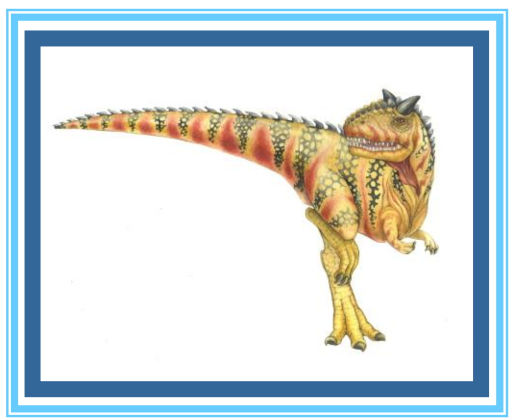
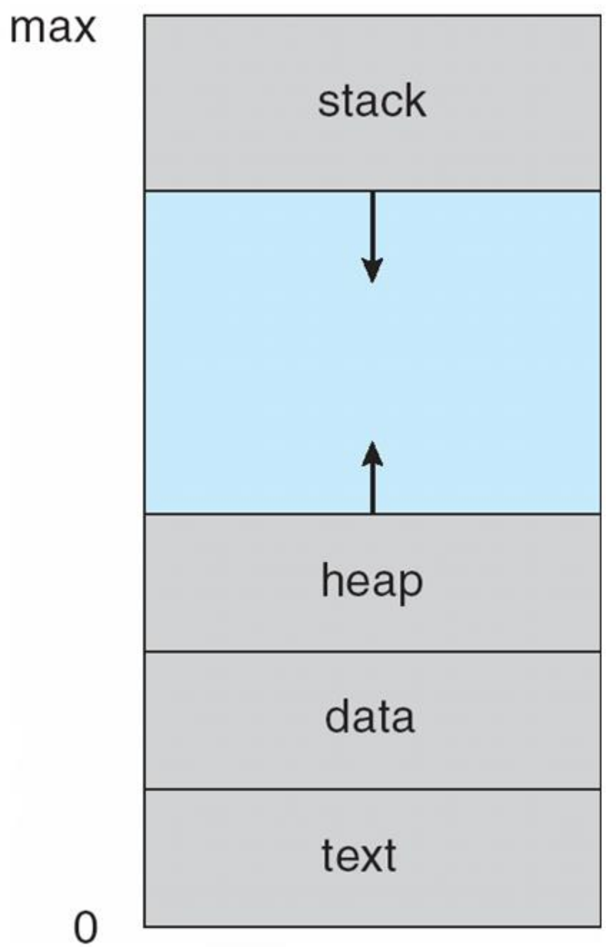
Process Concepts



Process Concept

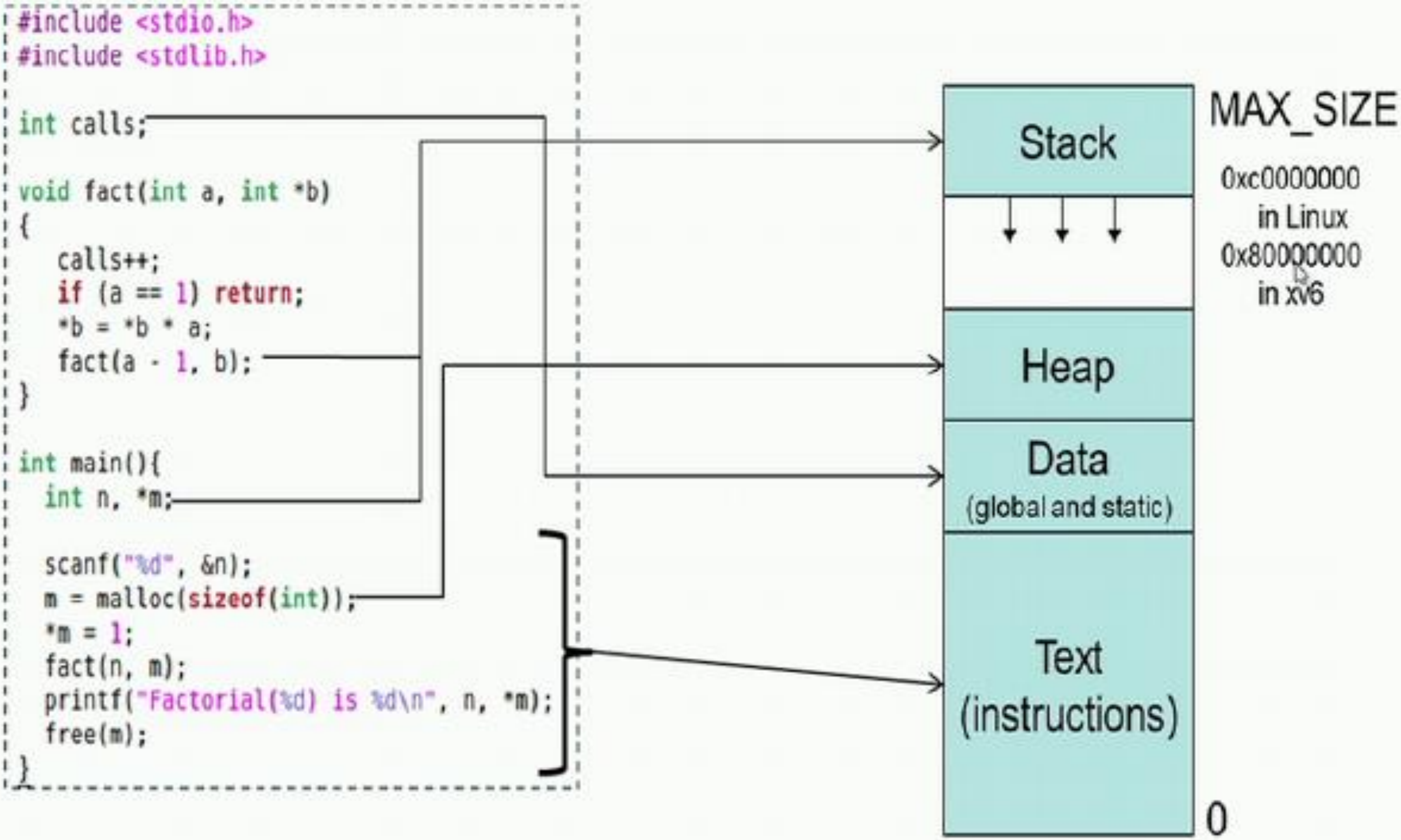
- **Process** – a program in execution;
  - A process will need certain resources—such as CPU time, memory, files, and I/O devices to accomplish its task.
  - These resources are allocated to the process either when it is created or while it is executing
- Process execution must progress in sequential 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



