Chapter 3: Processes

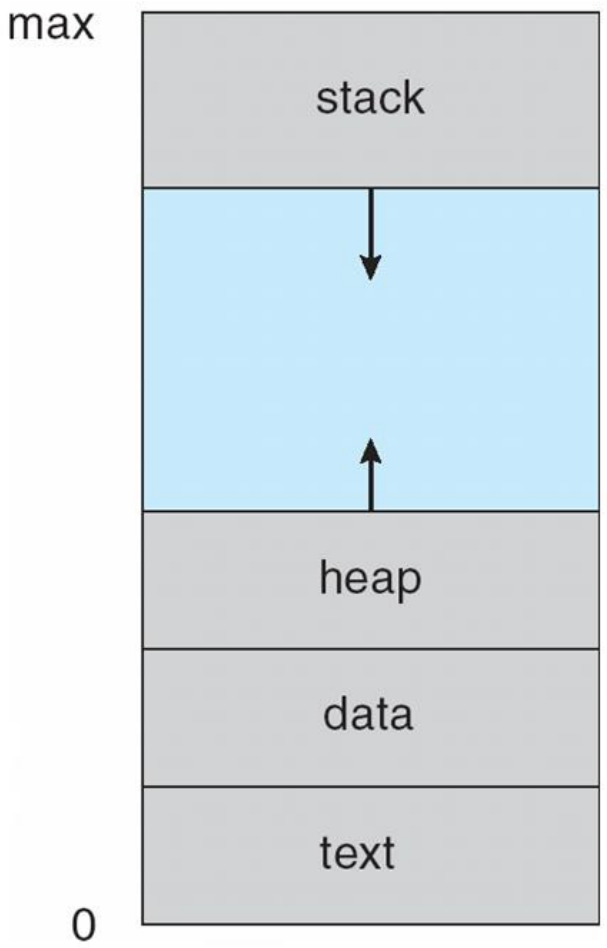
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

3.1 The Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs, processes** or **tasks**
- We use the terms ***job*** and ***process*** almost interchangeably
  - In conventional full-fledged operating systems, i.e. those running workstations or servers (not embedded systems running FreeRTOS):
    process = job = task
- **Process** – a program in execution; process execution generally (but not always) progresses in a sequential fashion.

The Process Concept – cont.

- Area occupied in main memory is divided into multiple sections:
  - The program code, also called **text section**
  - **Data section** containing global variables
    - Initialized sections (aka .data section), followed by
    - Uninitialized sections (aka .bss section)
  - **Stack** containing temporary data
    - Function parameters
    - Saved CPU registers (including return addresses)
    - Local variables
  - **Heap** containing memory dynamically allocated during run time. Grows opposite to the stack
    - e.g. using new/delete (in C++ or Java)
    - Malloc/free in C
- A process also occupies/uses CPU registers (not shown on Fig.).

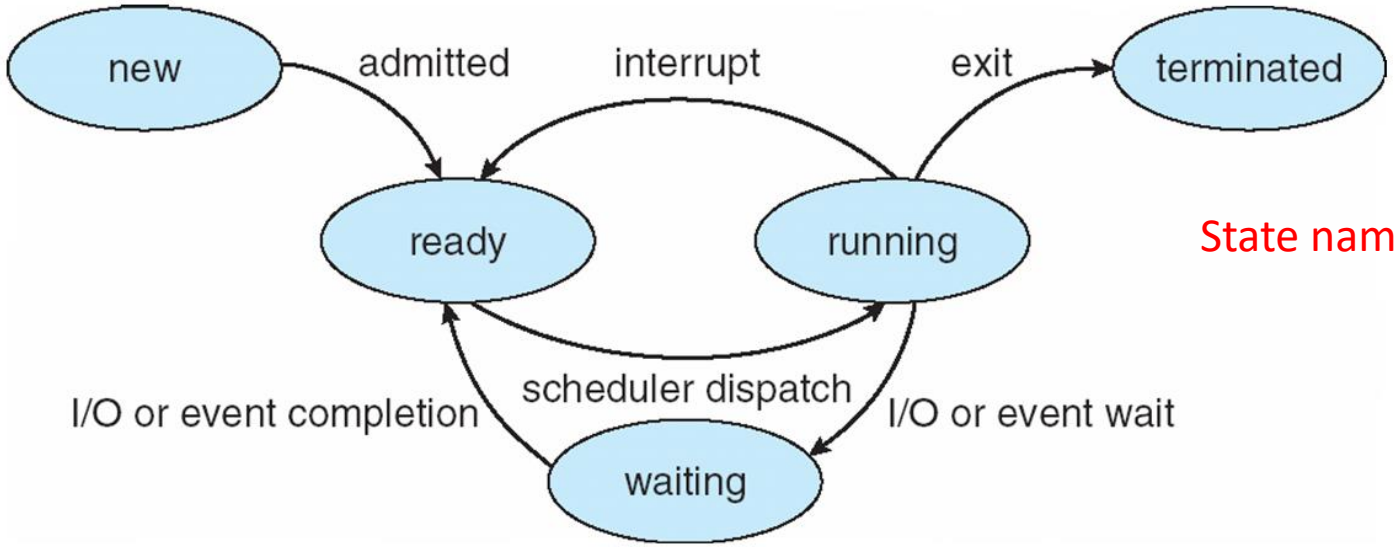


The Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when its executable file is loaded into main memory and is registered with the scheduler for execution.
- Execution of program is usually started via:
  - GUI mouse clicks
  - Command line entry of its name, etc.
- One program can be instantiated as several processes
  - Consider multiple users executing the same program

