OPERATING SYSTEMS
CSE-4041

Lecture-3

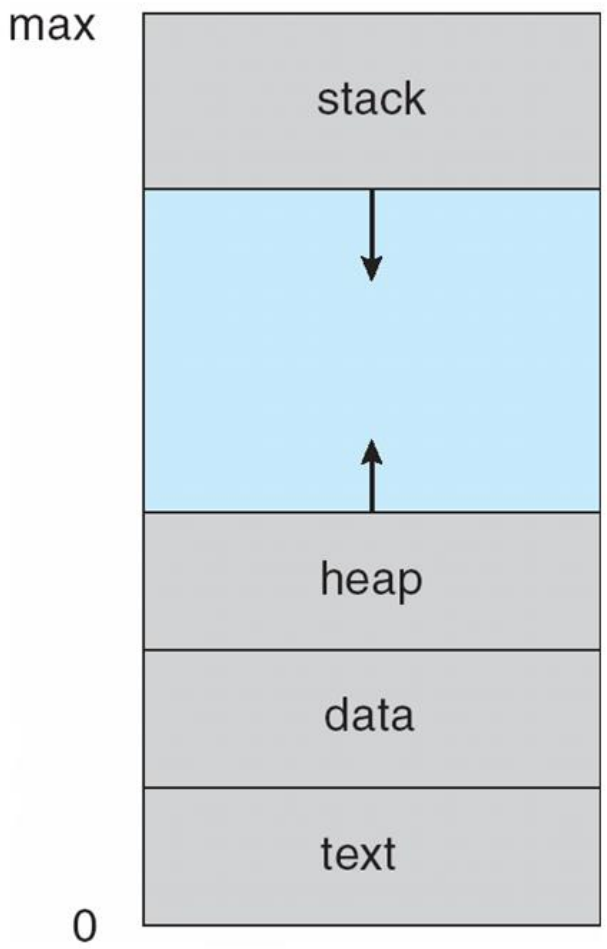
Process Management

Nitesh Kumar Jha
Assistant Professor
ITER S'O'A (Deemed to be University)

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 in Memory



Process Concept (Cont.)

- 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

Processes

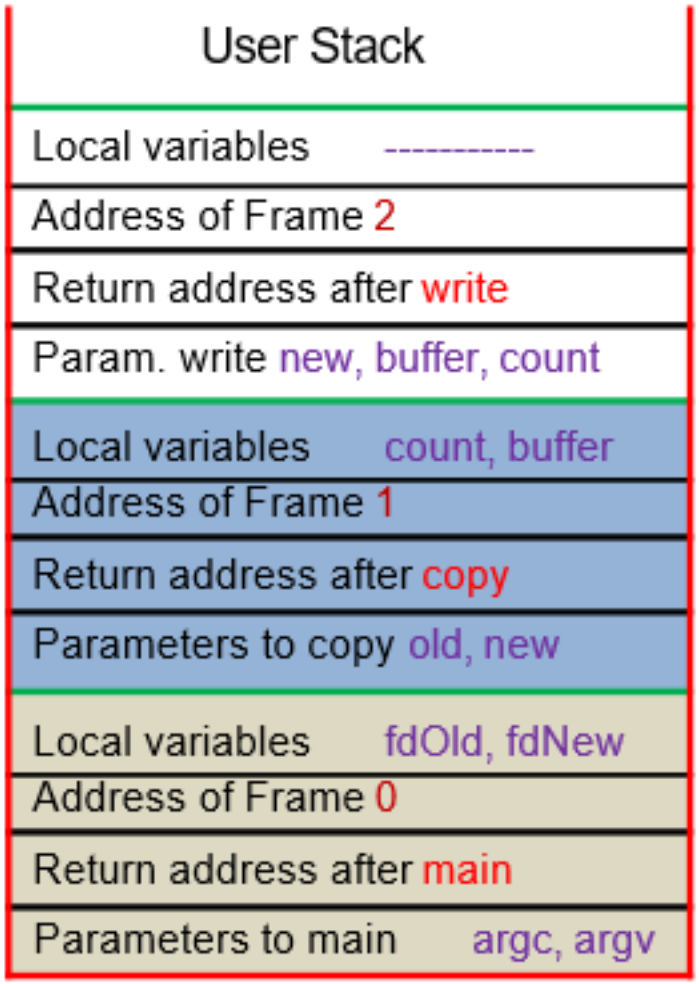
- A process is the execution of a program and consists of text, data and stack in memory.
- The kernel loads an executable file into memory during an exec system call,
- Program's text and data are loaded to corresponding regions in memory. But the stack region automatically created with dynamically adjusted size.
  - Logical frames are pushed when calling a function and popped when returns.
  - A special register called the stack pointer indicates the current stack depth.
  - Frame contains the parameters to a function, local variables, and data required to recover previous stack frame, value of the program counter and stack pointer.

