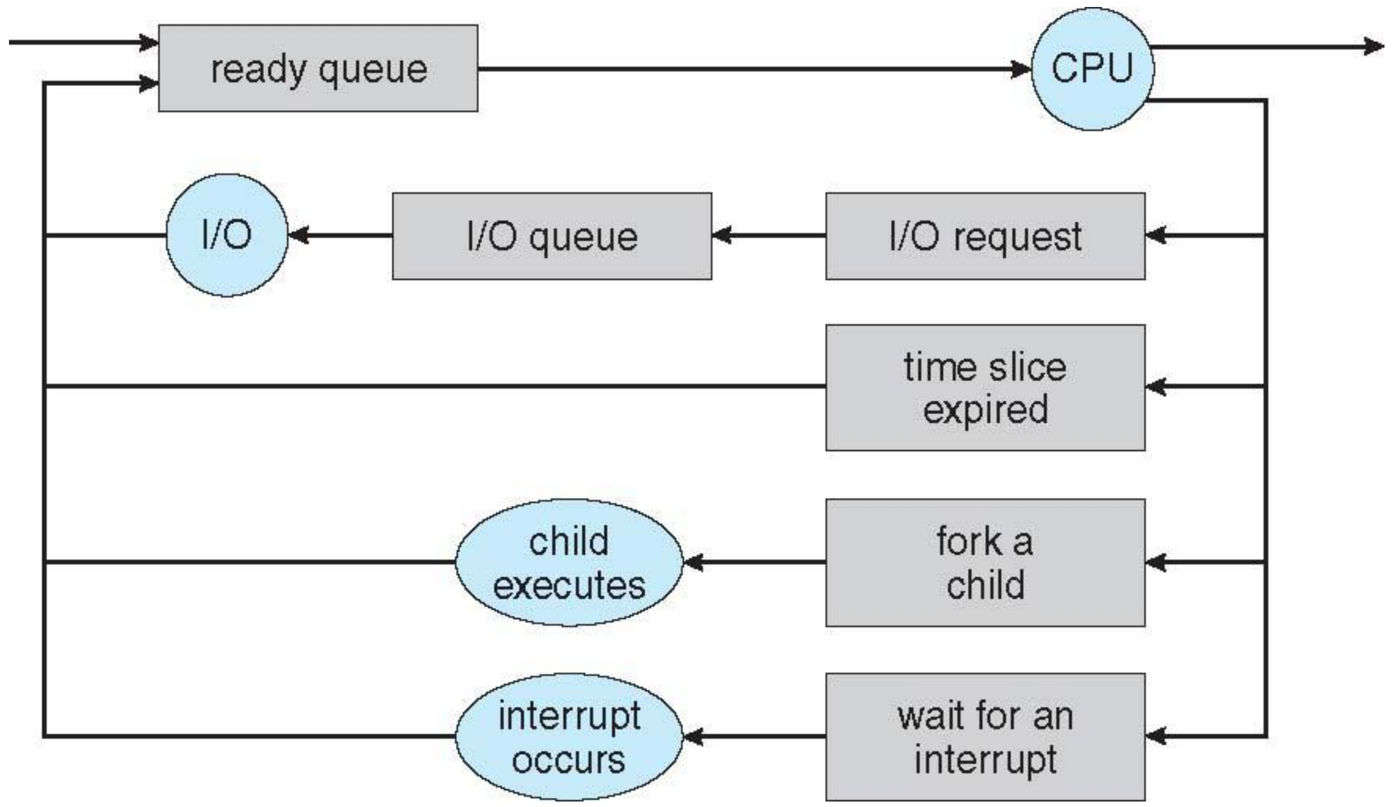
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

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



Schedulers

Short-term scheduler (or **CPU scheduler**) – selects which process should be executed next and allocates CPU

- Sometimes the only scheduler in a system
- Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)

Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the **degree of multiprogramming**