A process is the unit of work in a modern time-sharing system

Process Memory Map



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
- Operating system processes executing system code and user processes executing user code.
- Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them

Process State

- As a process executes, it changes **state**
- The state of a process is defined in part by the current activity of that process
- A process may be in one of the following states:
  - **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
- Only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*

Diagram of Process State

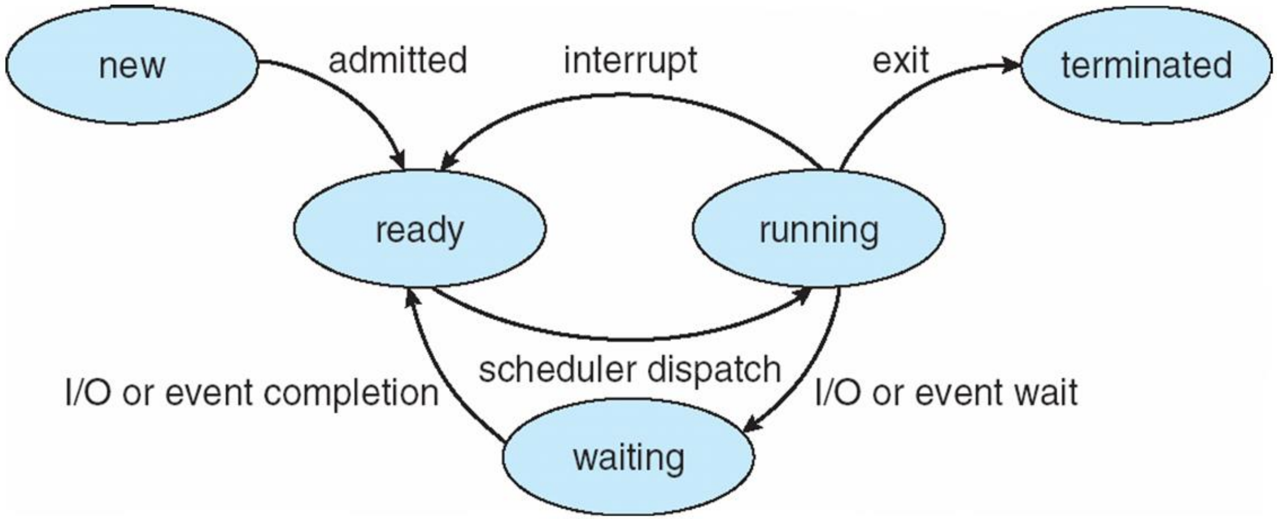
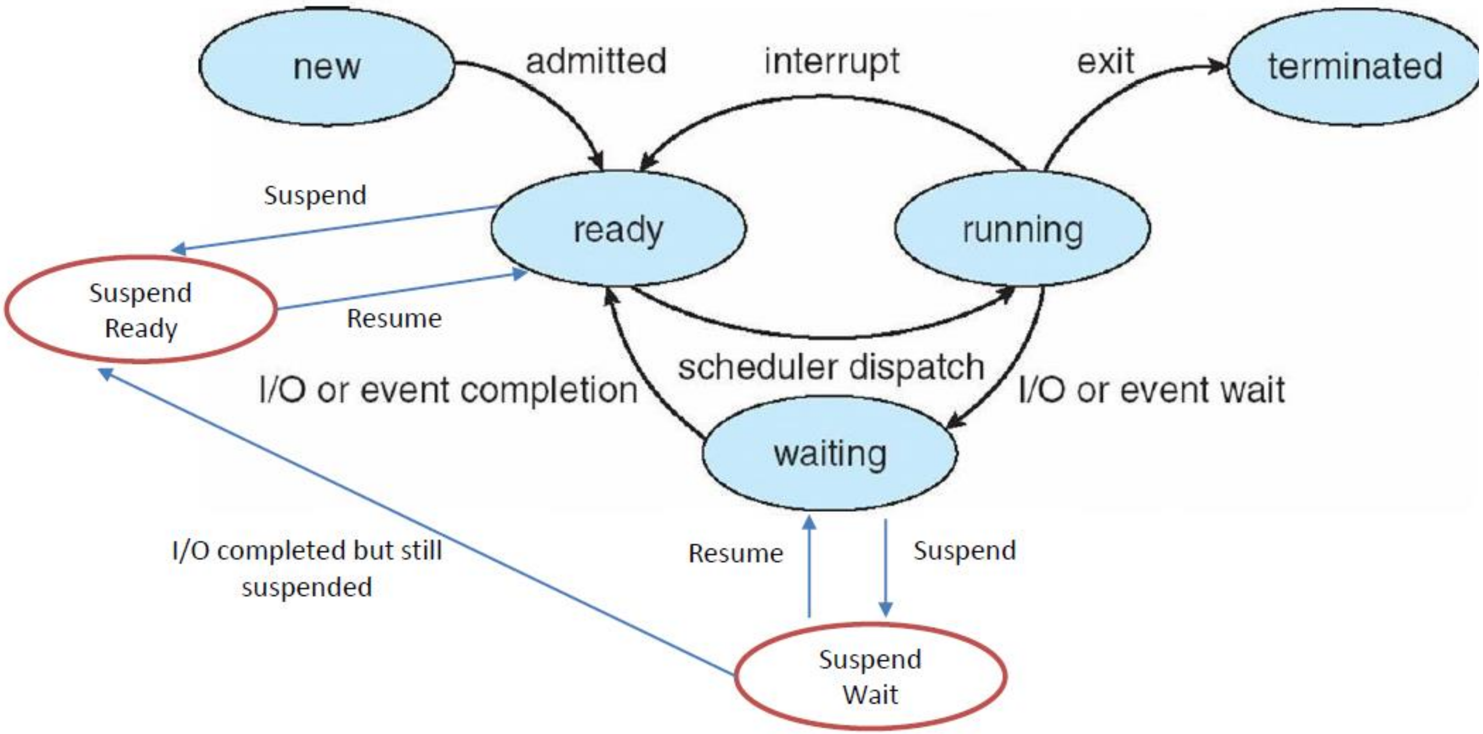


