Chapter 3: Processes

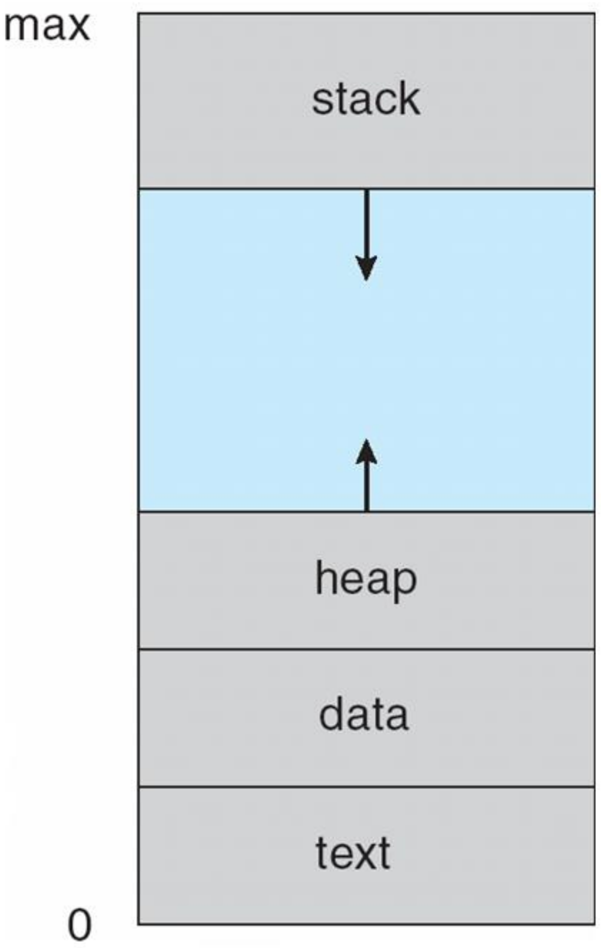
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

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 traditional operating systems (i.e. those running workstations or servers):
    process = job = task
- **Process** – a program in execution; process execution generally 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.).

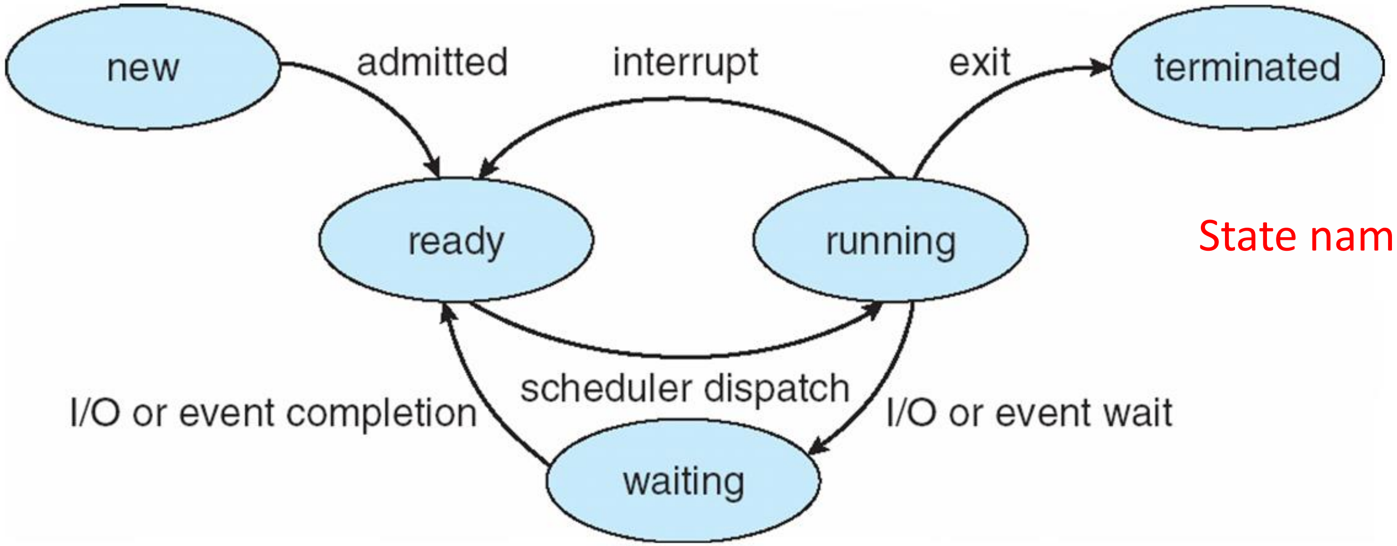


The Process Concept (Cont.)

