Chapter 3: Processes

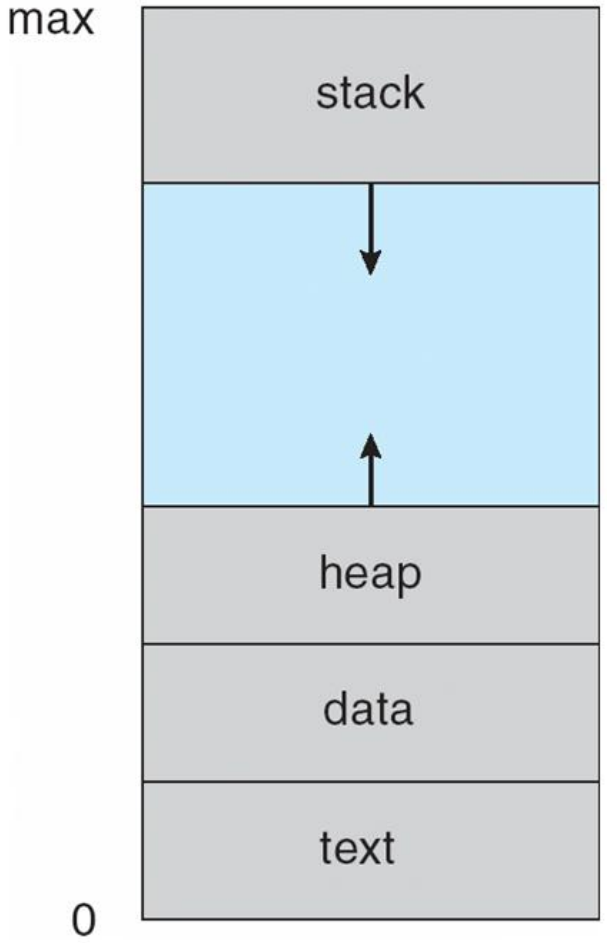
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

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, task*** 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 mostly progresses in a sequential fashion, but sometimes branches and calls occur.

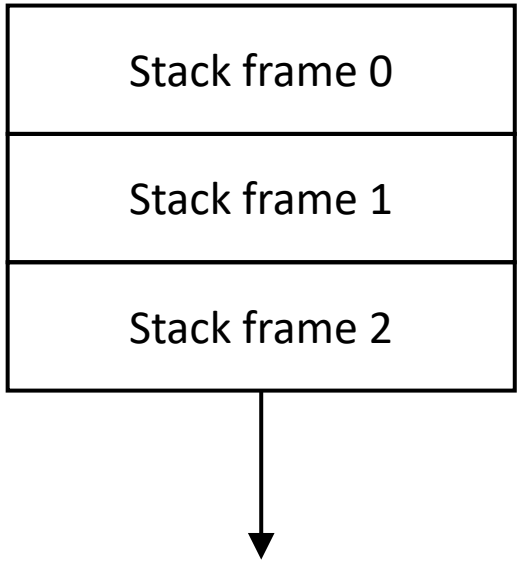
The Process Concept – cont.

- Area occupied in main memory is divided into multiple sections:
  - The program code, also called **text segment**
  - **Data segment** containing global variables
    - Initialized sections (aka .data section), followed by
    - Uninitialized sections (aka .bss section)
  - **Stack:** The **call stack** contains temporary data:
    - Saved CPU registers (including return addresses)
    - Function parameters
    - 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.).

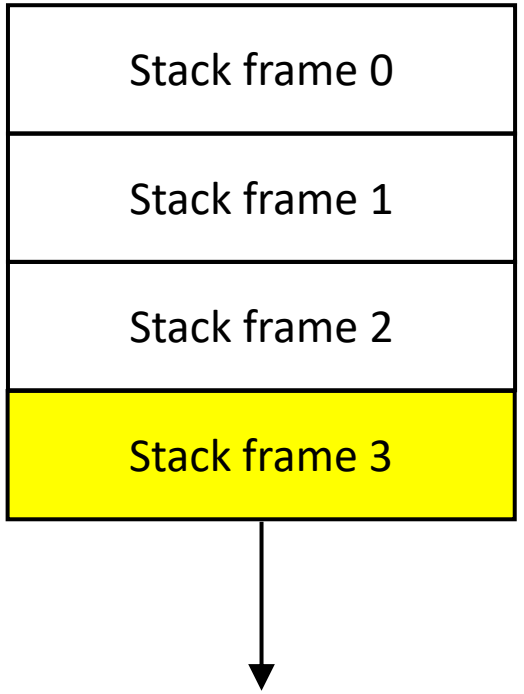


The Call Stack - example

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```

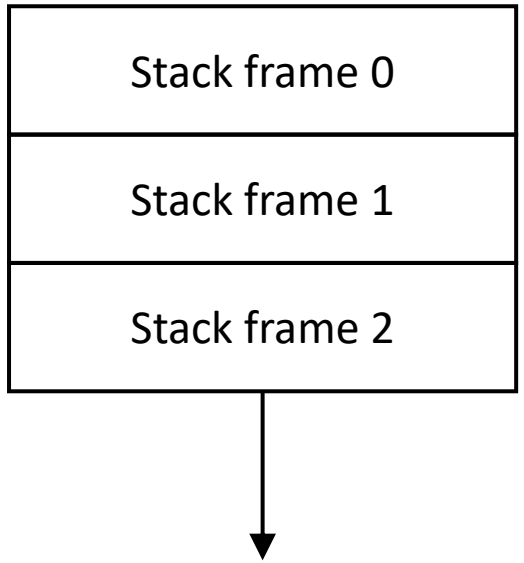


Before calling myfunc()



While myfunc() is executing:

- CPU registers are saved into frame 3
- Parameter x is pushed into stack frame 3
- Local variable y is allocated in frame 3



After myfunc() executed:

