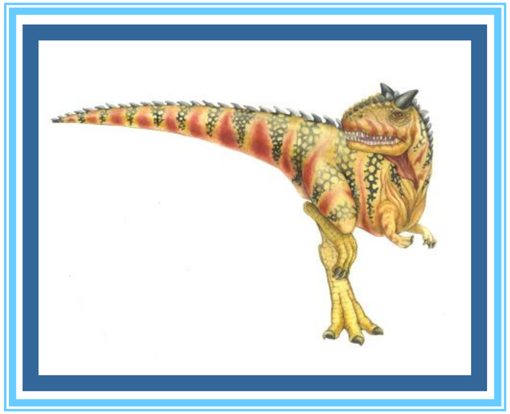
Chapter 3: Processes



Chapter 3: Processes

- Defining Process
- Process Scheduling
- Operations on Processes
- Interprocess Communication (IPC)
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including:
  - scheduling
  - creation and termination
  - and communication
- To explore interprocess communication using
  - shared memory, and
  - message passing

Process Concept

- An operating system executes a variety of programs:
  - Batch system – "**jobs**"
  - Time-shared systems – "**user programs**" or "**tasks**"
- We will use the terms ***job*** and ***process*** almost interchangeably
- **Process** – is a program in execution (informal definition)
- 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, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process In Memory

- In memory, a process consists of **multiple parts:**
  - **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

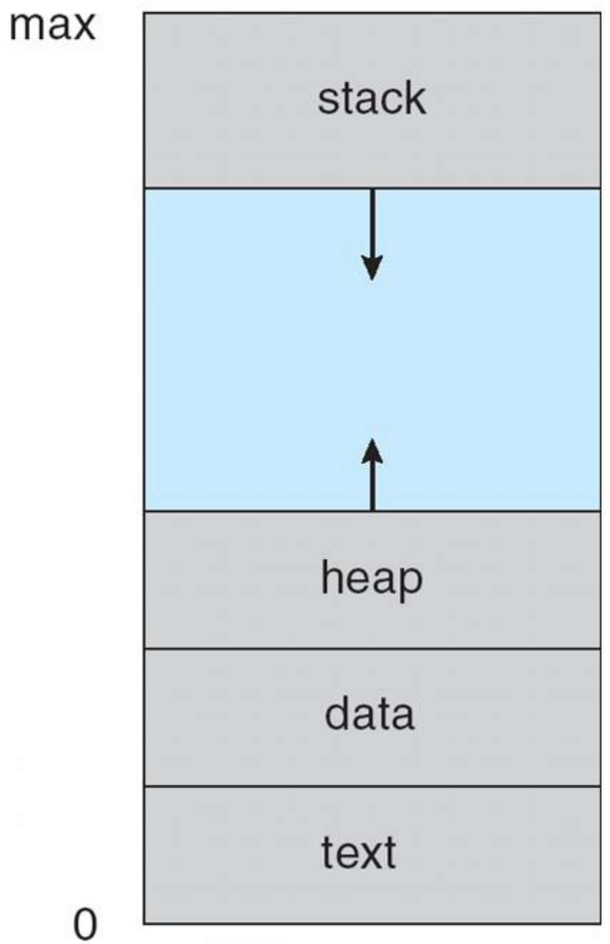
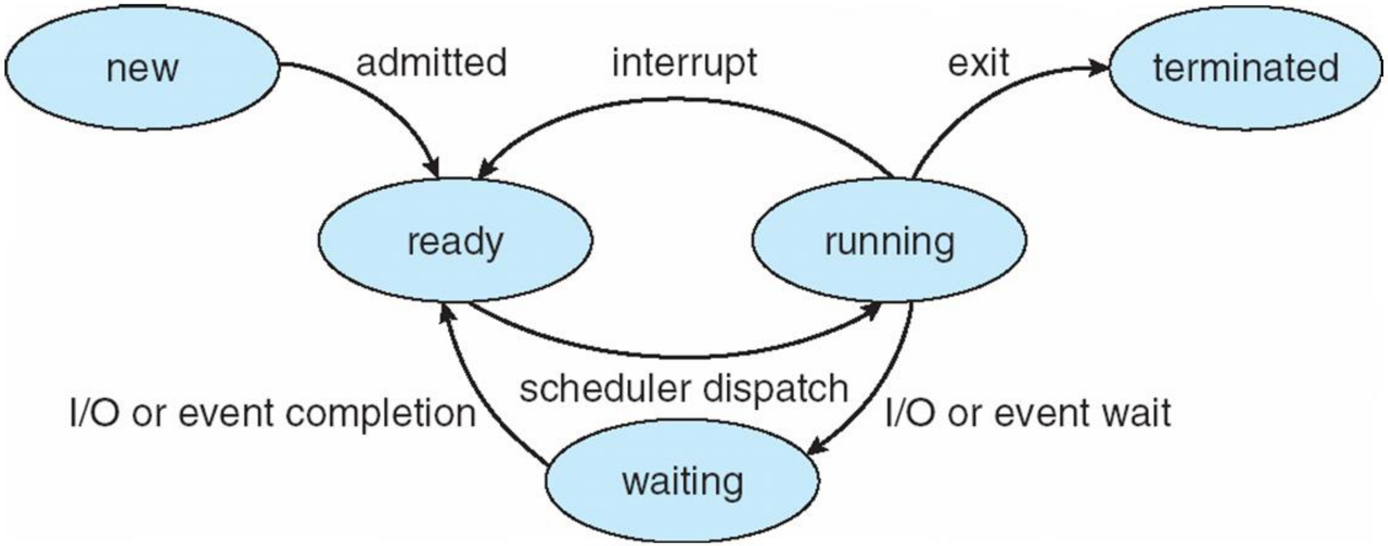