Example Program- copy a file

```
#include<fcntl.h>
main(int argc, char *argv[]){
   int fdOld, fdNew;
   fdOld = open(argv[1], O_RDONLY);
    fdNew = creat(argv[2], 0666);
   copy(fdOld,fdNew);
 }

copy(int old, int new)
{  int count;
    while((count = read(old, buffer, sizeof(buffer))) > 0)
          write(new, buffer, count);
 }
```

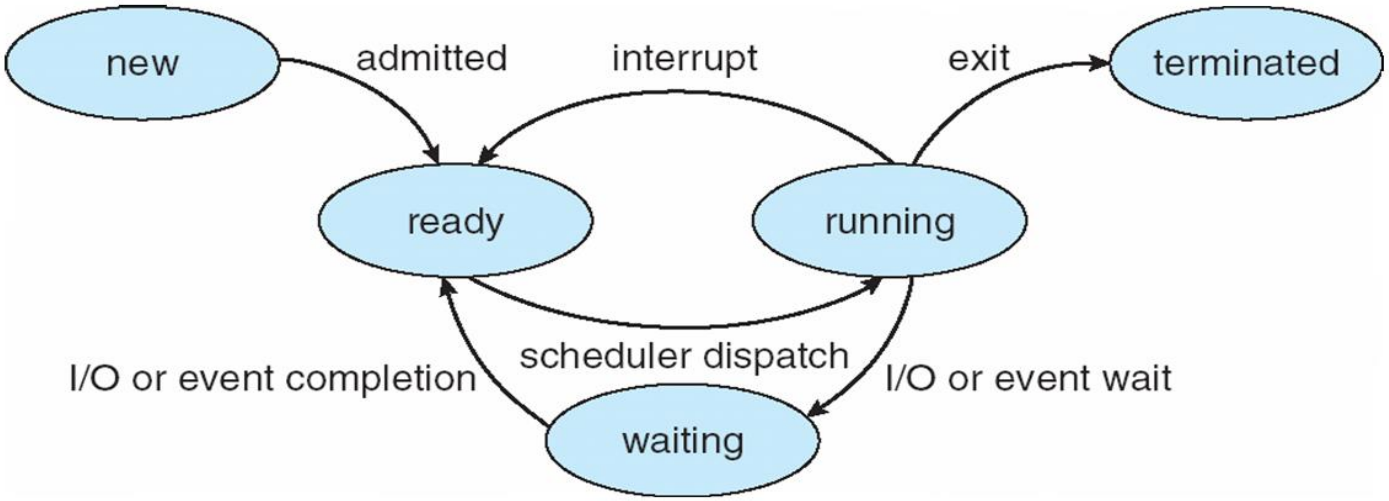
Ex. User stack for copy program



- The process startup procedure calls the main function with two parameters, thus pushing frame 1
- main then calls copy with two parameters and pushes Frame 2
- Finally process invoked system calls write that causes pushing frame 3
- When write is executed, causes an interrupt that results in a hardware switch to kernel mode.
- Now the process get executed in kernel code and uses kernel stack.
- Kernel stack contains stack frames for functions executing in kernel mode similar to user stack

Process States

- 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

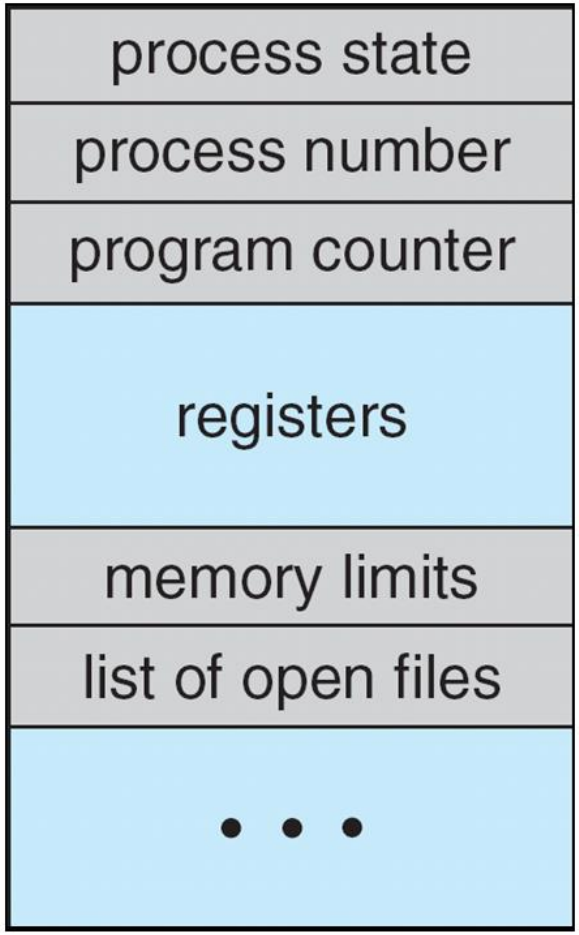


Process Control Block (PCB)

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



Context of a Process

- The context of a process is its state,
  - its text
  - the values of its global user variables and data structures
  - the values of machine registers it uses
  - The values stored in its process table
  - the contents of its user and kernel stacks.