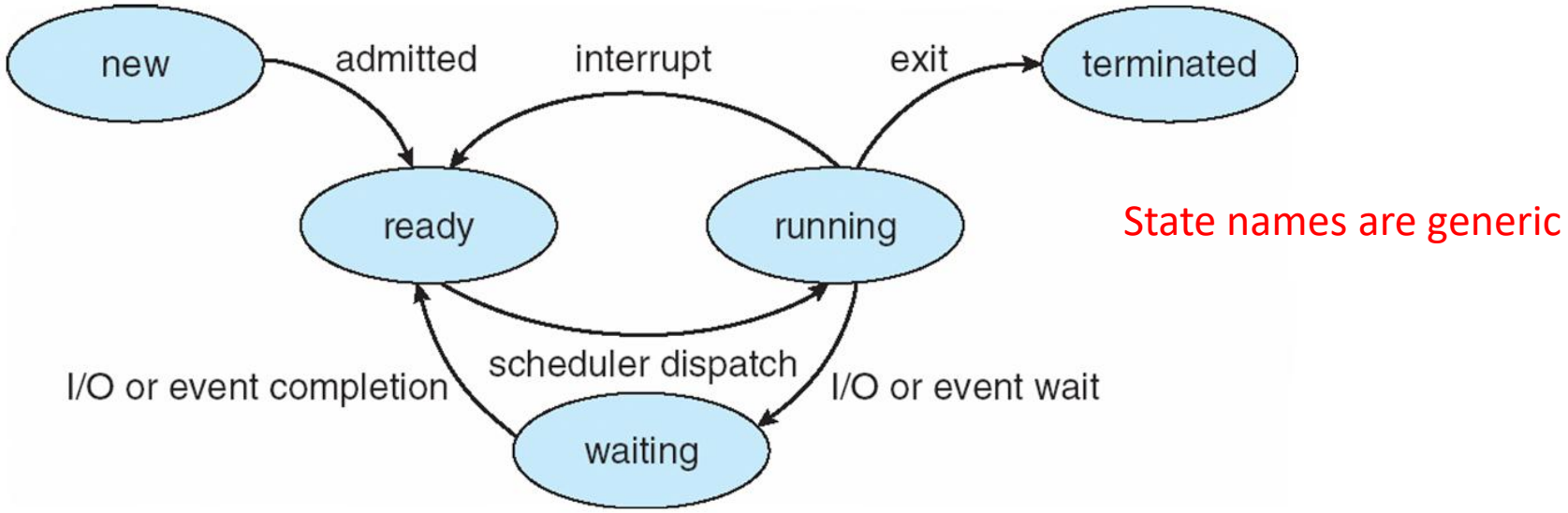
- CPU registers are restored from frame 3
- Parameter x is deallocated
- Local variable y deallocated
- Stack frame is no more

The Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active.*** A program becomes a 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



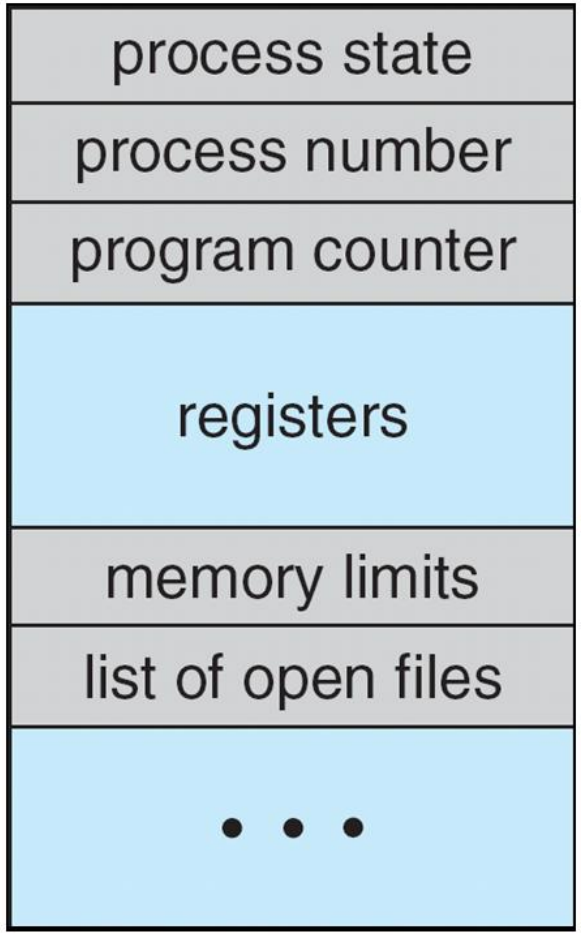
State names are generic

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



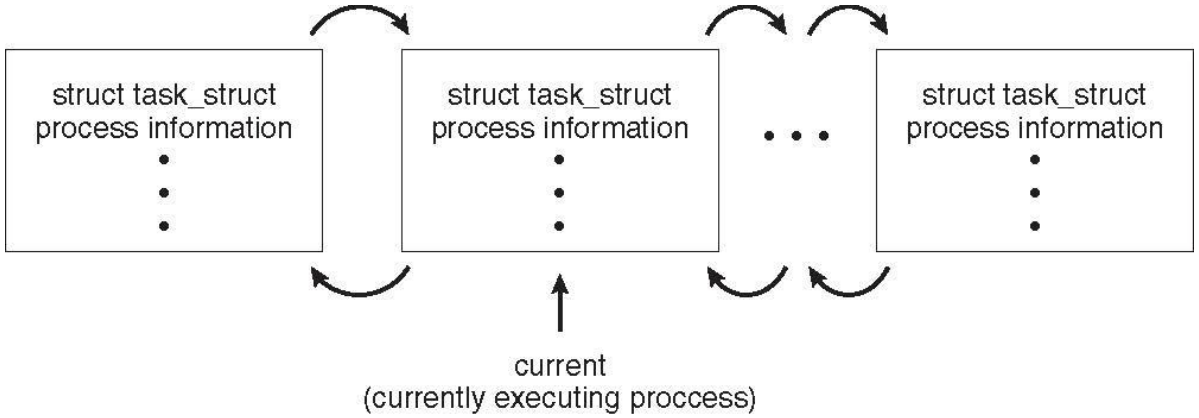
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple sequences of execution == **multiple 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.