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



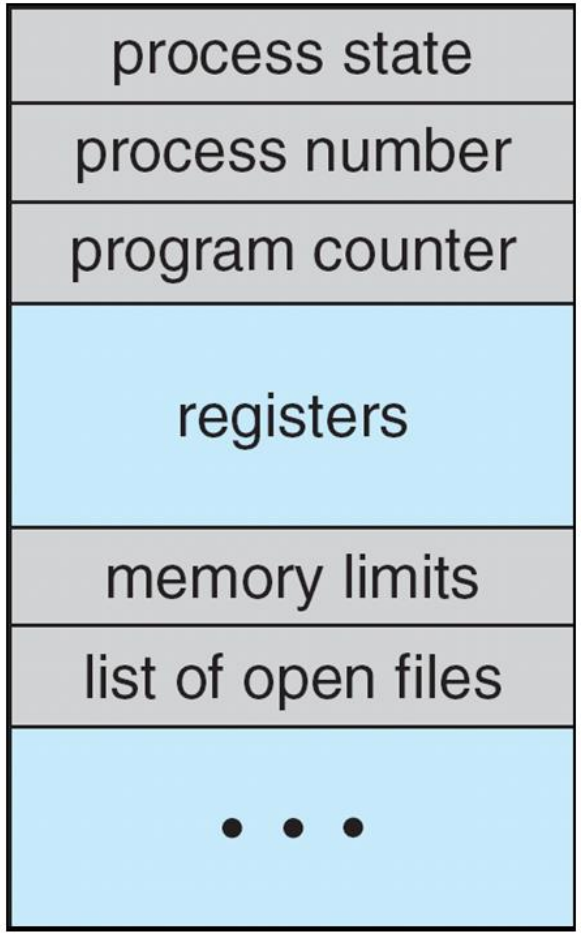
State names are generic

Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process ID
- Process state – running, waiting, etc
- CPU registers – contents of all process-centric registers **including the program counter** (which contains location of next instruction to execute)
- CPU scheduling information - priorities, scheduling queue pointers, etc.
- Memory-management information – memory allocated to the process (base and limit registers, page/segment tables, etc.)
- Accounting information – CPU and real time used, time limits
- Process numbers of parents or children
- Allocated resources – I/O devices allocated to process, list of open files



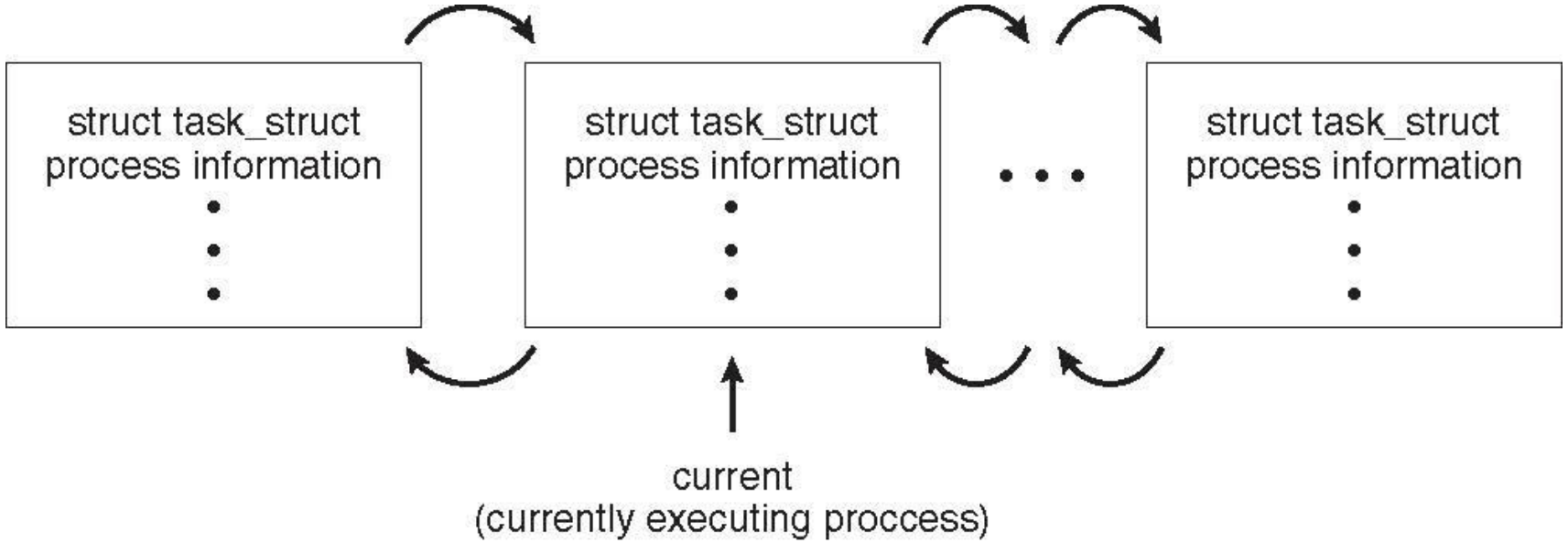
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple flows of executions == **threads**
    - Running concurrently OR
    - Running in parallel (in case of multiple processor cores)
- Must then have storage for thread details, and multiple program counters in PCB
- More about threads in the next chapter.

