Operating Systems - Processes

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

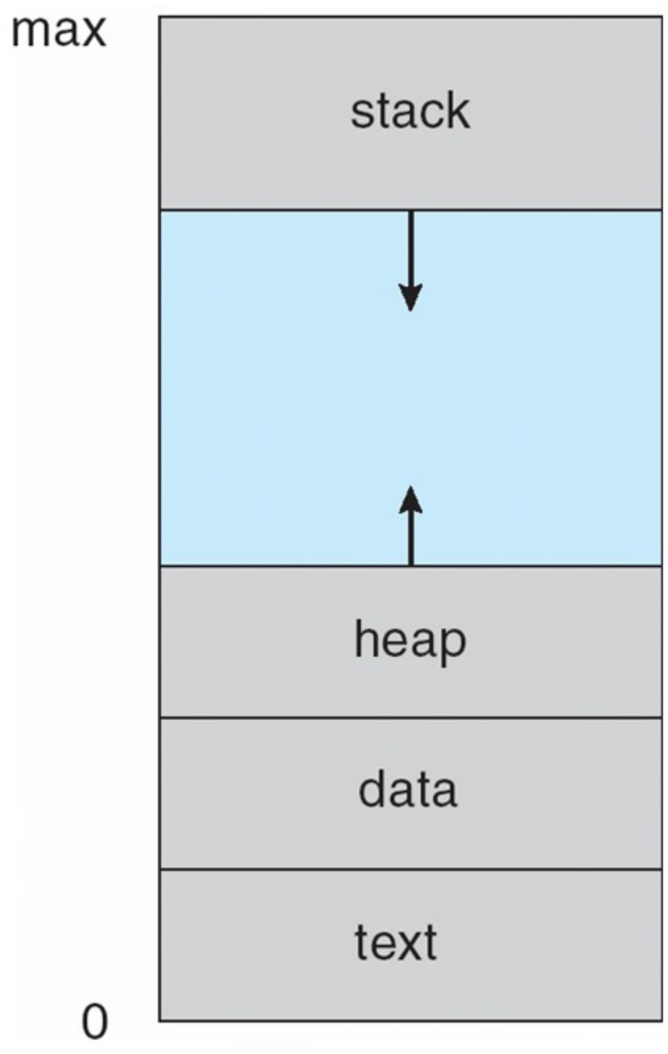
Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 3)

Process

- Process is a program in execution
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory

Process in Memory

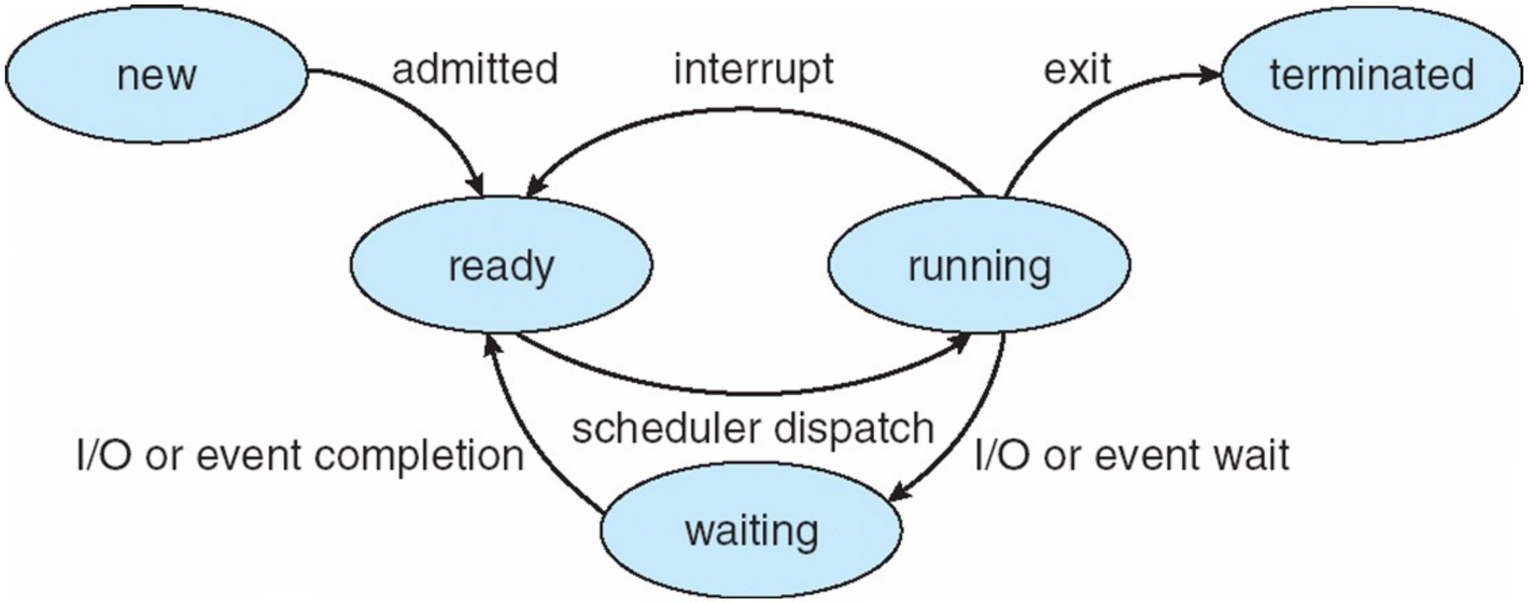
- **Text section** - compiled program code, read in from the non-volatile storage when the program is launched.
- **Data Section** - global and static variables, which are allocated and initialized prior to executing `main()`.
- **Heap** – Memory that is dynamically allocatedduring program run time and is managed via calls to `new`, `delete`, `malloc`, `free`, etc. in C.
- **Stack** - is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope.



