Processes

Course Code: CSC 2209 Course Title: Operating Systems

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**04**	**Week No:**	**04**	**Semester:**	
Lecturer:	*Name & email*				

Lecture Outline

1. Process Concept
2. Process Scheduling
3. Operations on Processes
4. Inter-process Communication (IPC)
5. Examples of IPC Systems
6. Communication in Client-Server Systems

Process Concept

- **Process** – a program in execution; process execution **must** progress in sequential fashion
- 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 Structure

- A process is more than the program code, which is sometimes known as the **text** section.

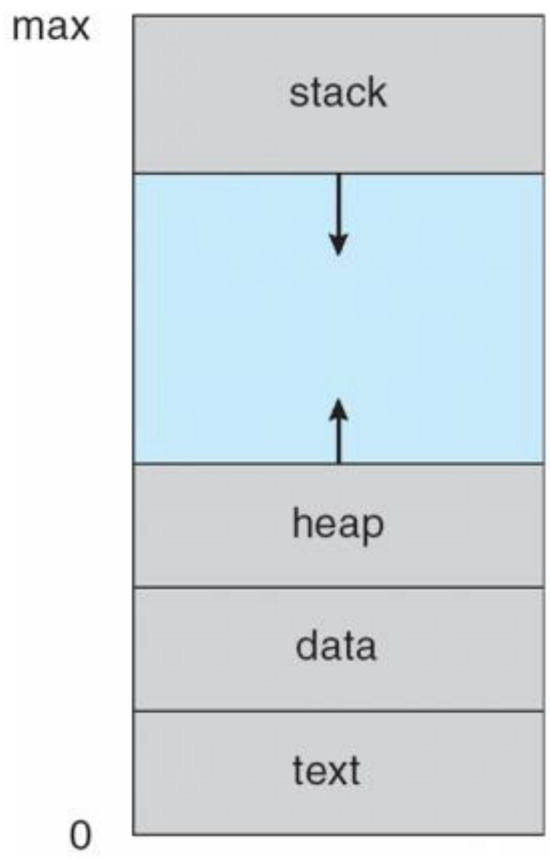
- It also includes the current activity:
  - The value of the **program counter**
  - The contents of the **processor's registers**.

- It also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables)

- It also includes the **data section**, which contains global variables, static variable (once created stay forever).

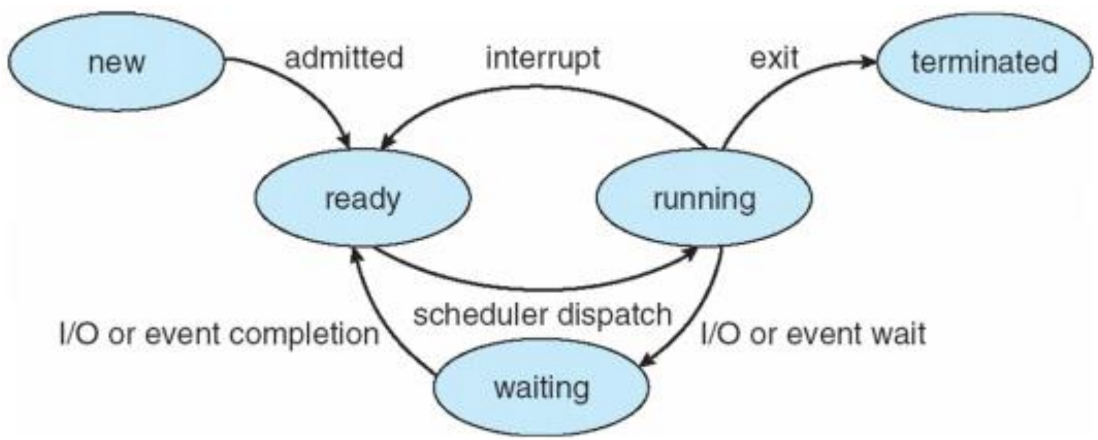
- It may also include a **heap**, which is memory that is dynamically allocated during process run time.