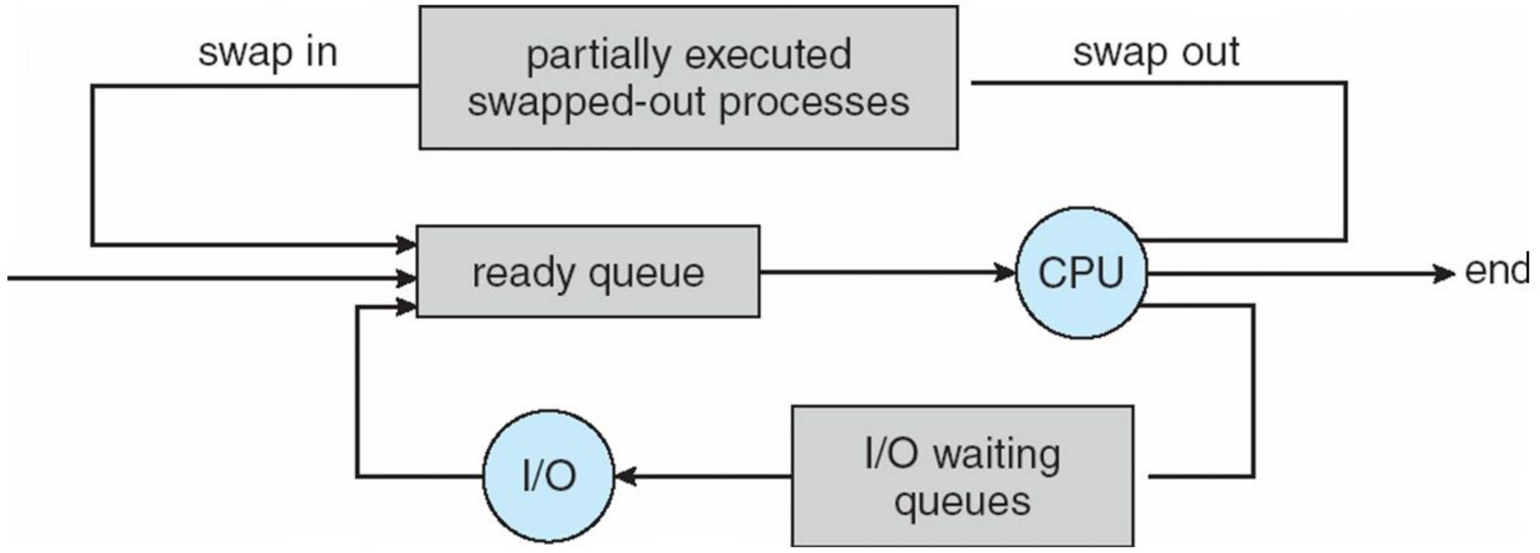
Processes can be described as either:

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

Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



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 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

Session-11

- Operations on Process - Process creation, Process termination
- Understanding the system calls – fork(),wait(),exit()