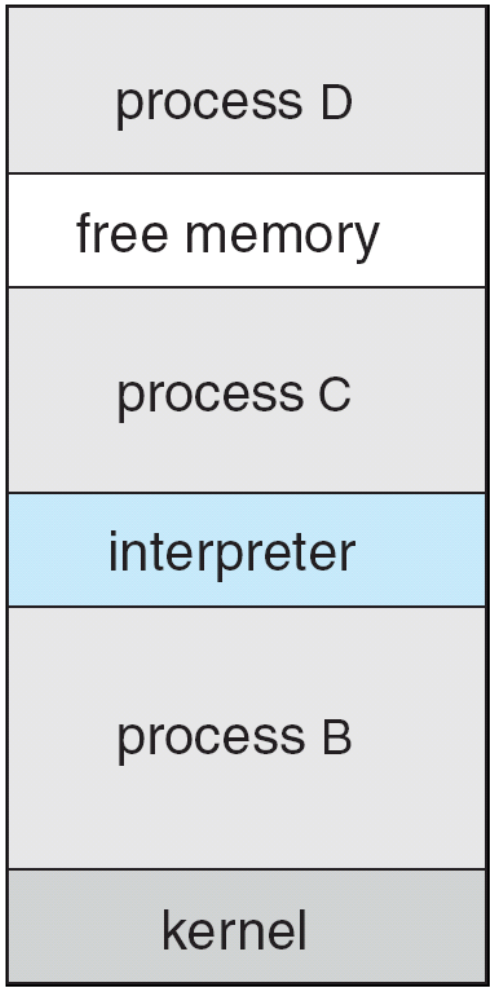
Process Representation in Linux

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 */
```



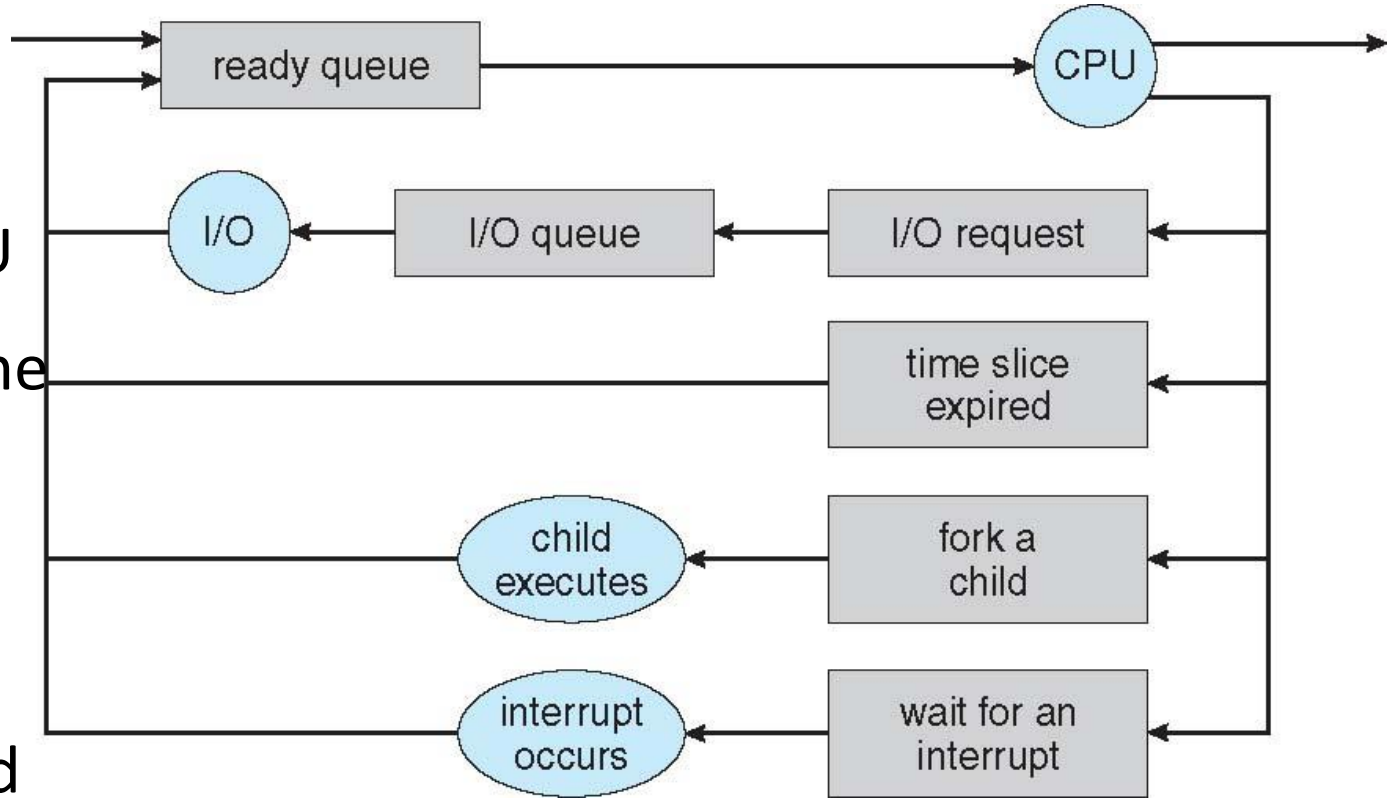
Where is the PCB for process C ?



main memory

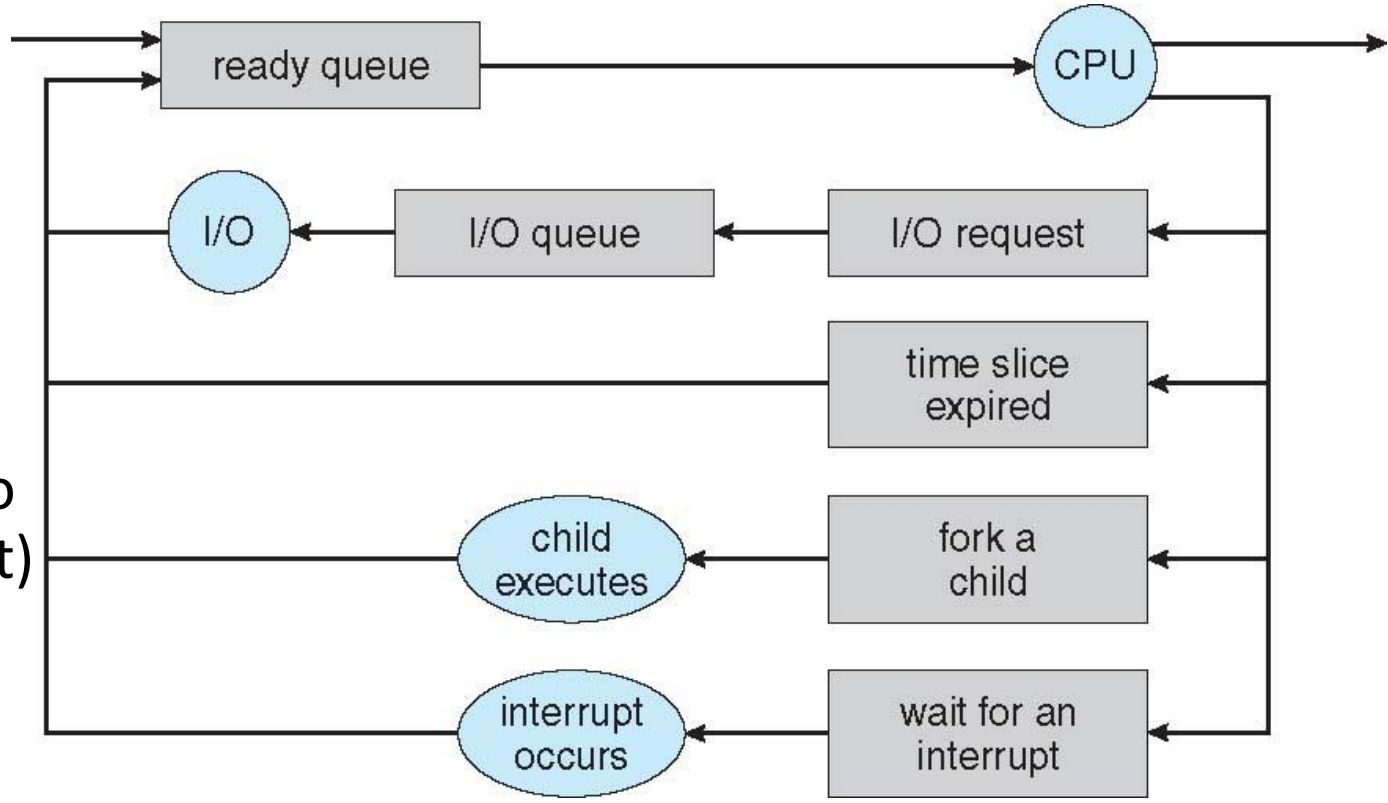
3.2 Process Scheduling

- In a single CPU system, the scheduler chooses only one process to run at a time, while the rest of available processes must wait till the CPU is free.
- The **kernel** must maximize CPU use and quickly switch processes onto the CPU for time sharing
- **Process scheduler** is a routine that selects among "ready" processes for next execution on CPU
- Often, the terms scheduler and process manager are used interchangeably



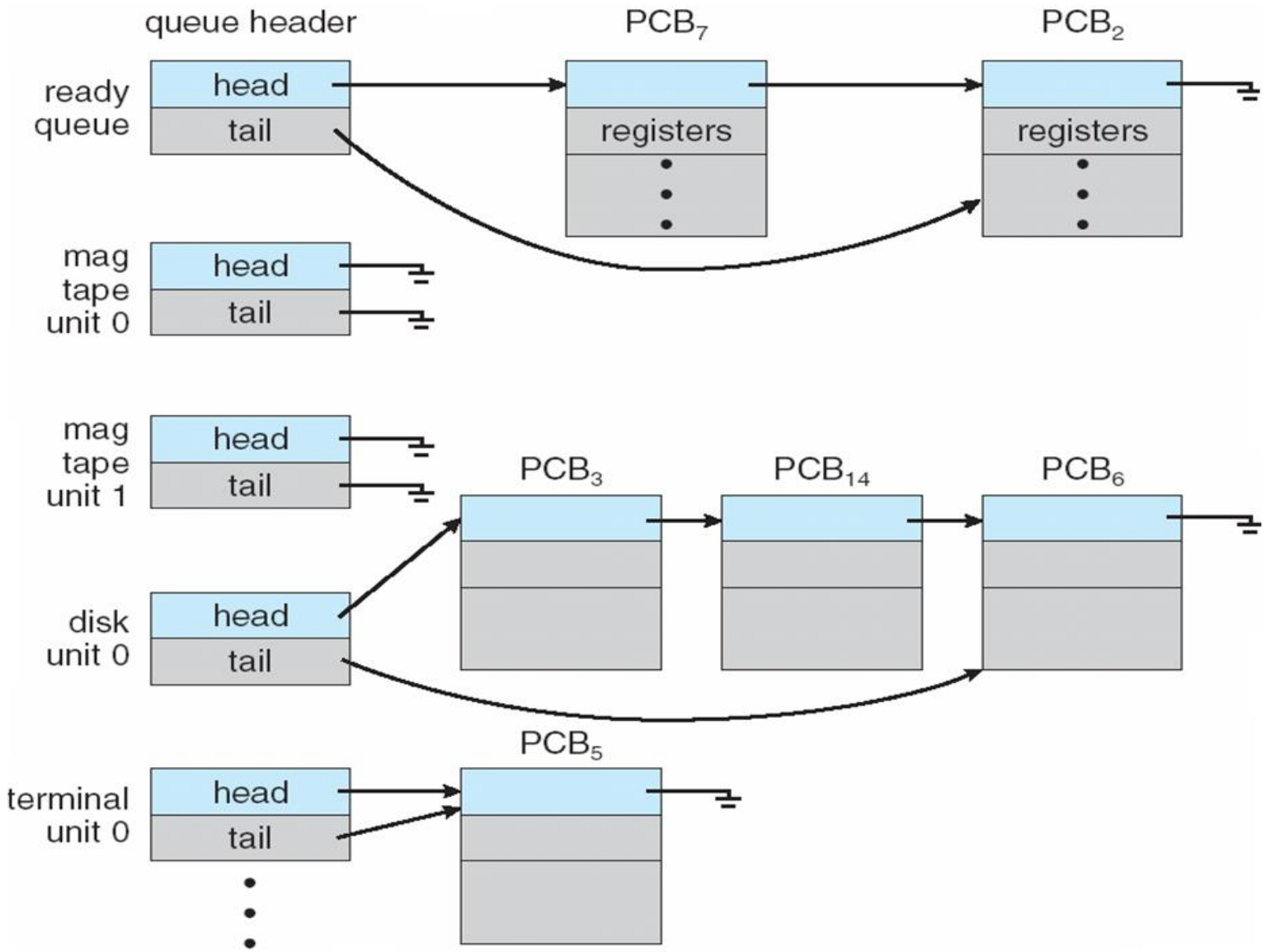
Queueing diagram represents queues, resources, flows

- **Process manager** 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 (stored as a linked list)
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