- 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



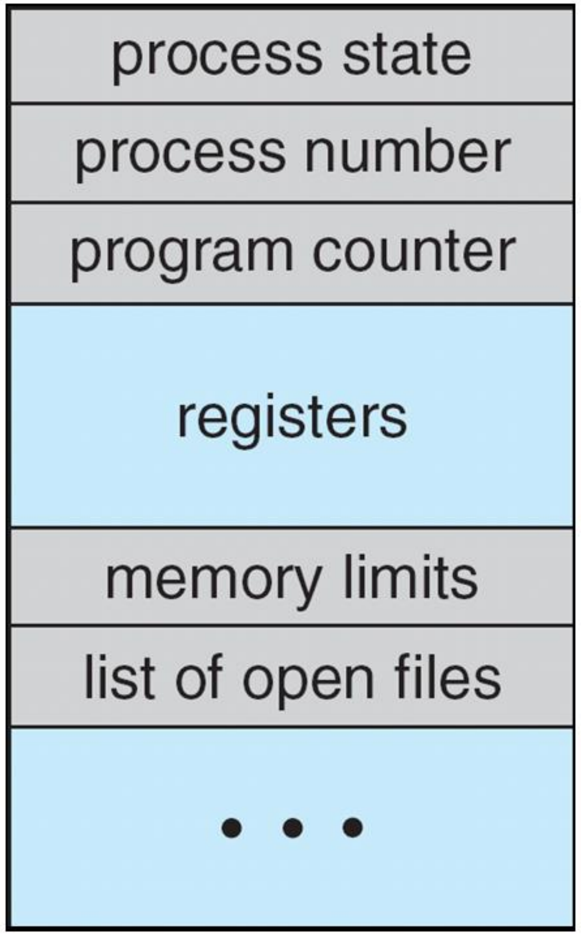
State names are generic

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



process state

process number

program counter

registers

memory limits

list of open files

• • •

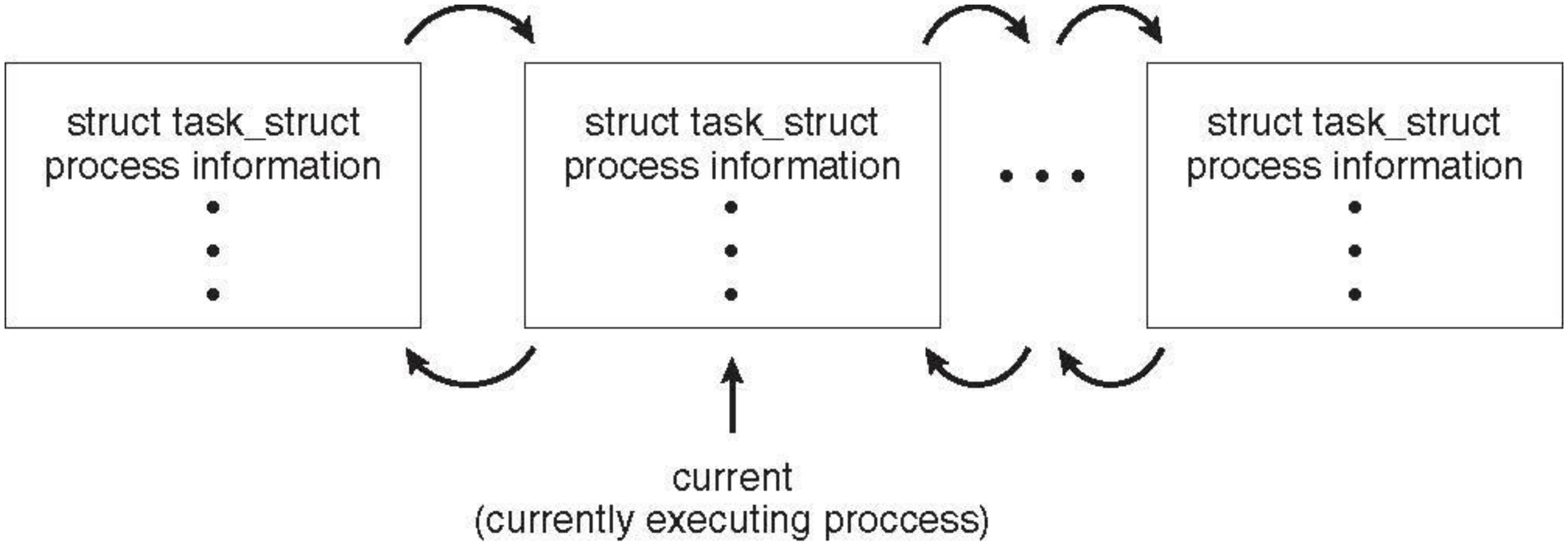
Threads

- 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.