Diagram of Process State

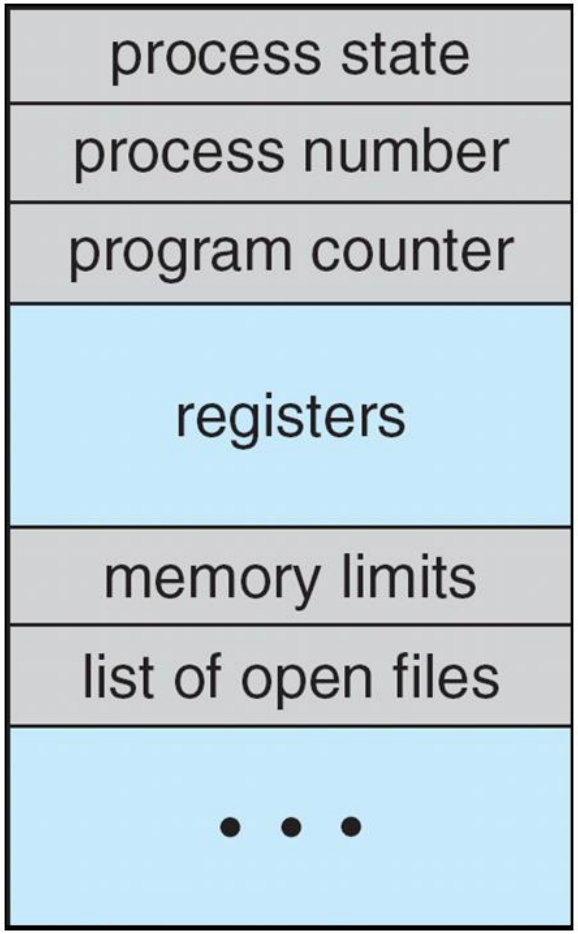


- As a process executes, it changes **state**
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process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