Diagram of Process State



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

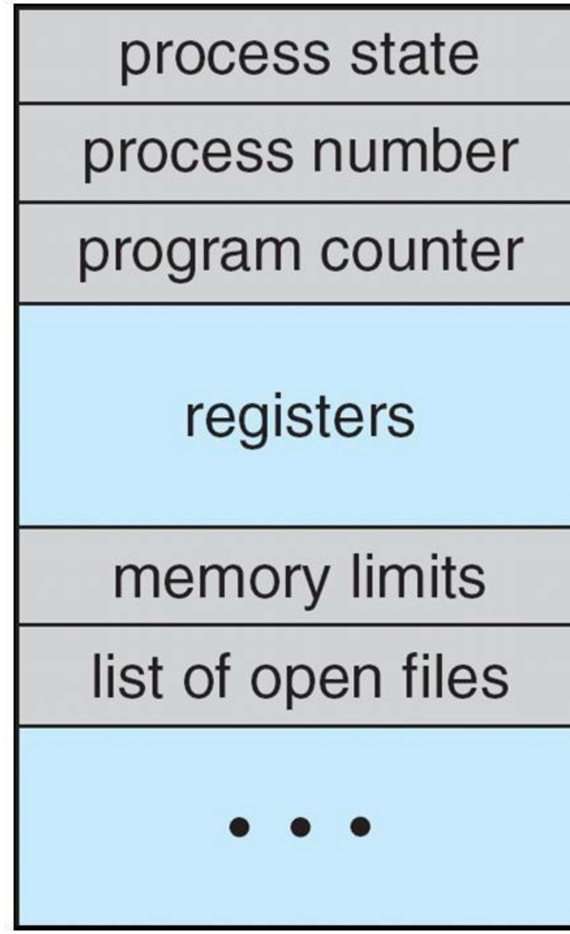
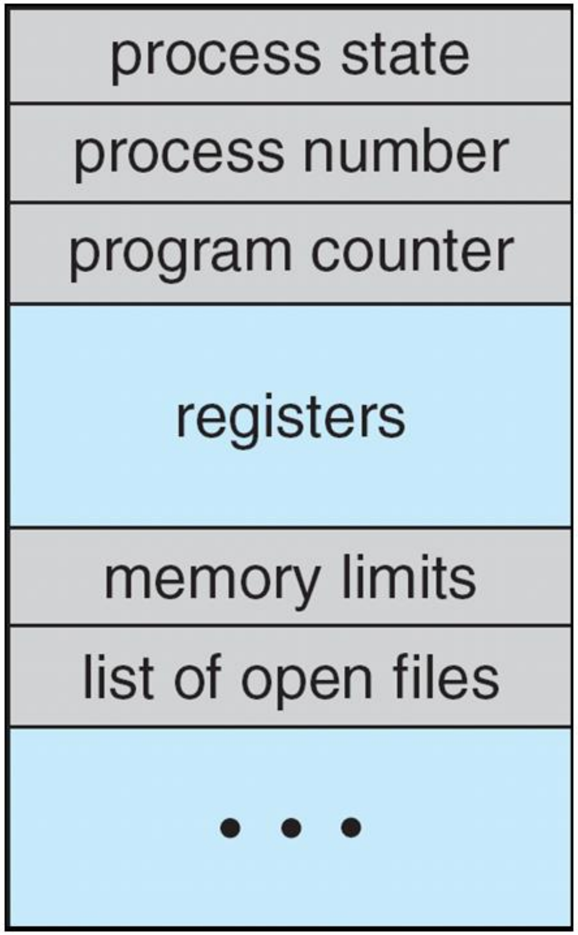


Fig:- Process Control Block (PCB)

Process Control Block (PCB)

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**
- It contains many pieces of information associated with a specific process, including these:



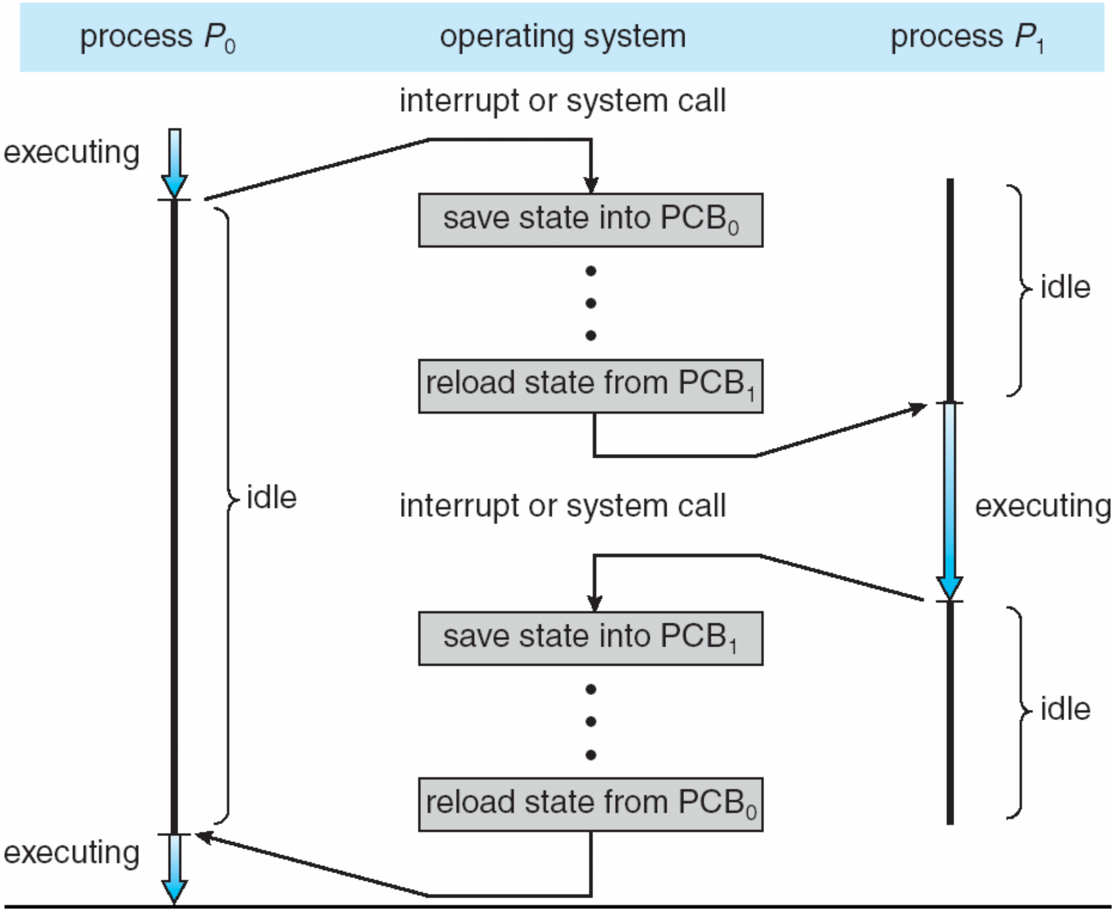
Process Control Block (PCB)

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. **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-scheduling information**. This information includes a **process priority**, **pointers to scheduling queues**, and any **other scheduling parameters**.

Process Control Block (PCB)

- **Memory-management information**. This information may include such items as the **value of the base and limit registers** and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

CPU Switch From Process to Process



Process Scheduling

- **WHY:**
  - Several Processes competing at a time to get the CPU for their execution
- **Scheduling**
  - strategy and methods used by OS to decide which process is going to be allocated to CPU next among the several process in the queue for CPU time
- The objective **of multiprogramming** is
  - to have some process running at all times, to maximize CPU utilization
- Time sharing multiprogramming system

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

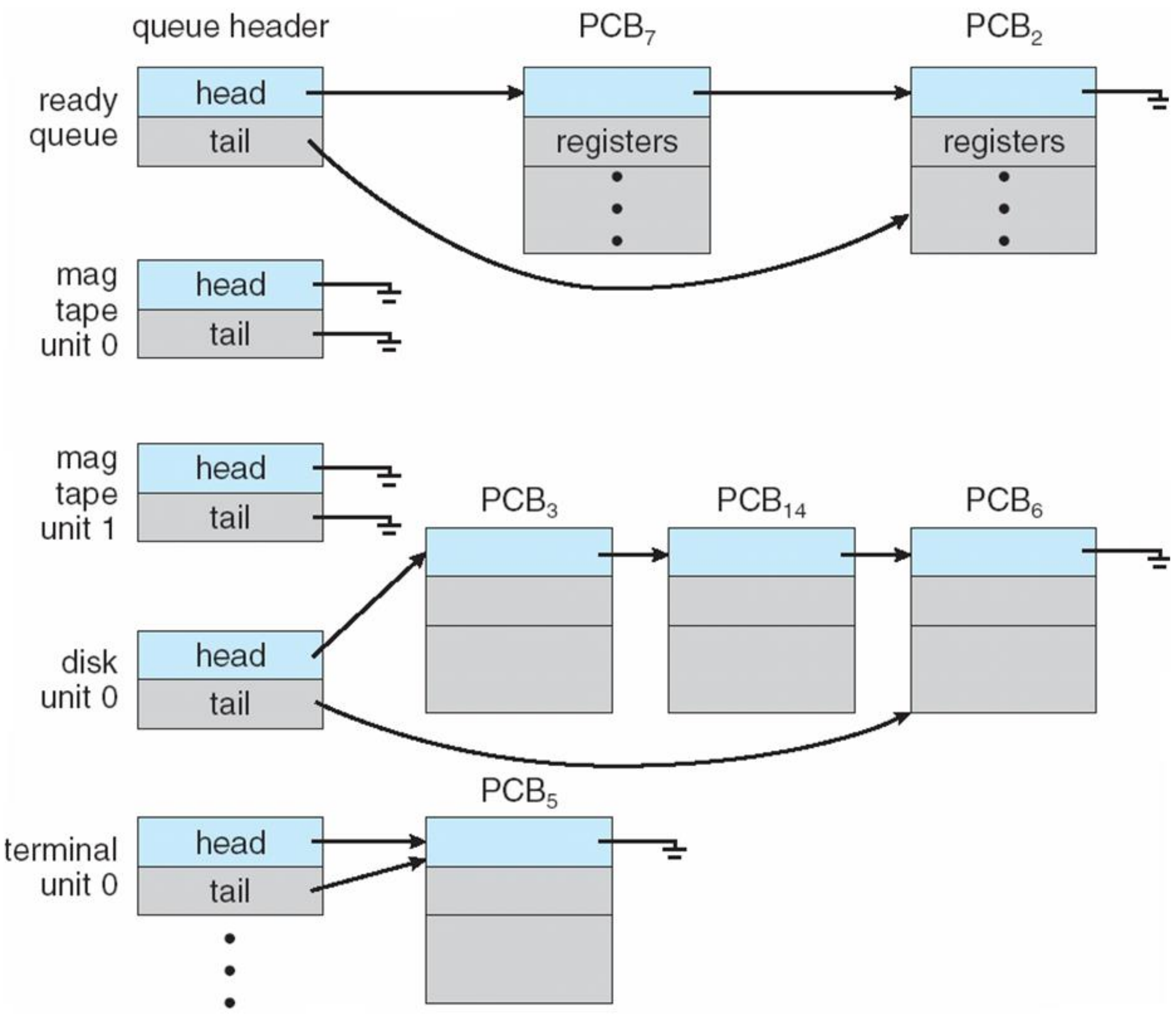
- Once a process gives up the CPU it is added to the ready queue