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

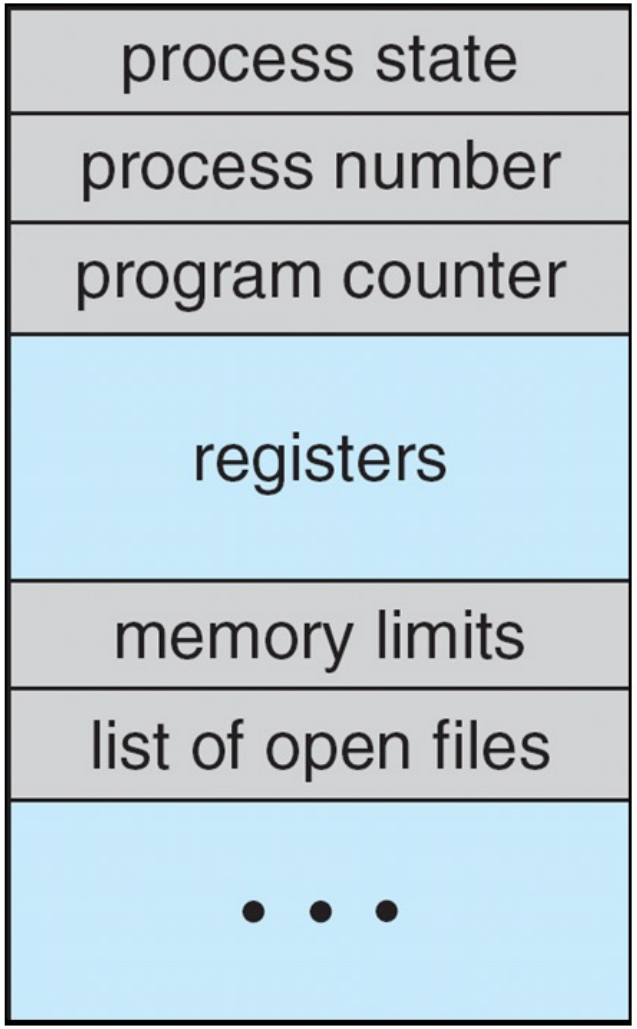
Diagram of Process State



Process Control Block (PCB)

PCB – Stores all the information associated with each process (also called **task control block**)

- **Process state** – running, waiting, etc
- **Process number** – Process ID
- CPU registers and program counter – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- And much more ….



Process Representation in Linux

Processes in Linux are referred to as tasks.

