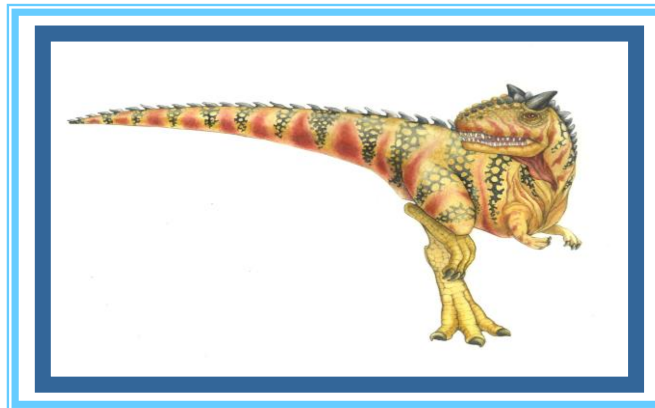
Chapter 3: Processes



Silberschatz, Galvin and Gagne ©2013

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Objectives

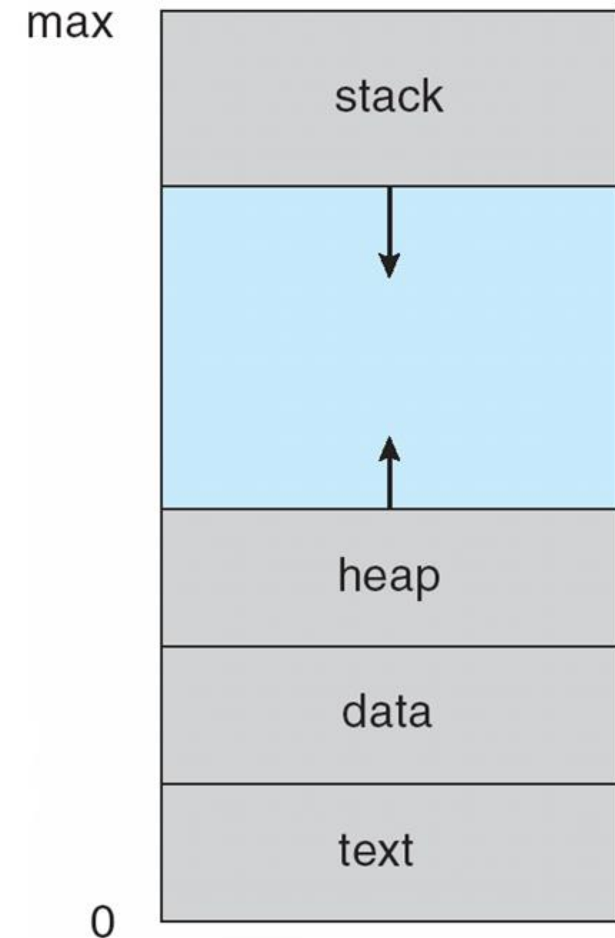
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication

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
- A process will need certain resources — such as CPU time, memory, files, and I/O devices —to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.

Process in Memory

- The structure of a process in memory is represented in multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time



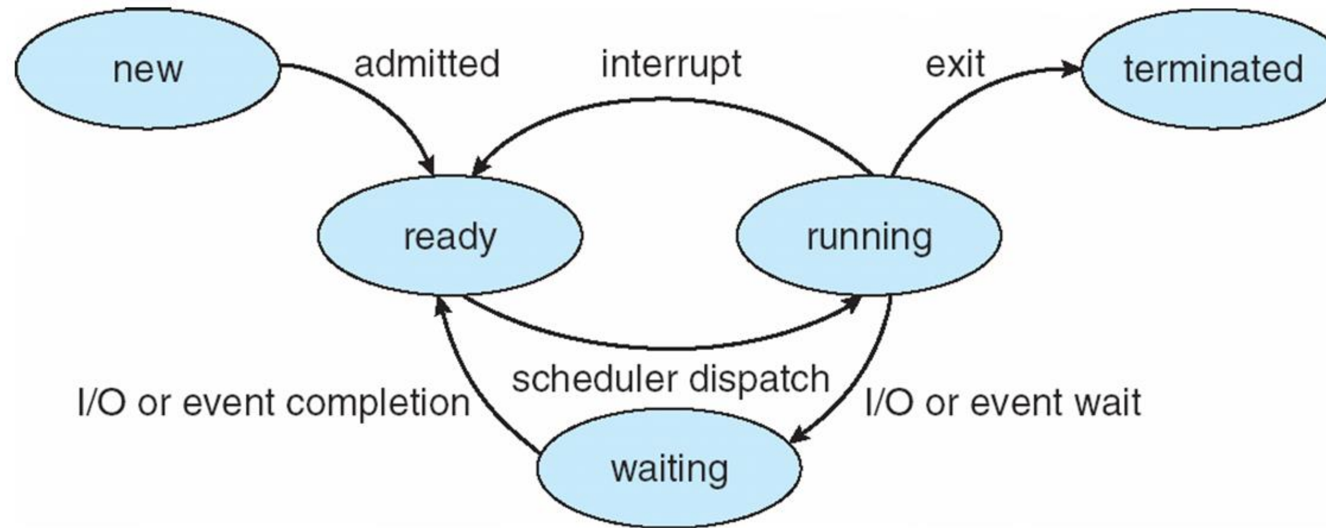
Process Concept (Cont.)