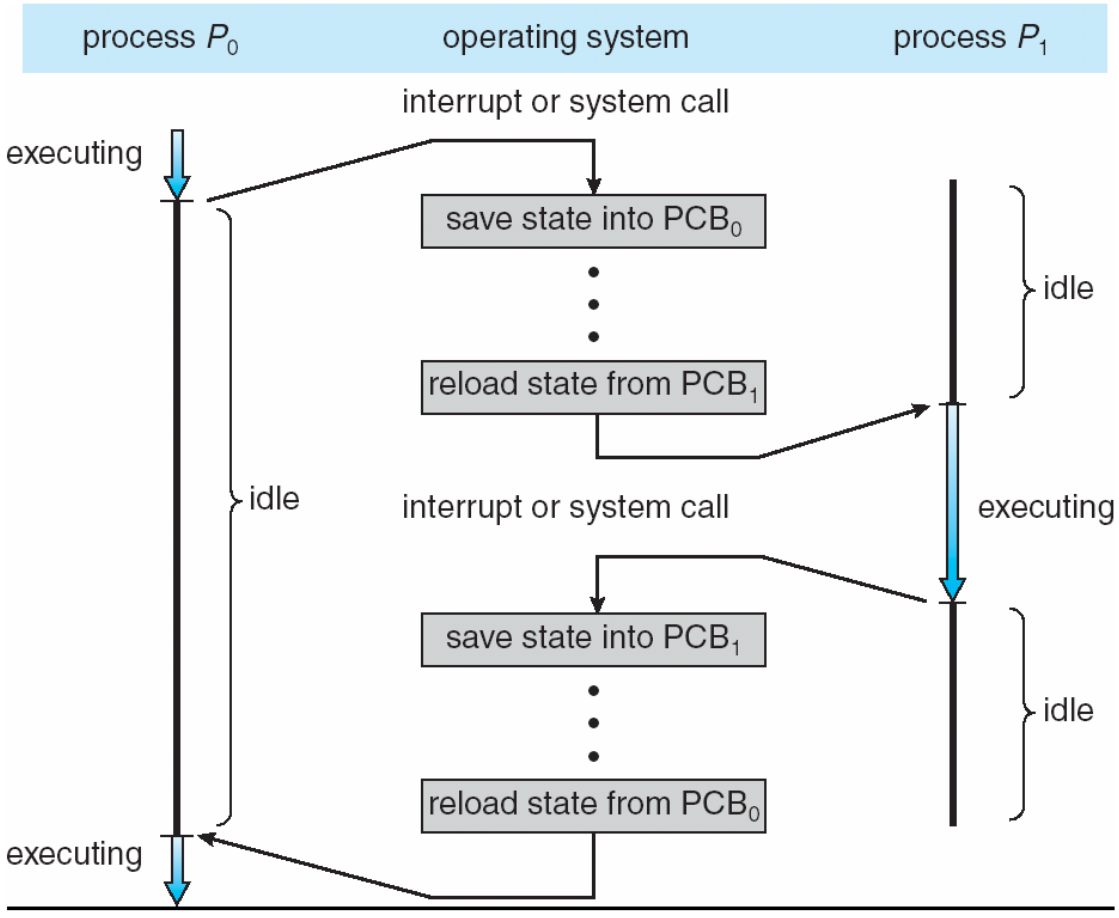
- When executing a process, the system is said to be executing in the context of the process.

- To execute another process, it does a context switch, so that the system executes in the context of the other process.

