Process Management: Processes

Lecturer: Kor Sokchea

Royal University of Phnom Penh

Faculty of Engineering

kor.sokchea@gmail.com

Administrivia

- **Textbook:** *Operating System Concepts*, 9th Edition, by Silberschatz, Galvin, and Gagne.
- Class
  - IT Engineering: Room **T002**
    - Monday Session: 13:00 - 14:30
    - Tuesday Session: 16:00 - 17:30
  - TEED: Room **T104**
    - Monday Session: 16:00 - 17:30
    - Tuesday Session: 13:00 - 14:30
- Exams
  - Final Exam: 60%
  - Assignment: 20%
  - Homework: 10%
  - Attendance: 10%

Course Topics

- Threads & Processes
- Concurrency & Synchronization
- Scheduling
- Deadlocks
- Main Memory
- Virtual Memory
- I/O systems
- Disk
- File System
- Protection & Security

Process Management

- Process
- Threads
- Process Synchronization
- CPU Scheduling

What is a Process?

- A process is an instance of a program running
  - Program = static file (image)
  - Process = executing program + execution state
- A Process is the basic unit of execution in an operating system
  - Each process has a number, its process identifier (**pid**)
- Different processes may run different instances of the same program
  - Ex: my gcc and your gcc process both run c compiler
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

What is a Process?

- Modern OS run multiple processes concurrently
- Examples: (can all run simultaneously):
  - gcc hello_world.c – compiler running on **hello_world** file
  - gcc read_A.c – compiler running on **read_A** file
  - emacs – text editor
  - firefox – web browser

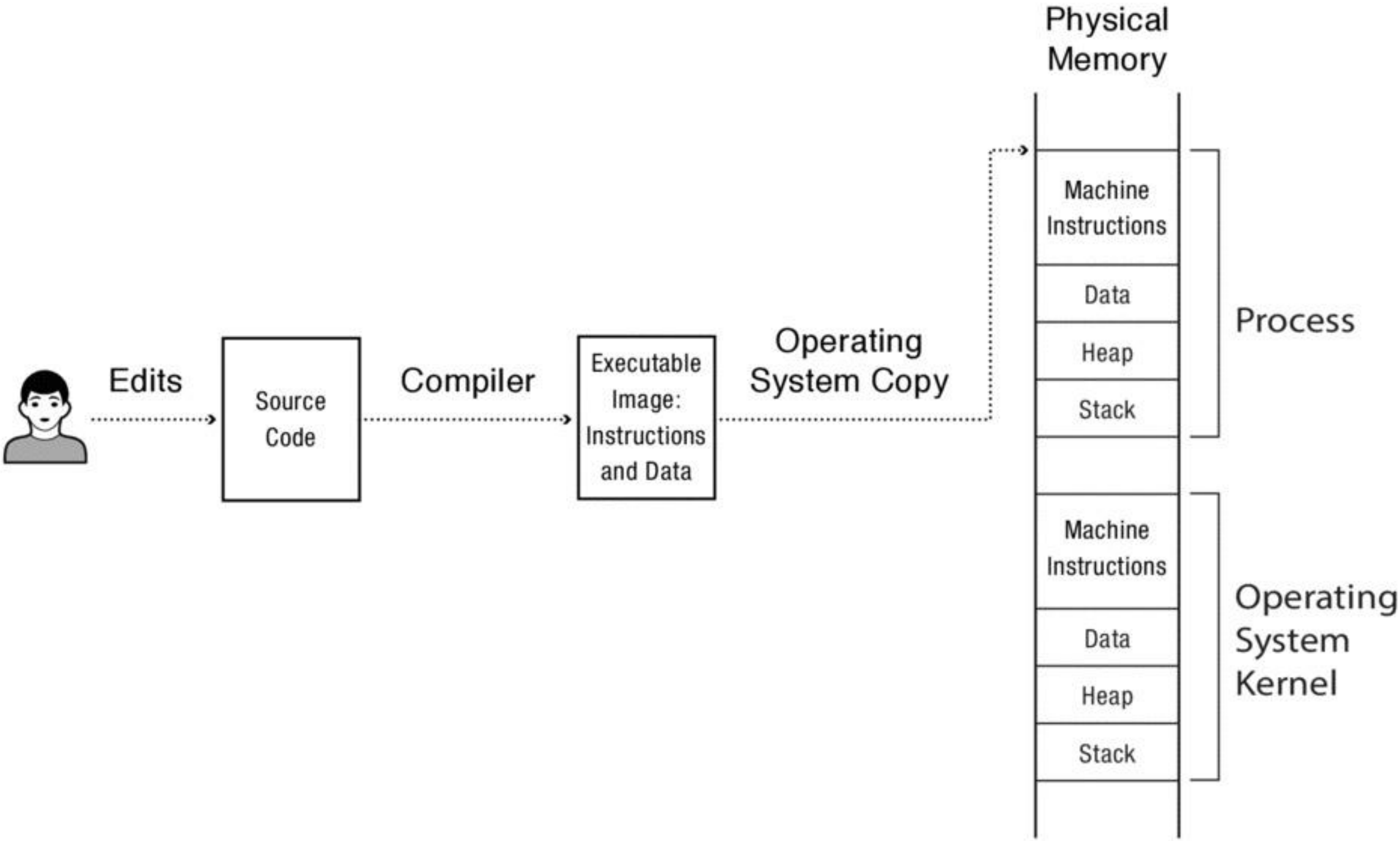
Program to Process

- We write a program in e.g. C
- A compiler turns that program into an instruction list
- The CPU interprets the instruction list (which is more a graph of basic blocks)

```
void add(int c) {
    if(c == 1){
...
int main(){
    int a = 2;
    add(a);
}
```