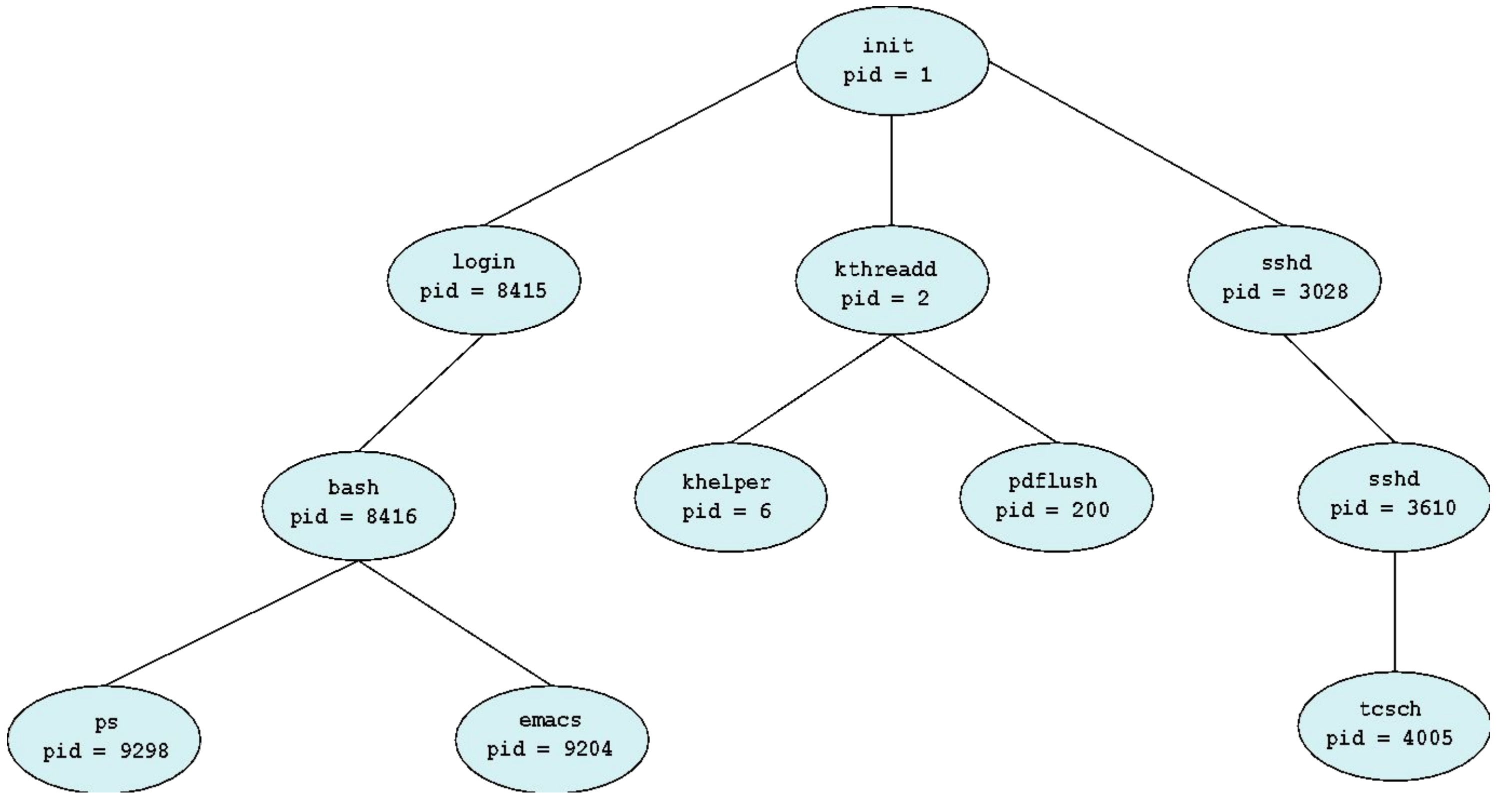
Operations on Processes

- Process creation,
- Process termination,
- Process execution

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

A Tree of Processes in Linux



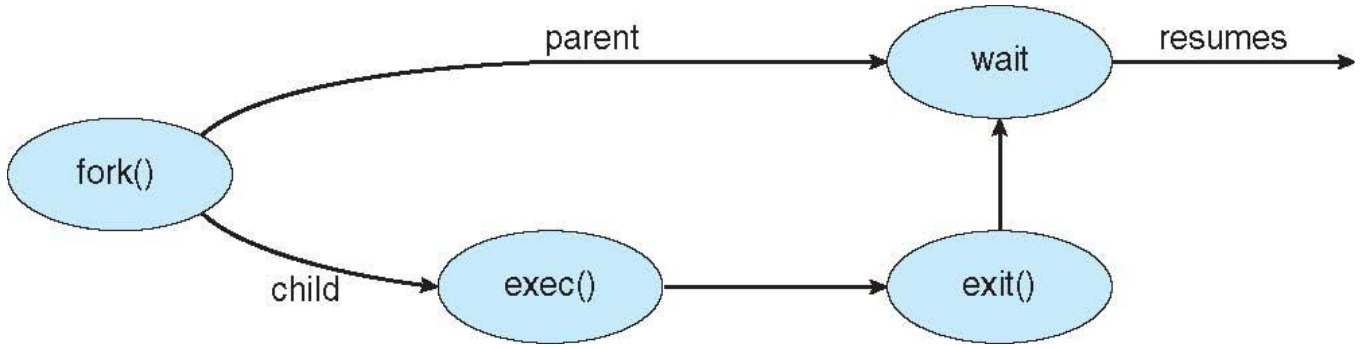
Process Creation (Cont.)