Process in Memory



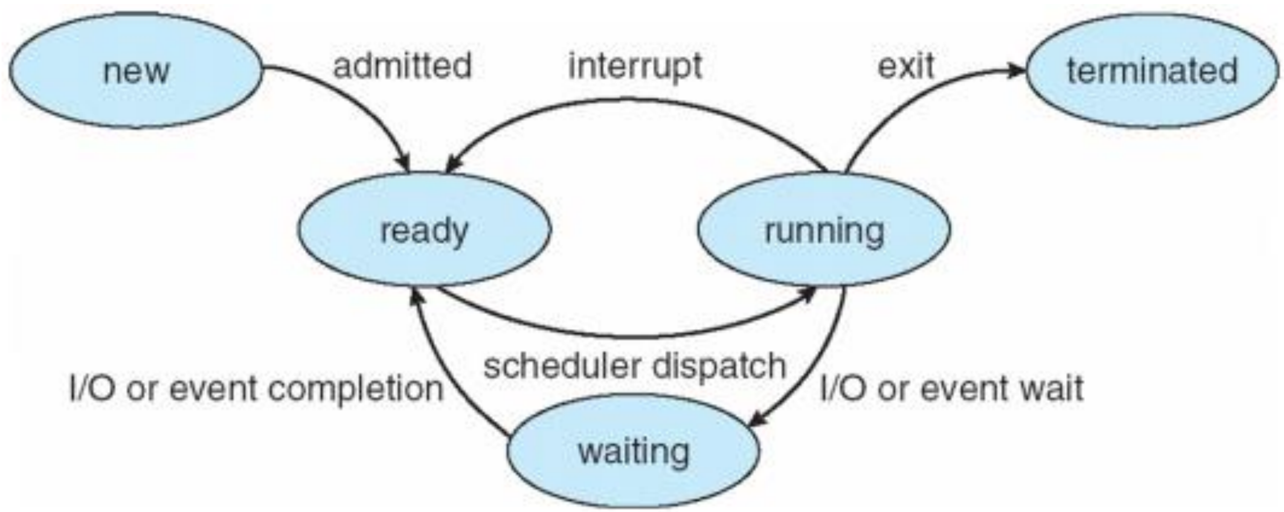
Process State

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



Process State

- Diagram of Process State



Process Control Block (PCB)

Information associated with each process (use to present a process in OS) (also called **task control block**)

- Process ID / Number – Unique ID.
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute (addr of next inst)
- 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

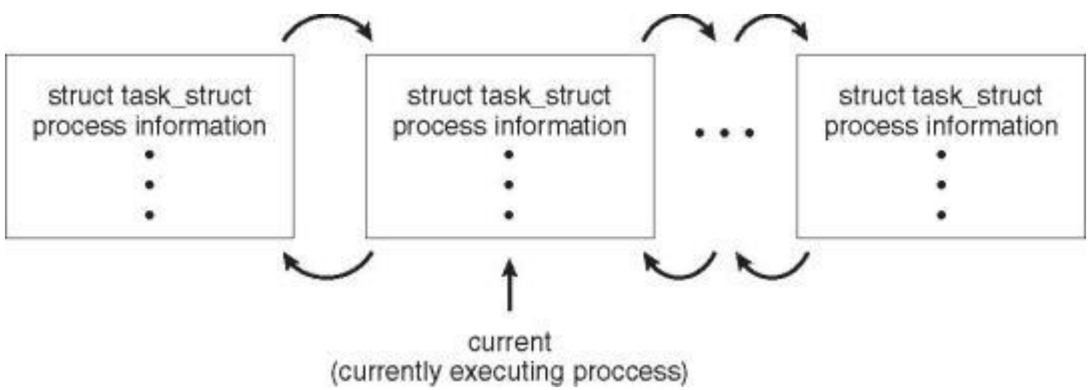
Threads

- 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**
- Need storage for thread details, multiple program counters in PCB
- Covered in the next chapter