- Program becomes process when an executable file is loaded into memory
- Loading executable files via GUI mouse clicks, command line entry of its name

Program to Process

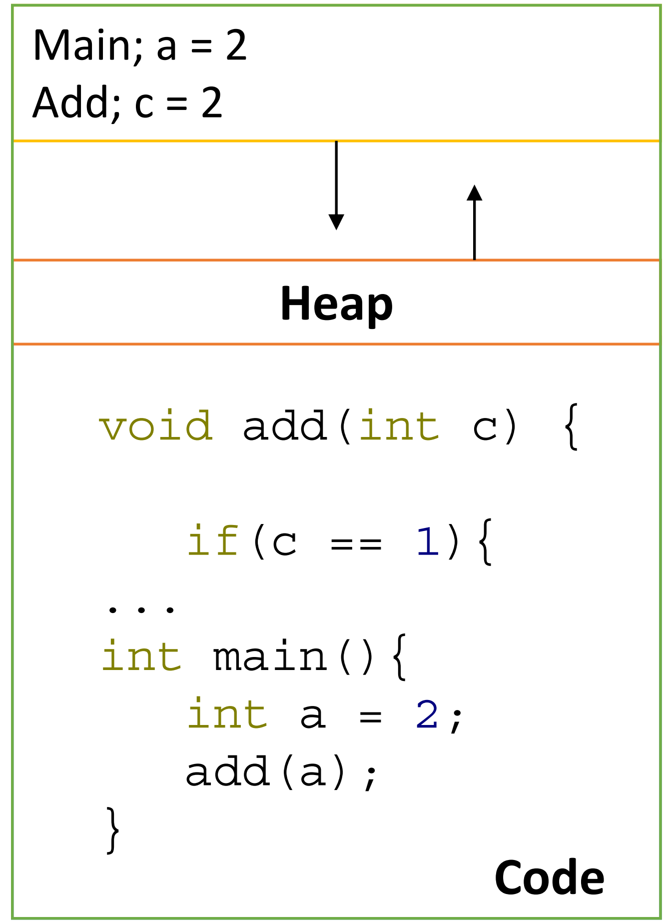


Process in Memory

- Program to process
  - What you wrote

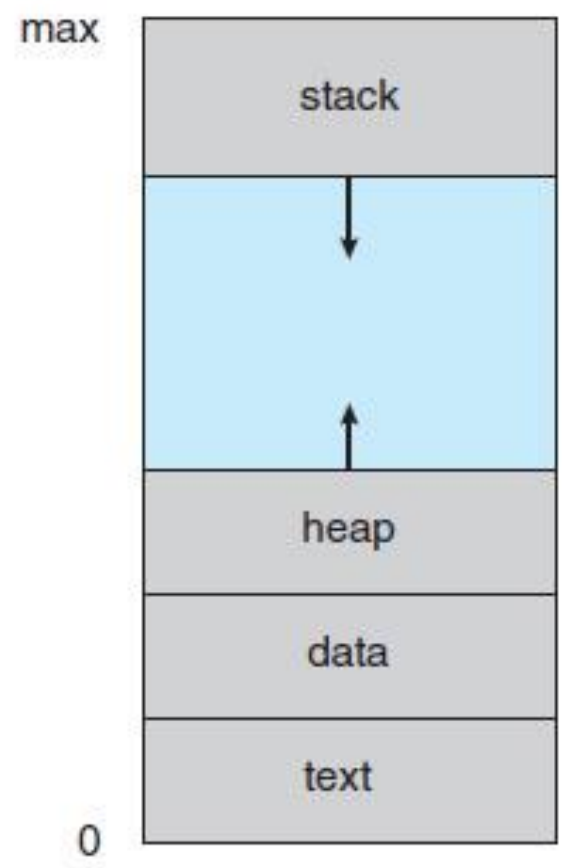
```
void add(int c) {

    if(c == 1){
...
int main(){
    int a = 2;
    add(a);
}
```

- What is in memory



Processes

- Process has its own address space:
  - 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