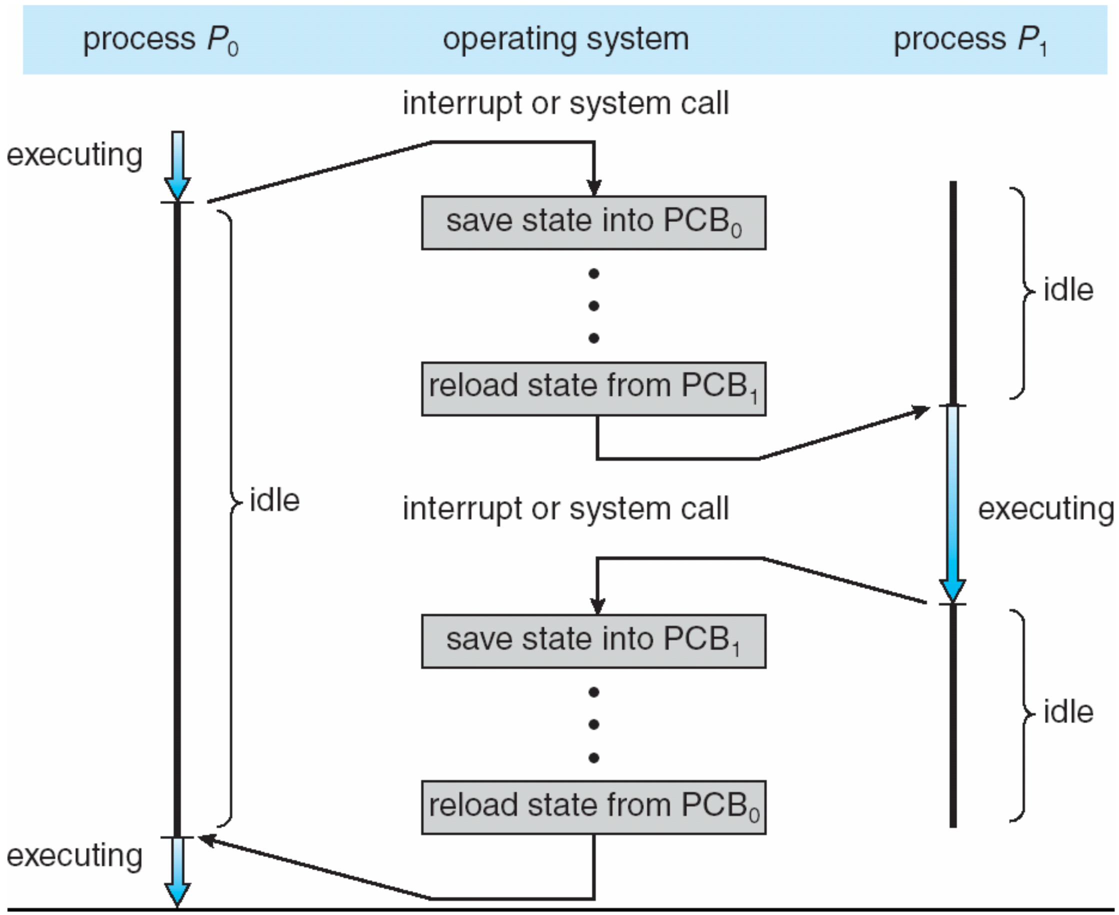
Process Control Block (PCB)

Each process is represented in OS by PCB

- PCB - info associated with the 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