- A program by itself **is not a process**!
  - A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (**executable file**),
  - In contrast, a process is an ***active*** entity.
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, or command line entry of its name.
- One program can be several processes; for example:
  - Consider multiple users executing the same program.
  - The same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

Process State

- As a process executes, it changes **state**
- The state of a process is defined by the current activity of that process.
- A process may be in one of the following five states:
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

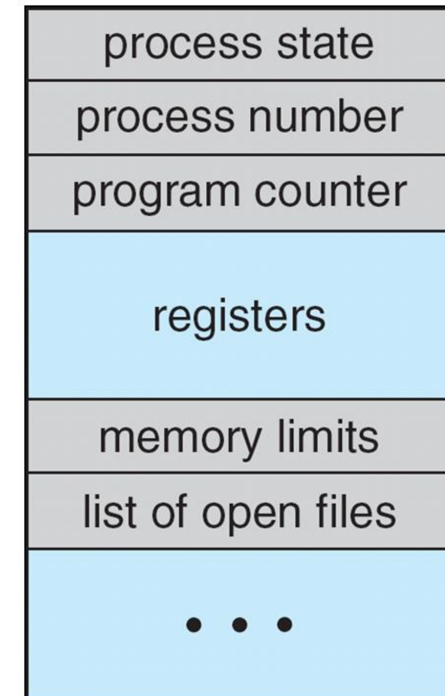
Diagram of Process State



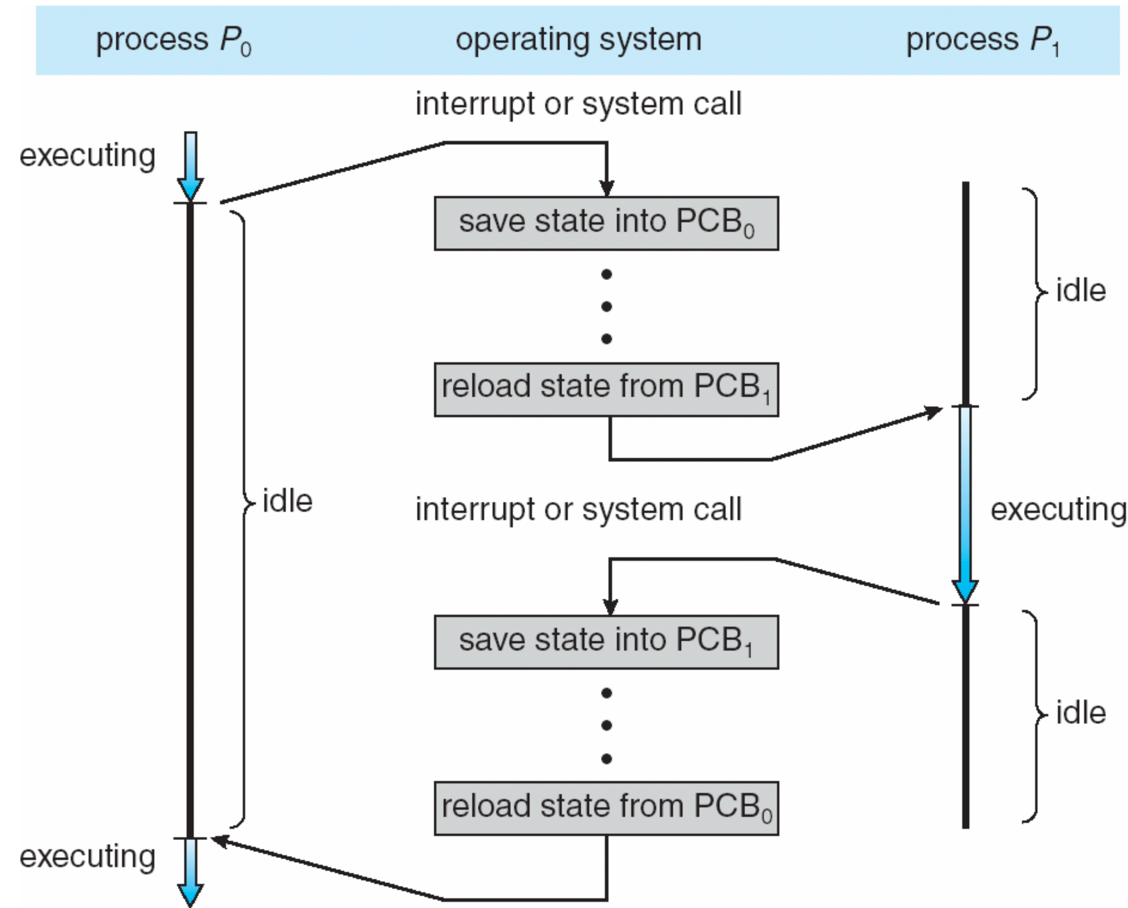
State	Description
New	The process is being created
Running	Instructions are being executed
Waiting	The process is waiting for some event to occur (such as an I/O completion or reception of a signal)
Ready	The process is waiting to be assigned to a processor
Terminated	The process has finished execution