- **Process scheduler** selects among available processes for next execution on CPU

Process Scheduling

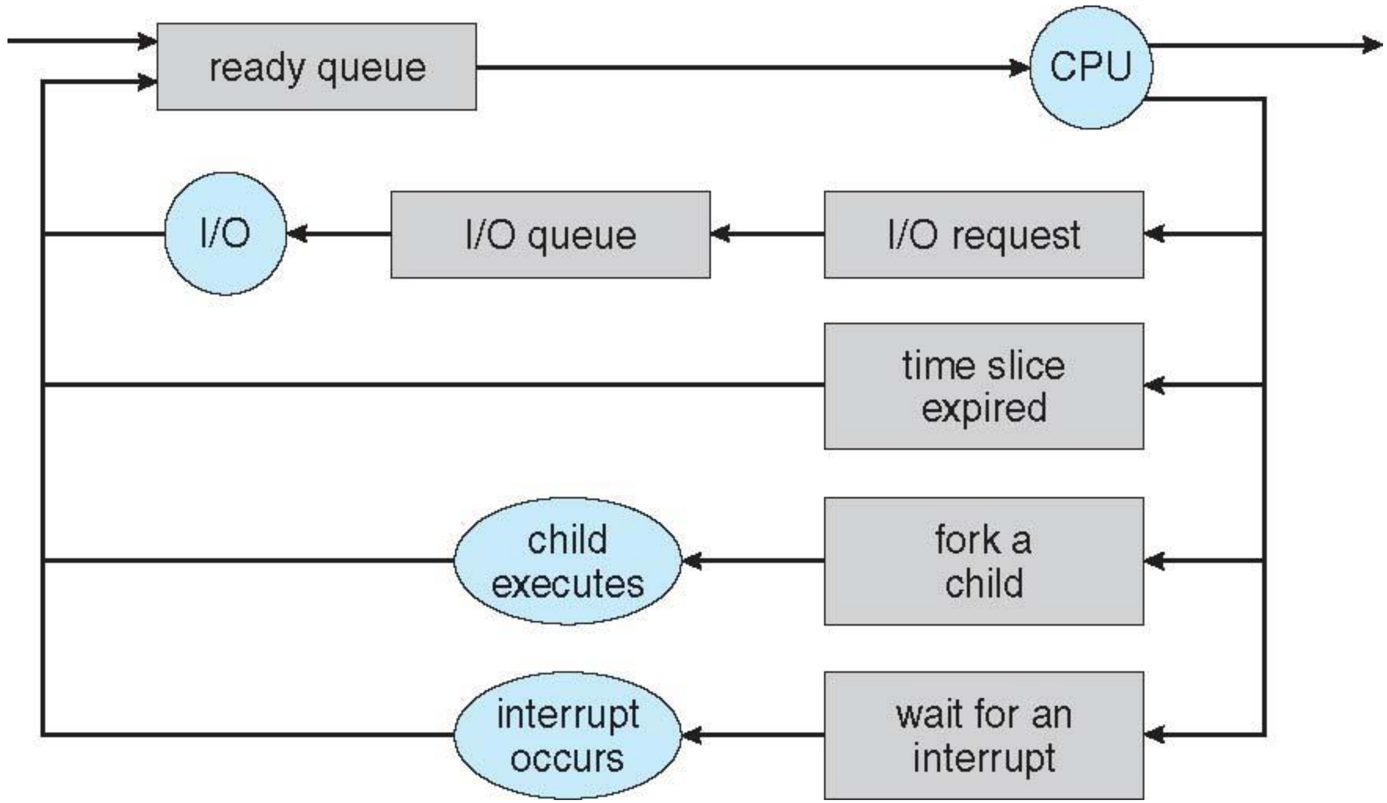
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing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