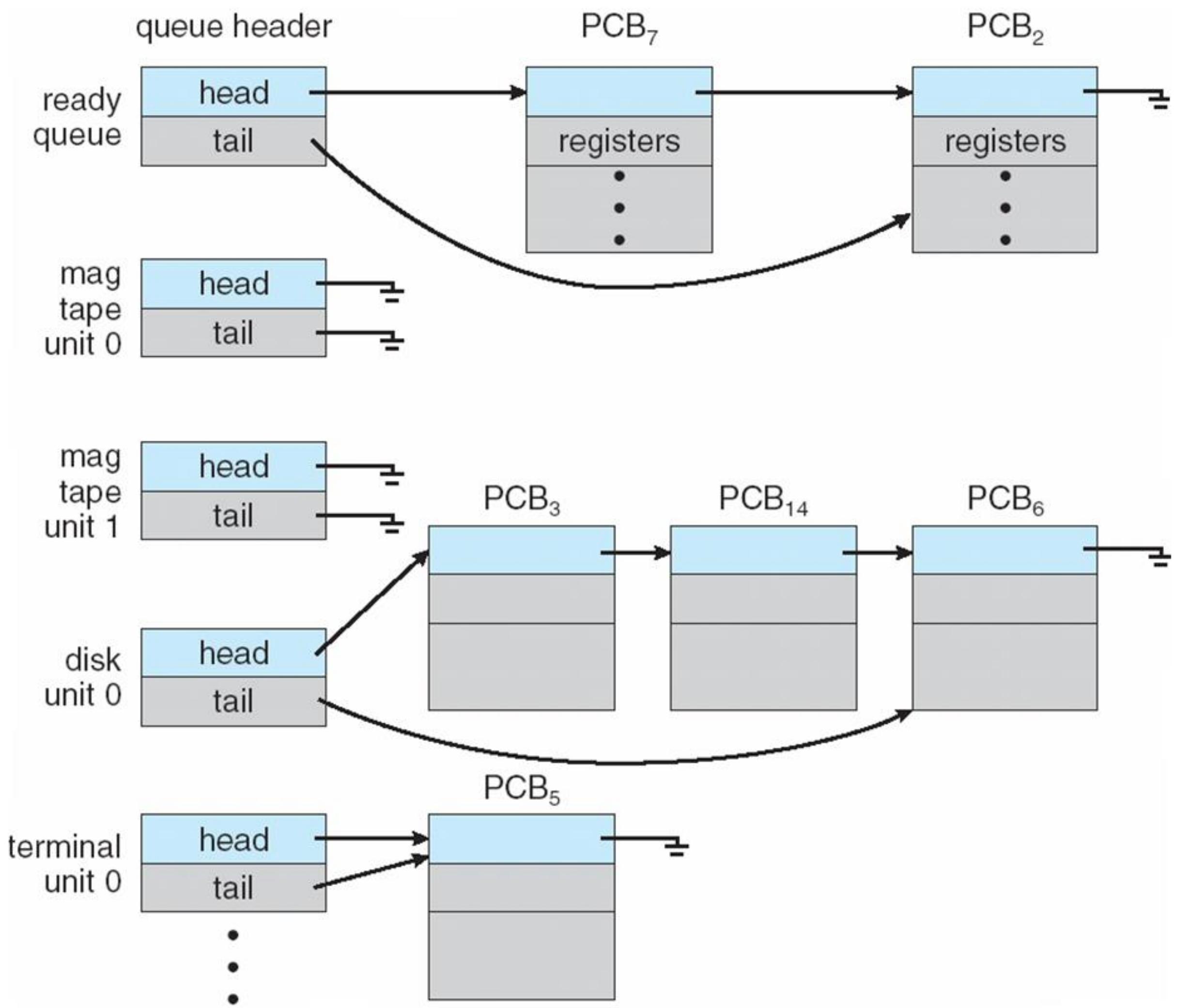
Threads

- So far, process has a single thread of execution
  - One task at a time
- Consider having multiple program counters per process
  - Multiple locations can execute at once
  - Multiple tasks at a time
  - Multiple threads of control -> **threads**
- PCB must be extended to handle threads:
  - Store thread details
  - Multiple program counters
- Details on threads in the next chapter