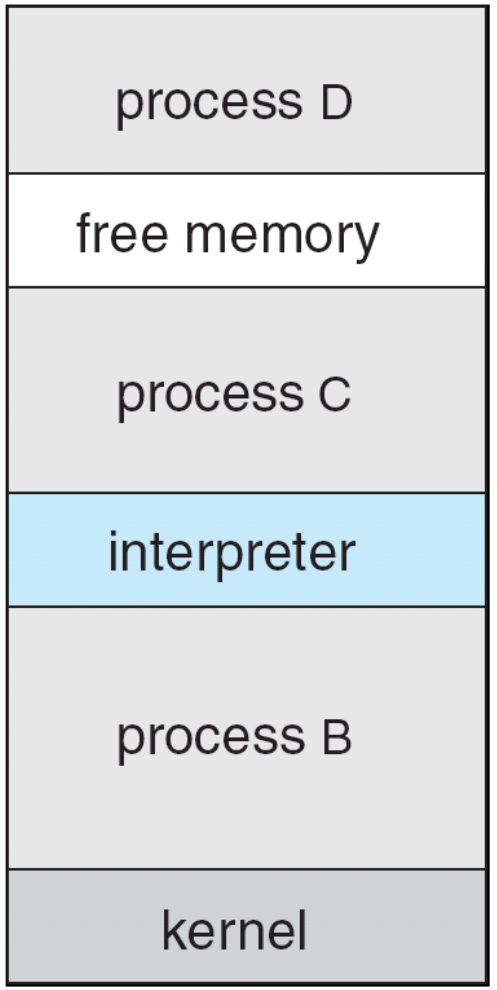
Process Representation in Linux

Represented by the C structure `task_struct`

```
pid_t pid; /* process identifier */
long state; /* state of the process */
long nice;

unsigned long policy;

struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct task_struct *next_task, *prev_task;

struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



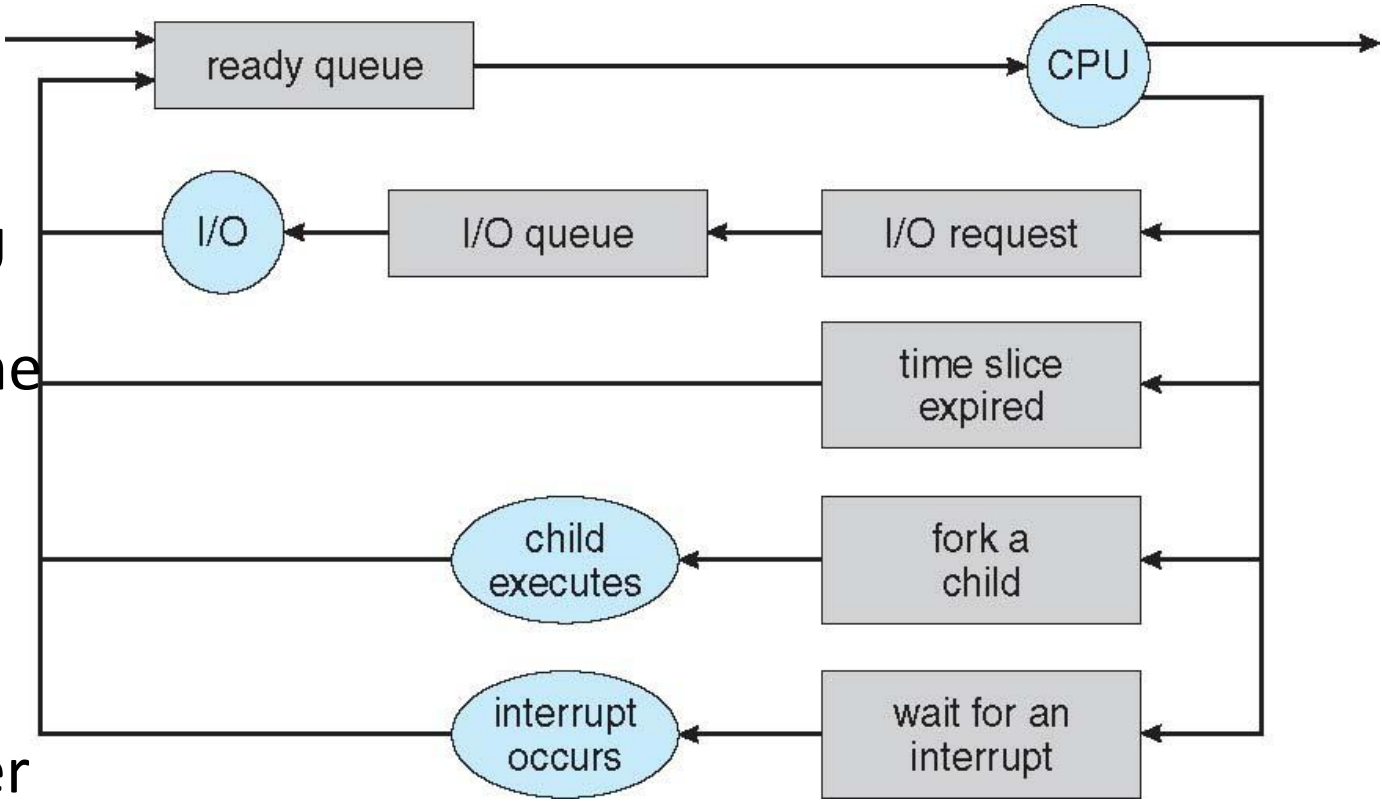
Where is the PCB for process C ?



main memory

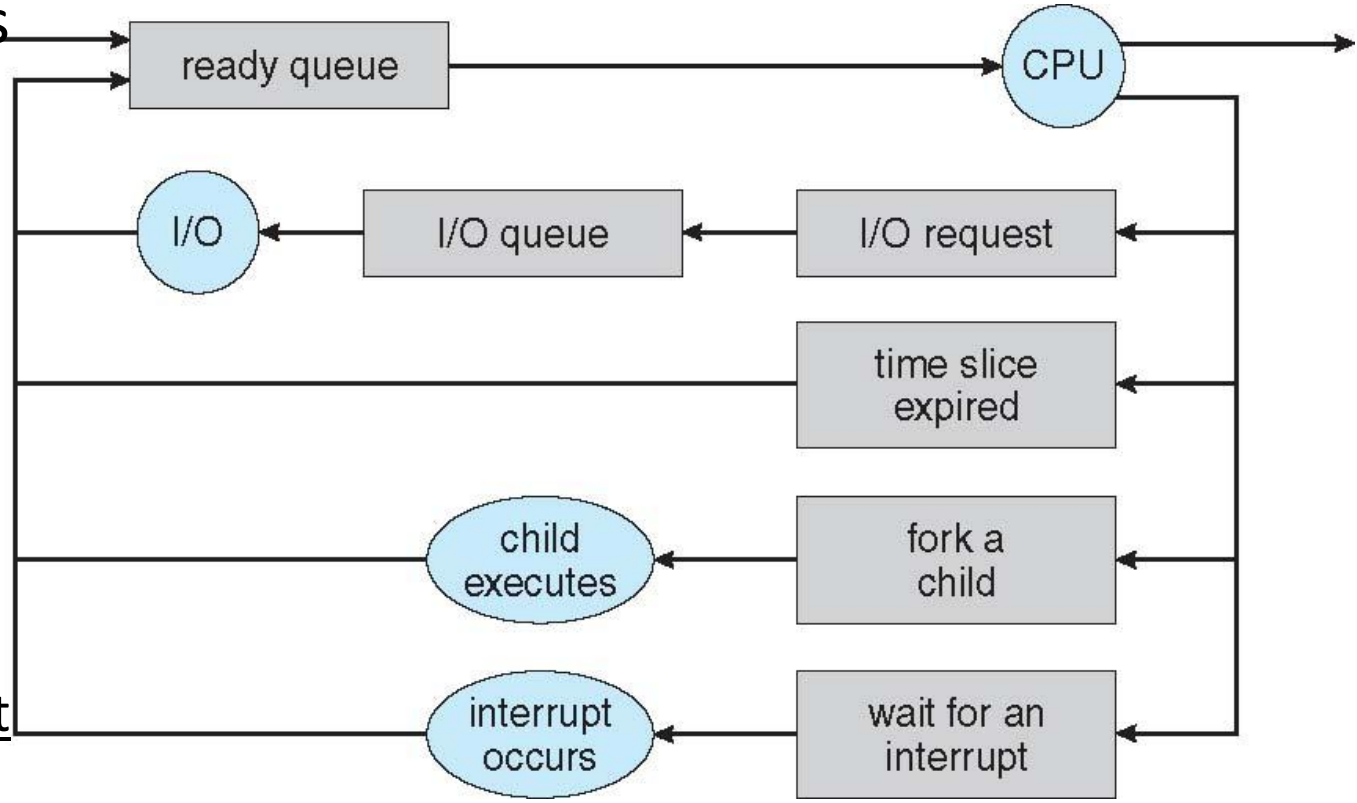
3.2 Process Scheduling

- 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 set of routines that selects among "**ready**" processes for next execution on CPU
- Sometimes the terms scheduler and process manager are used interchangeably



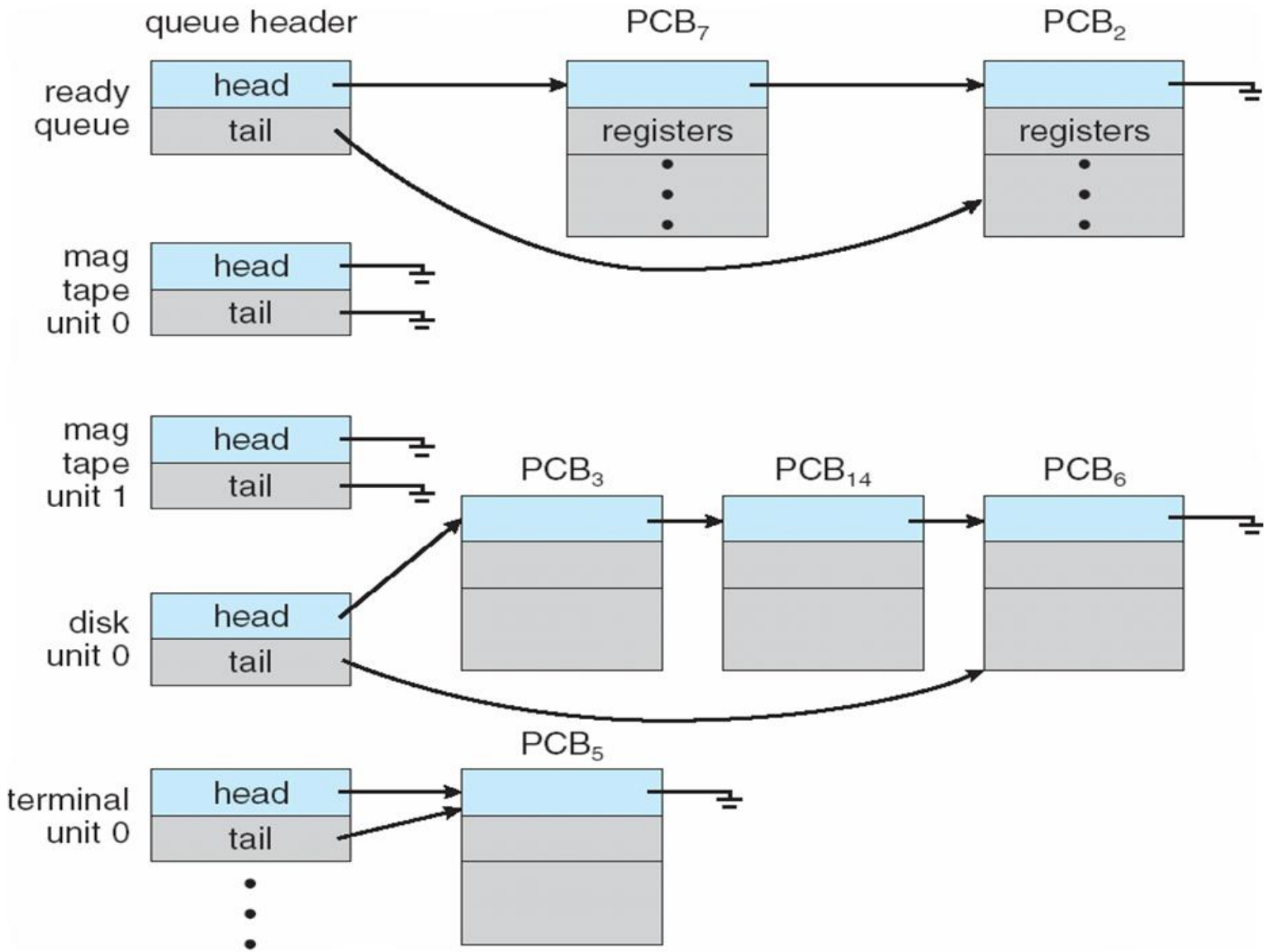
Queueing diagram represents queues, resources, flows

- **Process manager** maintains **scheduling queues** of processes (also contains the scheduler)