Queueing diagram represents queues, resources, flows

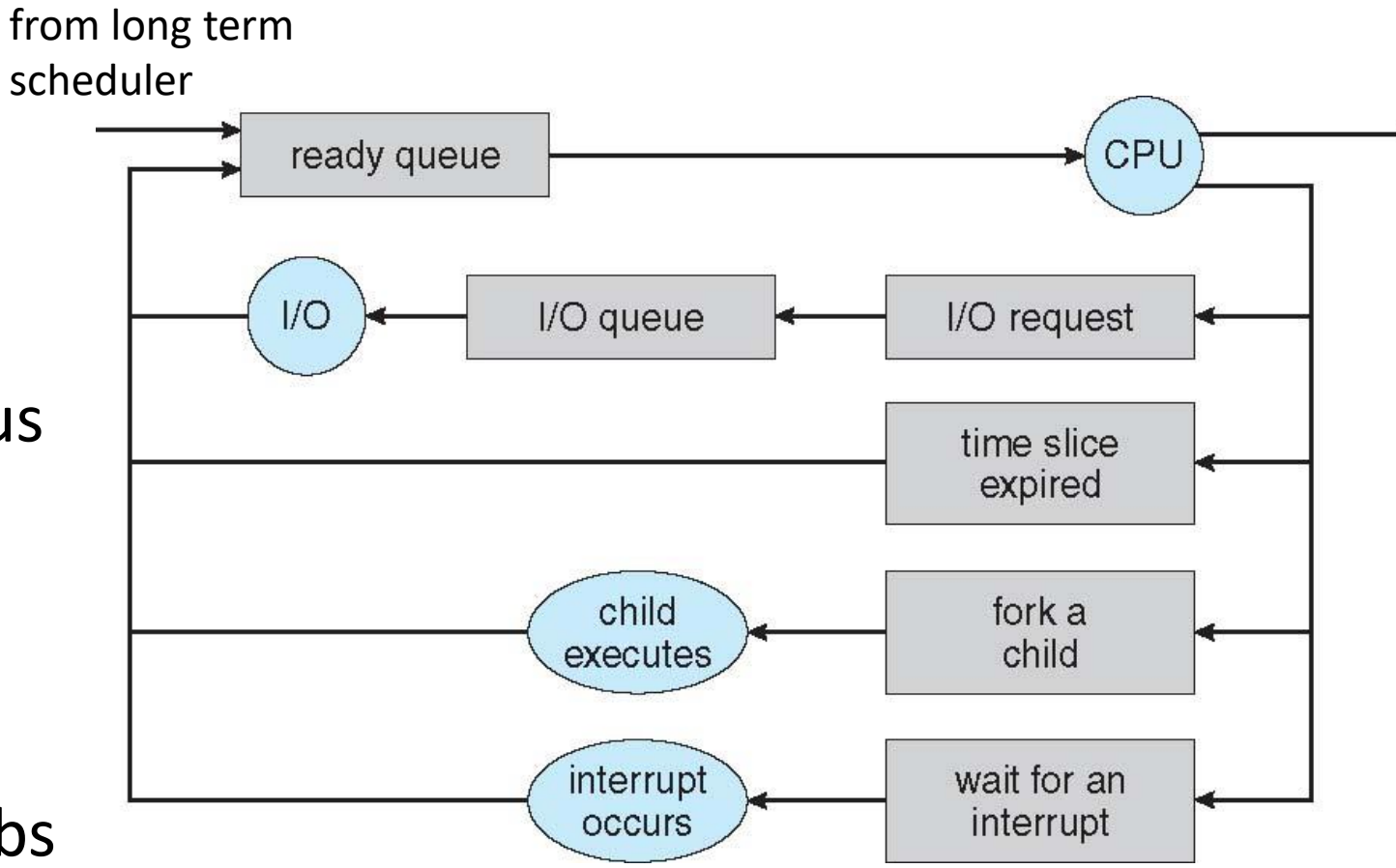
Ready Queue And Various I/O Device Queues



Schedulers

Long-term scheduler

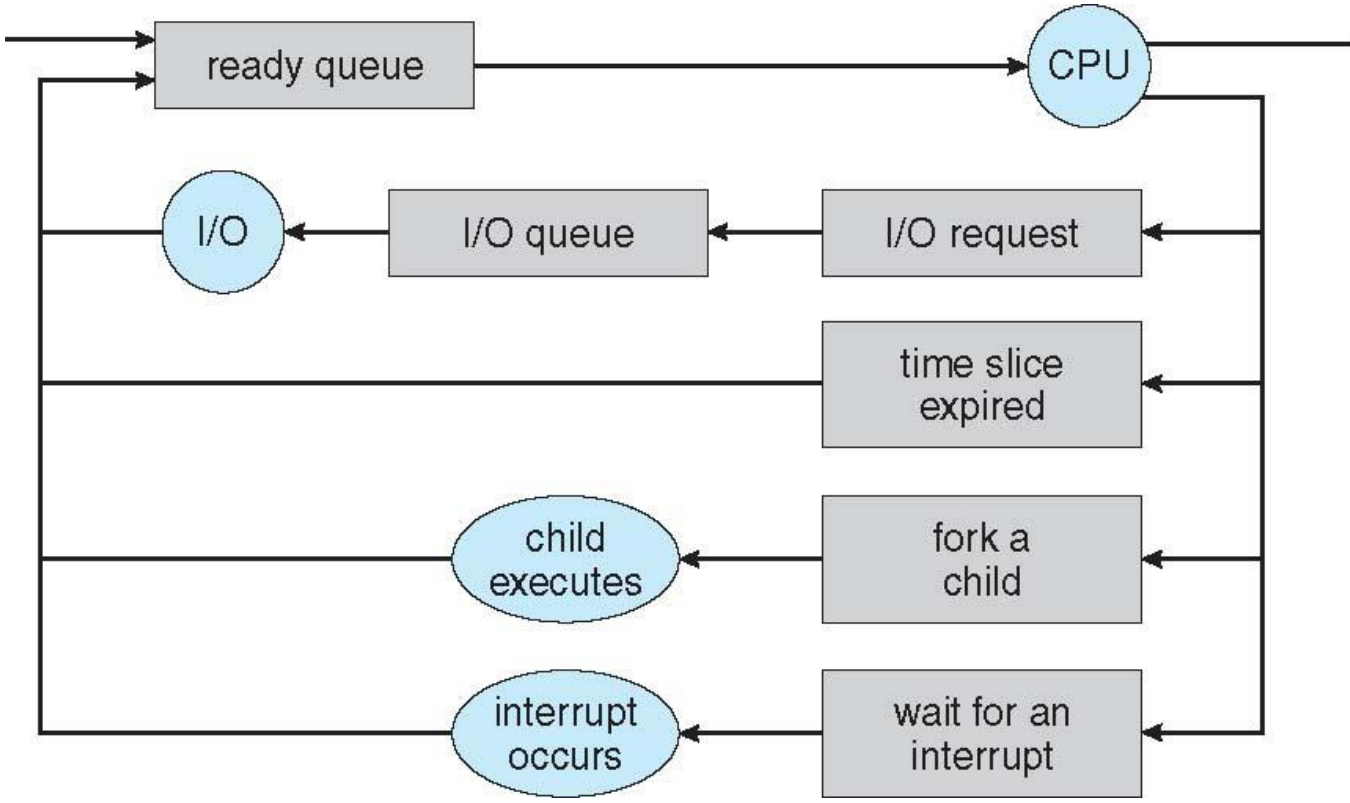
- Occurs in batch systems
- Invoked when a process exits, i.e. infrequently, in seconds or minutes ⇒ (thus is allowed to be slow)
- Selects which processes should be admitted, i.e. brought into the ready queue, from the pool of jobs waiting.
- Controls the **degree of multiprogramming**



Queueing diagram represents queues, resources, flows

Schedulers – cont.

- **Short-term scheduler** (or just **scheduler**, the one we already discussed) – selects which process should be executed next and allocates a CPU for it:
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) and thus must be fast.



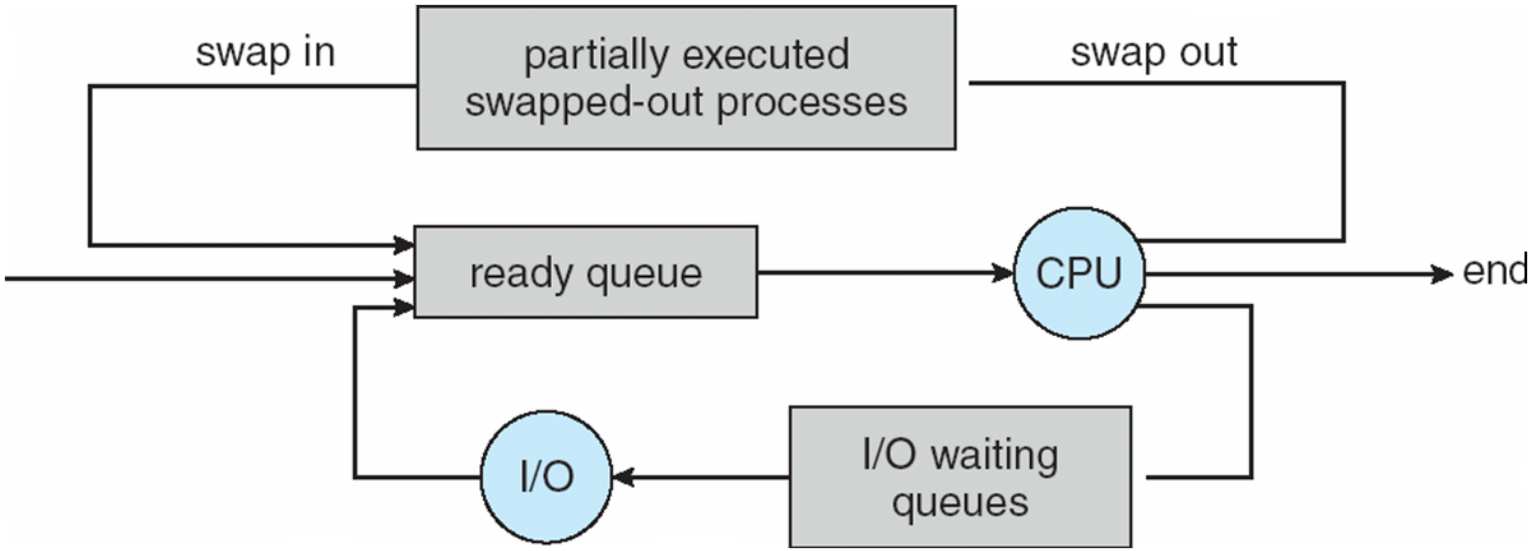
Queueing diagram represents queues, resources, flows

Schedulers – cont.