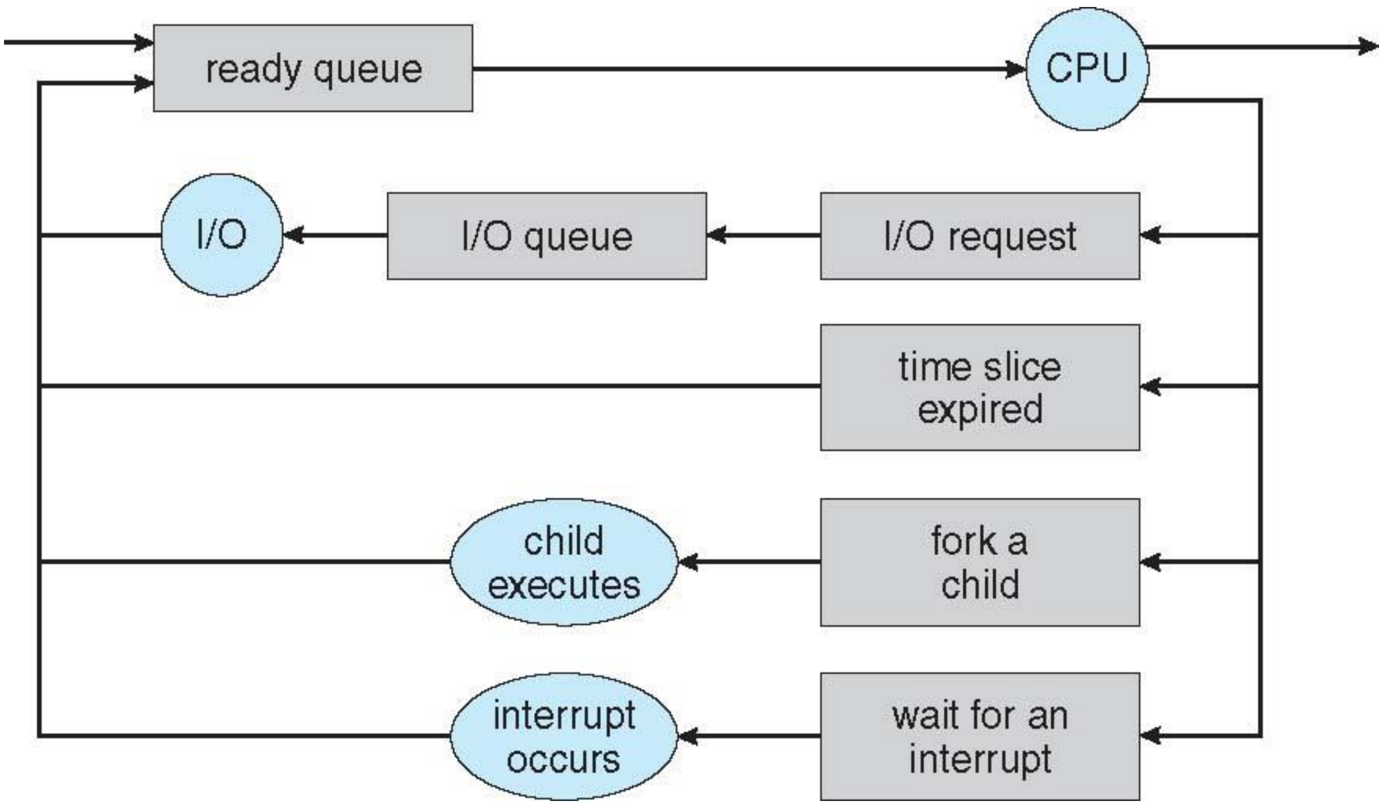
Process Scheduling

- Goa of multiprogramming:
  - Maximize CPU use
- Goal of time sharing:
  - Quickly switch processes onto CPU for time sharing
- **Process scheduler** – needed to meet these goals
  - Selects 1 process to be executed next on CPU
  - Among available processes
- 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**
  - a common representation of process scheduling
  - represents queues, resources, flows