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



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

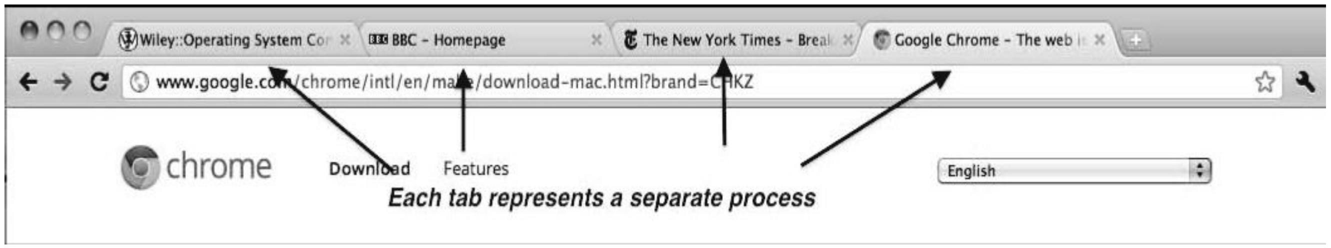
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The 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**

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


Wiley::Operating System Cor
BBC – Homepage
The New York Times – Breal
Google Chrome – The web
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English
Each tab represents a separate process

Session-12

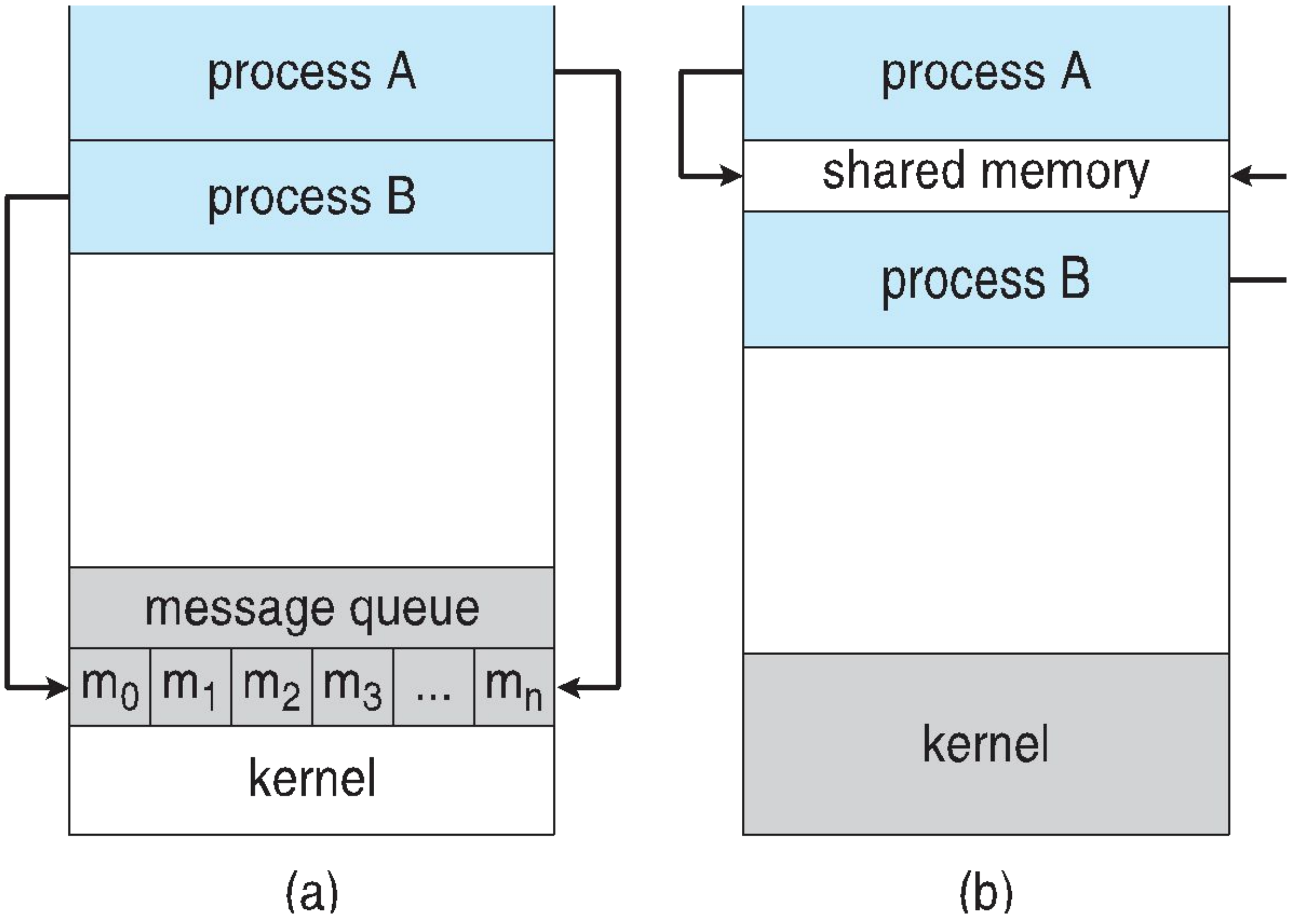
- Inter Process communication : Shared Memory, Message Passing ,Pipe()
- Understanding the need for IPC