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 */
```

Types of Processes

- 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

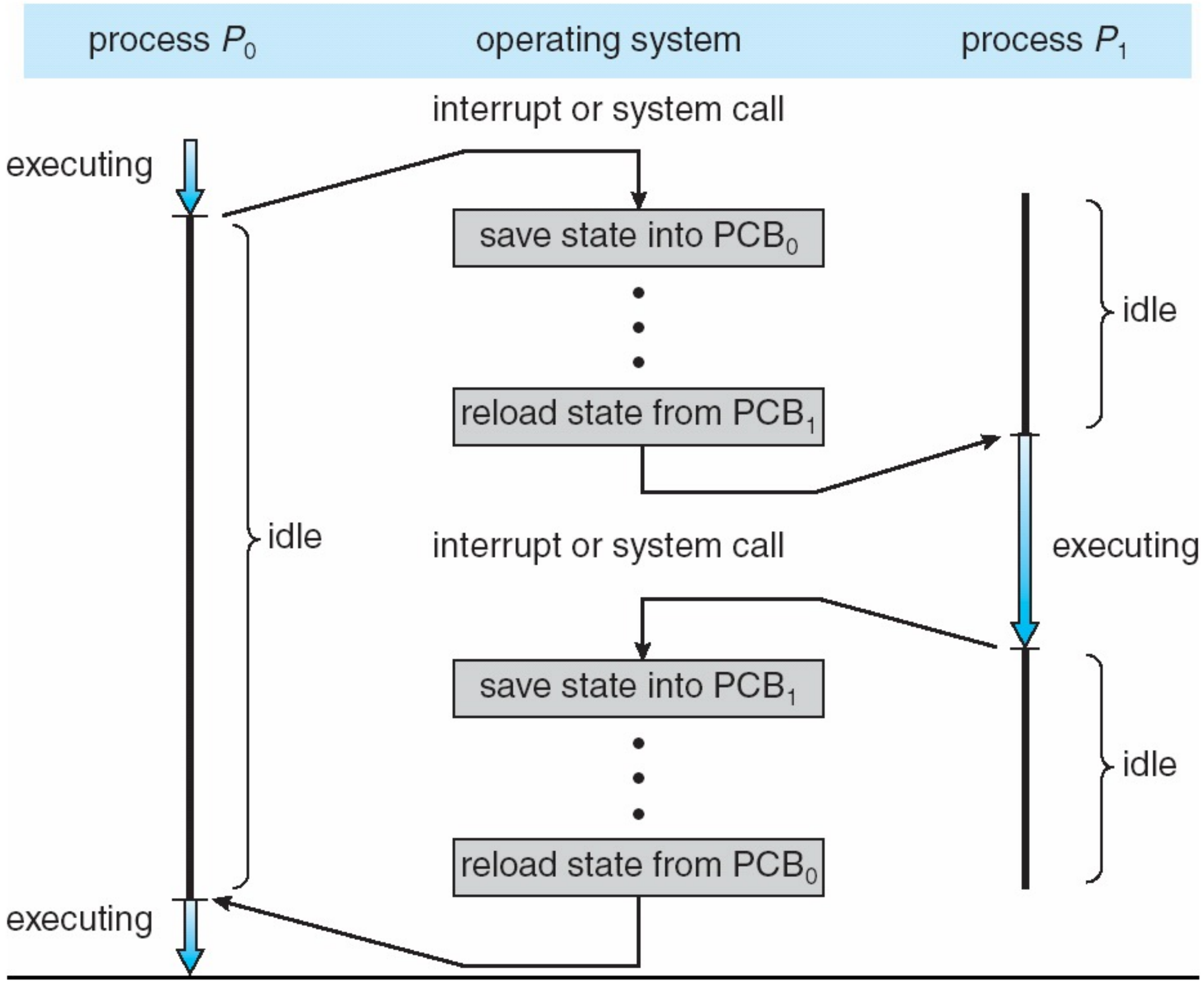
Threads

- Modern systems allow a process to have multiple threads associated with it.
- These threads can execute concurrently.
- More on Threads in the next chapter.

Context Switch

- **Context** of a process is represented in the PCB (value of CPU registers, the process state etc. See slide 3.9)
- When CPU switches to another process, the system must **save the context** of the old process and load the **saved context** for the new process. The process is called **context switching**
- Context-switch time is an 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

CPU Switch From Process to Process



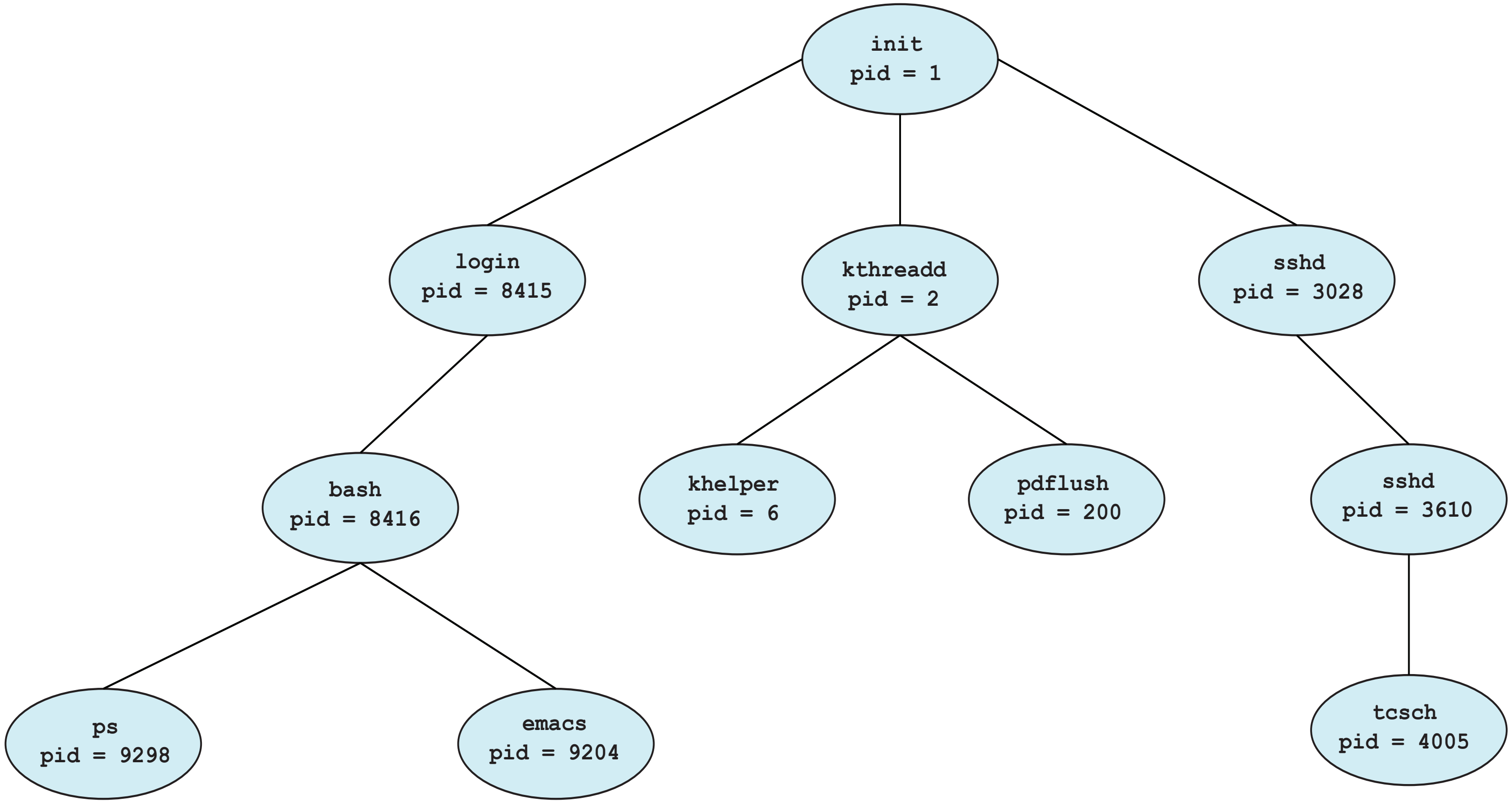
Operations on Processes

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

Process Creation

- Every process is given an integer identifier called the **process identifier** (**pid**):
- A process (**parent process**) can create another process (**child process**).
- **Parent** process creates **child** processes (using system calls), which, in turn creates other processes, forming a **tree of processes** or **process tree**.