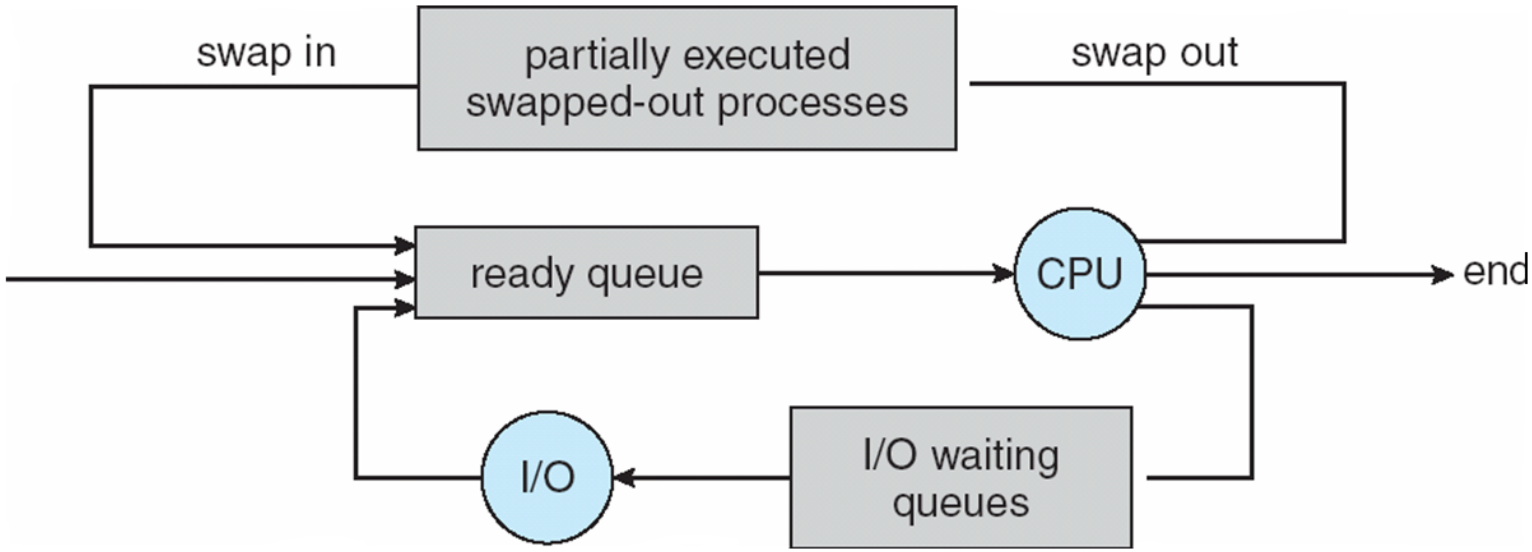
Schedulers

- **Scheduler –** component that decides how processes are selected from these queues for scheduling purposes
- **Long-term scheduler** (or **job scheduler**)
  - On this slide - “LTS” (LTS is not a common notation)
  - In a **batch system**, more processes are submitted then can be executed in memory
    - They are spooled to disk
  - LTS selects which processes should be brought into the ready queue
  - LTS is invoked infrequently
    - (seconds, minutes) ⇒ (may be slow, hence can use advanced algorithms)
  - LTS controls the **degree of multiprogramming**
    - The number of processes in memory
- 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
- LTS strives for good ***process mix***

Short-Term Scheduler

- **Short-term scheduler** (or **CPU scheduler**)
  - Selects 1 process to be executed next
    - Among ready-to-execute processes
      - From the ready queue
    - Allocates CPU to this process
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently
    - (milliseconds) $\Rightarrow$ (must be fast)
    - Hence cannot use costly selection logic

Addition of Medium Term Scheduling



- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Used by time-sharing OSes, etc
    - Too many programs →poor performance →users quit
- Key idea:
  - Reduce the degree of multiprogramming by **swapping**
  - **Swapping** removes a process from memory, stores on disk, brings back in from disk to continue execution

Context Switch

- **Context** of a process represented in its PCB
- **Context switch**
  - When CPU switches to another process, the system must:
    1. **save the state** of the old process, and
    2. load the **saved state** for the new process
- 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
    - more details in Chapter 8
- This overhead time is dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU
    - multiple contexts are loaded at once
    - switch requires only changing pointer to the right set