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 */
```

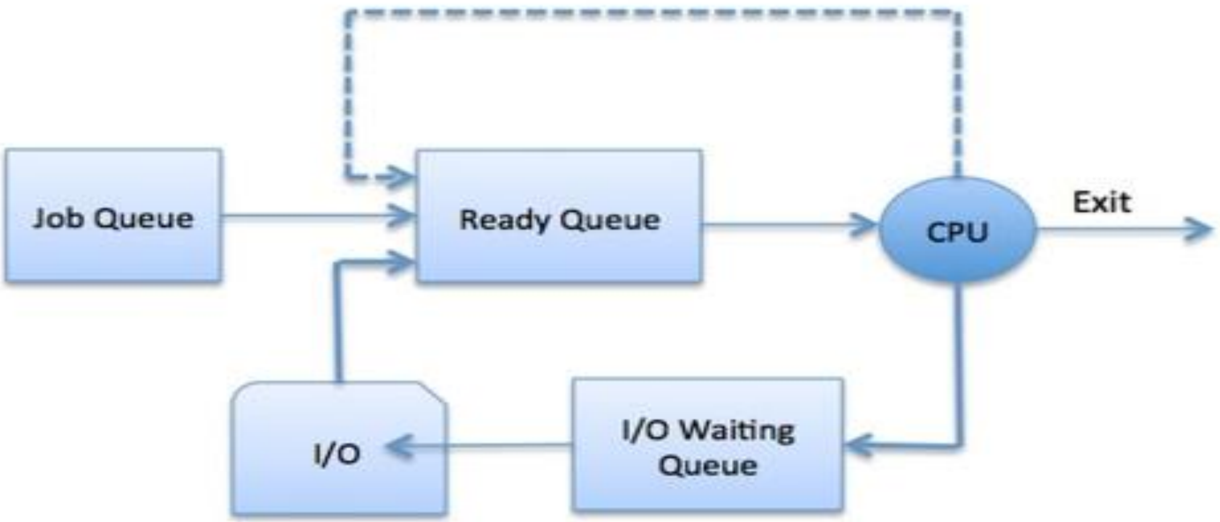


Process Scheduling

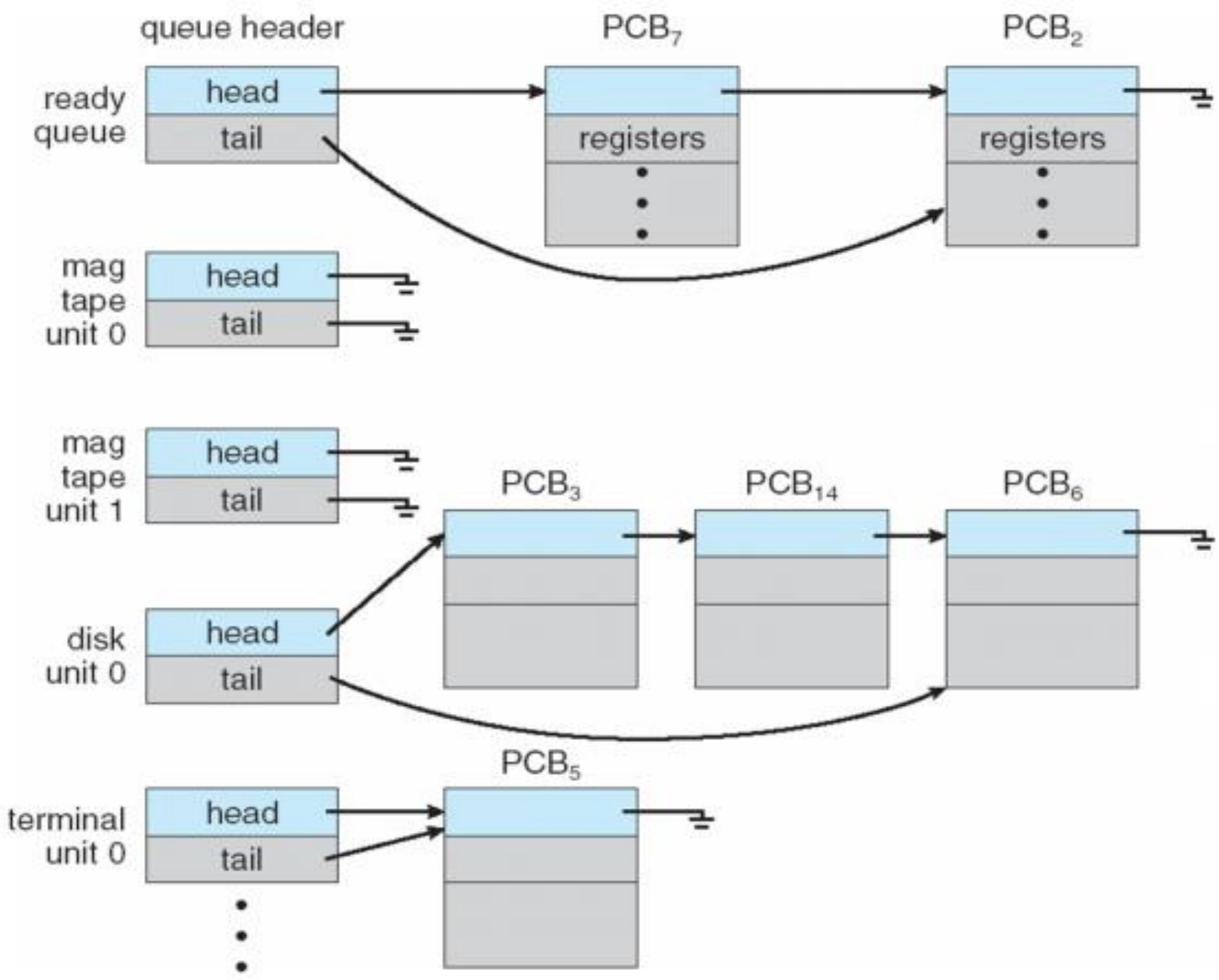
- Maximize CPU use
  - Quickly switch processes onto CPU for time sharing
- Process “gives” up the CPU under two conditions:
  - I/O request
  - After *N* units of time have elapsed (need a timer)
- Once a process gives up the CPU it is added to the “ready queue”
- **Process scheduler** selects among available processes in the ready queue for next execution on CPU