- In addition to PID of a process, its parent PID (termed as PPID) is stored as well.

A Tree of Processes in Linux



- At system startup, **`init`** process is executed
  - Its process identifier is 1.
  - **`init`** then launches all system daemons and user logins, and becomes the parent of all other processes.

Parent – Child Sharing

- **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
- **Address space sharing options:**
  - Child is a duplicate of parent (has the same program and data as the parent)
  - Child has a new program loaded into it

Creating Processes in Linux/Unix

- `fork()` system call is used to create a new process
- `fork()` takes no arguments and returns a process ID (in the parent)
- The new process created by fork becomes the child process of the calling process.
- This child process has the same environment as its parent; that is, it is an exact copy of the parent with only a different process ID.
- After a new child process is created, both the parent and child will execute the next instruction following the `fork()` system call.

Question?

How many processes (including the parent) are created when the below code is executed?

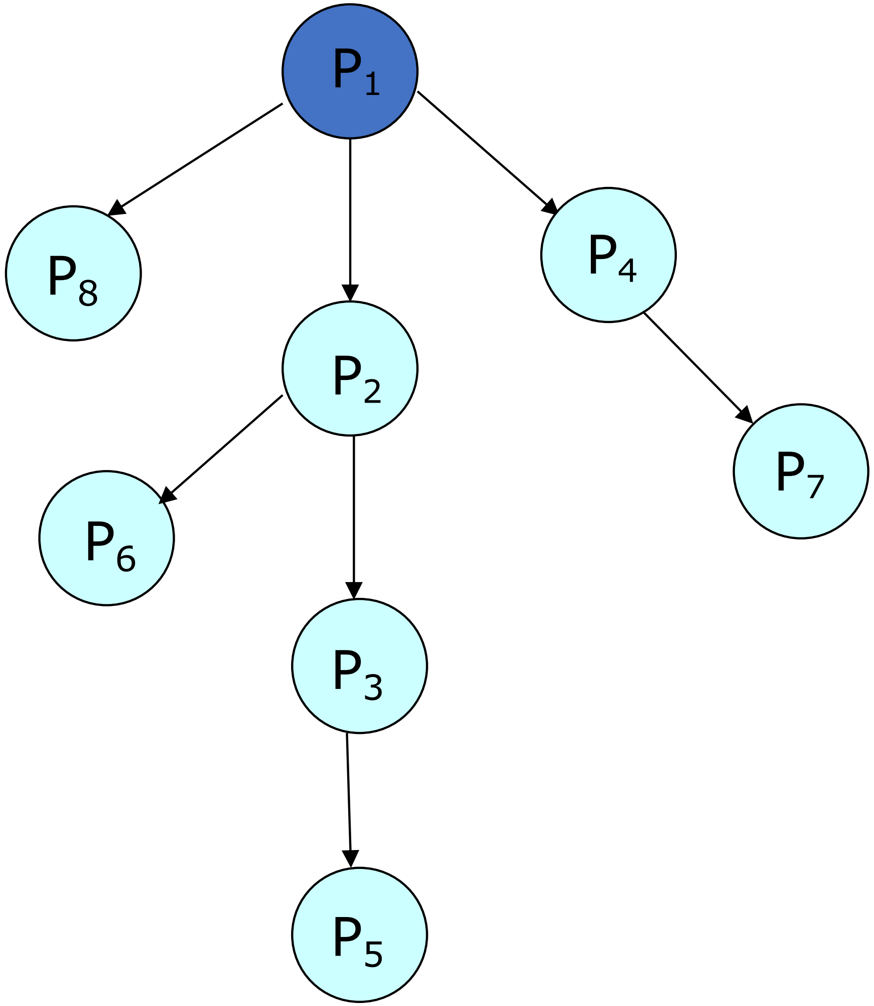
Draw the process tree for the below code.

