# CSC 4320/6320: Operating Systems

Georgia State University

# Chapter 03: Processes

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

# Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

# Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

# Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.

# Process Concept

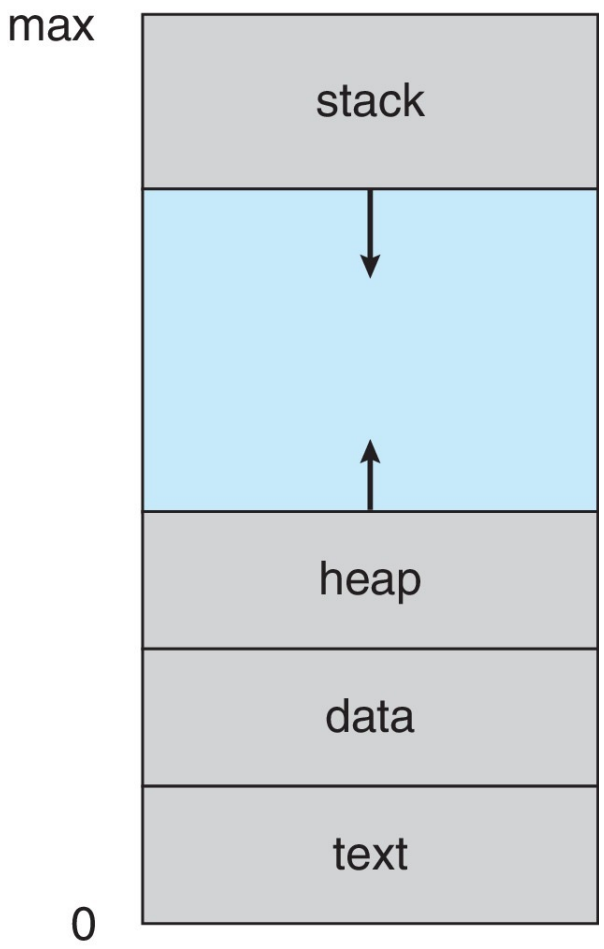
- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - 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.)

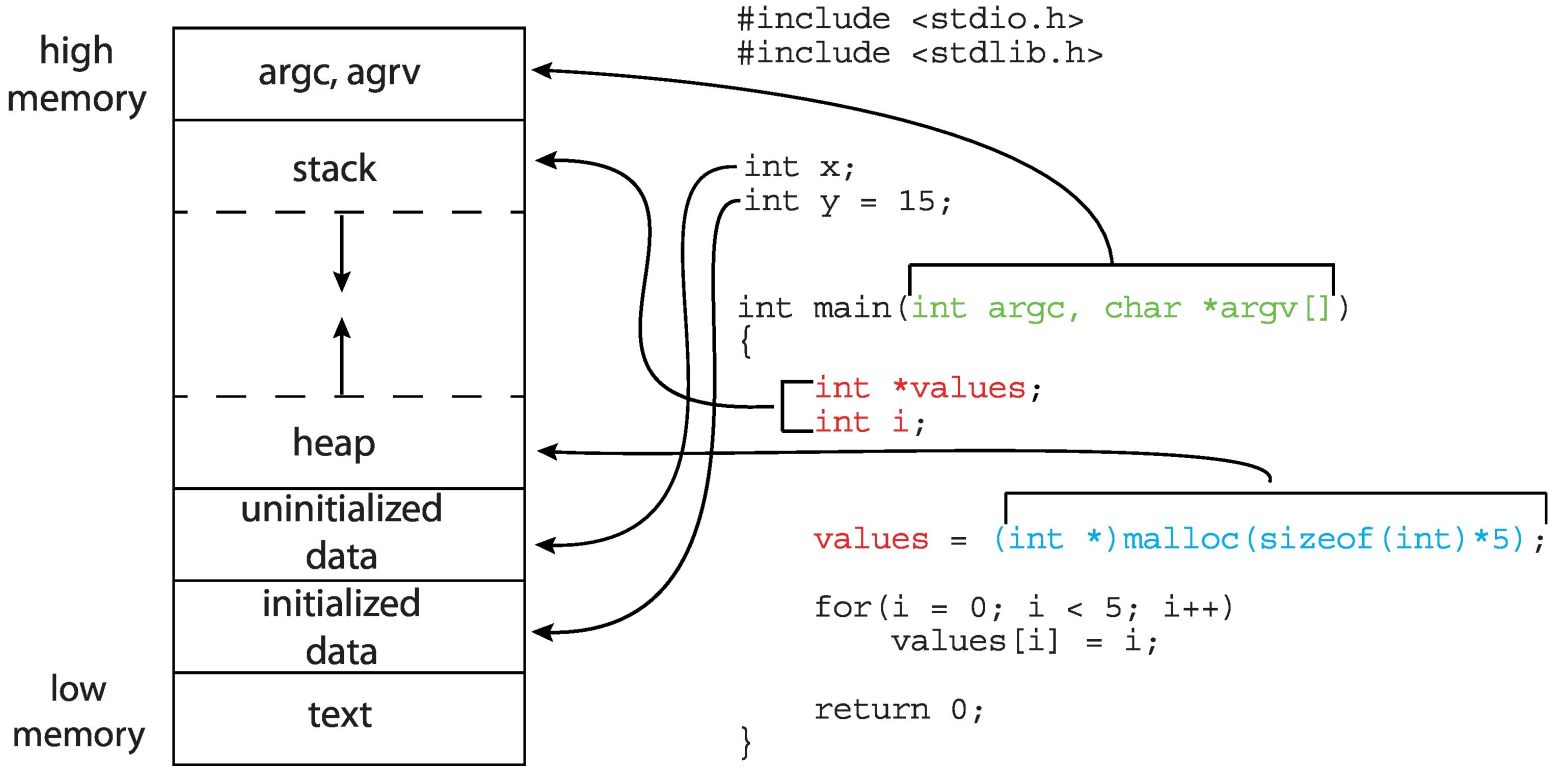
- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