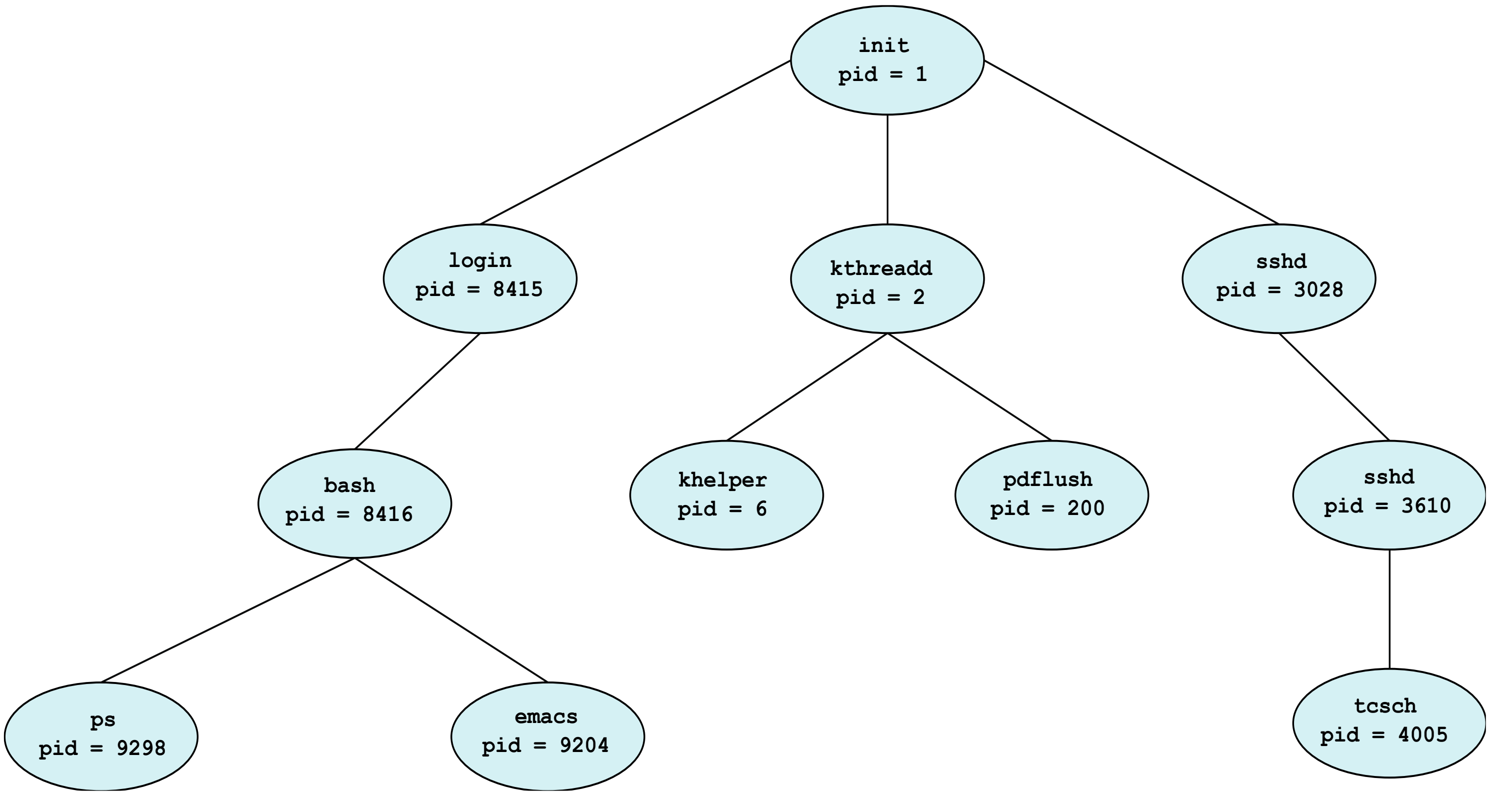
Operations on Processes

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

Process Creation

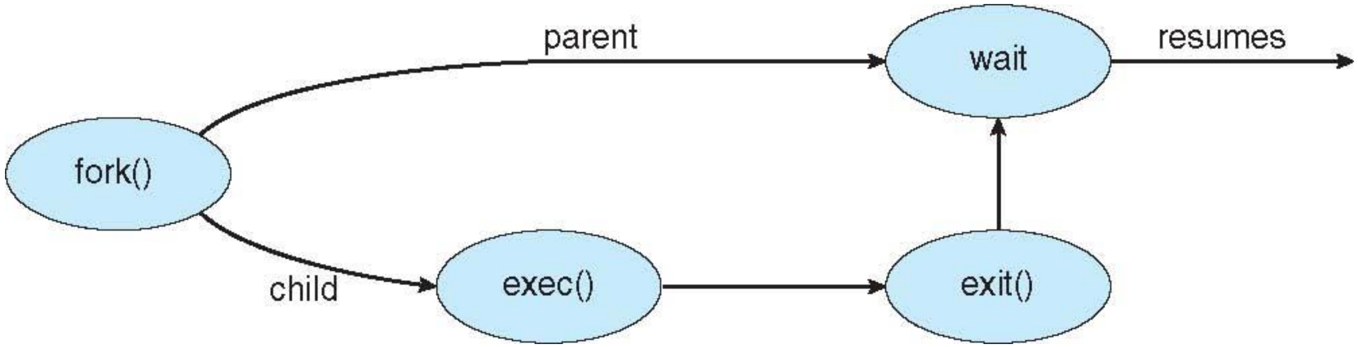
- A **(parent)** process can create several (**children)** processes
  - Children can, in turn, create other processes
  - Hence, a **tree** of processes forms
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options (of process creation)
  - Parent and children share all resources
  - Children share subset of parent's resources
    - One usage is to prevent system overload by too many child processes
  - 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 options
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
    - Child is a copy of parent's address space
      - except fork() returns 0 to child and nonzero to parent
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