```
int main()

{

        fork();

        fork();

        fork();

        return 0;

}
```

After executing this fork(), the child process created from it, executes instructions starting from the next statement; that is, fork().

return 0; return the exit status of main, where 0 indicates the program executed normally.

Process tree for question in slide 19



Creating Processes in Linux/Unix

- **How do we distinguish between a child process and parent process?**
- **`fork()`** returns a zero in the child process
- **`fork()`** returns the PID of the child process in parent
- Only if the creation of the child process was unsuccessful, **`fork()`** returns a negative value.

`getpid()` and **`getppid()`**

- Every process can query its own PID using the **`getpid()`**.
- Every process can query its parent PID using **`getppid()`**.

Question?

How many processes (including the parent) are created when the below code is executed?

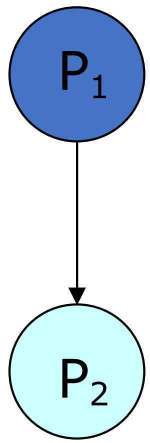
Draw the process tree for the below code.

```
int main()
{
    pid_t pid;
    pid = fork();
    if(pid == 0) { /*Child process*/
        printf("Child Process with PID: %d\n",getpid());
    }
    else { /*Parent process*/
        printf("Parent Process with PID: %d\n",getpid());
    }
    return 0;
}
```

pid_t is the type to store Process ID

fork() returns the PID of the child process in the parent process and returns 0 in the child process.