# Process in Memory

The blank (blue) space is **available memory** for the heap and stack.



# Memory Layout of a C Program




global data section is divided into (a) initialized data and (b) uninitialized data. A separate section is provided for the argc and argv parameters passed

# Watching the size

One can see the size of a process using ‘size’ command:

Syntax: size executable_file_name

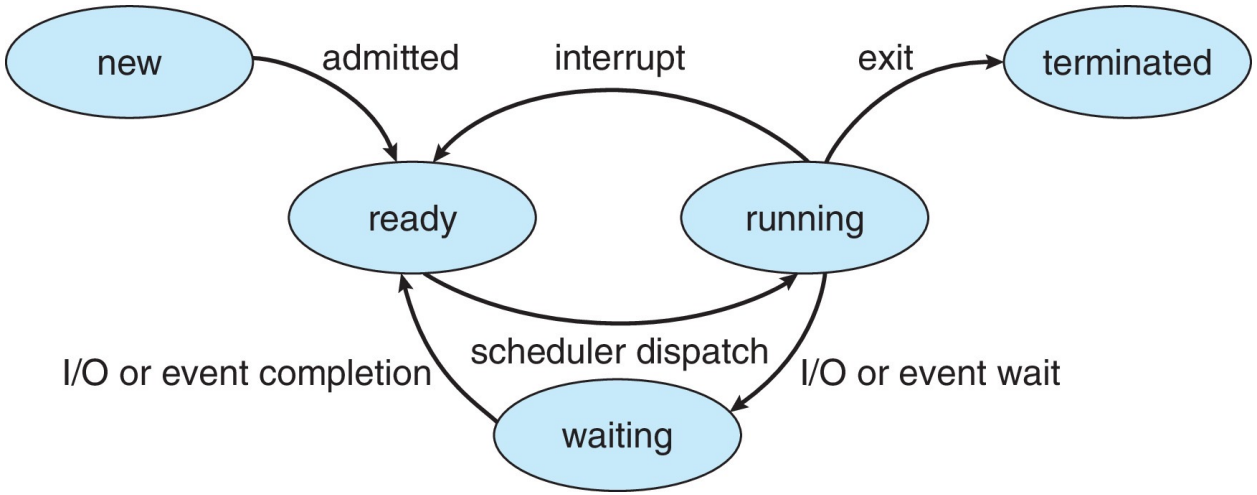
```
text    data    bss    dec    hex    filename
1158    284     8      1450   5aa    memory
```