Process Control Block (PCB)

- Each process is represented in the operating system by a **process control block (PCB)** (also called **task control block**)
  - Process state – running, waiting, etc
  - Program counter – location of instruction to next execute
  - 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 registers – contents of all process-centric registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.



CPU Switch From Process to Process



Threads

- So far, a process has **a single thread of execution**.
  - For example, when a process is running a word-processor program, a single thread of instructions is being executed.
  - This single thread of control allows the process to perform only one task at a time.
  - The user cannot simultaneously type in characters and run the spell checker within the same process!

- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**

    → and thus, perform more than one task at a time.

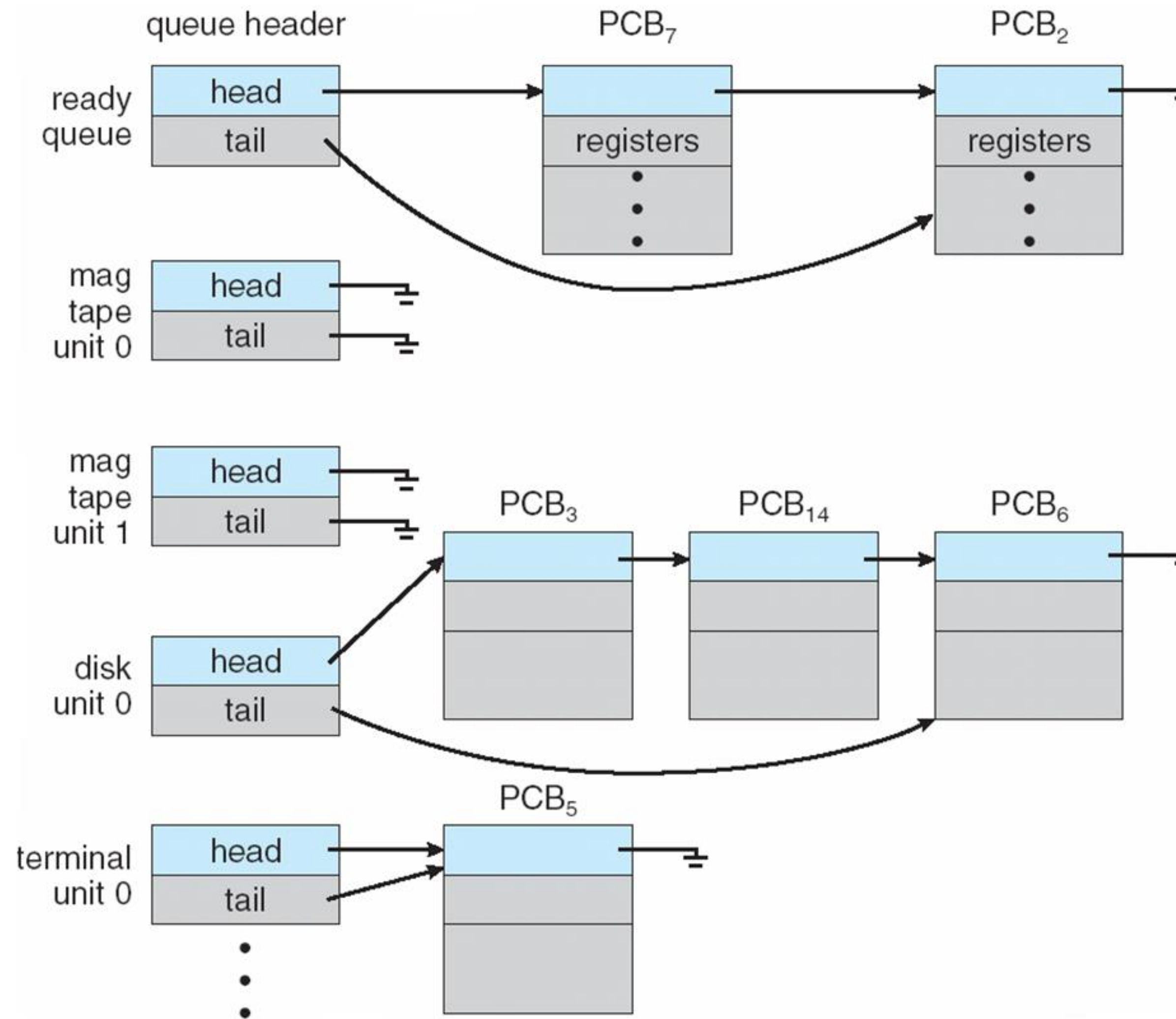
- Must have storage for <u>thread details</u>, multiple program counters in PCB

Process Scheduling

- The objective of ***multiprogramming*** is to have some process always running, to maximize CPU utilization.
- The objective of ***time-sharing*** is to switch the CPU among processes so frequently that users can interact with each program

- **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
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then moved to I/O queue.
  - The process could create a new child process and wait for the child's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

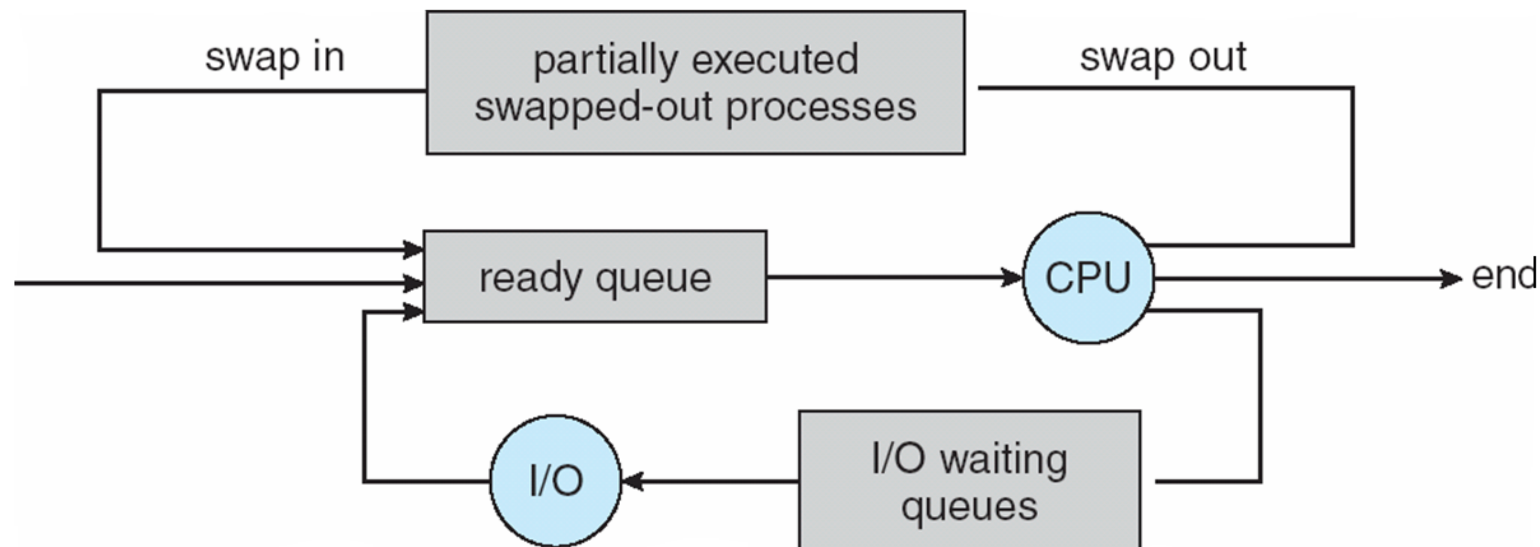
ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