- 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*** of I/O-bound and CPU-bound processes.
- Unix and windows do not implement long term scheduling, and thus their stability (which relies on degree of multi-programming) relies on the user's behavior, i.e. the user won't try to run more processes if the system is slowing down and becoming irresponsive.

Schedulers – cont.

- **Medium-term scheduler (aka swapping scheduler)** can be added if degree of multiple programming needs to decrease, or if the process mix (I/O-bound vs CPU-bound) needs to be adapted.
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

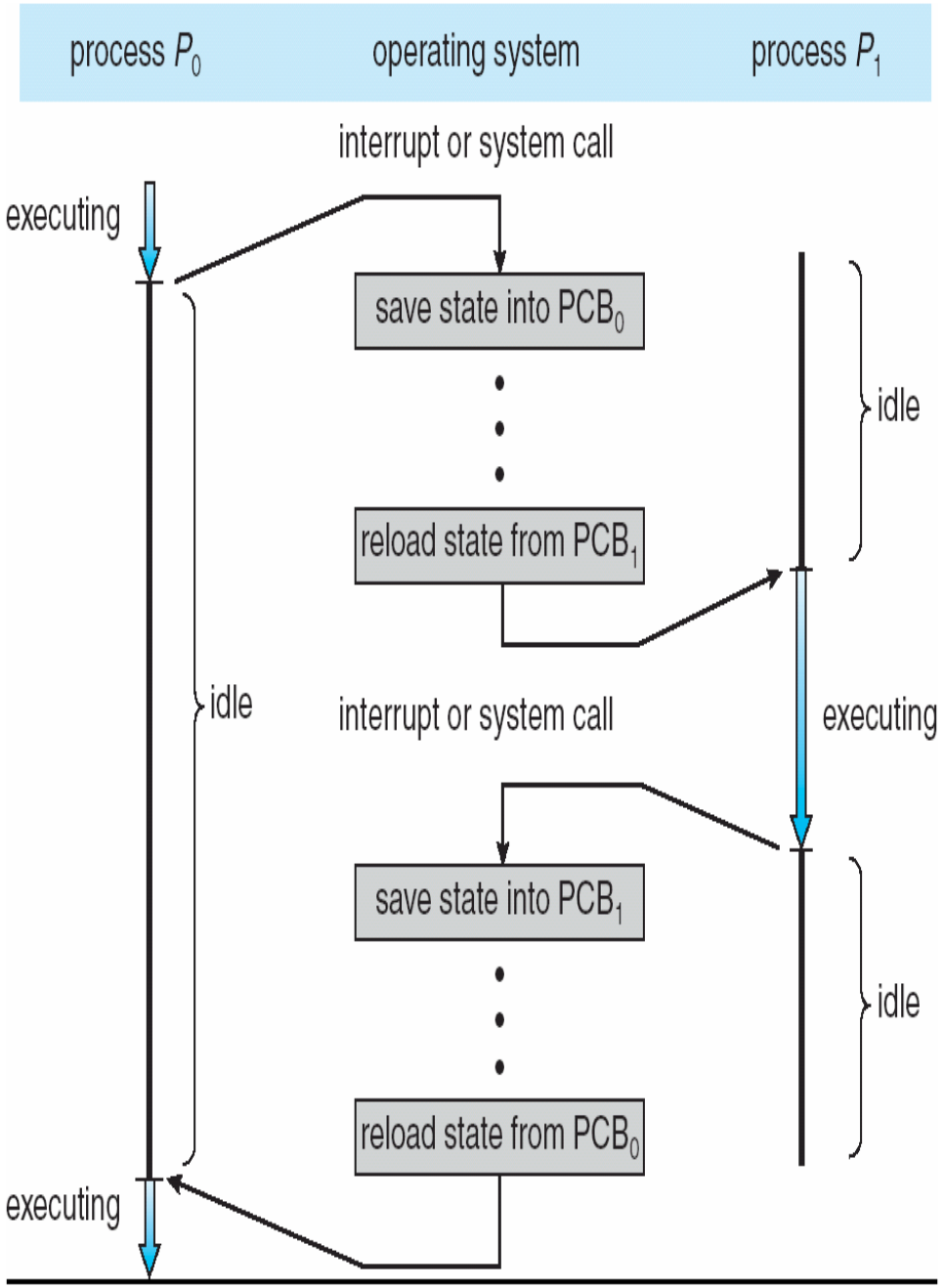


Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allowed only one user process to run, while all others were suspended (many OS processes ran concurrently).
- 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 were limited to apps that are either:
    - Running a single, short task,