Scheduling Queues

- OS 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

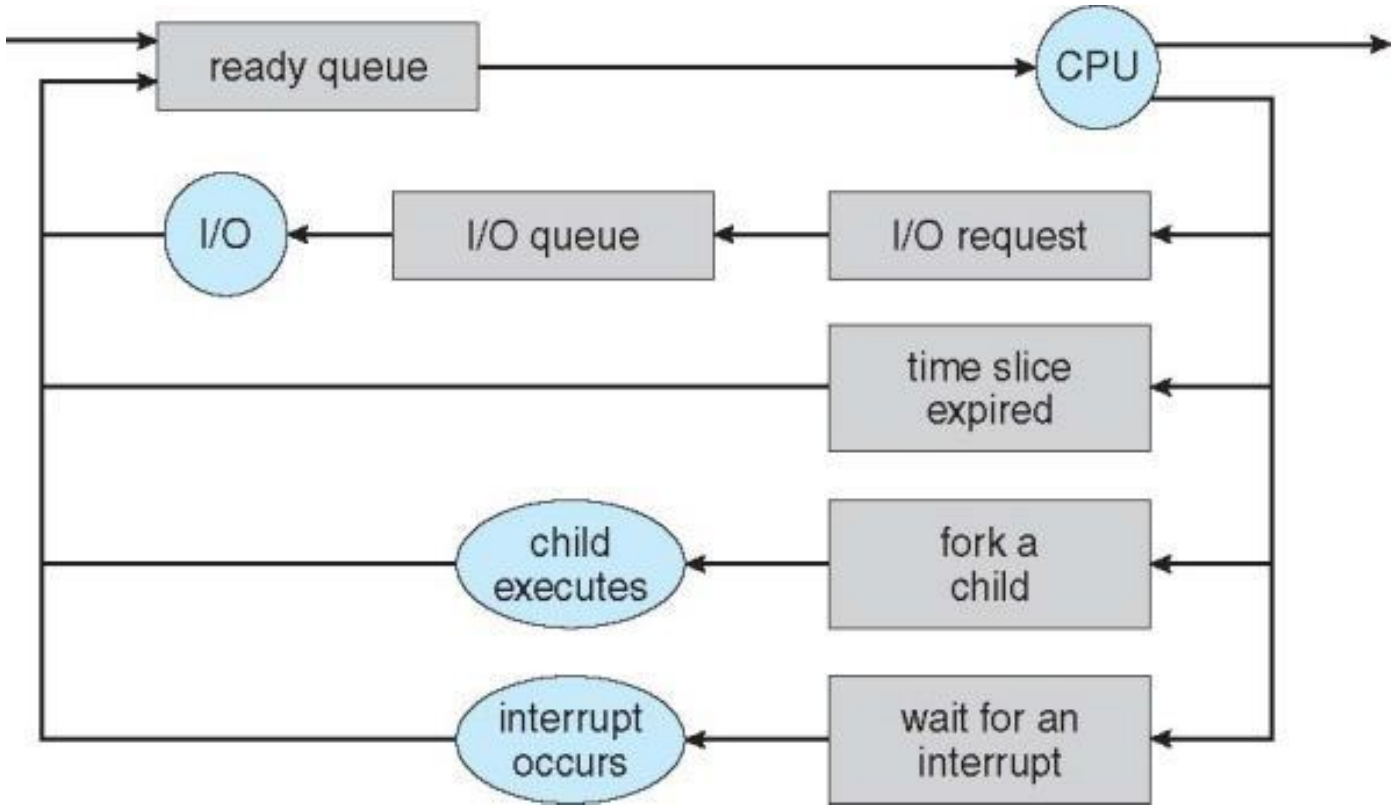


Ready Queue And Various I/O Device Queues

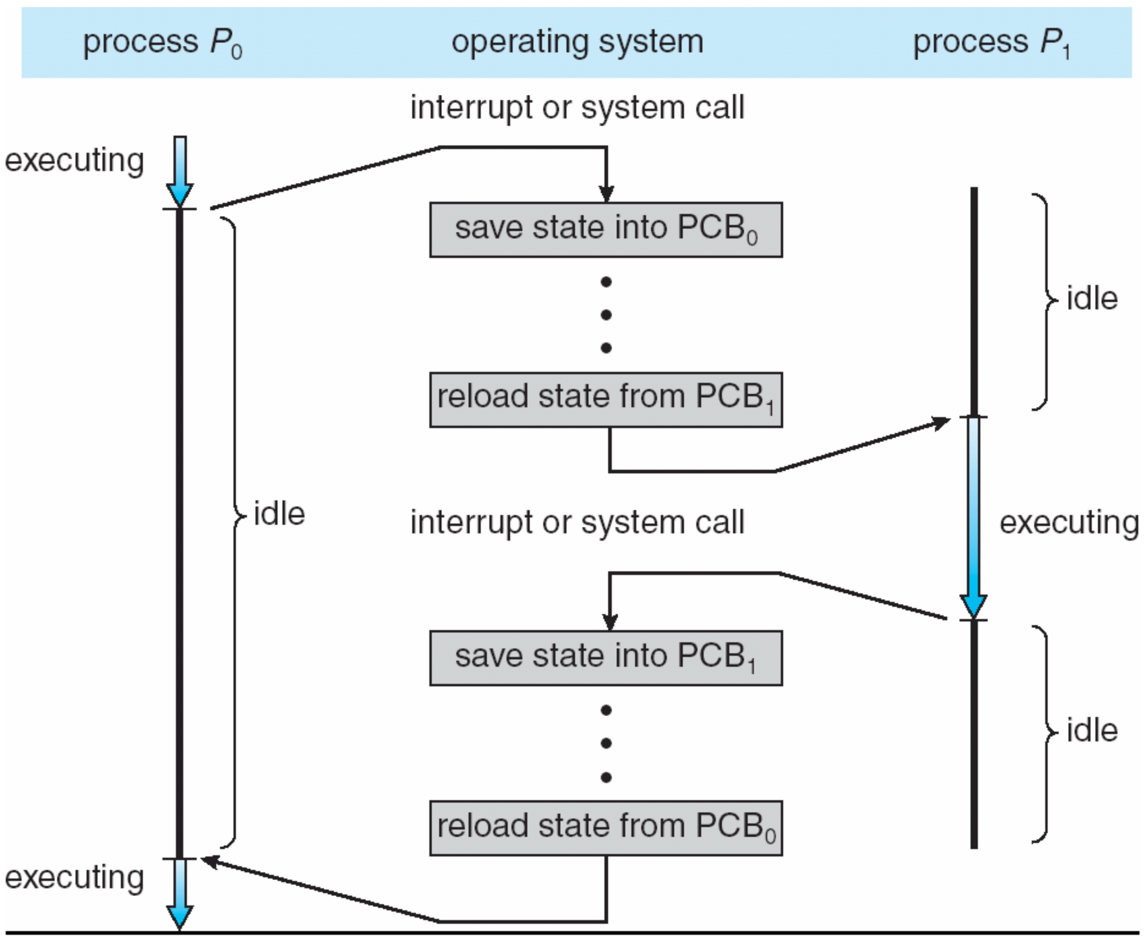


Representation of Process Scheduling

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



CPU Switch From Process to Process



Schedulers

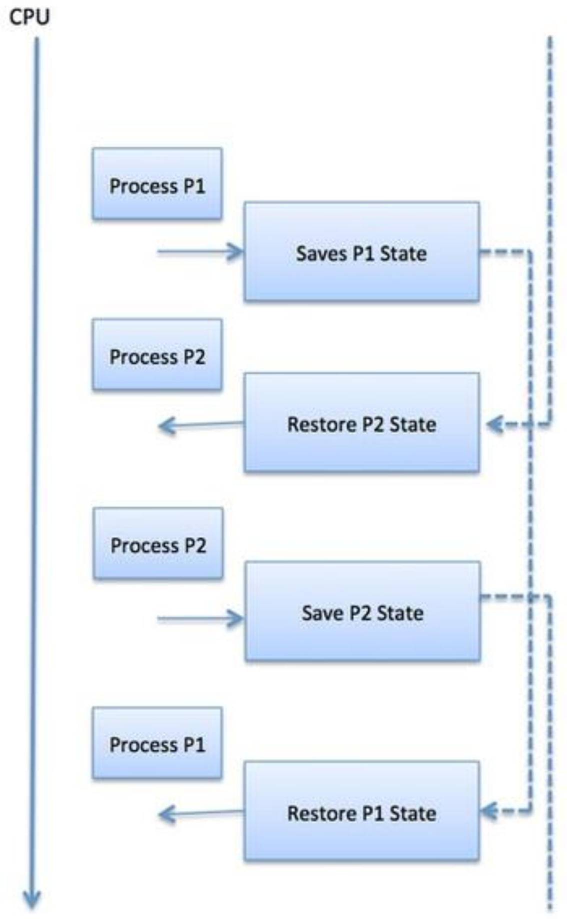
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates a CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects 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***