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) Message passing. (b) shared memory.



Cooperating Processes

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

Producer-Consumer Problem

Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

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

Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

```
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 */
}
```

Interprocess Communication – 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 Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – 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?

Message Passing (Cont.)

Implementation of communication link

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

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 to share it
  - Set the size of the object

```
ftruncate(shm fd, 4096);
```

  - Now the process could write to the shared memory

```
   sprintf(shared memory, "Writing to shared
memory");
```

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 obect */
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;
}
```

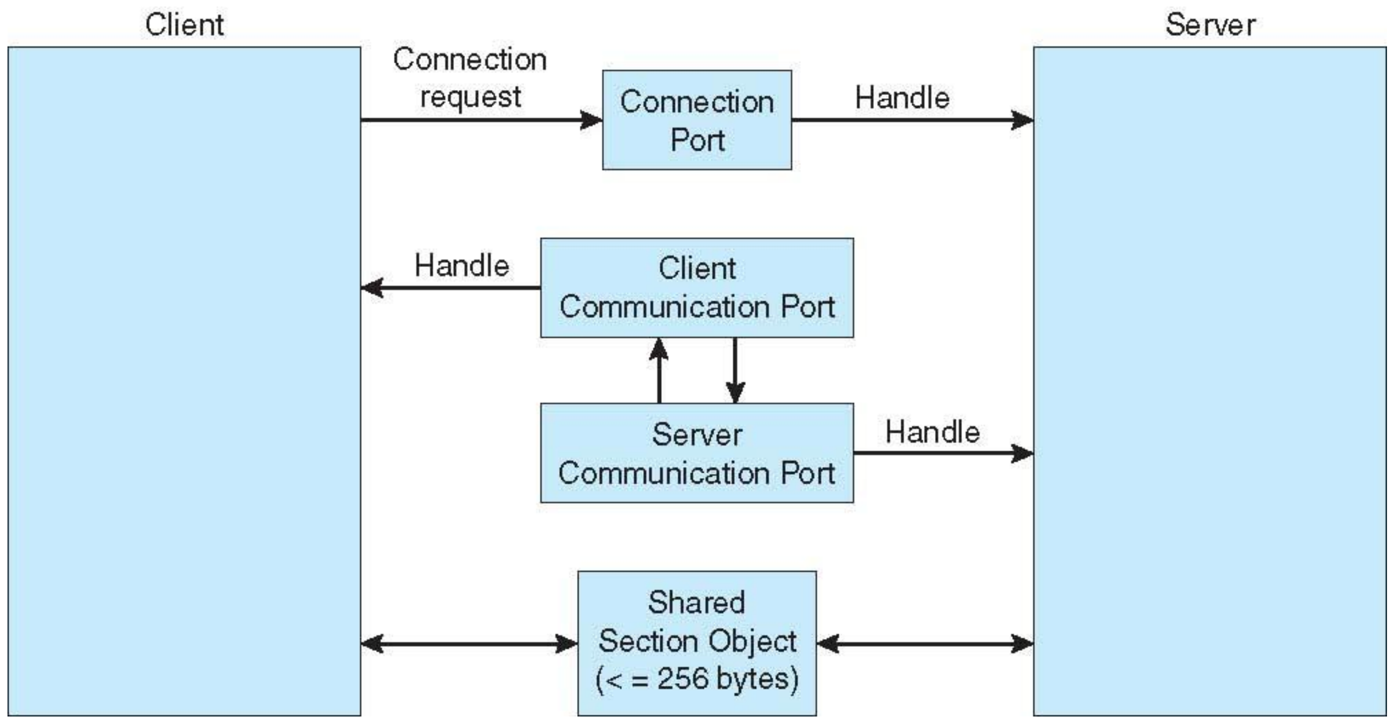
Examples of IPC Systems - Mach

- Mach communication is message based
    - ◦Even system calls are messages
    - ◦Each task gets two mailboxes at creation- Kernel and Notify
    - ◦Only three system calls needed for message transfer

      `msg_send(), msg_receive(), msg_rpc()`
    - ◦Mailboxes needed for commuication, created via

      `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

Local Procedure Calls in Windows

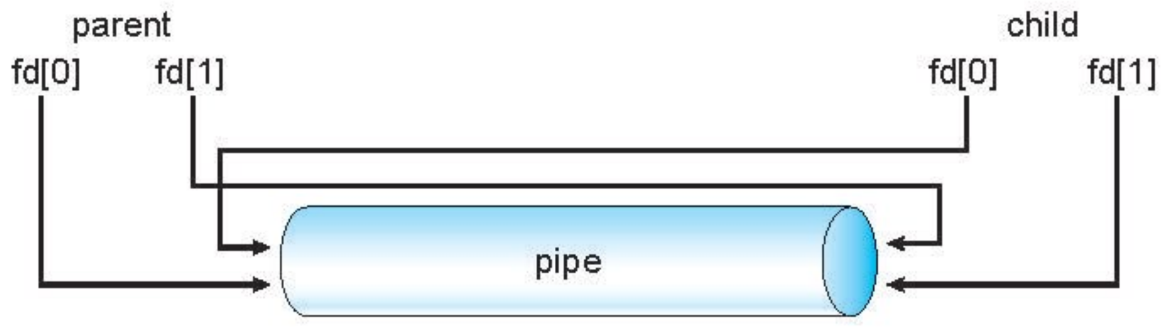

Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

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



- Window
- See Unix and Windows code samples in textbook

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

Session-13

- Process synchronization: Background, Critical section Problem
- Understanding the race conditions and the need for the Process synchronization

Process Synchronization

- **Process Synchronization** means sharing system resources by **processes** in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure **synchronized** execution of cooperating **processes**.

In other ways,

- 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

- Illustration of the problem:

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. It is incremented by the producer after it produces a new buffer 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

- Occurs when multiple processes or threads read and write shared data items
- The final result depends on the order of execution
    - the “loser” of the race is the process that updates last and will determine the final value of the variable

Race Condition

- Assume P1 and P2 are executing this code and share the variable **a**
- Processes can be preempted at any time.
- Assume P1 is preempted after the input statement, and P2 then executes entirely
- The character echoed by P1 will be the one read by P2 !!