    - Receiving notification of events (e.g. new mail message) OR
    - Long-running background tasks like audio playback
- Android runs foreground and background processes, with fewer limits
  - A background process uses a **service** to perform tasks.
  - A service is a separate application component that runs on behalf of the background task.
  - A service can keep running even if background process is suspended.
  - A service has no user interface and a small memory footprint.

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**
- The **context** of a process is represented in the PCB.
- It involves a state save, then a state restore.



Context Switch

- 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
  - The address space needs to be preserved as part of the PCB. How it is preserved and what amount of work is needed depends on the memory management method of the OS.

3.3 Operations on Processes

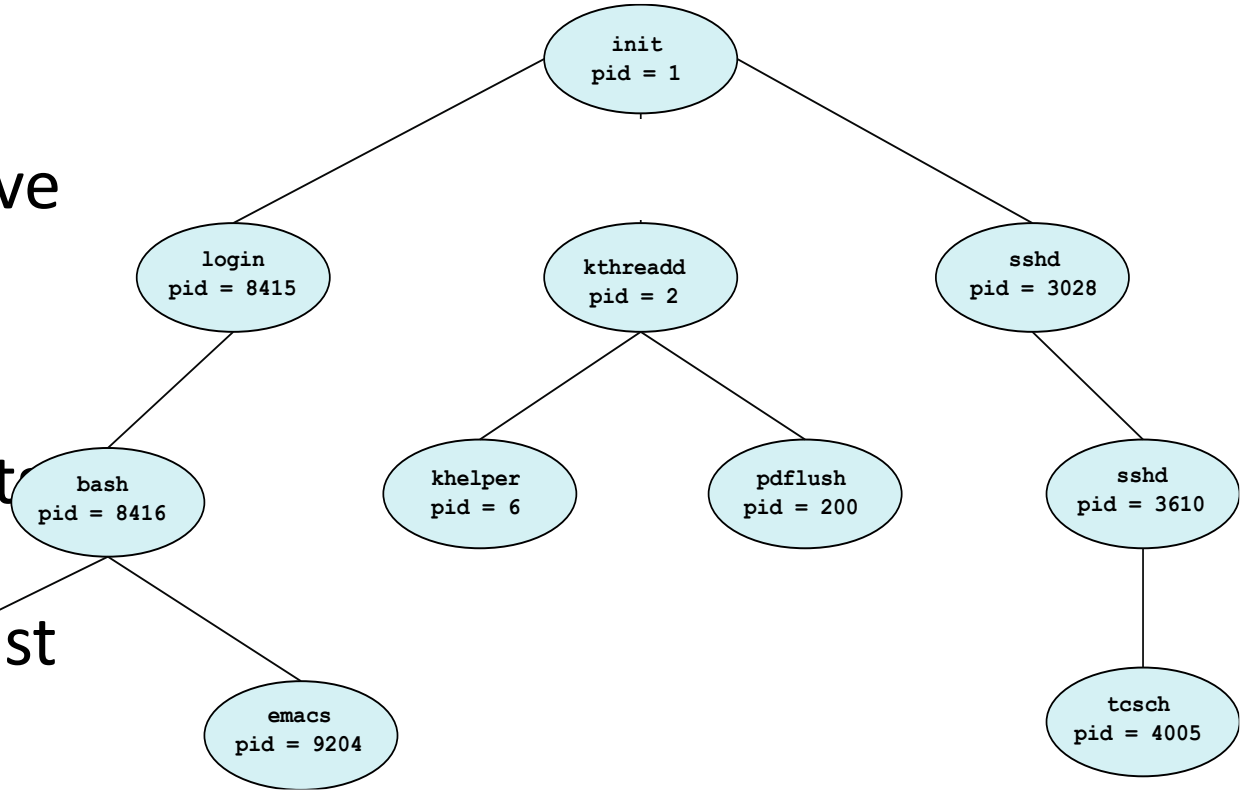