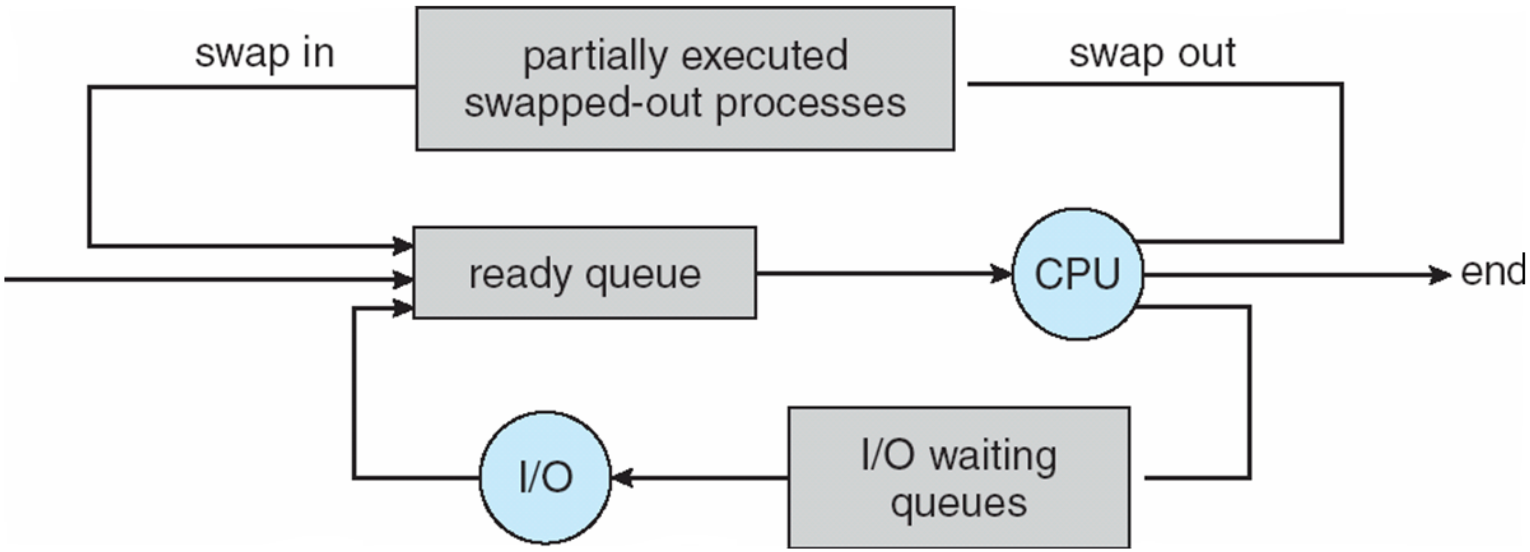


Schedulers

- **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) $\Rightarrow$ (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**
- 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

Operations on Processes

- During the course of execution, a process may create several new processes
  - The creating process is called a parent process, and
  - the new processes are called the children of that process
- System must provide mechanisms for:
  - process creation,
  - process termination,

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Process identified and managed via a **process identifier** (**pid**)
  - PID is an integer number
  - The PID provides a unique value for each process in the system, and
  - it can be used as an index to access various attributes of a process within the kernel.

- ## getpid ()

This function returns the process identifiers of the calling process.

```
#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void);   // this function returns the process identifier (PID)
pid_t getppid(void);  // this function returns the parent process identifier (PPID)
```

Process Creation

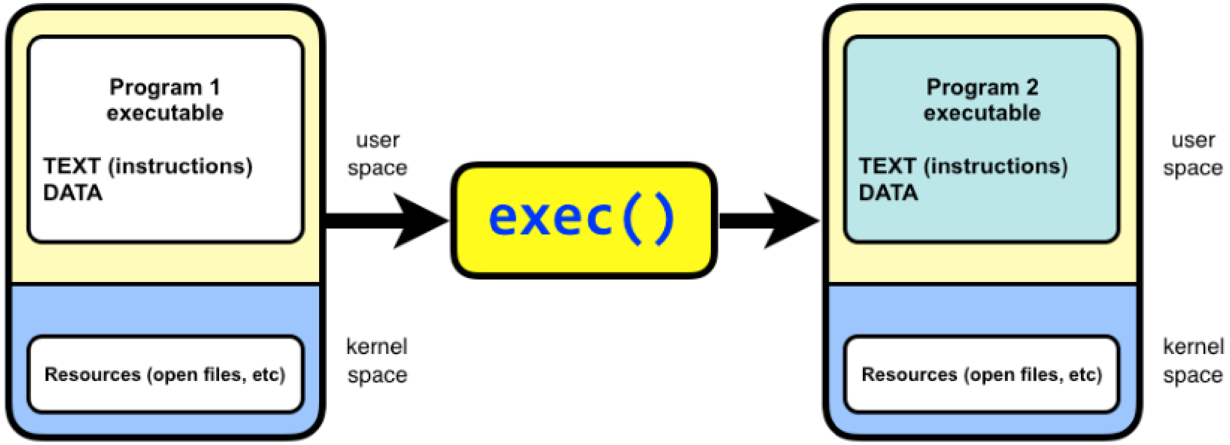
```
sandhya@telnet:~/DSEOS2022/process$ cat pid.c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
        pid_t getpid(void); //return PID
        pid_t getppid(void); //return PPID
        printf("PID is %d",getpid());
        printf("\n PPID is %d",getppid());
        return 0;
}
sandhya@telnet:~/DSEOS2022/process$ ./a.out
PID is 2682
 PPID is 2554sandhya@telnet:~/DSEOS2022/process$ ps
  PID TTY          TIME CMD
 2554 pts/0    00:00:00 bash
 2683 pts/0    00:00:00 ps
sandhya@telnet:~/DSEOS2022/process$
```

Process Creation

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
    - ***Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes***

Process Creation

- When a process creates a new process, two possibilities for execution exist:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- There are also two address-space possibilities for the new process:
  - **1.** The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - **2.** The child process has a new program loaded into it.