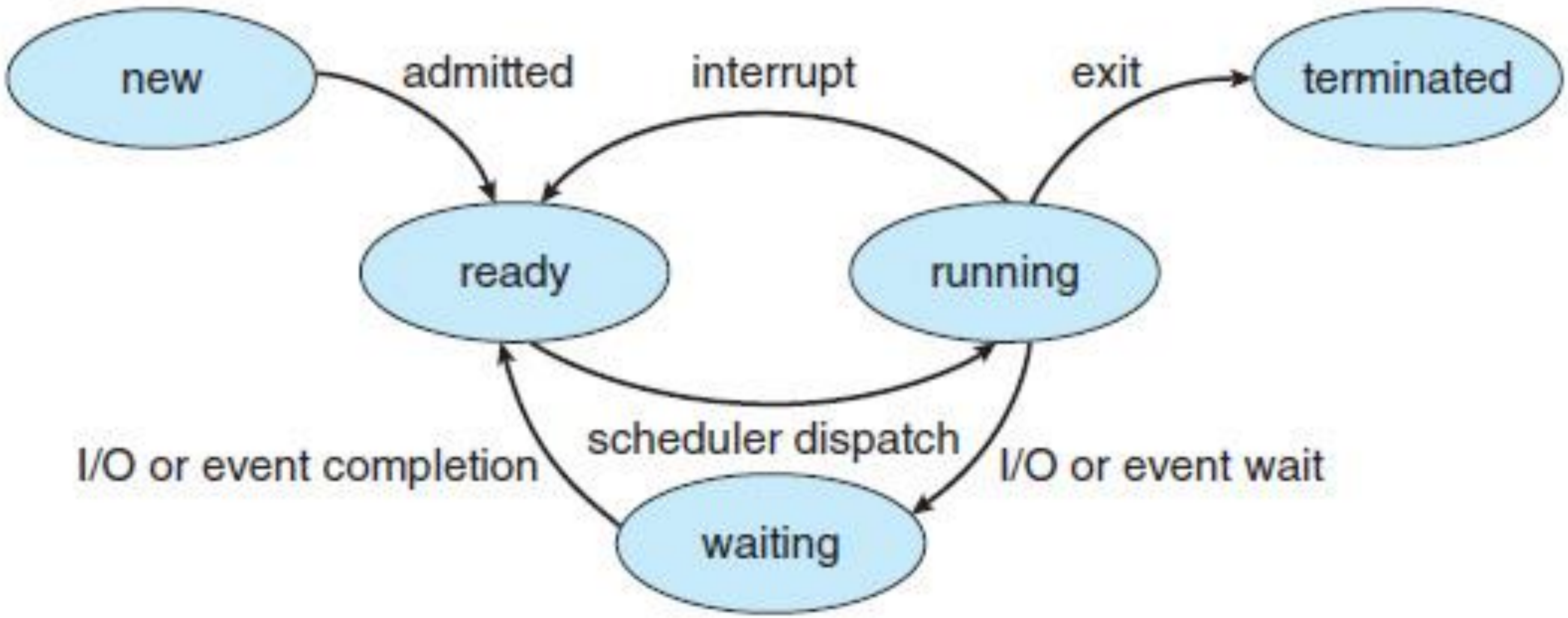


Process in Memory

Process State

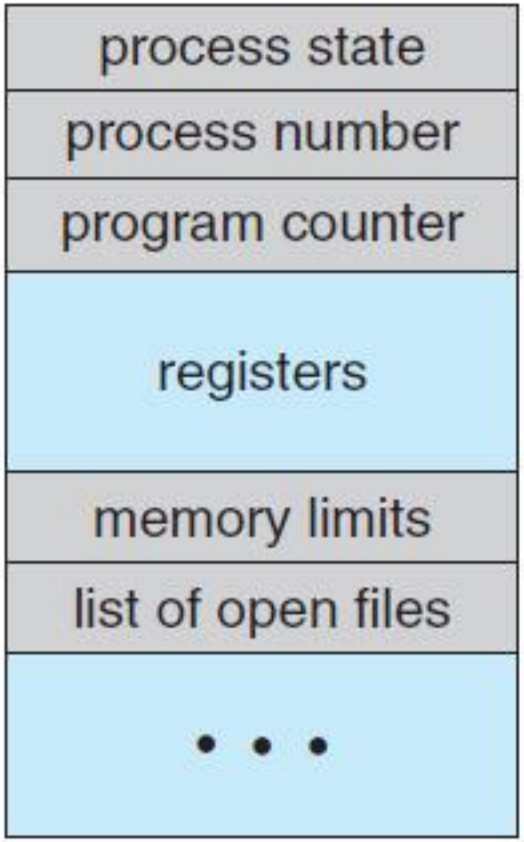
- As a process executes, it changes **state**
  - **new:** The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready:** The process is waiting to be assigned to a processor
  - **terminated:** the process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