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) $\Rightarrow$ (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) $\Rightarrow$ (may be slow)
  - The long-term scheduler 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
- 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**



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Apple probably limits multitasking due to battery life and memory use concerns.
- 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

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** **(i.e.,** This task is known as 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

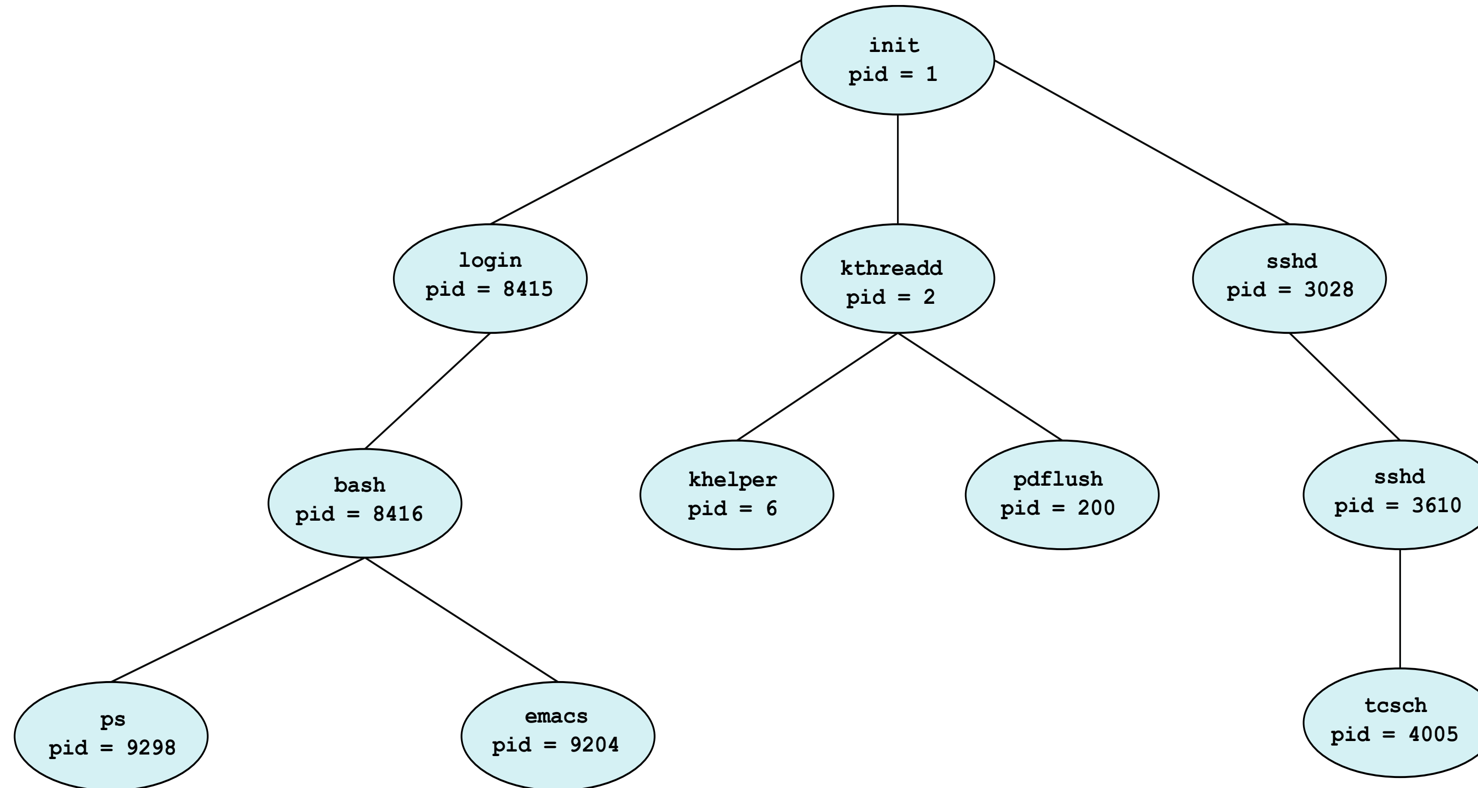
Operations on Processes

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

Process Creation

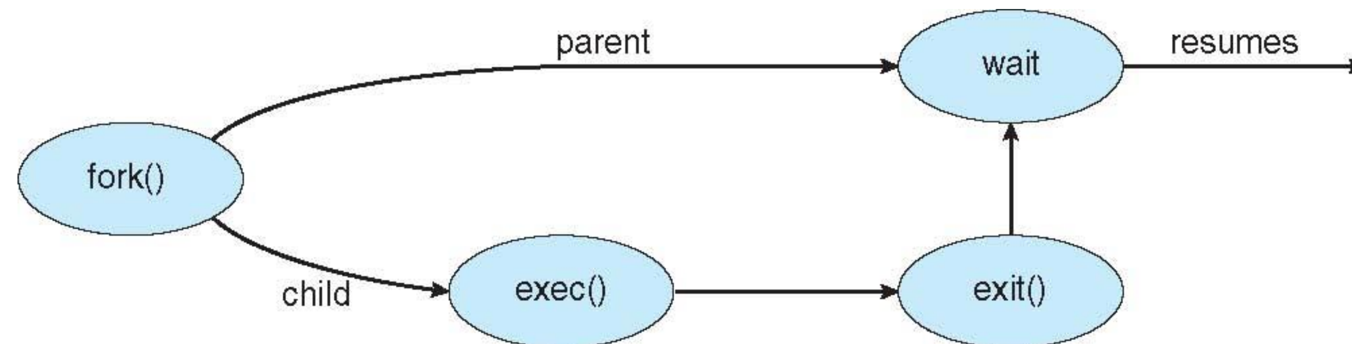
- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset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

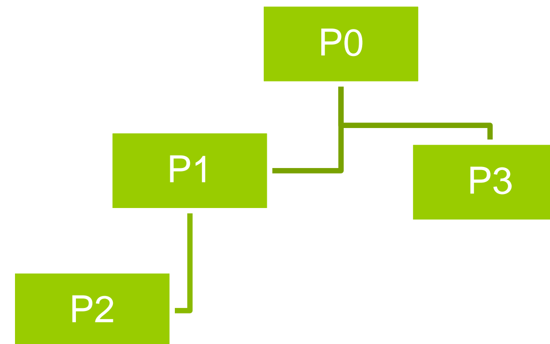
- **fork** system call creates new process
- **exec** system call is used after a **fork** system call by one of the two processes to replace the process's memory space with a new program
- The parent waits for the child process to complete
- When the child process completes the parent process resumes
- This is illustrated in the figure



Process Creation (Cont.)

- **Example1.**
  - **How many processes are created by the following fork calls?**
  - **What is the output- assuming parent processes wait until children terminate?**

```
fork();
cout<<"Hi"<<endl;
fork();
cout<<"OK"<<endl;
```

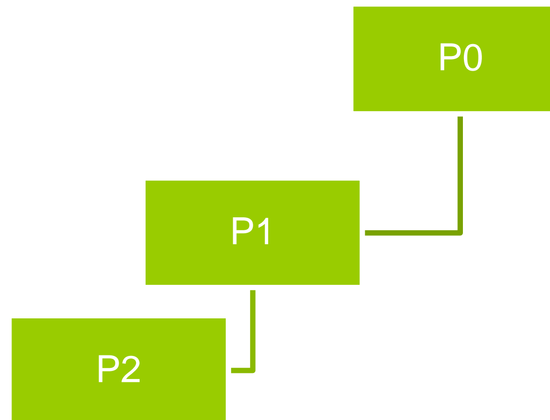



```
Hi
OK
OK
Hi
OK
OK
```

Process Creation (Cont.)