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Starting with **iOS 4**, it 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

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

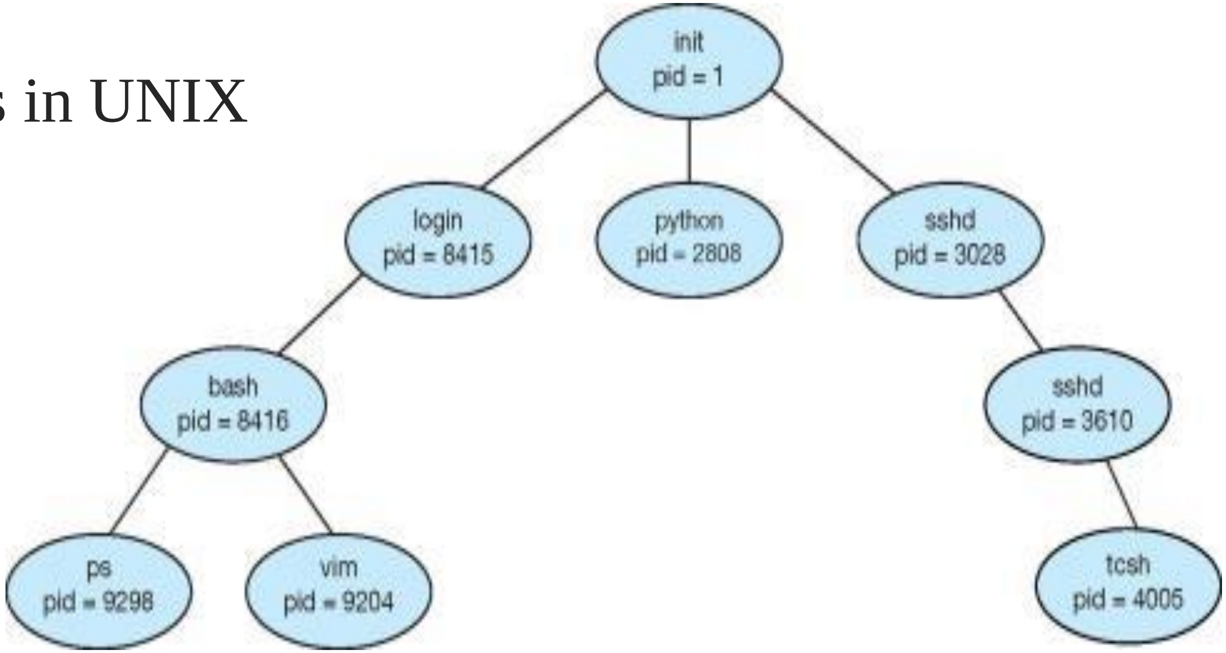


Operations on Processes

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

Process Creation

- A **process** may create other processes.
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, a process is identified and managed via a **process identifier** (**pid**)