Diagram of Process State

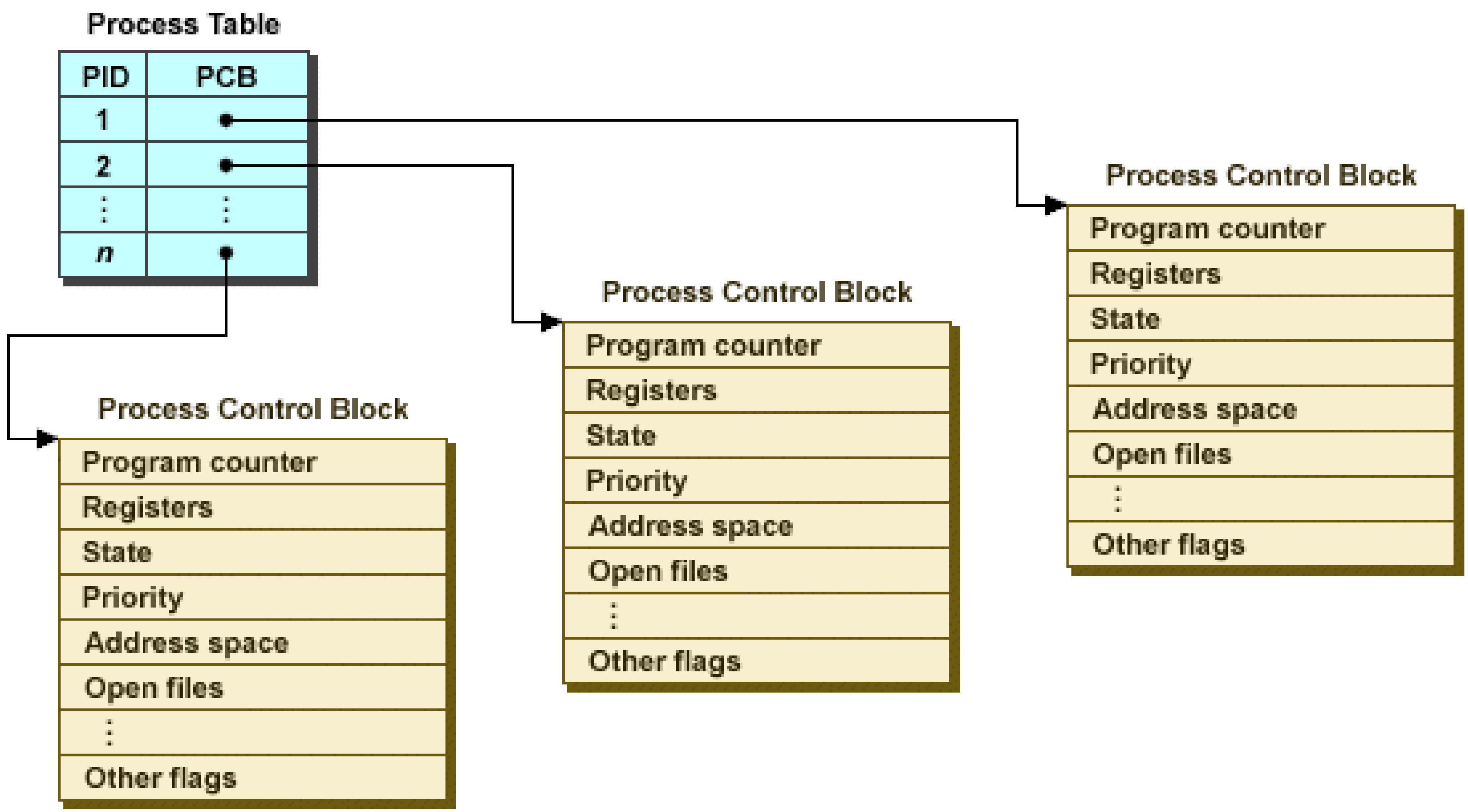


Process Control Blocks

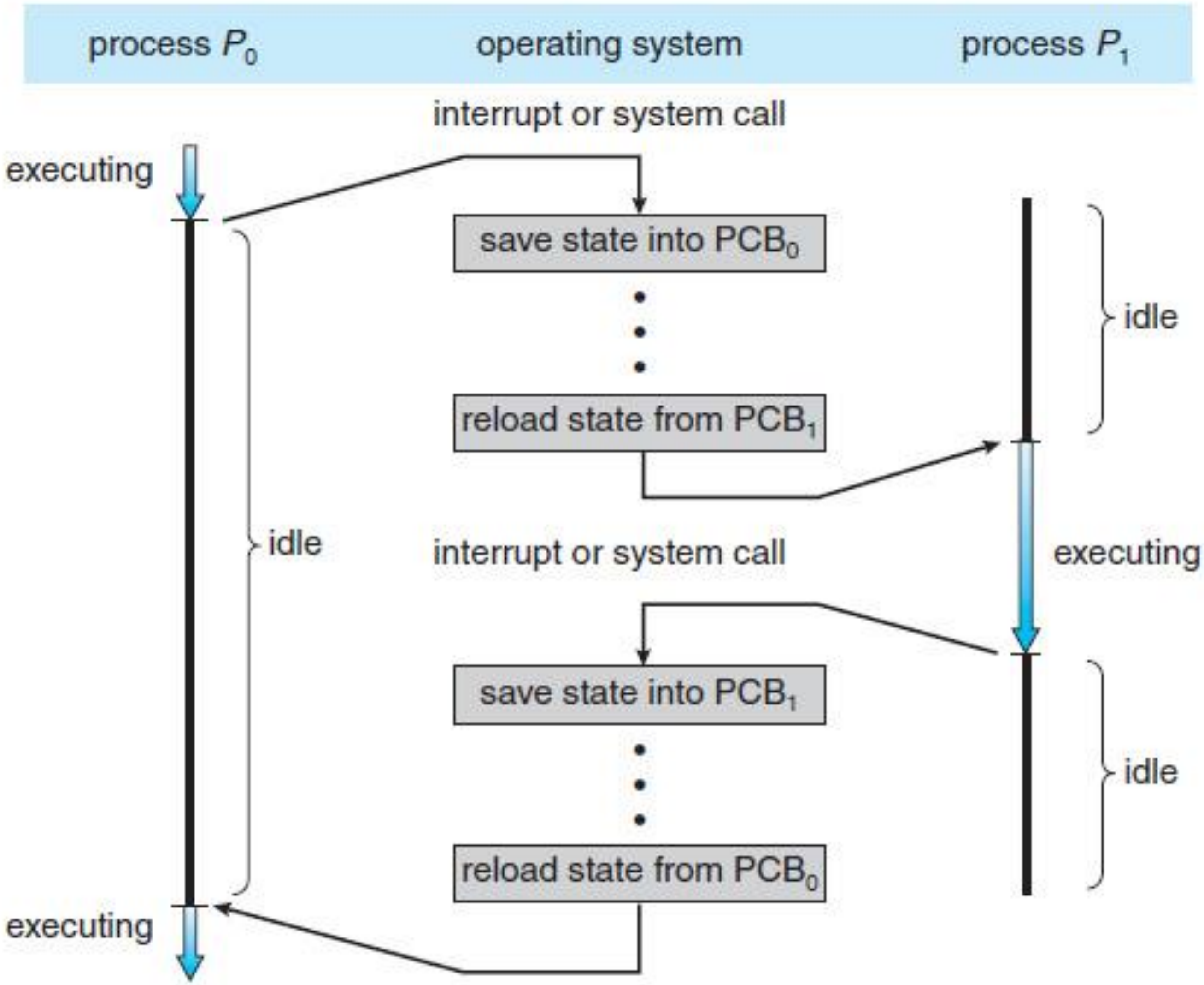
- Each process is represented in the operating system by a **process control block** (PCB)
- PCB contains many pieces of information associated with a specific process:
  - **Process state:** running, waiting, etc
  - **Program counter:** location of instruction to next execute
  - **CPU registers:** contents of all process-centric register
  - **CPU scheduling information:** priorities, scheduling queue pointers
  - **Memory-management information:** memory allocated to the process
  - **Accounting information:** CPU used, time limits, account numbers, job, or, process numbers
  - **I/O status information:** I/O devices allocated to process list of open files