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

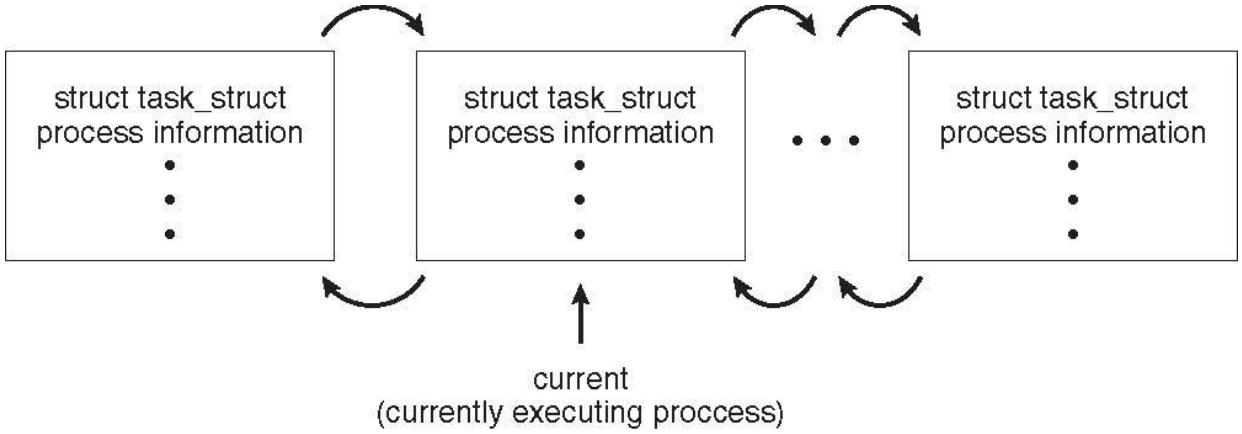
CPU Switch From Process to Process



Process Representation in Linux

Represented by the C structure

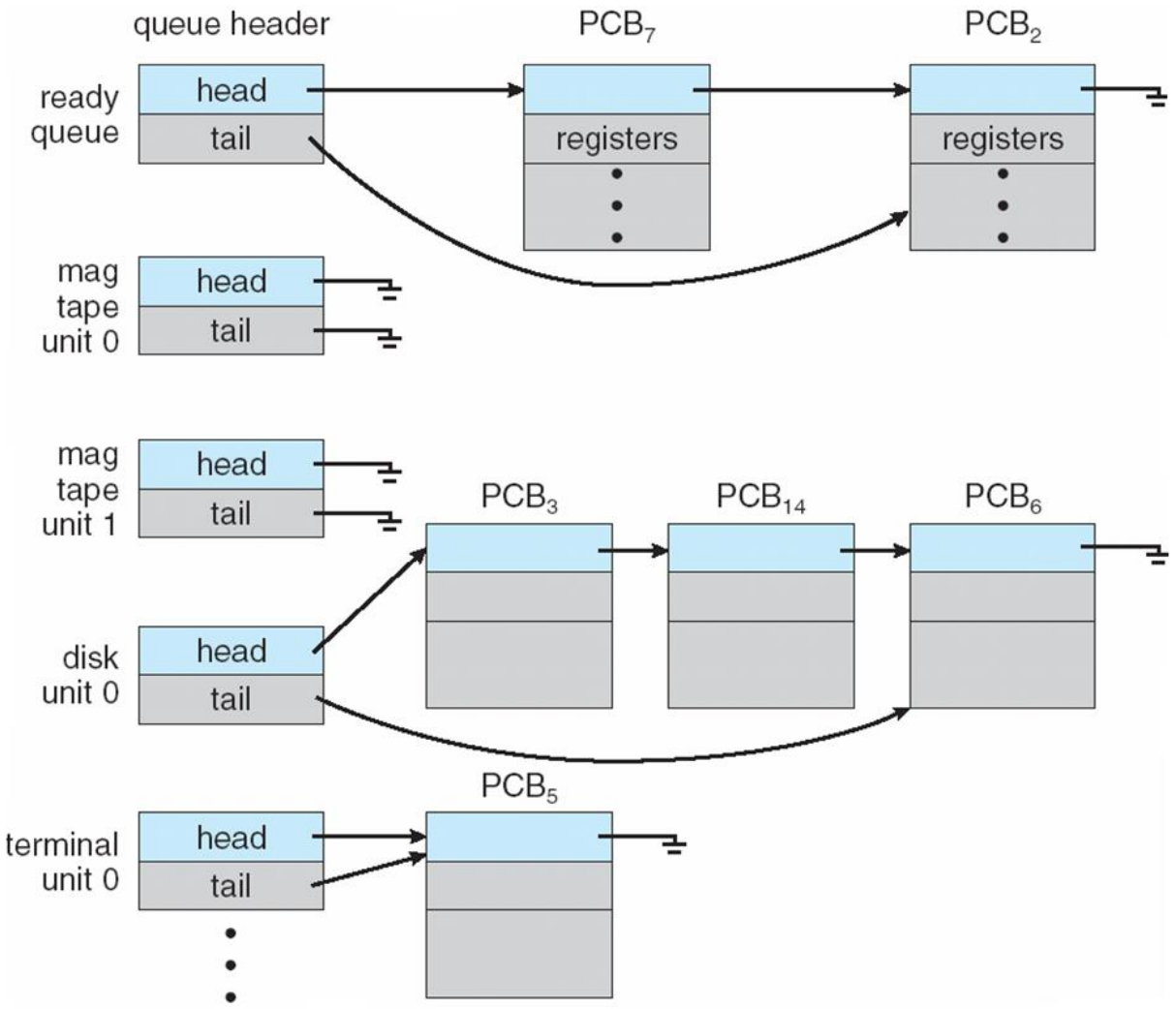
```
task_struct
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open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



Process Scheduling

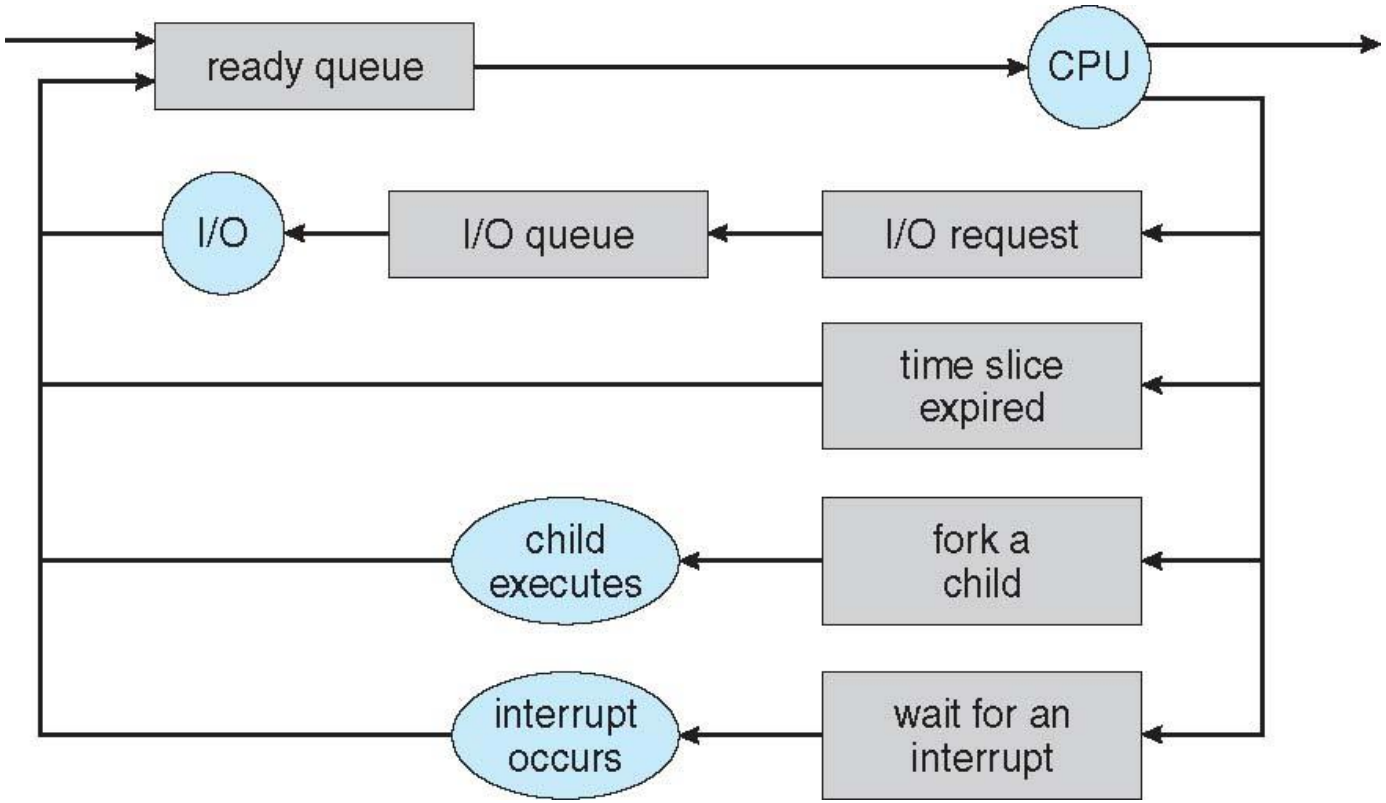
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing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



Process Scheduling: Queueing Diagram

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



Schedulers

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types –