  - **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)
  - **Wait queues** – set of all processes blocked/waiting on an I/O device, a resource or an event
    - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



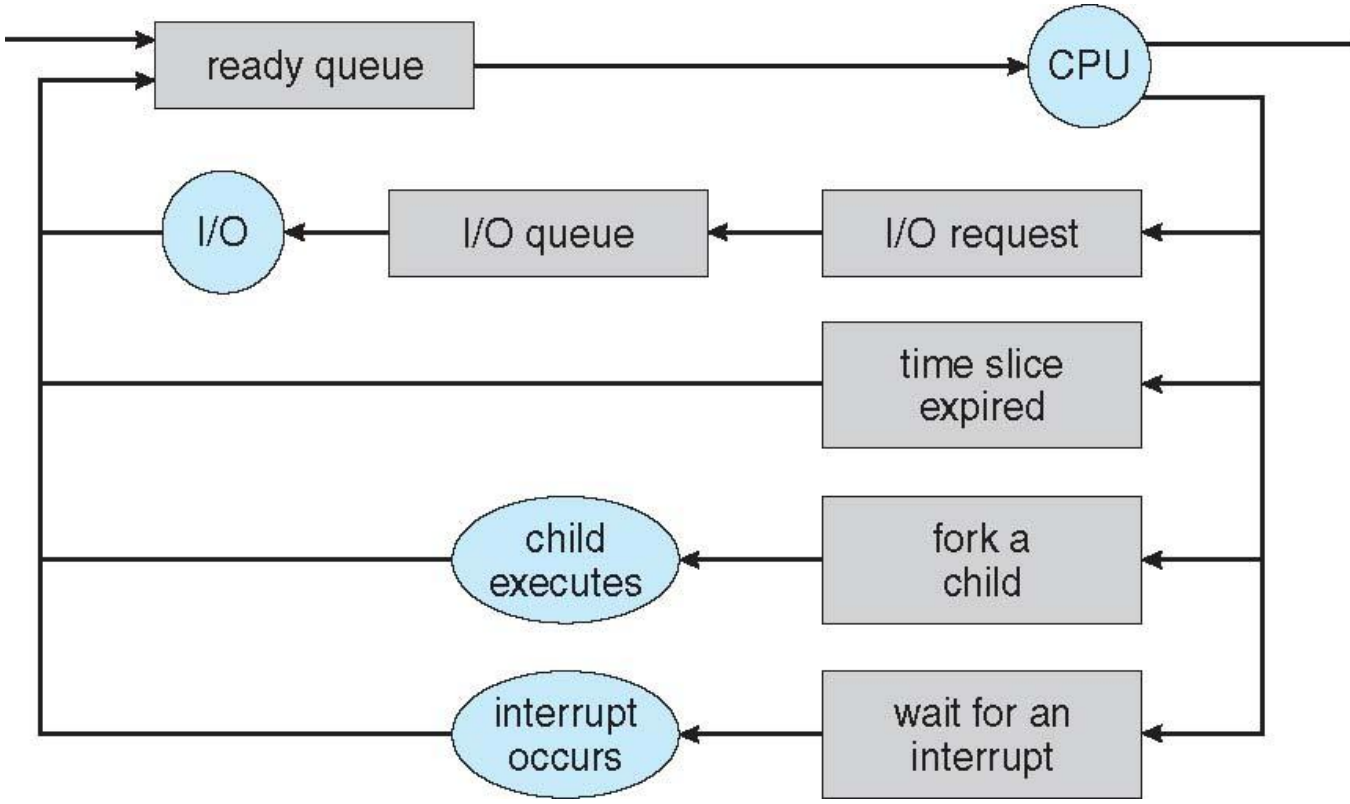
Queueing diagram represents queues, resources, flows

Ready Queue And Various I/O Device Queues



Schedulers

- **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.

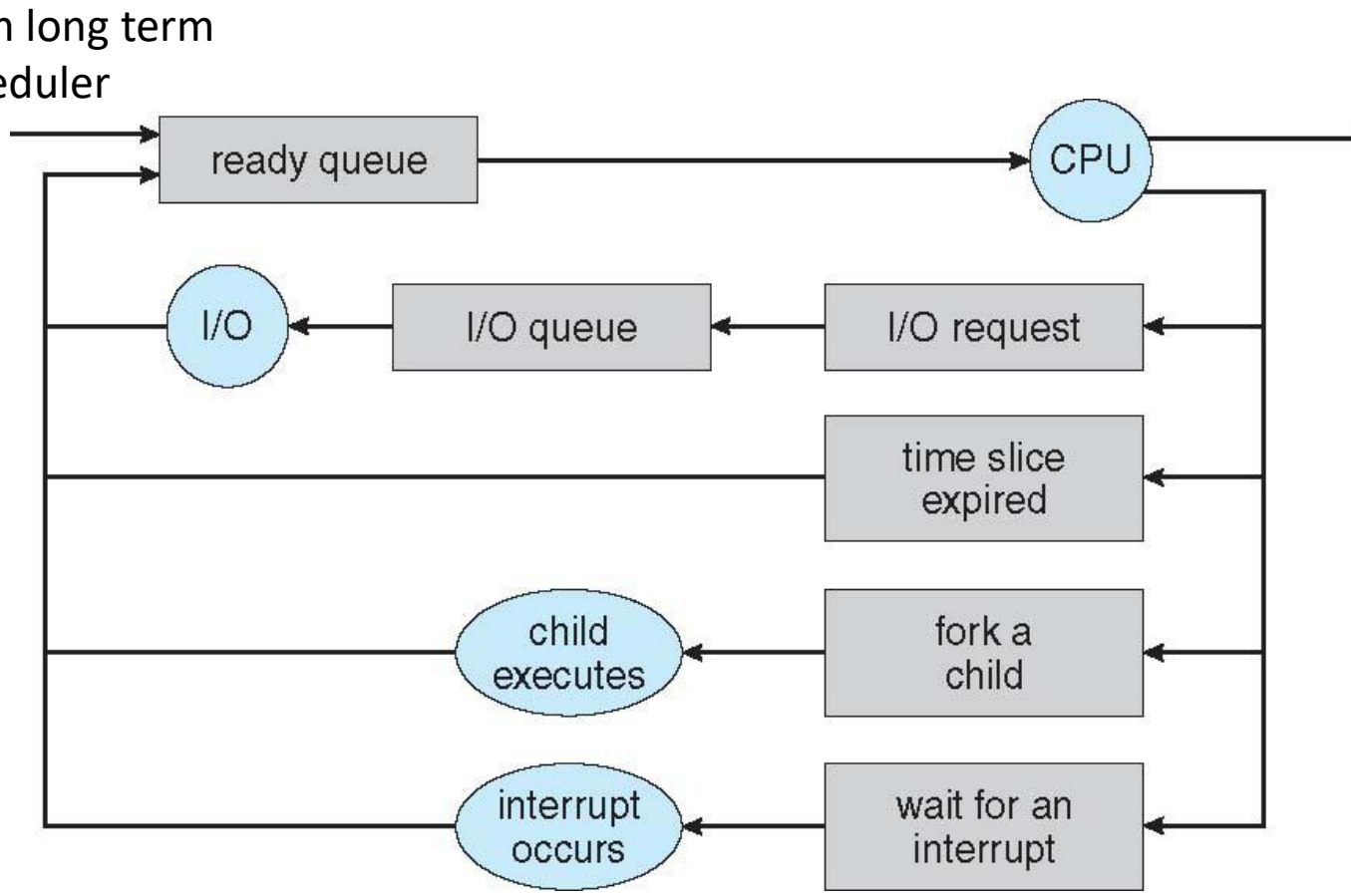


Queueing diagram represents queues, resources, flows

Schedulers – cont.

Long-term scheduler

- Occurs in batch systems
- Selects which processes should be admitted, i.e. brought into the ready queue, from the pool of jobs waiting.
- Invoked infrequently, in seconds or minutes ⇒ (thus is allowed to be slow)
  - Invoked when a process is attempting to be admitted.
  - May be invoked when a process exits (to decide on processes pending admission).