- `data` field refers to uninitialized data, and bss refers to initialized data. (bss: block started by symbol)
- `dec` and `hex` are sums of the items

# 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




I/O or event may include:

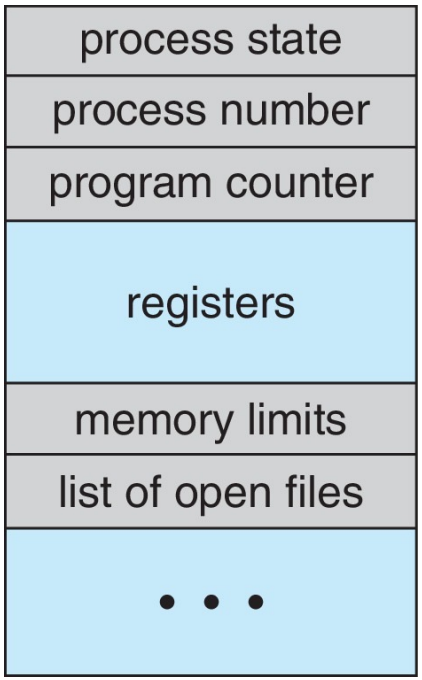
- File I/O
- Device I/O
- Network Communication (for data via network)
- IPC for a message via wait() call
- Waiting for shared memory access turn
- Waiting for a resource (e.g., a file access, GPU, mutex lock, semaphore, etc.)

When a process gets created, it becomes a new process. It moves to **Ready State** once all necessary resources are allocated.

# 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- a process priority, 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



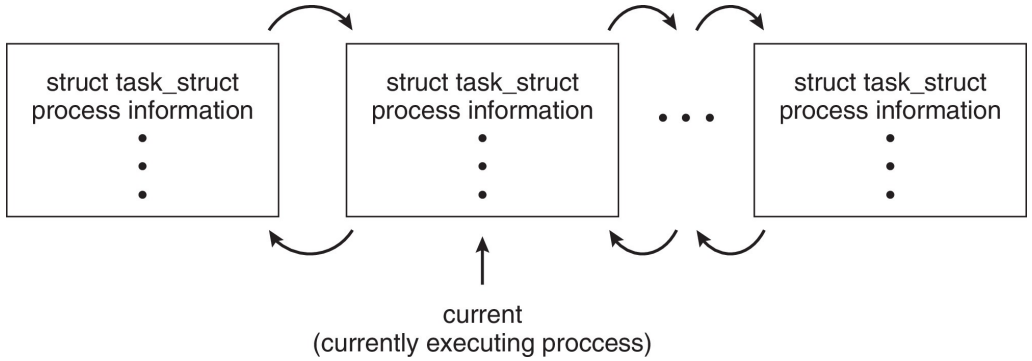
# Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
    - Consider multithreaded word processor (handles user input, spell checking, etc.)
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