- A Tree of Processes in UNIX

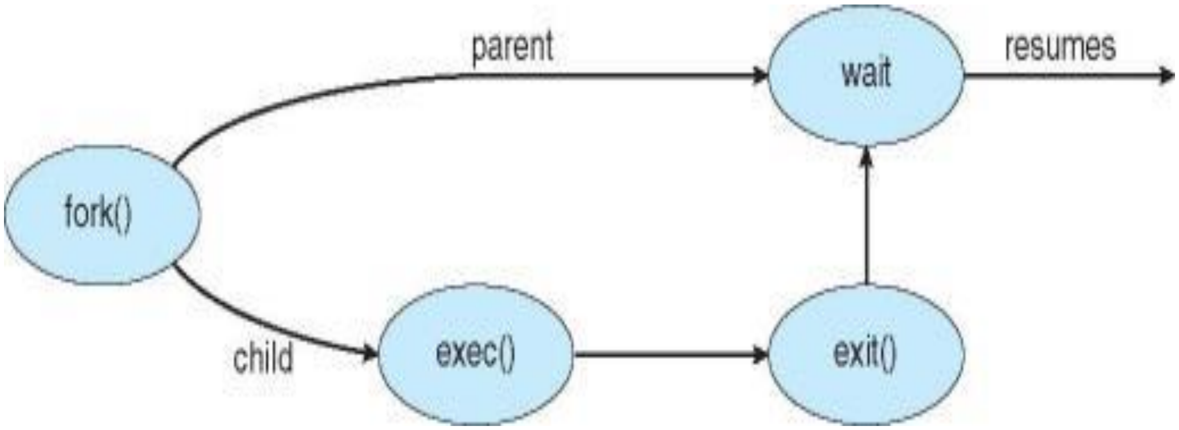


Process Creation (cont'd)

- Resource sharing among parents and children 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'd)

- Address space
  - A child is a duplicate of the parent address space.
  - A child loads a program into the address space.
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** replaces the process' memory space with a new program



C program to create a separate process in UNIX