- **Example2.**
  - **How many processes are created by the following fork calls?**
  - **What is the output- assuming parent processes wait until children terminate?**

```
id = fork();
if(id>0)
  cout<<“Hi”<<endl;
else{
  fork();
  cout<<“OK”<<endl;
}
```




```
OK
OK
Hi
```

Process Creation (Cont.)

- **Exercise.**
  - **How many processes are created by the following fork calls?**
  - **What is the output- assuming parent processes wait until children terminate?**

```
id = fork();
if(id>0)
  cout<<“Hi”<<endl;
else{
  fork();
  cout<<“OK”<<endl;
}
fork();
cout<<“Done”<<endl;
```

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.

Interprocess Communication

- Processes within a system may be ***independent processes*** or ***cooperating processes.***

- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
- ***Cooperating process*** can affect or be affected by other processes, including sharing data.

 Silberschatz, Galvin and Gagne ©2013

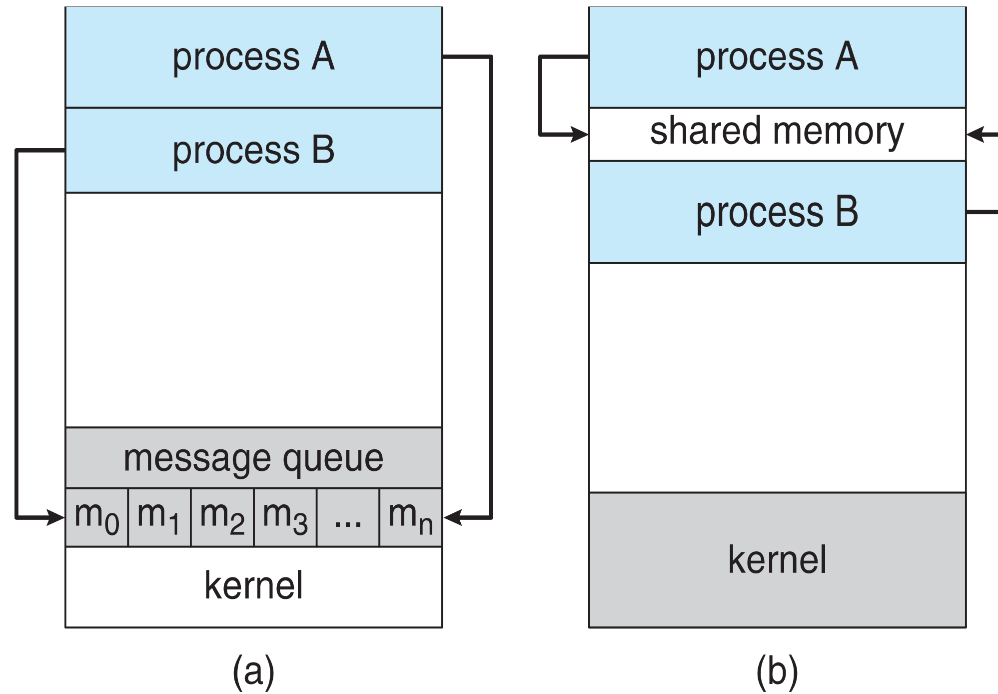
Cooperating Processes advatages

- There are several reasons for providing an environment that allows process cooperation:

Information sharing	• Several users may be interested in the same piece of information (shared files)
Computation speedup	• Tasks to be run faster, it must be broken into subtasks, each of which will be executing in parallel with the others
Modularity	• The system may be constructed in modular fashion, in which its functions are divided into separate processes
Convenience	• Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel

Communications Models

- Cooperating processes need **interprocess communication** (**IPC**) mechanism that will allow them to exchange data and information.
- Two models of IPC
  - **Shared memory** **and** **Message passing**



(a) Message passing. (b) shared memory.

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the communicating processes not the operating system.
- Major issues is to provide mechanism that will allow the cooperating processes to synchronize their actions when they access shared memory (i.e., ensuring that they are not writing to the same location simultaneously)
- **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 the need have shared variables
- IPC facility provides two operations:
  - `send`(*message*)
  - `receive`(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

Producer-Consumer Paradigm

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- Information, messages, are placed in a buffer
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- Roughly speaking, the interaction follows the pattern below.

```
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 */
}
```

- More in chapter 5.

End of Chapter 3