- Controls the **degree of multiprogramming** which is the number of processes admitted for scheduling (i.e. in short-term and medium-term schedulers)



Queueing diagram represents queues, resources, flows

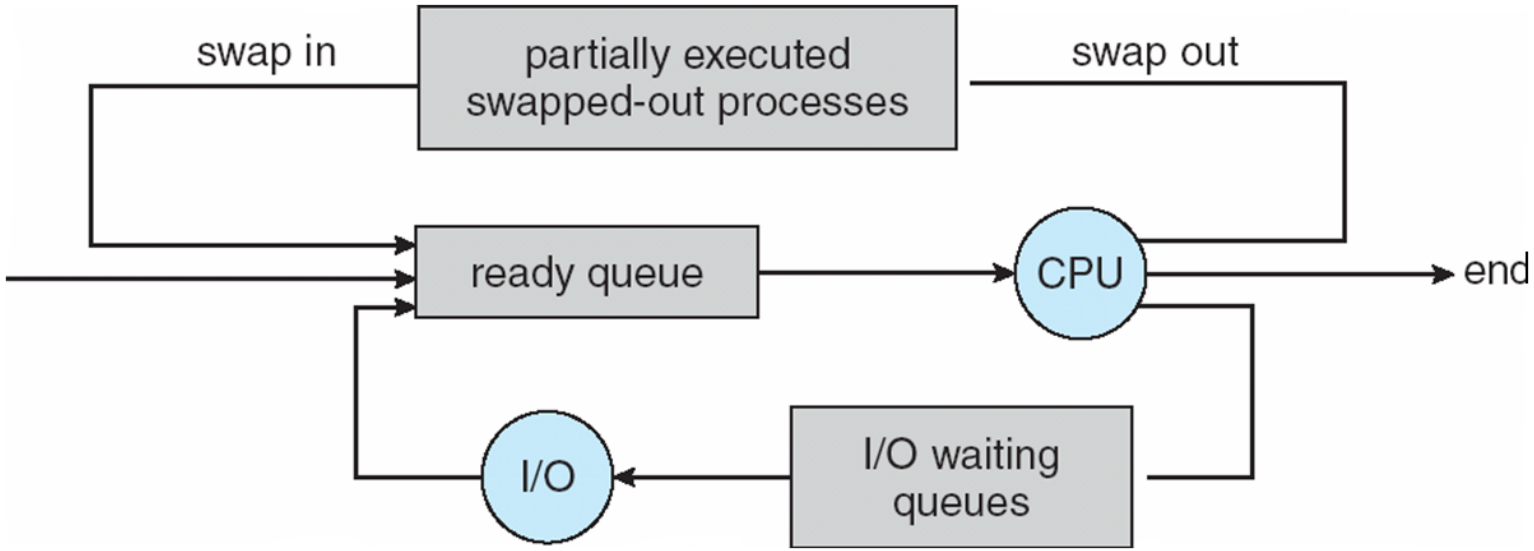
Schedulers – cont.

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than CPU 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.

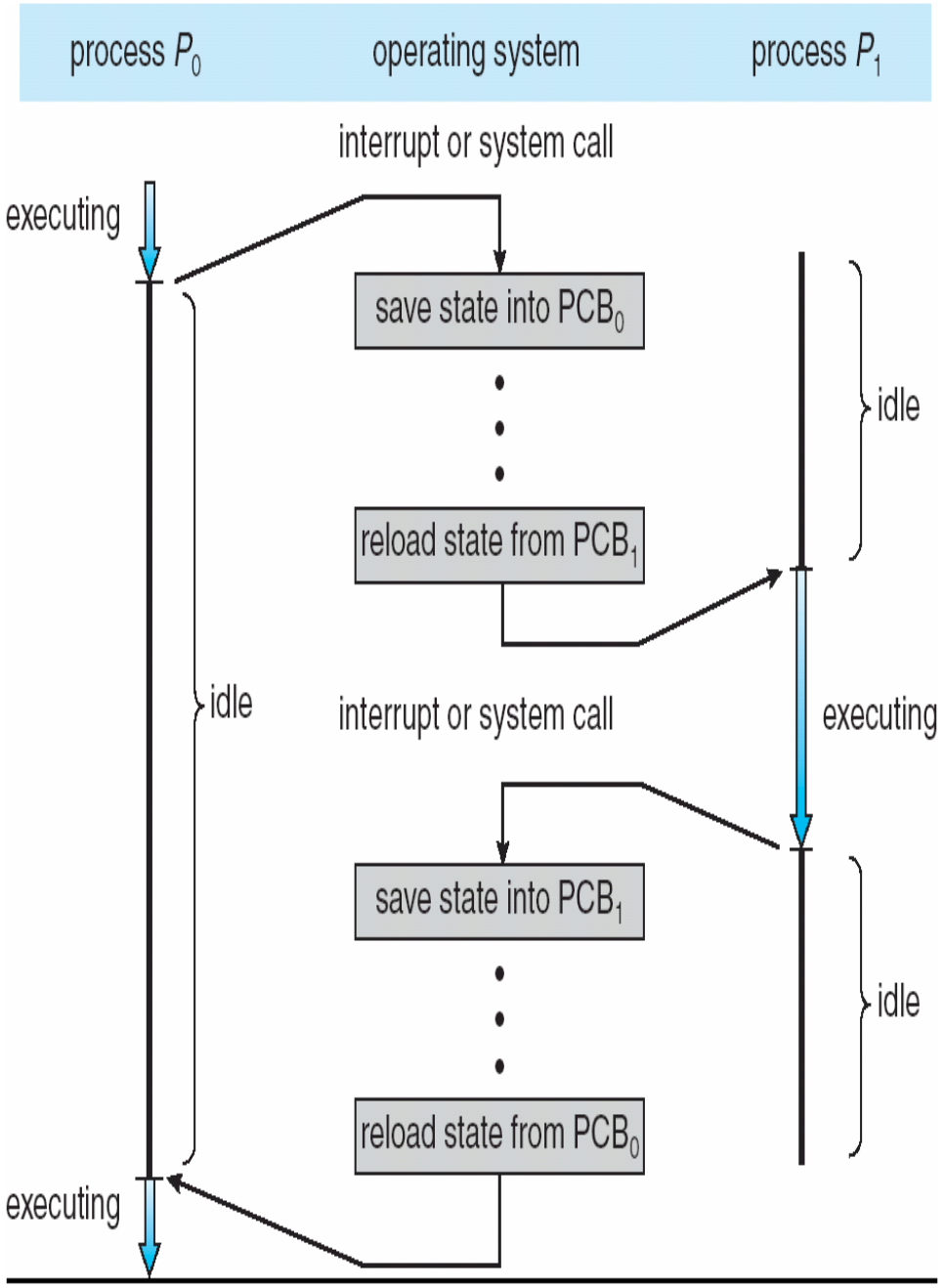
Schedulers – cont.

- **Medium-term scheduler (aka swapping scheduler)** can be added if degree of multiple programming is too high for efficient operation, or if the process mix (I/O-bound vs CPU-bound) for the short-term scheduler needs to be adapted.
  - Remove process from memory and store on disk (→ process state is now = "**suspended**"). Entire process is thus swapped out.
  - Bring back in from disk to continue execution. Entire process is swapped back in.



Process 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**
  - Note that a functional context switching is not shown here. It involves saving CPU registers so that the interrupt service routine can freely use the CPU.