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- A **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

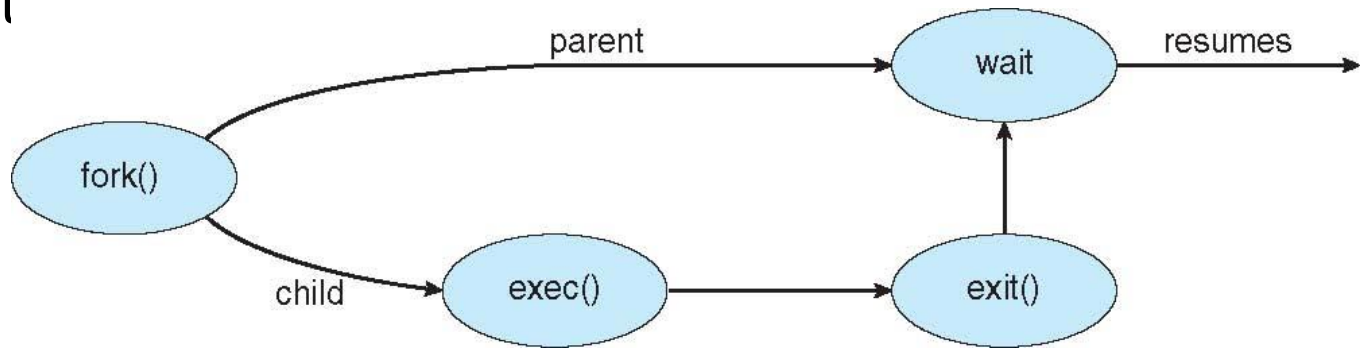
A Tree of Processes in Linux

- Kernel-space processes usually have a pid <1000
- User-space processes usually have a pid >=1000
- pid<=0 is not valid
- init and kthreadd have no parent (parent's PID, PPID = 0), because they are created by the kernel just after the system boots.
- init is the mother of all user processes, while kthreadd is the mother of all kernel-space processes



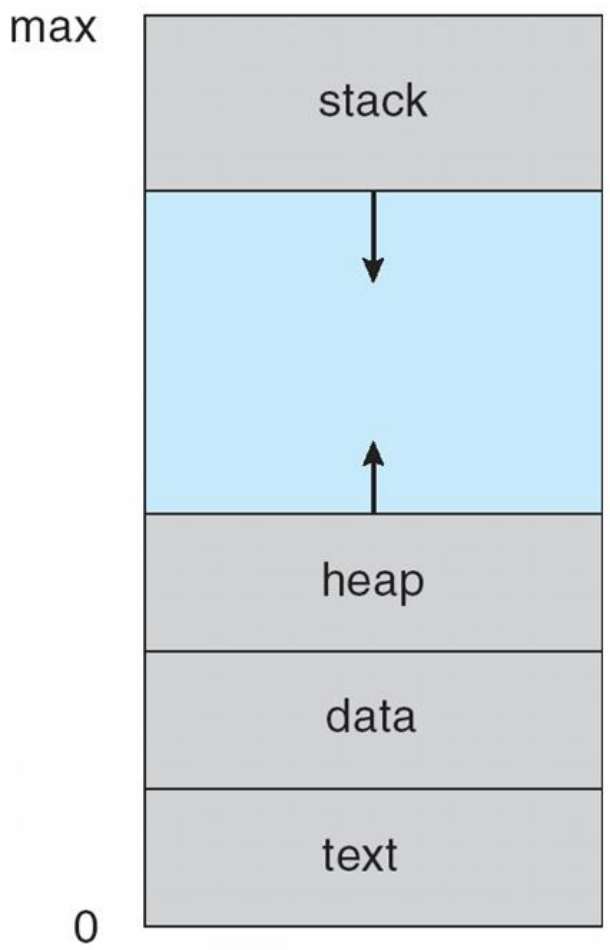
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates (aka spawns) a new process.
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program, i.e. loads new program from disk.
  - **`wait()`** system call allows the parent process to wait for the child to exit



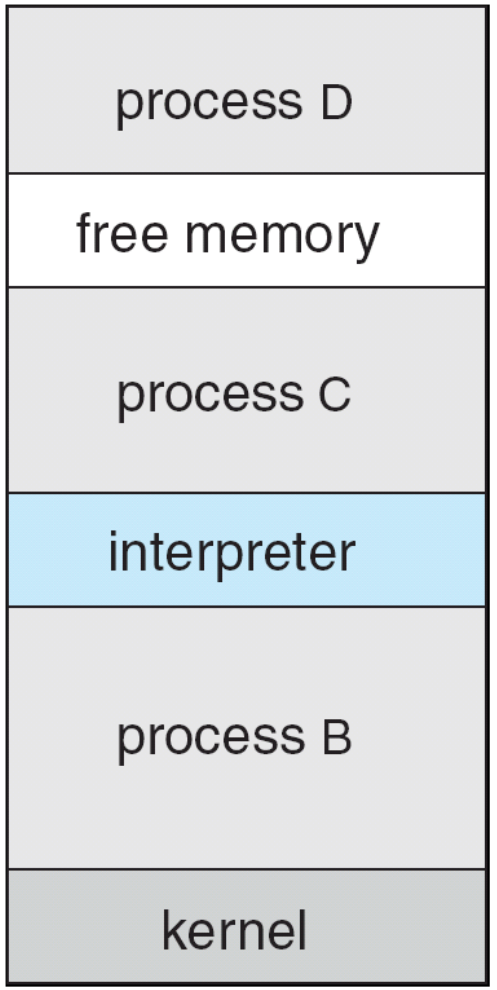
Creating a separate process in Unix/Linux