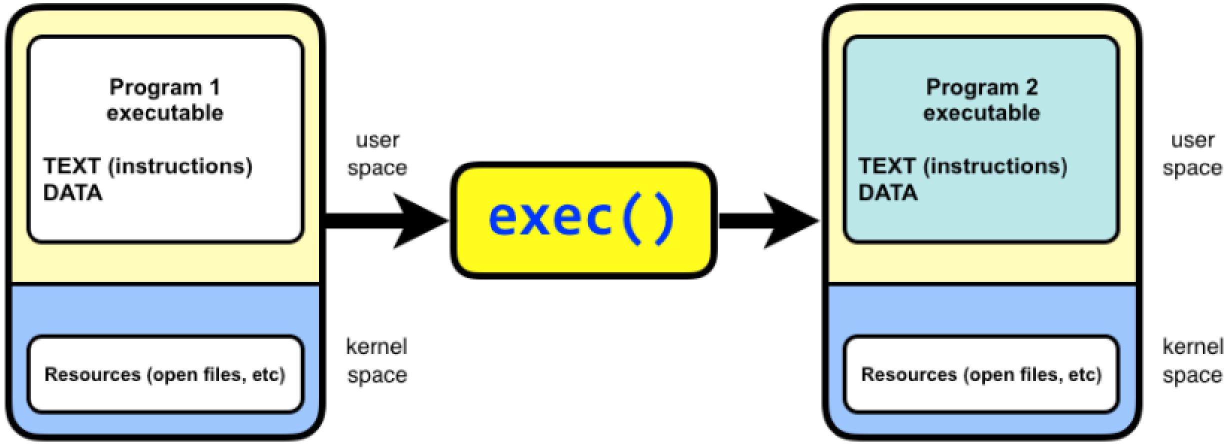


Process Creation: fork()

- A new process is created by the **fork()** system call
  - A new process is created by calling fork.
  - This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process.
  - The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors.
  - After a new child process is created, both processes will execute the **next instruction following the fork() system call.**

Process Creation: fork()

- 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.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program
- The exec() system call loads a binary file into memory and starts its execution
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child



Example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    int sharedVar = 10;
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == -1) {
        perror("fork");
        return 1;
    }

    if (child_pid == 0) {
        // This code will be executed by the child process
        sharedVar = 20;
        printf("Child - sharedVar: %d\n", sharedVar);
    }
    else {
        // This code will be executed by the parent process
        printf("Parent - sharedVar: %d\n", sharedVar);
    }

    return 0;
}
```

Output

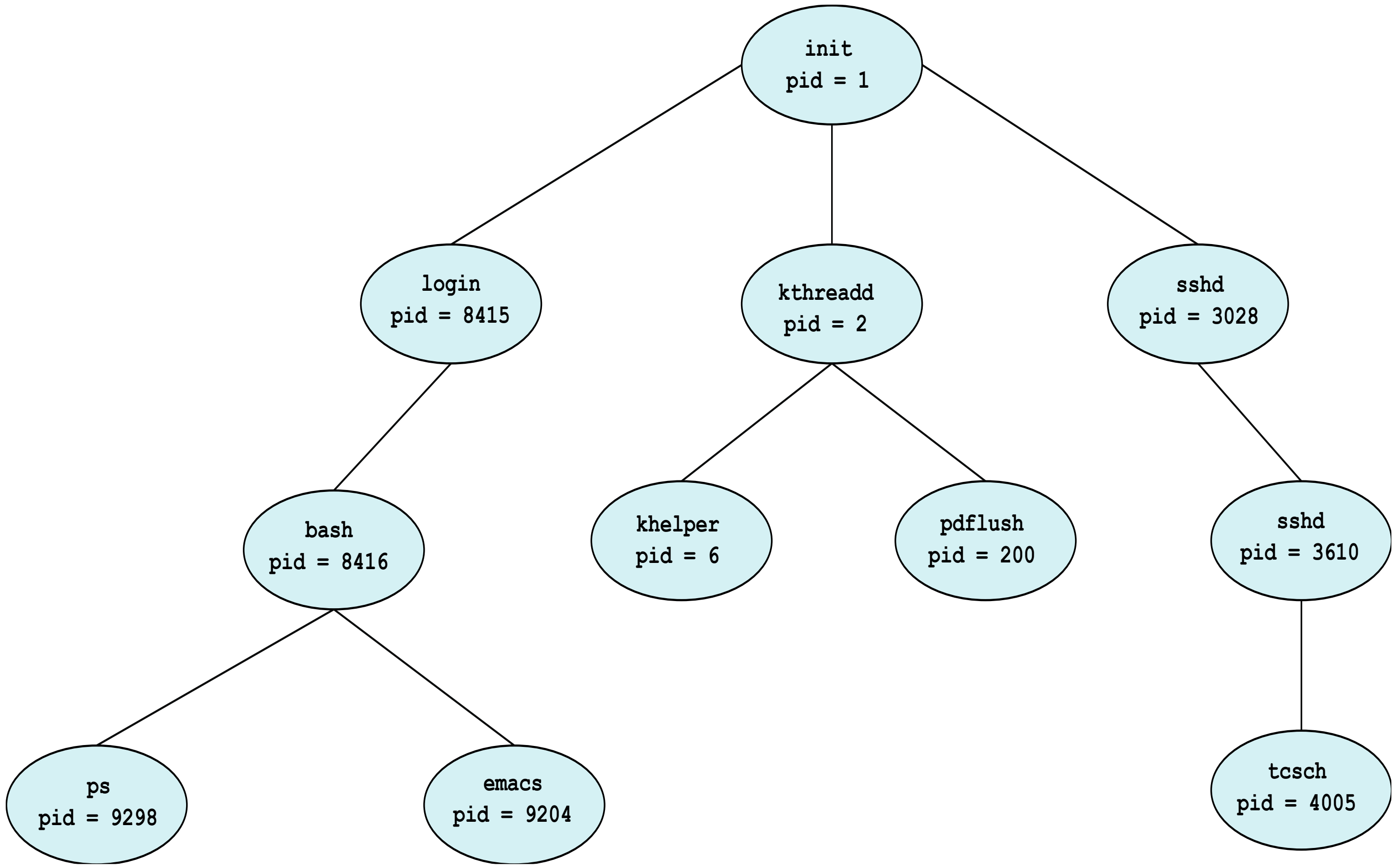
- If Parent process executes first:

```
Parent - sharedVar: 10
Child - sharedVar: 20
```

- If Child process executes first:

```
Child - sharedVar: 20
Parent - sharedVar: 10
```

A Tree of Processes in Linux



A Tree of Processes in Linux

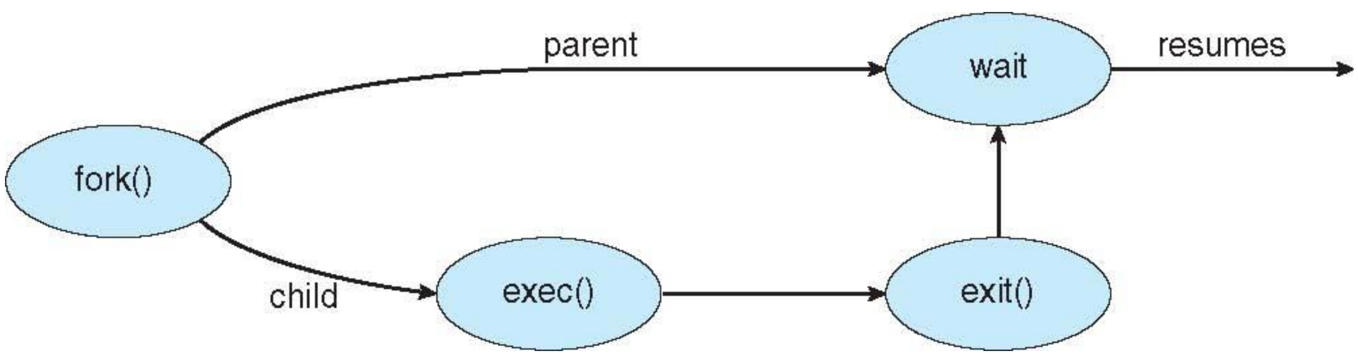
- The **init process** (which always has a pid of 1) serves as the root parent process for all user processes.
- The **kthreadd process** is responsible for creating additional processes that perform tasks on behalf of the kernel
- The **sshd process** is responsible for managing clients that connect to the system by using ssh (Secure Shell)
- The **login process** is responsible for managing clients that directly log onto the system.
- In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
- Using the bash command-line interface, this user has created the process ps as well as the emacs editor.
- Parent waits until children terminate
- **Pdflush: a set of kernel threads which are responsible for writing the dirty pages to disk**

A Tree of Processes in Linux

- On UNIX and Linux systems, the `ps` command is used to list all the user process
  - For example, `ps -el`
    - will list complete information for all processes currently active in the system.

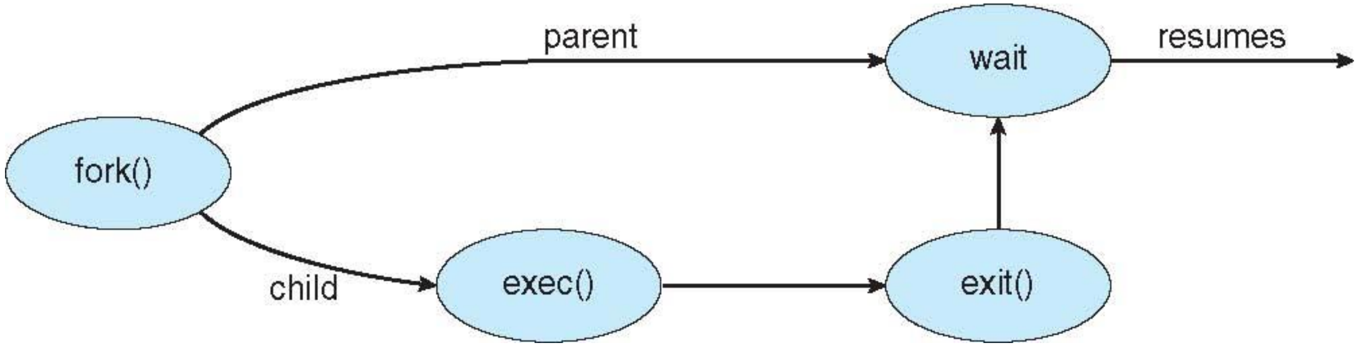
Process Creation (Cont.)

- 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 (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 Termination

1. 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
2. A process can cause the termination of another process via an appropriate system call
   - Usually, such a system call can be invoked only by the parent of the process.
   - Otherwise, users could arbitrarily kill each other's jobs.
   - Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

Process Termination

- 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

1. 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.

Process Termination

- 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**
- Linux and UNIX address this scenario by assigning the **init process** as the new parent to orphan processes

Silberschatz, Galvin and Gagne ©2013

Example

```c
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
          if (fork()== 0)
                    printf("HC: hello from child\n");
          else
          {
                    printf("HP: hello from parent\n");

                    wait(NULL);

                    printf("CT: child has terminated\n");
          }


          printf("Bye\n");

          return 0;
}
```

One Possibility of Output

```
HC: hello from child
Bye
HP: hello from parent
CT: child has terminated
Bye
```