- The **context** of a process is represented in the PCB.
- It involves a state save, then a state restore.



Process Context Switching

- 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 memory manager information also 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.

Context Switching

Note that so far, we have talked about **two types of context switching:**

- **Process context switching** - CPU registers of a running process (and many other state information) are saved prior to selecting another process to the “Run” state.

- **Functional context switching** – CPU registers of a calling function (the caller) are saved prior to invoking the called function (the callee). This responsibility is split (possibly unevenly) between the caller and the callee
  - An **interrupt function** is a function that is called from the interrupt vector table.
  - Saving the CPU registers of the interrupted routine is the sole responsibly of the ISR.

3.3 Operations on Processes

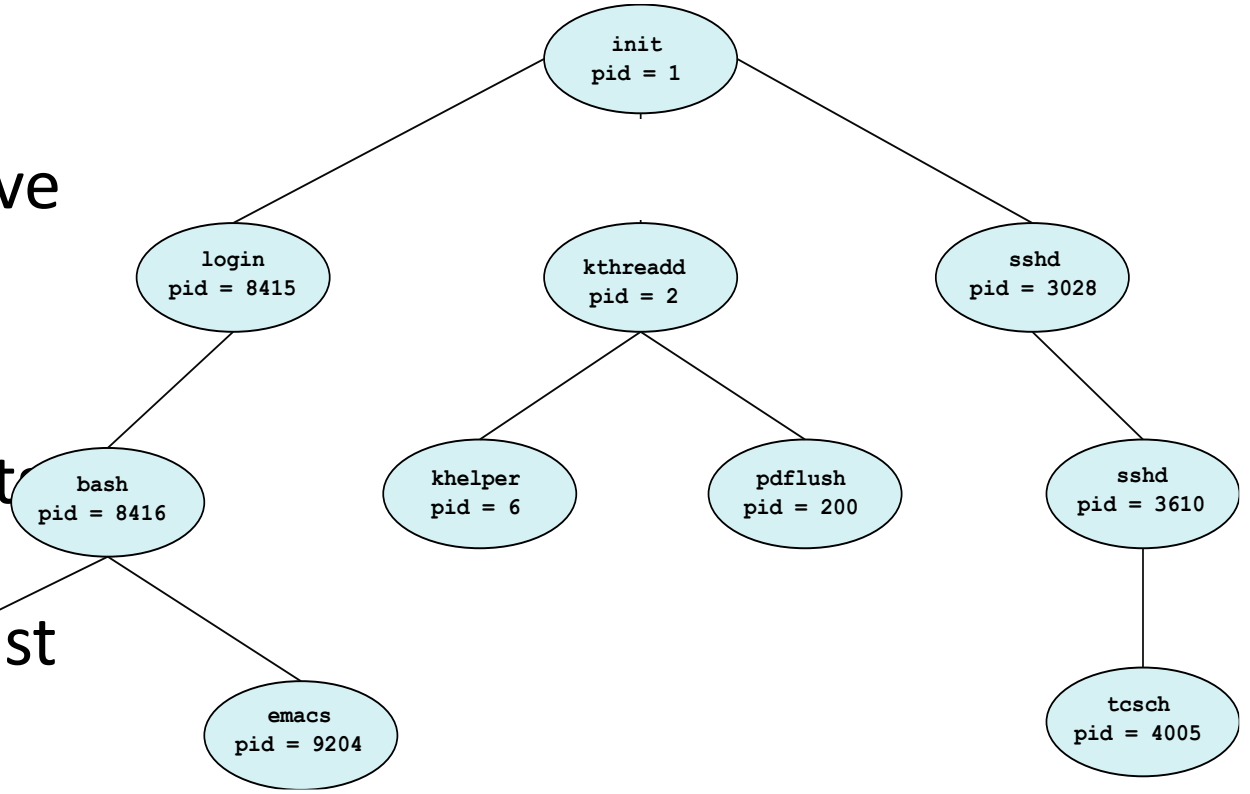
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

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
    - In Linux, they share all open files but do NOT share memory spaces.
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

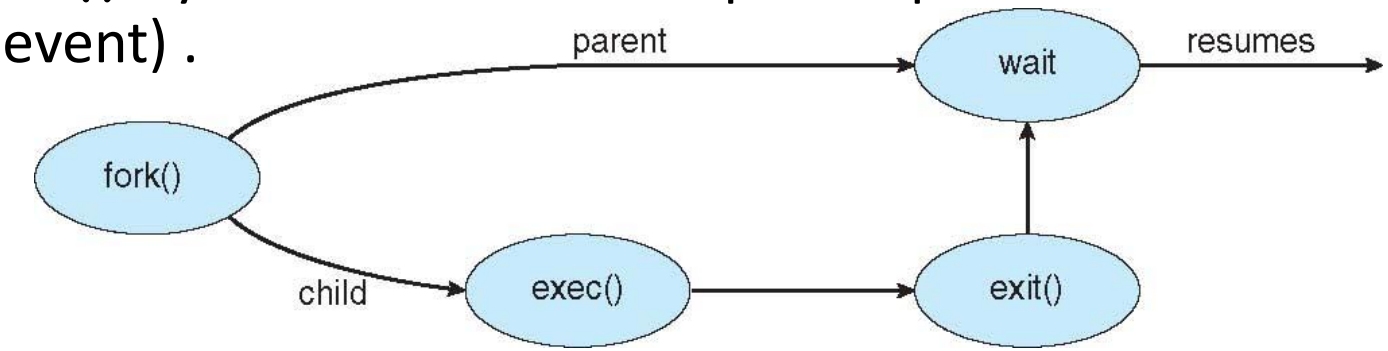
A Tree of Processes in Linux

- 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 may duplicate address space of parent (**duplicate ≠ share**), OR
  - Child may have another program loaded into it
- UNIX examples
  - **`fork()`** system call creates (aka **spawns**) a new process – In Unix-like systems, the address space is duplicated.
  - **`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 (an event) .

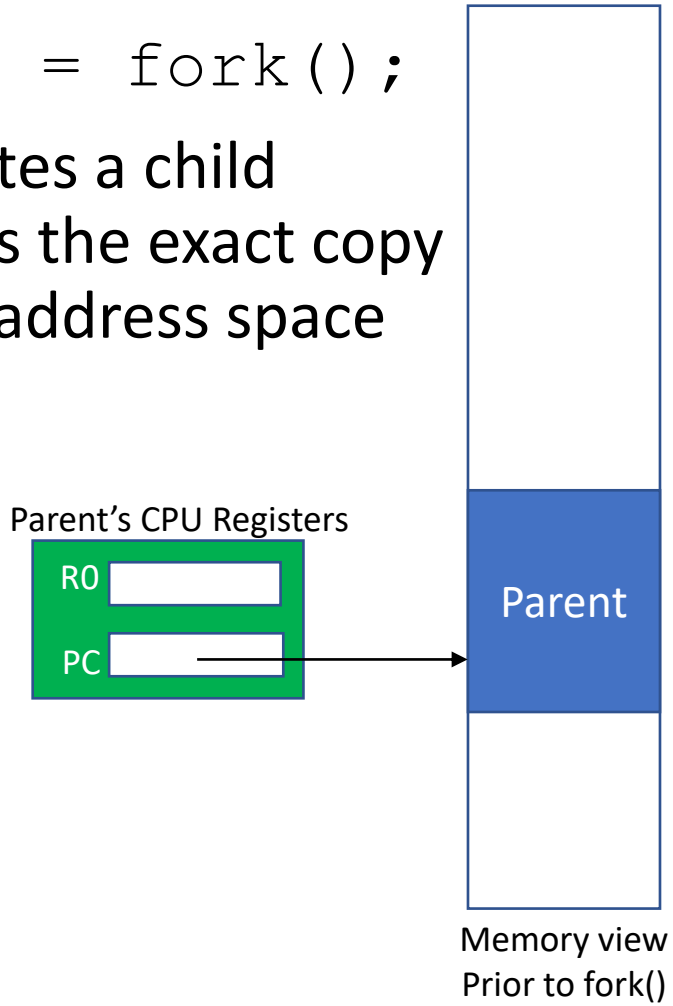


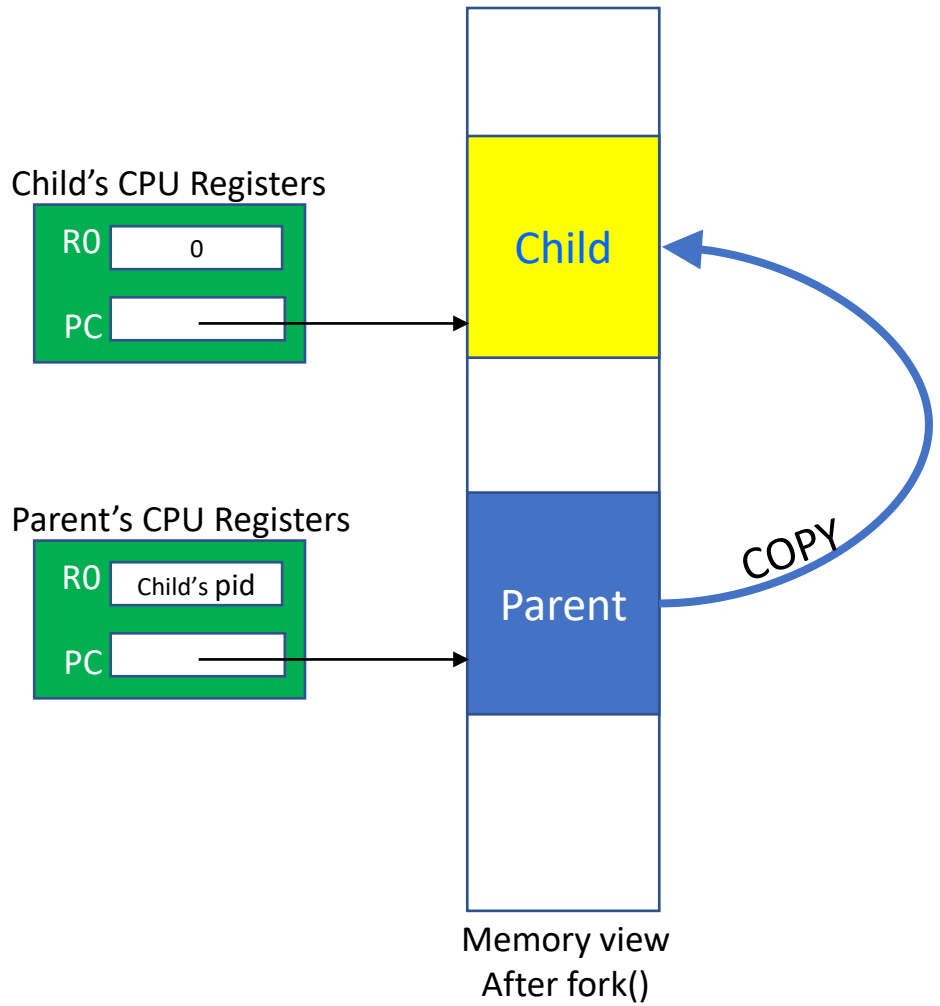
Creating a separate process in Unix/Linux - cont.

When using fork(), e.g.:

```
int ret = fork();
```

- The kernel creates a child process that has the exact copy of the parent's address space content

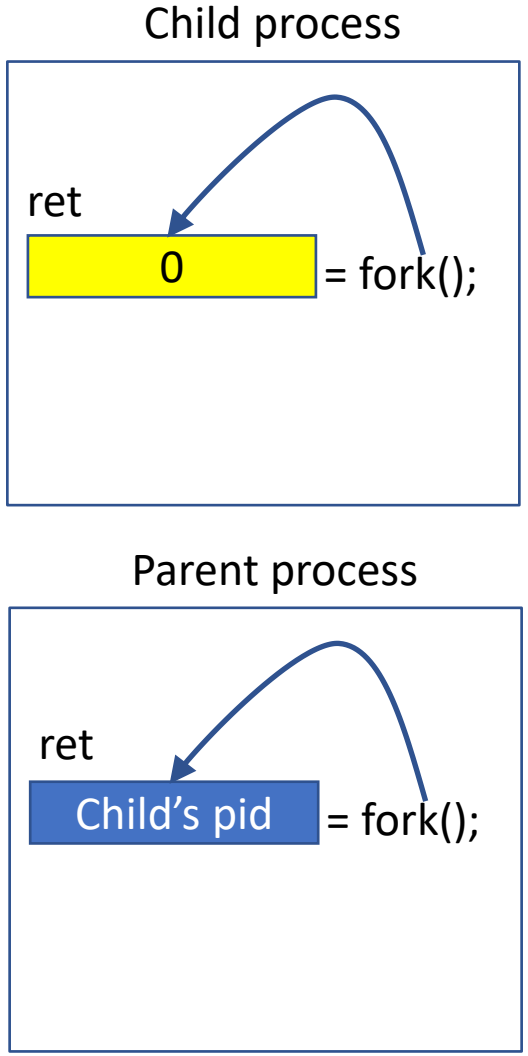




Creating a separate process in Unix/Linux - cont.

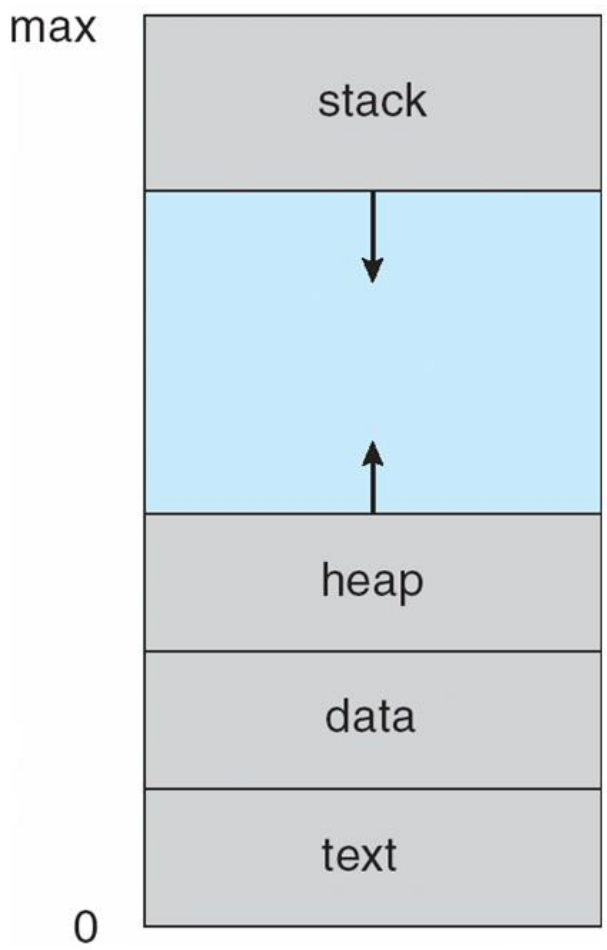
When using "`int ret=fork()`" – cont.

- The parent executes and then invokes the fork() function within the API library
  - which invokes the system call (via a **trap instruction, obviously!**)
- Upon return from the system call, both the parent and the child processes resume at the instruction following the fork() call.
  - That would be the assignment statement!
- Note that 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 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 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 implicitly 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

```
int status;
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.

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Mozilla Firefox and Google Chrome Browsers are multi-process with 3 different types of (**communicating**) processes:
  - A **browser** process manages user interface, disk and network I/O
  - One or more **renderer** processes renders web pages, deals with HTML, Javascript, etc. A new renderer created for each tab/website opened
    - Since each tab is a separate process, they don't share memory. They also do not share file resources based on how their parent process (the browser) created them, thus minimizing effect of security exploits.
    - If a renderer crashes, it doesn't bring down the entire browser.
  - One or more **plug-in** processes for each type of plug-in