```
int main()
{
pit.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;
}
```

C Program Forking Separate Process

- The C program illustrates how to create a new process UNIX.
- After **fork()** there are two different processes running copies of the same program. The only difference is that the value of pid for the child process is zero, while that for the parent is an integer value greater than zero
- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.
- The child process then overlays its address space with the UNIX command “ls” (used to get a directory listing) using the execlp() (a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes, the parent process resumes from the call to wait(), where it completes using the exit() system call.

Creating a Separate Process via Windows API

```
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 threat handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
     {
```

Creating a Separate Process via Windows API (cont'd)

```
    {
    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

- A process terminates when it finishes executing its final statement and it asks the operating system to delete it by using the **exit()** system call.
  - At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call.
  - All the resources of the process are deallocated by the operating system.
- A 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 (cont'd)

- Some operating systems do not allow a child process 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 **zombie**
  - **Status table yes but running no but parent process need to call exit() system**
- If parent terminated without invoking **wait**, process is **orphan**
- **Running yes but parent running no**

Inter process Communication (IPC)

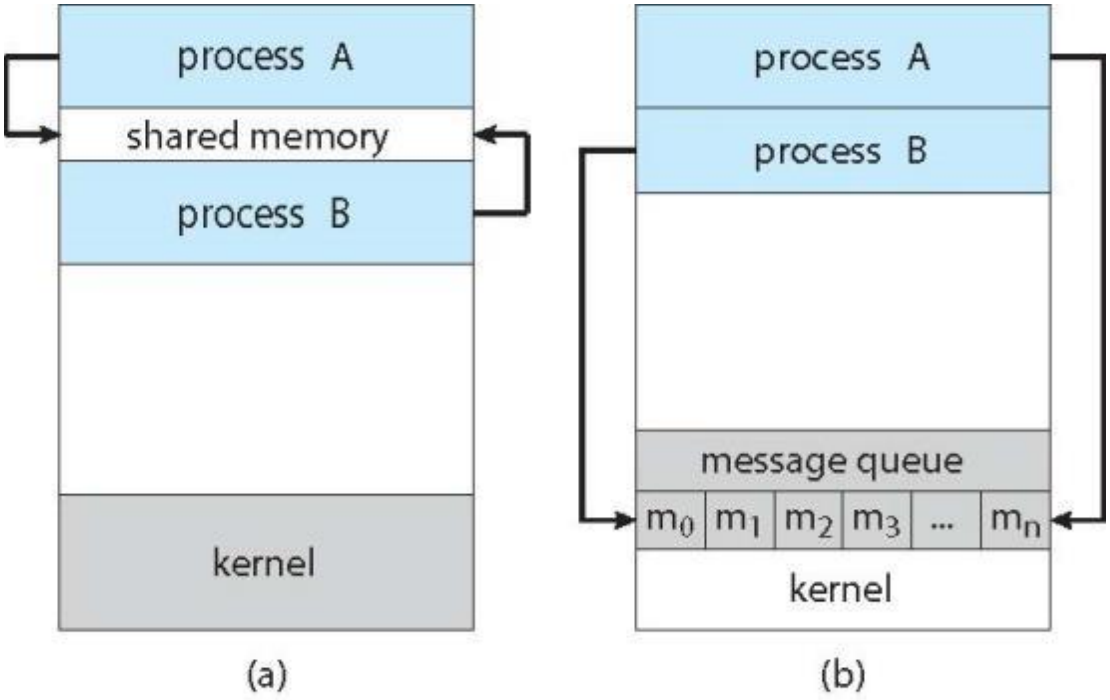
- Processes within a system may be ***independent*** or ***cooperating***
  - Cooperating processes can affect or be affected by other processes, including sharing data
  - Independent processes cannot affect other processes

- Reasons for having **cooperating** processes:
  - Information sharing
  - Computation speedup (multiple processes running in parallel- subtasks divided into processes)
  - Modularity (system design- several modules/process need comm.)
  - Convenience (multitasking-multiprogramming make user life easy)

- Cooperating processes need **inter process communication** (**IPC**)