  - Long-Term Scheduler
  - Short-Term Scheduler
  - Medium-Term Scheduler
- **Long-term scheduler (or job scheduler)** – A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution i.e., 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
  - The degree of multiprogramming describes the maximum number of processes that a single-processor system can accommodate efficiently.

Schedulers

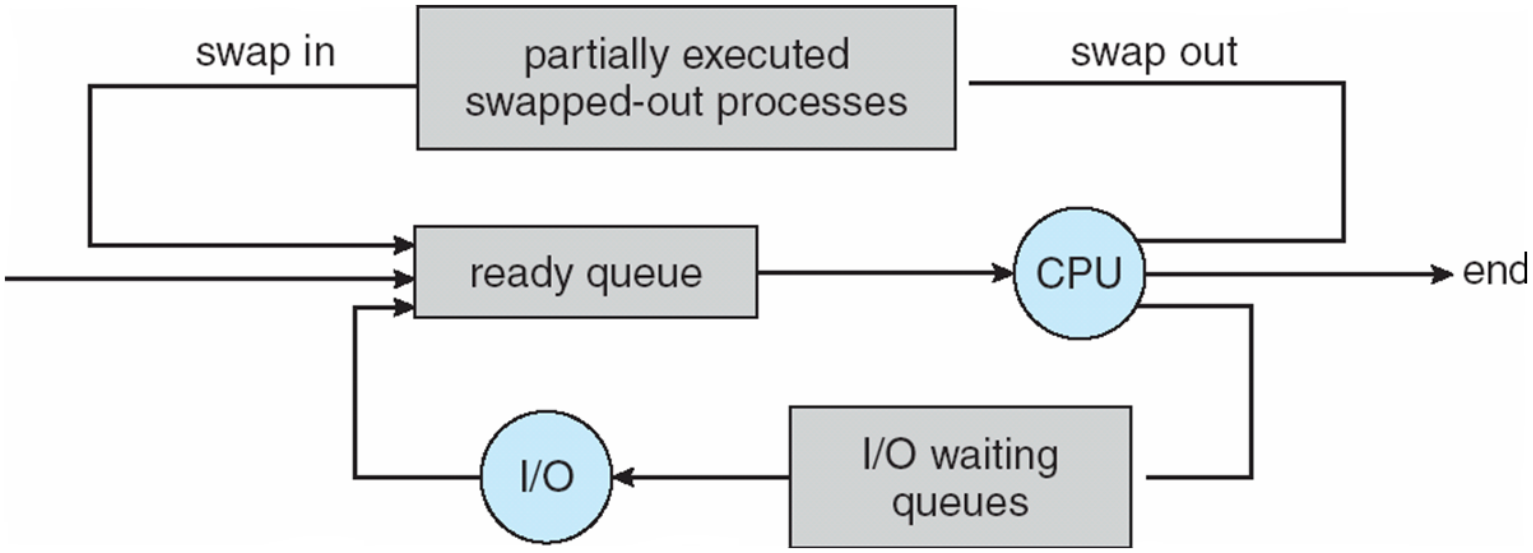
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of *ready state to running state* of the process.
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- 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***

Impact of Types of processes