# Process Representation in Linux

PCB is 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 */
```



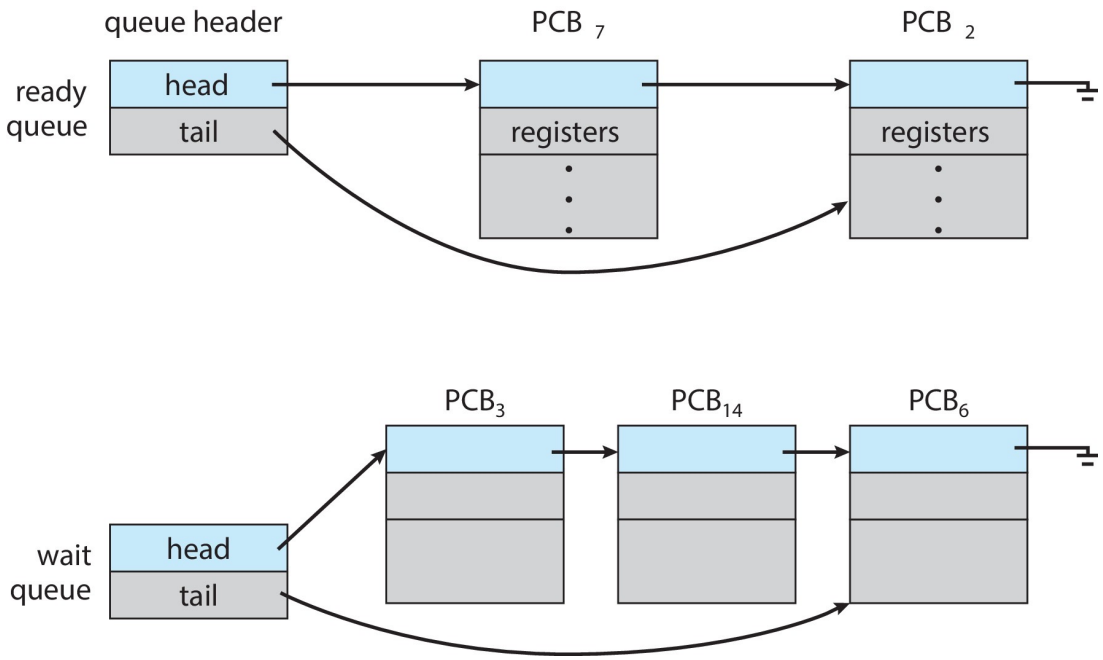

The number of processes currently in memory is known as the degree of multiprogramming.

# Process Scheduling

Georgia State University

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues

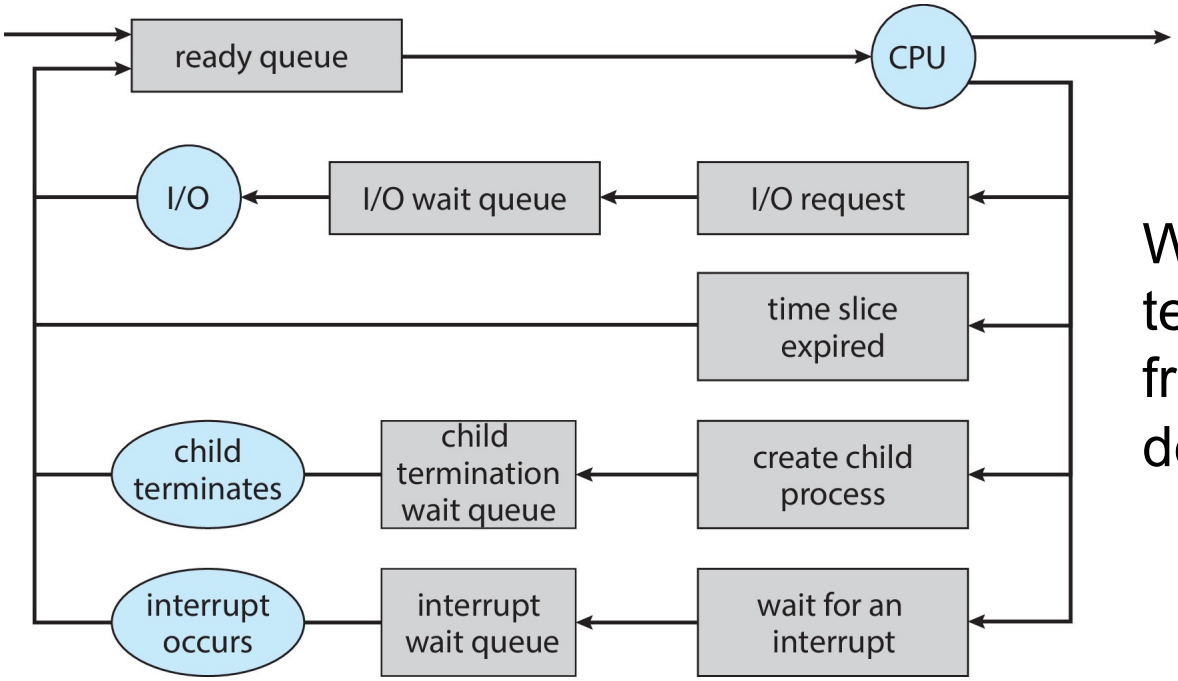
# Ready and Wait Queues




The ready queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB

Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a wait queue