Process tree for question in slide 22



exec() system call

- The **`exec()`** system call used right after **`fork()`** enables the child process to execute a different program than the one it inherits from its parent process.
- This act is also referred to as an **overlay**.
- There is a family of exec() functions, all of which have slightly different characteristics. See the Linux manual for more details.

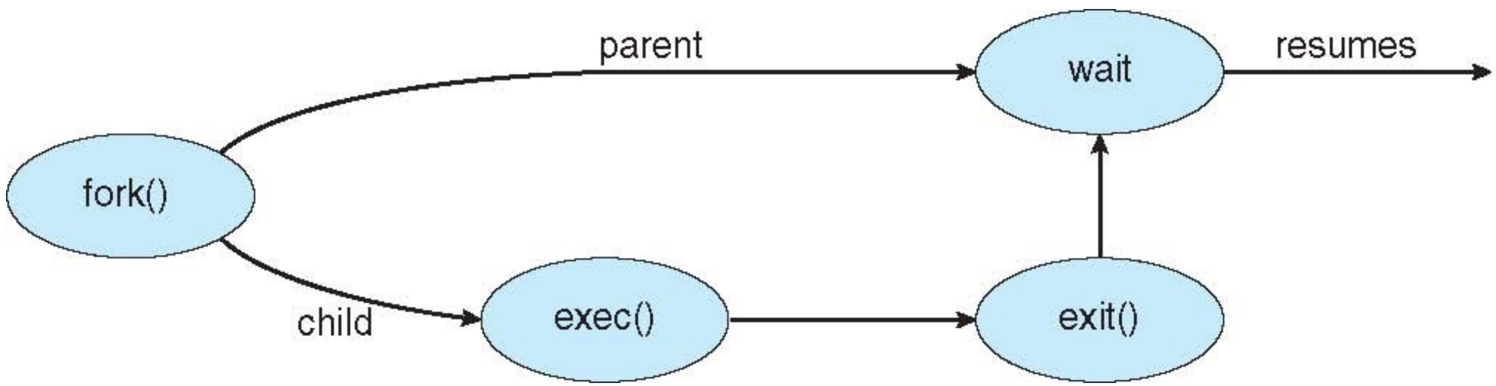
Process Termination

- After executing the last statement, a process is terminated
  - Implicitly – using the **`return`** statement
  - Explicitly – using the **`exit()`** system call
- 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 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.

Process creation using the fork() system call



Child Process – Termination status

- When a child process terminates, the parent can know the child's exit status using the **`wait()`** system call.
- The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- Unix maintains a **table of processes**. This table contains the list of all processes running and includes the process status.
- If a parent process terminates, its entry is removed from the table.
- If a child process terminates, its entry is removed from the table only after the parent process invokes a **`wait()`**.
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**

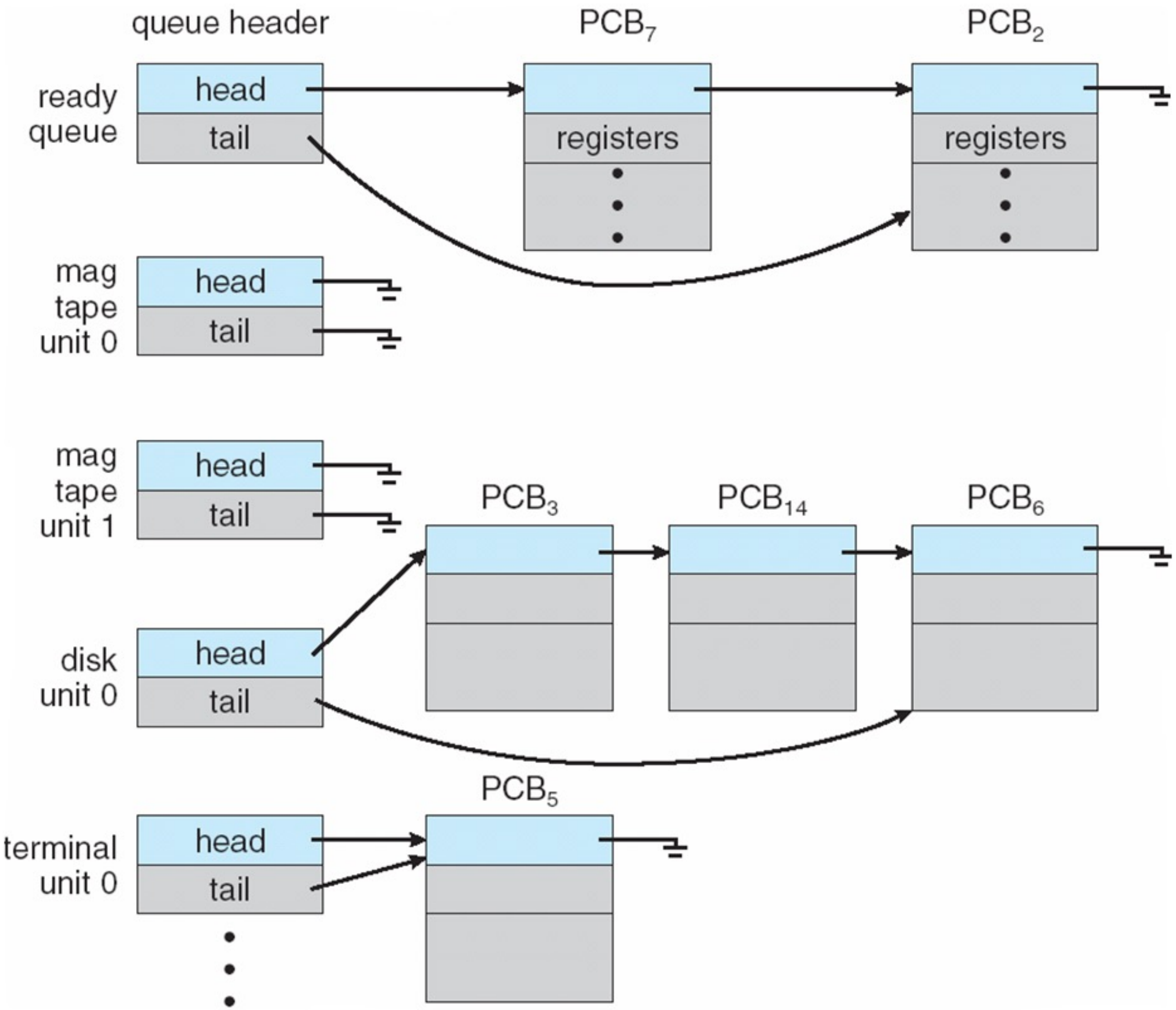
C Program Forking Separate Process