- All processes are I/O bound => ready queue almost empty
- All processes are CPU bound => I/O queue almost empty
- Therefore Long-term scheduler should select a good
- ***process mix***
- In UNIX and MS-Windows, there is no long-term scheduler.
- Some system may introduce medium-term scheduler

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - A running process may become suspended if it makes an I/O request.
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



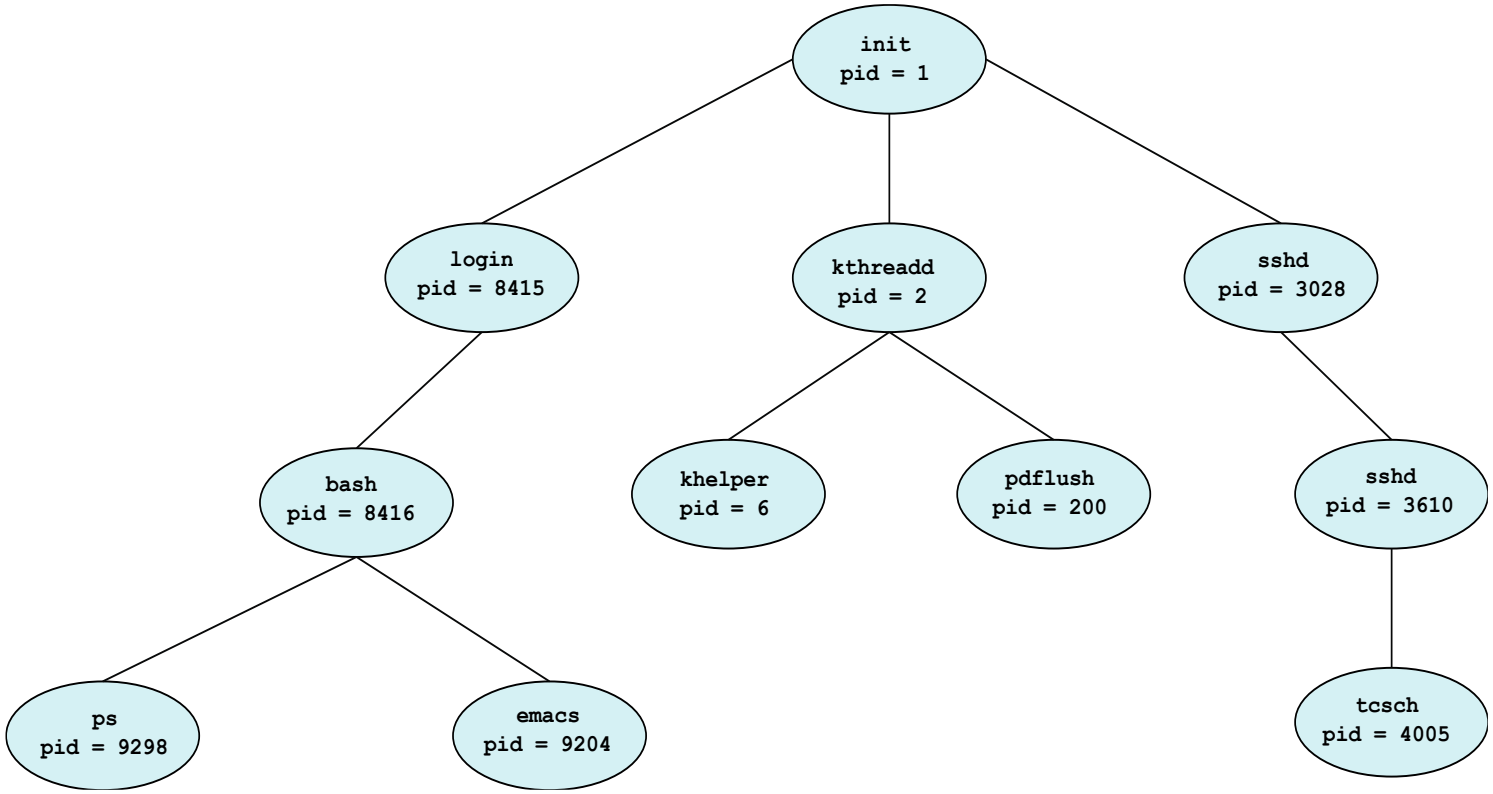
Process Scheduling In short

- LONG TERM SCHEDULER decide the number of process going to execute.
- SHORT TERM SCHEDULER improves the performance of the CPU by minimizing/remove the indolence of the CPU by processing as many process it can.