Process Control Blocks



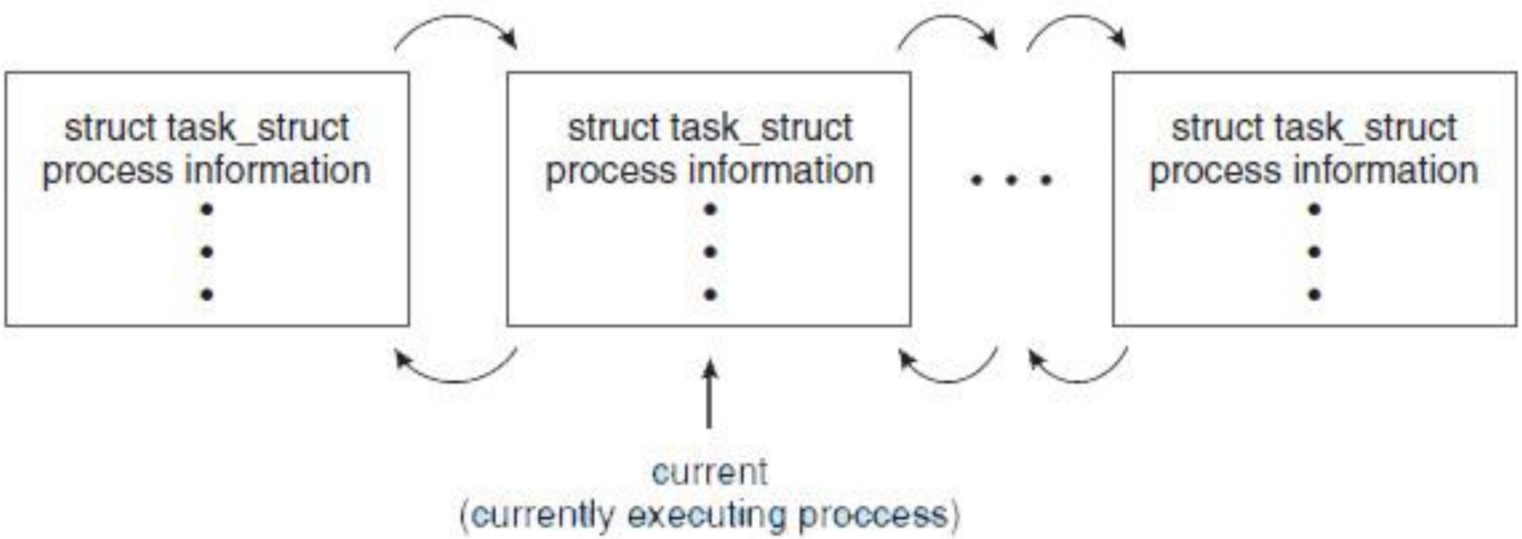
CPU Switch From Process to Process



Process Representation in Linux

- Represented by the C structure task_struct

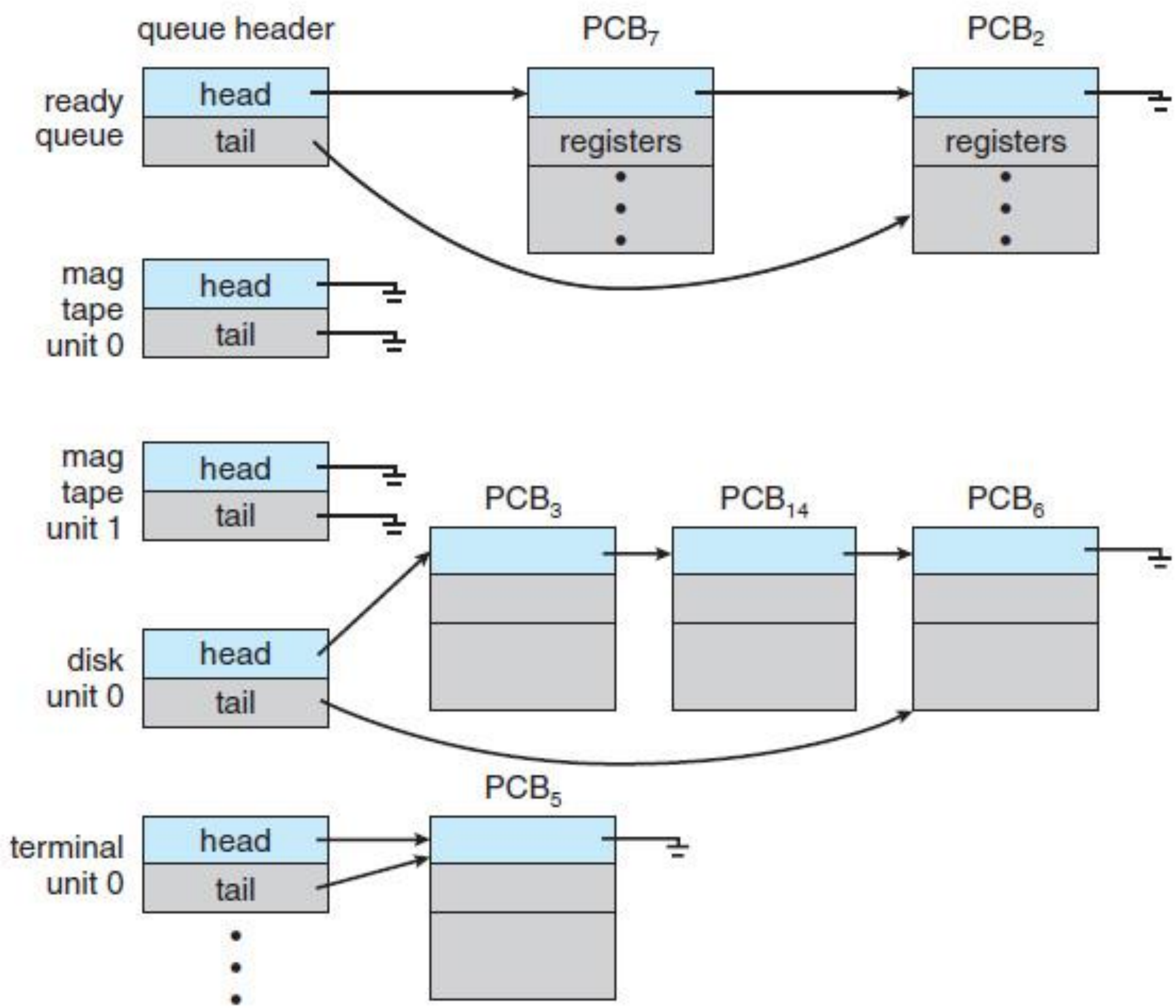
```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
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, read, and waiting to execute
  - **Device queues**: set of processes waiting for an I/O device
  - Processes migrate among the various queues

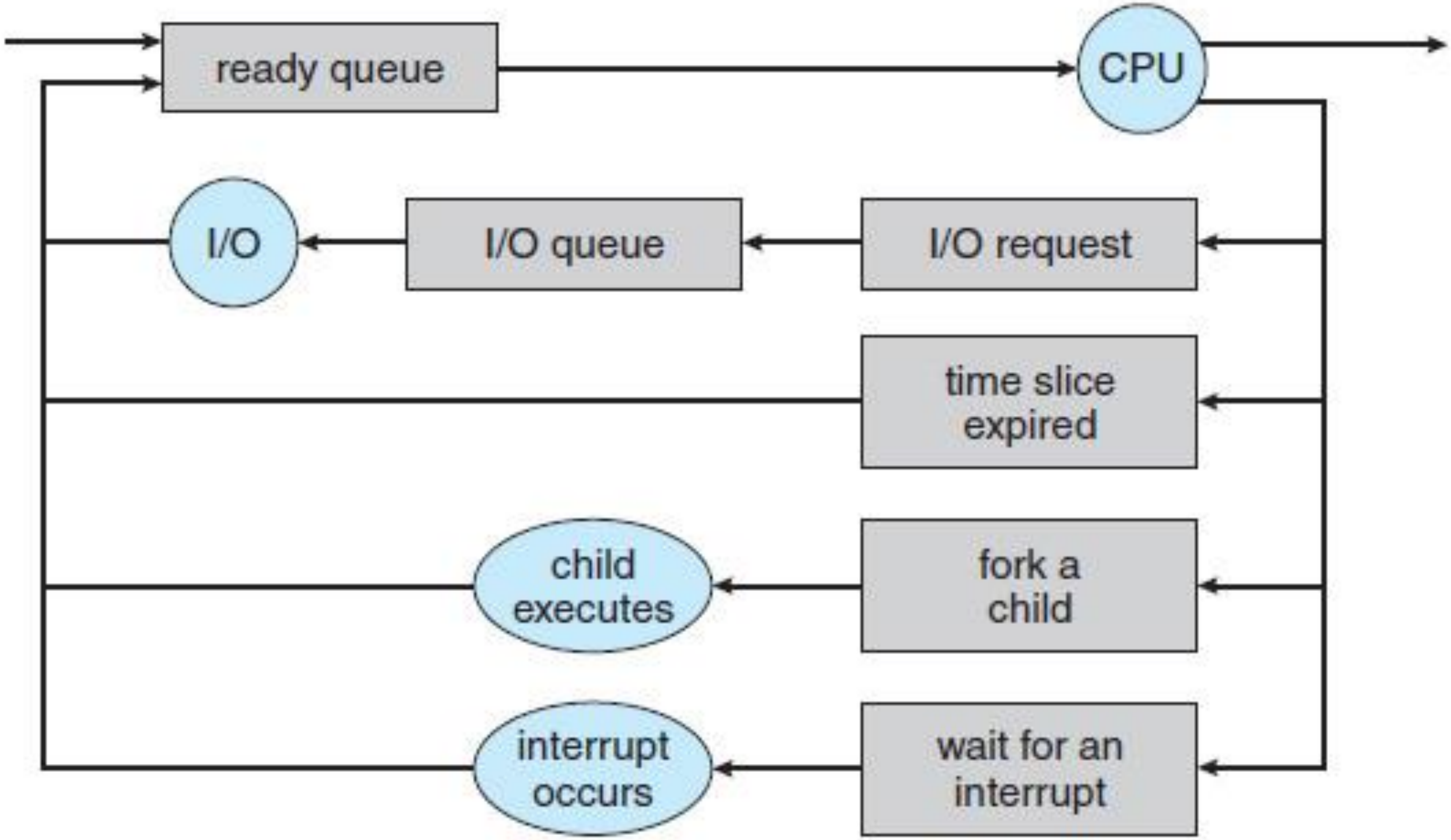
Ready Queue and Various I/O Device Queues



Scheduling Queues

- As processes enter the system, they are put into a **job queue** consisting of all processes in the system