```
static char a;

void echo()
{
    cin >> a;
    cout << a;
}
```

- This is an example of a *race condition*
- Individual processes (threads) execute sequentially in isolation, but concurrency causes them to interact.
- We need to prevent concurrent execution by processes when they are changing the same data. We need to enforce *mutual exclusion*.

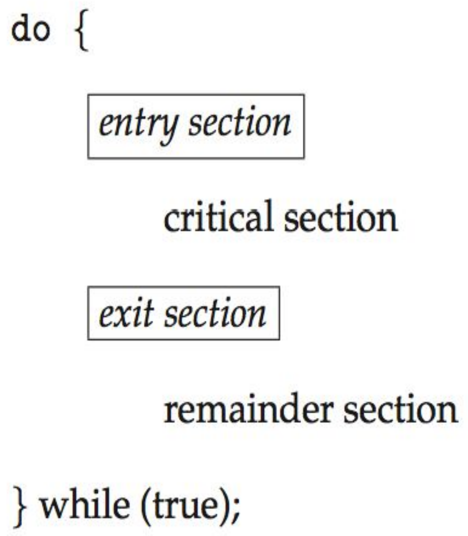
Critical Section Problem

- When a process executes code that manipulates shared data (or resources), we say that the process is in its critical section (CS) for that shared data
- We must enforce mutual exclusion on the execution of critical sections.
- Only one process at a time can be in its CS (for that shared data or resource).

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

General structure of process P_i



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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

2. William Stallings, Operating Systems-Internals and Design Principles, 7th ed., Prentice Hall, 2012

Thank You !!!