- As short term scheduler load many process so it may cause CPU overloading so here MIDDLE TERM SCHEDULER comes in to picture by removing process So, the processor can run the processes smoothly.

Process Creation

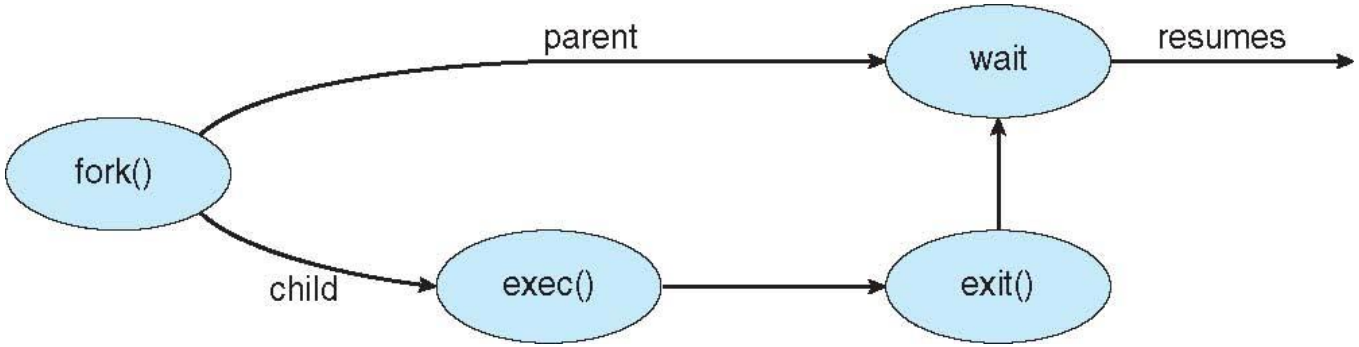
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset 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



Process Creation

- Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork().
- **Negative Value**: creation of a child process was unsuccessful.
  Zero: Returned to the newly created child process.
  Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

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;
}
```