- A new process is initially put in the ready queue
- It waits there until it is selected for execution
- 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 be placed in an 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

Scheduling Queues



Schedulers

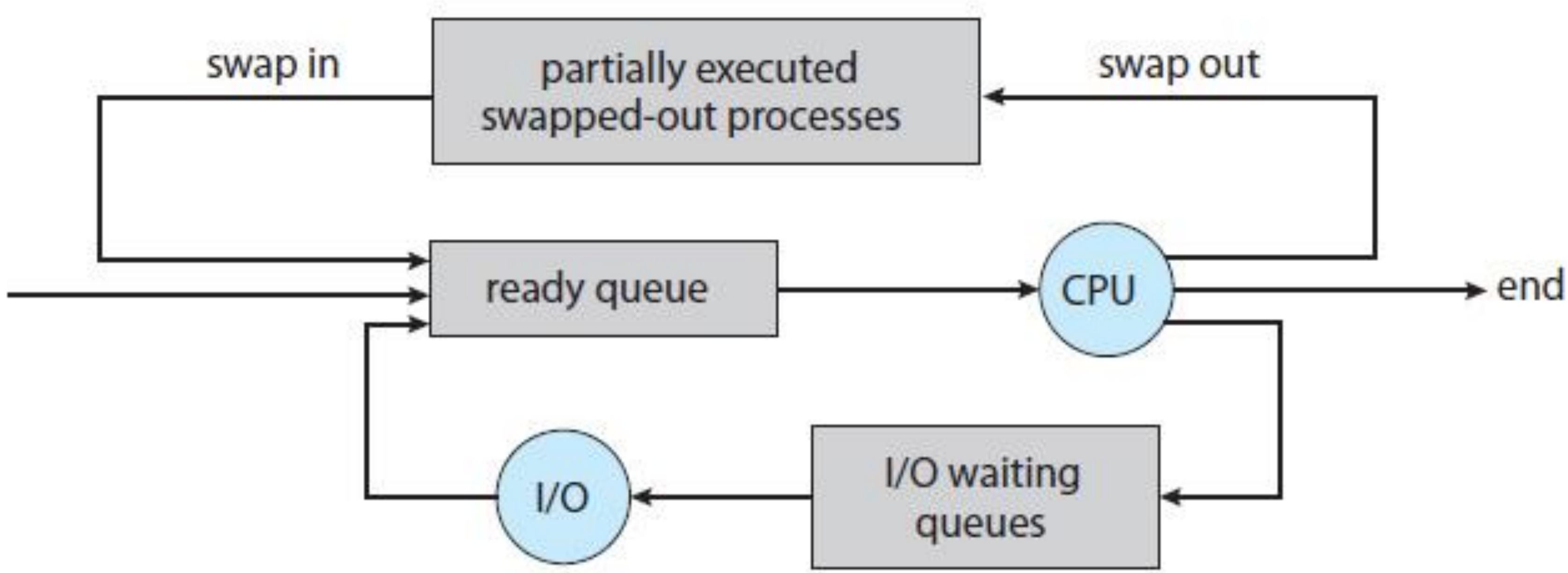
- A process migrates among the various scheduling queues throughout its lifetime
- The OS must select processes from the queues for scheduling
- The selection process is carried out by the appropriate **scheduler**
- There are three scheduler
  - Long-term scheduler or job scheduler
  - Short-term scheduler or CPU scheduler
  - Medium-term scheduler