Communications Models

- Two models of IPC (Inter process communication)
  - **Shared memory**
  - **Message passing**



Shared Memory Systems

- **A region of memory that is shared by cooperating processes.**
- 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

- Cooperating processes that access shared data need to synchronize their actions to ensure data consistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Illustration of the problem – The producer-Consumer problem
  - Producer process produces information that is consumed by a Consumer process.
  - The information is passed from the Producer to the Consumer via a buffer.
  - Two types of buffers can be used:
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that a fixed buffer size

Bounded-Buffer Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

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

- Solution presented in the next two slides is correct, but only 9 out of 10 buffer elements can be used

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 */

}
```

Message Passing Systems

- Mechanism for processes to communicate and to synchronize their actions (messages are exchanged between processes)
  - 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'd)

- 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
- 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'd)

- 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 Issues

- 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. (Algorithm like RR)

Blocking and Non-blocking schemes

- 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**

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in 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

Example of IPC Systems

- There are four different IPC systems.
  - POSIX API for shared memory
  - Mach operating system, which uses message passing
  - Windows IPC, which uses shared memory as a mechanism for providing certain types of message passing.
  - Pipes, one of the earliest IPC mechanisms on UNIX systems.

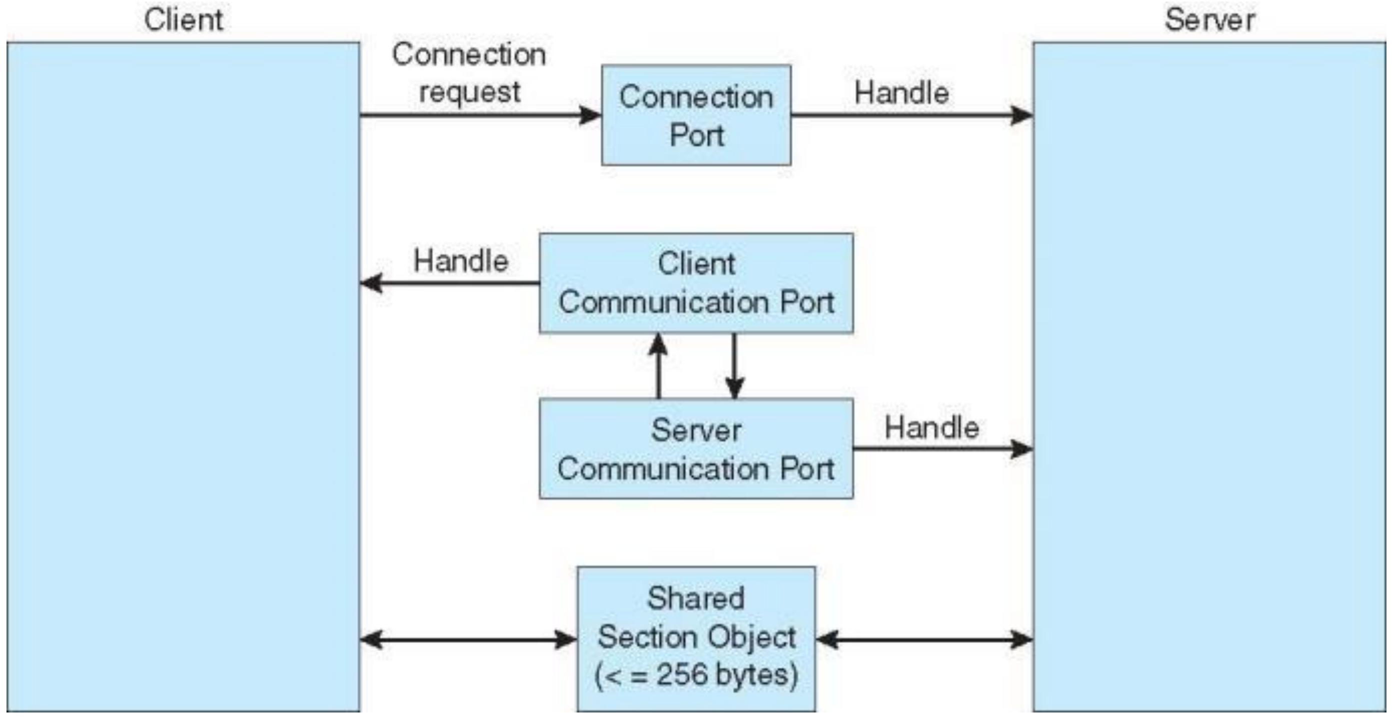
Mach

- 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

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 (an abstract reference to a resource) to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates a private **communication port** and returns the handle 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



Communications in Client-Server Systems

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

Books

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9th

References

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9^{th}