```
#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;
}
```

Process Scheduling

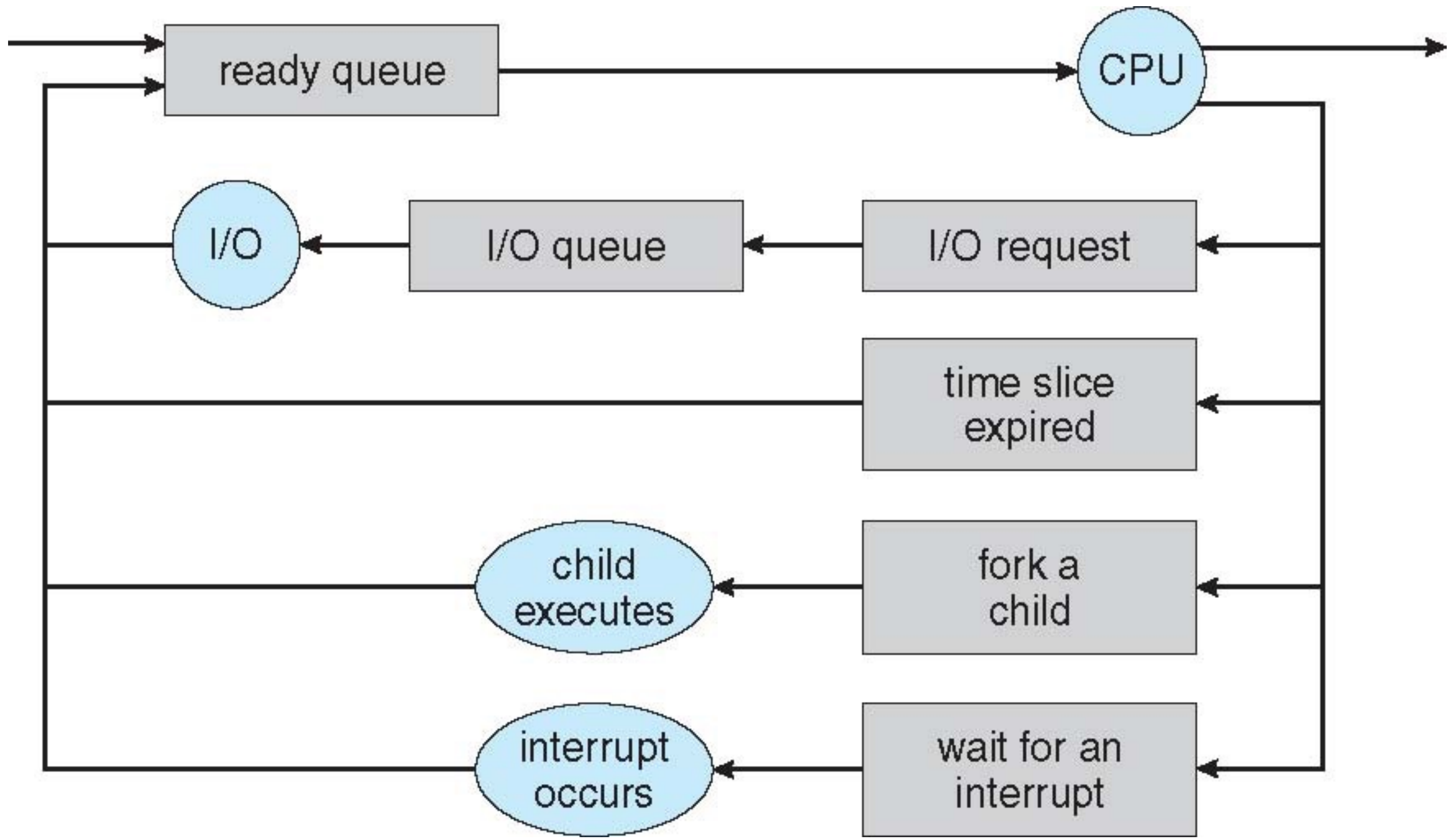
- Goal: Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler must meet the above objectives by implementing suitable scheduling policies.
- Operating System maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready 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. Usually a separate device queue for each device.
- Processes migrate among the various queues
- **Scheduler:** selects a process from a queue.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



Schedulers

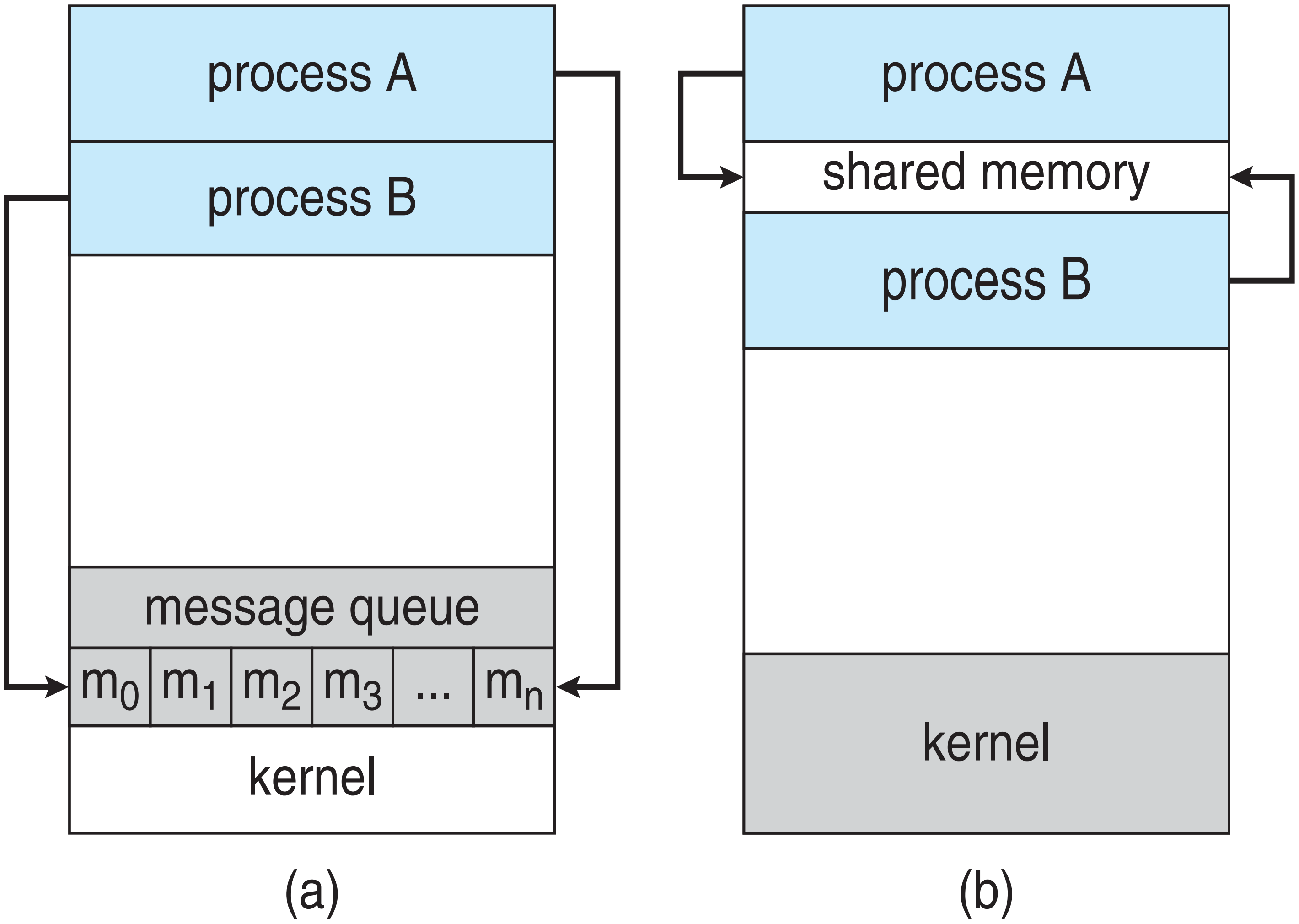
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow). Therefore, can use sophisticated scheduling algorithms.
  - Typically, seen on a process intensive systems.
  - The long-term scheduler controls the **degree of multiprogramming and** good ***process mix***
- **Swapping:** is the term used for removing a process from main memory store on disk and later bring it back in from disk to continue execution
- **Medium – term scheduler:** used to queue swapped jobs
  - can be added if degree of multiple programming needs to decrease.

Interprocess Communication (IPC)

- Processes within a system may be
  - ***Independent*** or
  - ***Cooperating*** *(it* can affect or be affected by other processes, including sharing data)
- Reasons for cooperating processes:
  - Information sharing, computation speedup, modularity and convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

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



IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the **control of the user** processes not the operating system.
- It is more complicated to set up and doesn't work as well across multiple computers.
- Used for sharing large amount of data.
- Major issues – synchronize process actions when accessing shared memory.

IPC – Message Passing

- Operating system provides message passing capability.
- As a result, Message Passing requires system calls for every message transfer, and is therefore slower.
- However, it is simpler to set up and works well across multiple computers.