```c
#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 in Unix/Linux - cont.

- When using fork():
  - The parent executes and then invokes the fork() function within the API library which invokes the system call (via a trap)
  - The kernel creates a child process that has the exact copy of the parent's address space contents
  - Upon return from the system call, both the parent and the child processes resume at the instruction following the fork() call.
  - Generally, each process has its own **address space in memory**. Unless explicitly requested, the kernel ensures that no process infringes on the address space of another process



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, appearance and handles to standard input/output.
- PROCESS_INFORMATION contains a handle and the ID of the new process and IDs of its threads.

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call. This causes:
  - Returns status data from child to parent (via the parent calling **`wait()`**)
  - Process' resources are deallocated by operating system
  - In Linux, `exit` usually takes one parameter indicating an error if non-zero.
  - `exit` is called inherently upon return from the main routine of a program.
- Parent may terminate the execution of children processes using the **`abort()`** system call (`TerminateProcess()` in windows). 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 a child to exist if its parent has terminated. Hence, in such systems, if a process terminates, then the OS terminates all its children.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.

  (this is not the case in Linux)
- 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);
```

- When a process exits, its resources are deallocated
  - except its entry in the process table (containing the exit status).
  - Only after the parent invokes the wait() function which reads that status that its entry in the process table is released.
  - Till then, the terminated child process is a **zombie.**
- If a parent terminated before the child (i.e. without invoking **`wait)`**, the running child process is an **orphan** (**if allowed by OS,** e.g. Linux) and its new parent becomes the init process (whose PID is 1).
- The init process periodically invokes **`wait`** in order to release orphan zombie processes.