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) => (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 computation; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

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

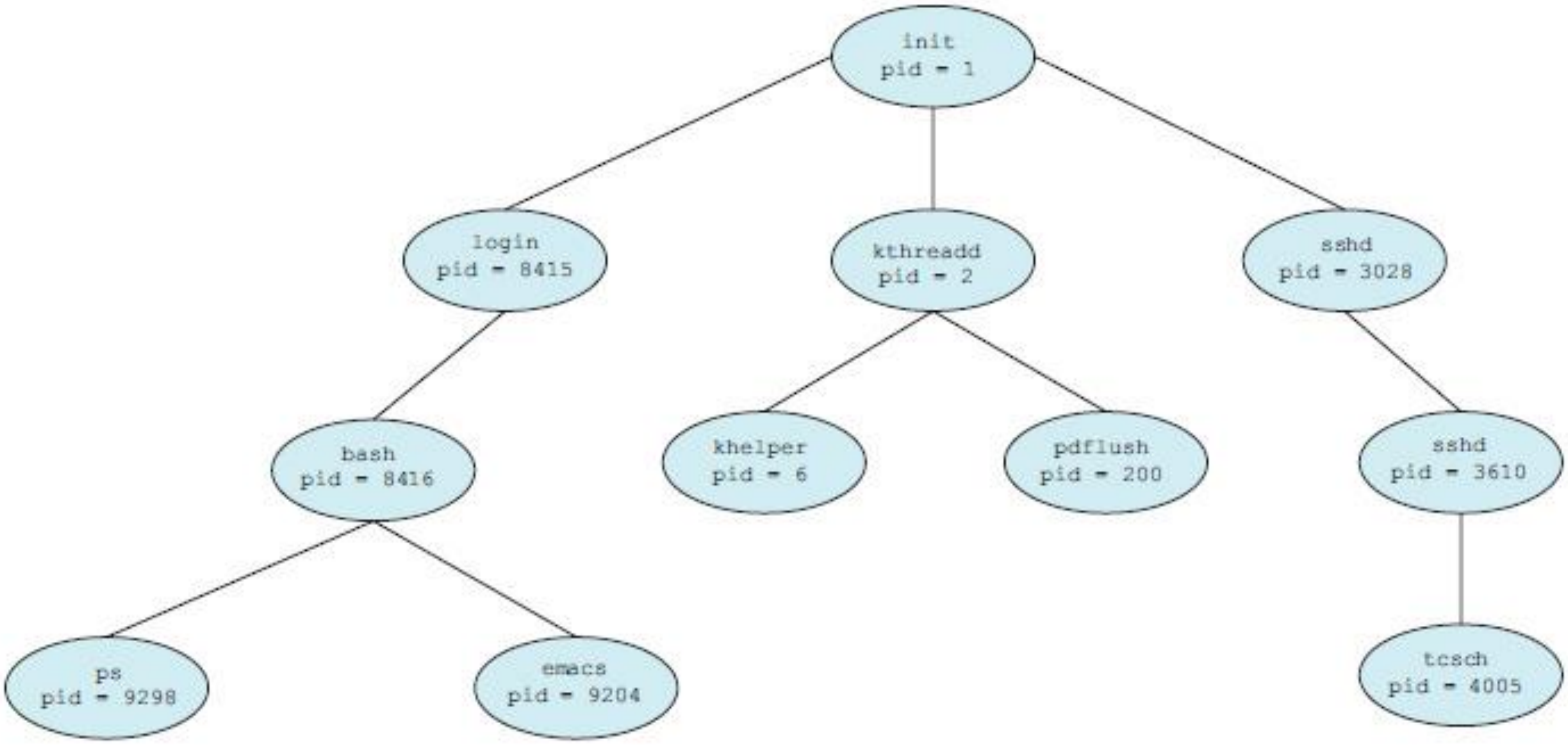
Operations in Processes

- Processes in most systems can execute concurrently
- They may be created and deleted dynamically
- System must provide mechanisms for:
    - Process creation
    - Process termination

Process Creation

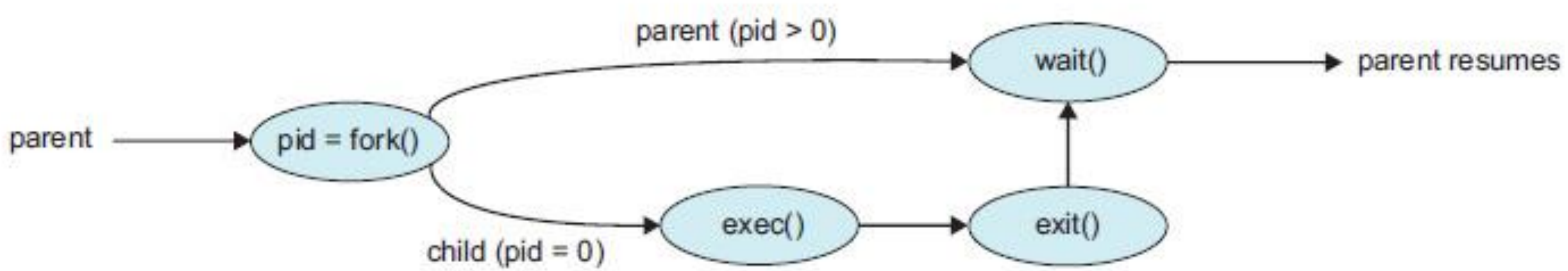
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Pid provides a unique value for each process in the system and can be used as an index to access various attributes of a process
- 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



A Tree of Processes in Linux

- **Address space**
  - Child duplication 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



Creating a Separate Process using the Unix fork

```
#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;
}
```

Creating a Separate Process via Window API

```
#include <stdio.h>
#include <windows.h>

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

- 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
- 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 exist 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**