# Representation of Process Scheduling




Queueing diagram

When a process finally terminates, it is removed from all queues and PCB deallocated

The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

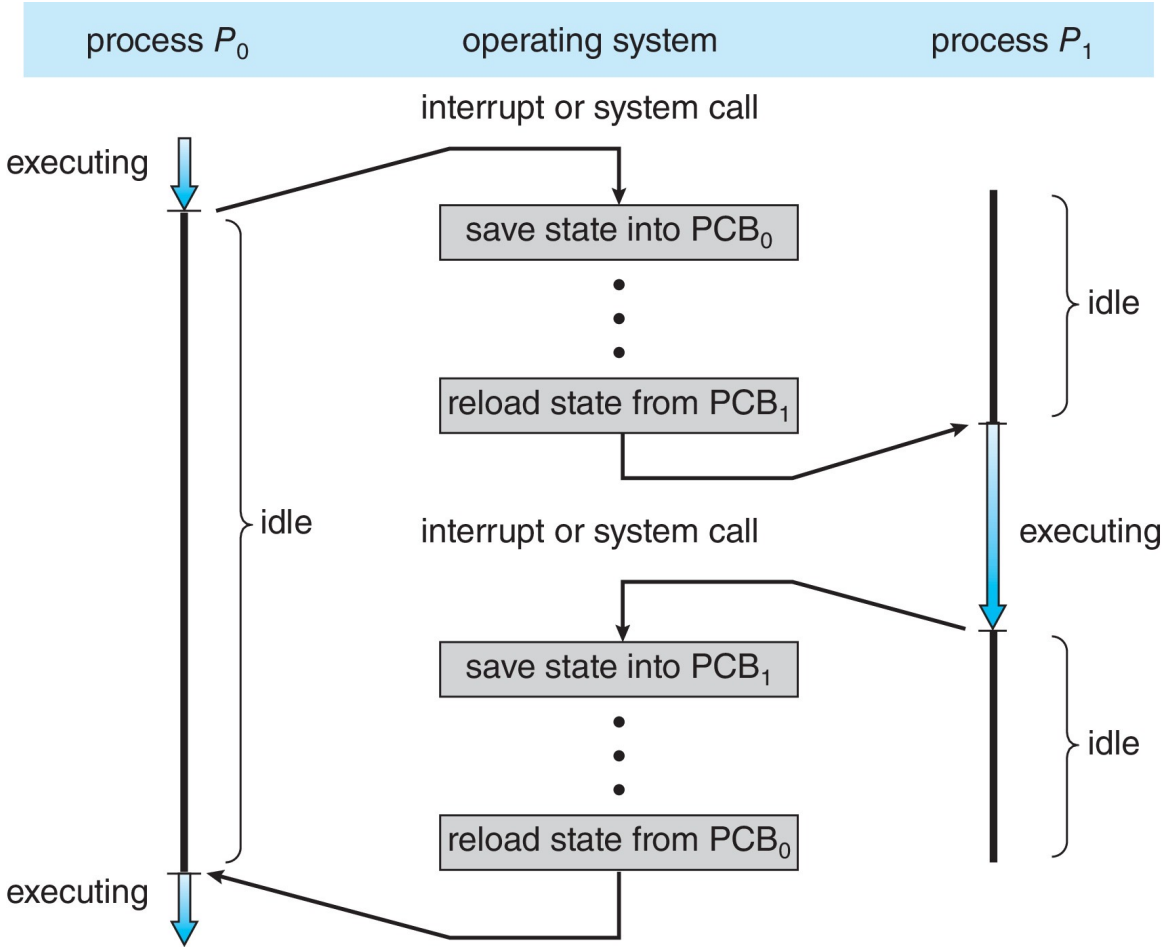
# Context Switch

- 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 pure 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
  - A context switch here simply requires changing the pointer to the current register set.

# CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

Context-switch time is pure overhead