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 OSes don't allow child to exists if its parent has terminated
  - **cascading termination -** if a process terminates, then all its children, grandchildren, etc must also be 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);
```

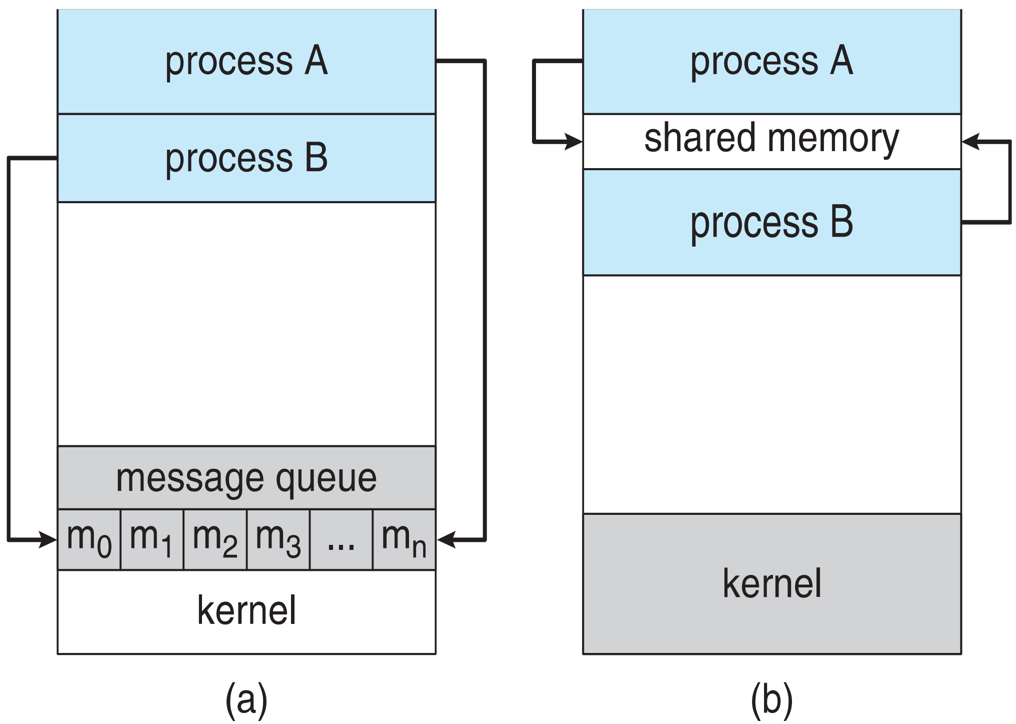
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
  - All its resources are deallocated, but exit status is kept
- If parent terminated without invoking **`wait`** , process is an **orphan**
  - UNIX: assigns **init** process as the parent
  - Init calls wait periodically

Cooperating Processes

- Processes within a system may be ***independent*** or ***cooperating***
  - When processes execute they produce some **computational results**
  - ***Independent*** process cannot affect (or be affected) by such results of another process
  - ***Cooperating*** process can affect (or be affected) by such results of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication

- For fast exchange of information, cooperating processes need some **interprocess communication** (**IPC**) mechanisms
- Two models of IPC
  - **Shared memory**
  - **Message passing**



Interprocess Communication – Shared Memory

- An area of memory is shared among the processes that wish to communicate
- The communication is under the control of the users processes, not the OS.
- Major issue 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.

Producer-Consumer Problem

- **Producer-consumer problem** – a common paradigm for cooperating processes
  - Used to exemplify one common generic way/scenario of cooperation among processes
  - We will use it to exemplify IPC
  - Very important!
- ***Producer*** process
  - produces some information
  - incrementally
- ***Consumer*** process
  - consumes this information
  - as it becomes available
- Challenge:
  - Producer and consumer should run concurrently and efficiently
  - Producer and consumer must be synchronized
    - Consumer cannot consume an item before it is produced

Bounded-Buffer – Shared-Memory Solution

- Shared-memory solution to producer-consumer
  - Uses a buffer in shared memory to exchange information
    - **unbounded-buffer:** assumes no practical limit on the buffer size
    - **bounded-buffer** assumes a fixed buffer size

- 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) {
        next_produced = ProduceItem();

        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing, no space in buffer */
                   //wait for consumer to get items and
                   //free up some space

       /* enough space in buffer */
        buffer[in] = next_produced; //put item into
buffer
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing, no new items produced
*/



                   //wait for items to be produced



        /* some new items are in the buffer */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        ConsumeItem(&next_consumed);
}
```

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 (buffer size) 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.)

- Logical implementation of communication link
  - 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 a direct 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 an indirect 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 issues
  - 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 continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking – called a **rendezvous**

Synchronization (Cont.)

- Producer-consumer is trivial via rendezvous

```
message next_produced;

while (true) {
          ProduceItem(&next_produced);


          send(next_produced);
}
```

```
message next_consumed;

while (true) {
   receive(next_consumed);

   ConsumeItem(&next_consumed);
}
```

Buffering in Message-Passing

- Queue of messages is 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 Chapter 3