C Program Forking separate process

```
#include <sys/types.h>                    /// Identify the values of pid at lines A,B,C and D.
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid, pid1;
        pid = fork(); /* fork a child process */                    ///Assume that the actual pid's
        if (pid < 0) { /* error occurred */                  //of the parent and child are 2600 and 2603
                fprintf(stderr, "Fork Failed");
        return 1;
        } else if (pid == 0) { /* child process */
                pid1 = getpid();
                printf("child: pid = %d",pid); /* A */
                printf("child: pid1 = %d",pid1); /* B */
        }else { /* parent process */
                pid1 = getpid();
                printf("parent: pid = %d",pid); /* C */
                printf("parent: pid1 = %d",pid1); /* D */
                wait(NULL);
        }
        return 0;
        }
```

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**

Multiprocessing 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 multiprocessing 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

Multitasking in Mobile Systems

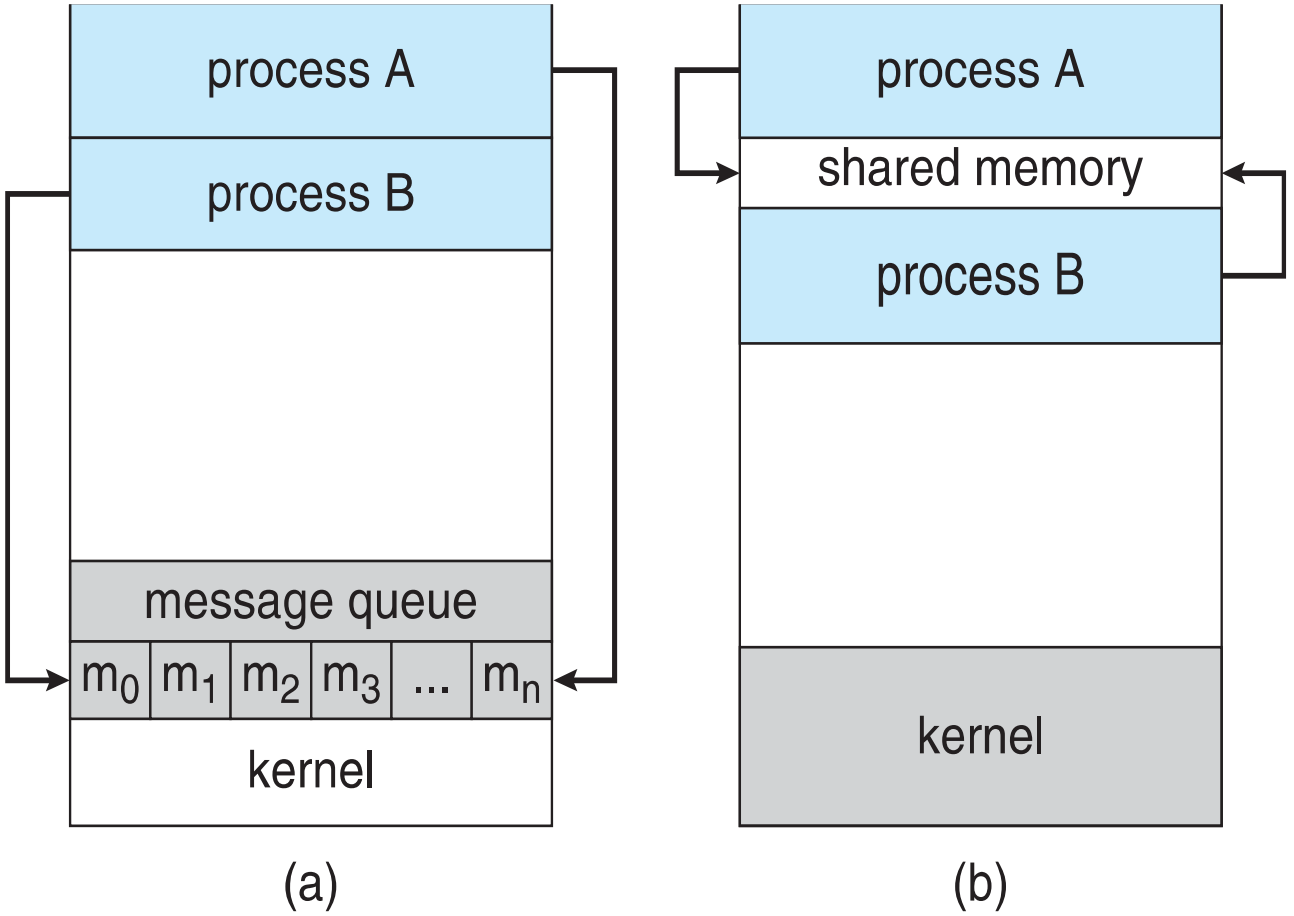
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not 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

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- **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.



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;
```

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 */
}
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

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 will be discussed in great details in future lecture

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

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

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy 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

- 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

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

```
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

END OF LECTURE

THANK YOU