# Operations on Processes

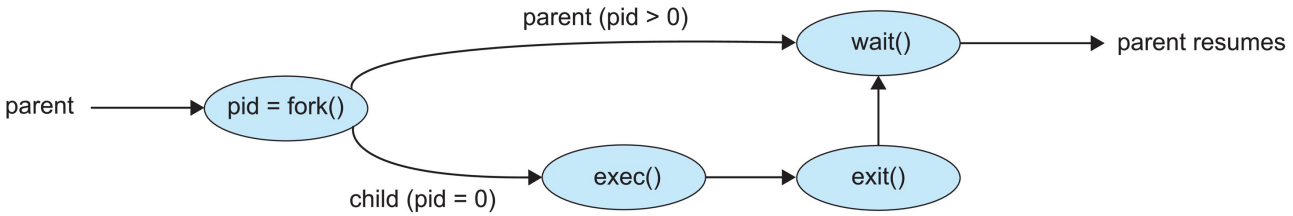
- System must provide mechanisms for:
  - Process creation
  - Process termination

# 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**)
- 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

# Process Creation (Cont.)

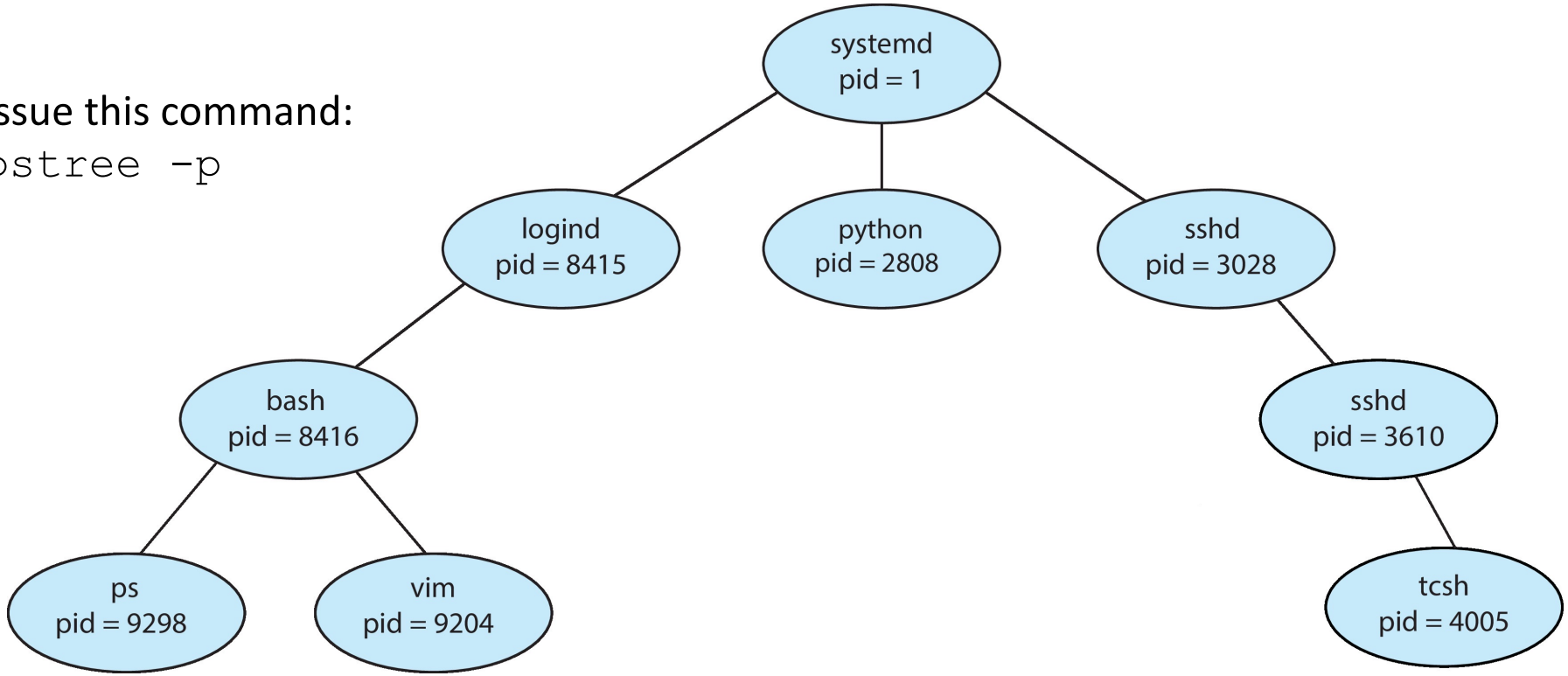
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program
  - Parent process calls `wait()` waiting for the child to terminate



# A Tree of Processes in Linux

Issue this command:

```
pstree -p
```




`systemd` – root parent process

`logind` -- responsible for managing clients that log onto the system.

- a client has logged on and is using the bash shell
- Using CLI, this user has created the process ps and the vim editor.

# 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;
}
```

- `pid_t` is a data type defined in the header `<sys/types.h>`
- Called, “process ID type”
- Usually integer/long int

Fork() Video: https://www.youtube.com/watch?v=IbcMFYvHL9g&t=12s

# Creating a Separate Process via Windows 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);
}
```

STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files.

PROCESS_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread.

ZeroMemory(): to allocate memory for each of these structures before proceeding with CreateProcess().

# Example Using execl()

```
#include<unistd.h>
#include<stdio.h>
int main()
{
  printf ("I'm process %d and I'm about to exec a ls -l\n", getpid());
  execl ("/bin/ls", "ls", "-l", NULL);  /* Execute ls */
  printf ("This line should never be executed\n");
  return 0;
}
```

```
$myexec
I'm process 61713 and I'm about to exec a ls -l
total 6
-rwxr-xr-x   1 mrahman21  staff   33736 Nov 18 00:08 a.out
-rwx------@  1 mrahman21  staff    8572 Sep  4 18:38 array
-rw-r--r--   1 mrahman21  staff     408 Oct 18 16:59 array_modify.c
-rwxr-xr-x   1 mrahman21  staff   33592 Oct 22 21:40 array_ptr
-rw-r--r--   1 mrahman21  staff     473 Oct 22 21:40 array_ptr_increment.c
-rw-r--r--   1 mrahman21  staff     422 Oct  8 21:32 array_side_effect.c
```

# 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**

# Review Questions

1. The ____ of a process contains temporary data such as function parameters, return addresses, and local variables.

A) text section
B) data section
C) program counter
D) stack ✓

2. A process control block ____.

A) includes information on the process's state ✓
B) stores the address of the next instruction to be processed by a different process
C) determines which process is to be executed next
D) is an example of a process queue

3. A process may transition to the Ready state by which of the following actions?

A) Completion of an I/O event
B) Awaiting its turn on the CPU
C) Newly-admitted process
D) All of the above ✓

# Questions

1. The _____ refers to the number of processes in memory.

A) process count
B) long-term scheduler
C) degree of multiprogramming ✓
D) CPU scheduler

2. The list of processes waiting for a particular I/O device is called a(n) ____.

A) standby queue
B) device queue ✓
C) ready queue
D) interrupt queue

3. A _____ saves the state of the currently running process and restores the state of the next process to run.

A) save-and-restore
B) state switch
C) context switch ✓
D) none of the above

# Questions

1. When a child process is created, which of the following is a possibility in terms of the execution or address space of the child process?

A) The child process runs concurrently with the parent.

B) The child process has a new program loaded into it.

C) The child is a duplicate of the parent.

D) All of the above ✓

2. A process that has terminated, but whose parent has not yet called wait(), is known as a _____ process.

A) zombie ✓

B) orphan

C) terminated

D) init

3. The _____ process is assigned as the parent to orphan processes.

A) zombie

B) init ✓

C) main

D) renderer