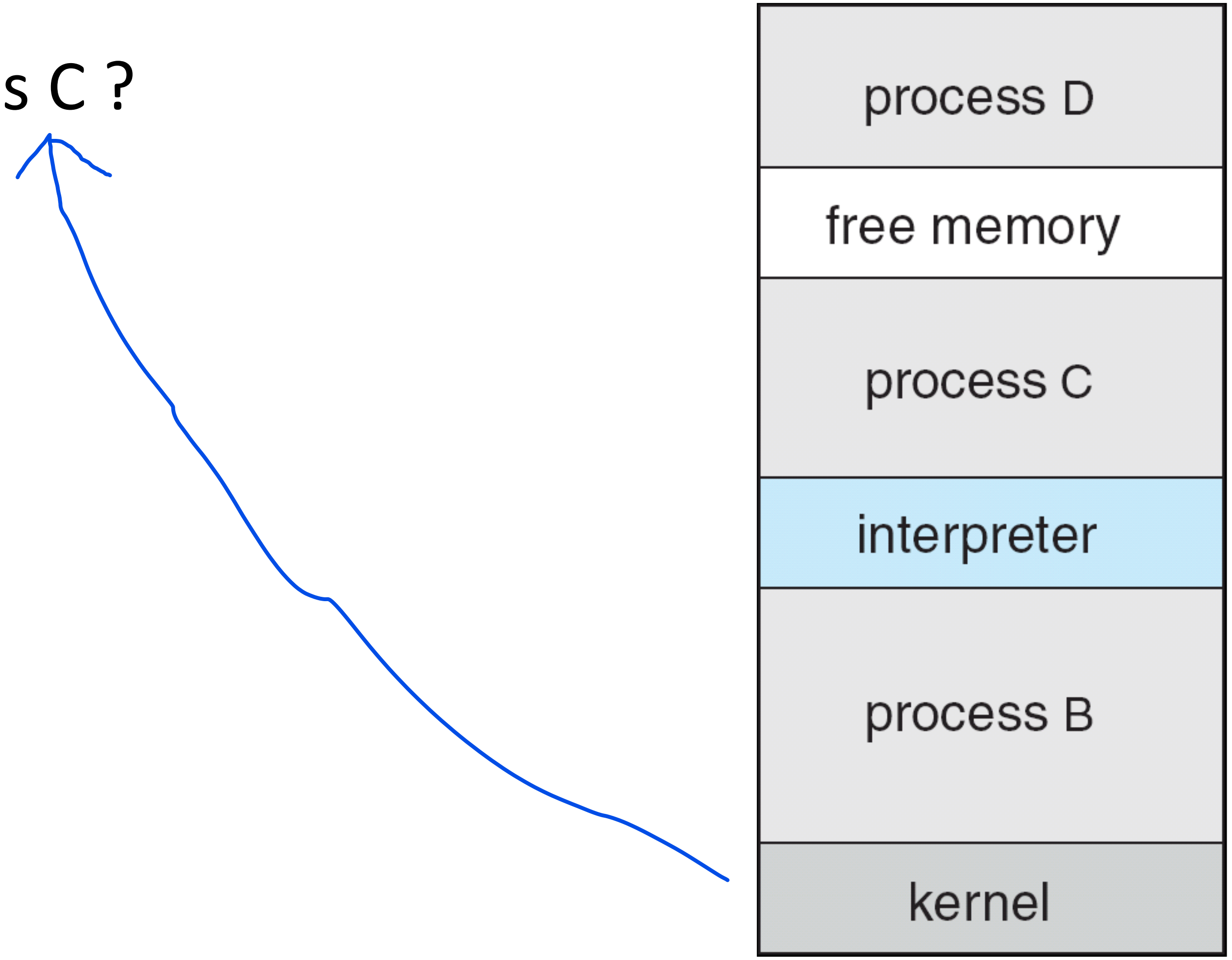
Process Representation in Linux

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 */
```



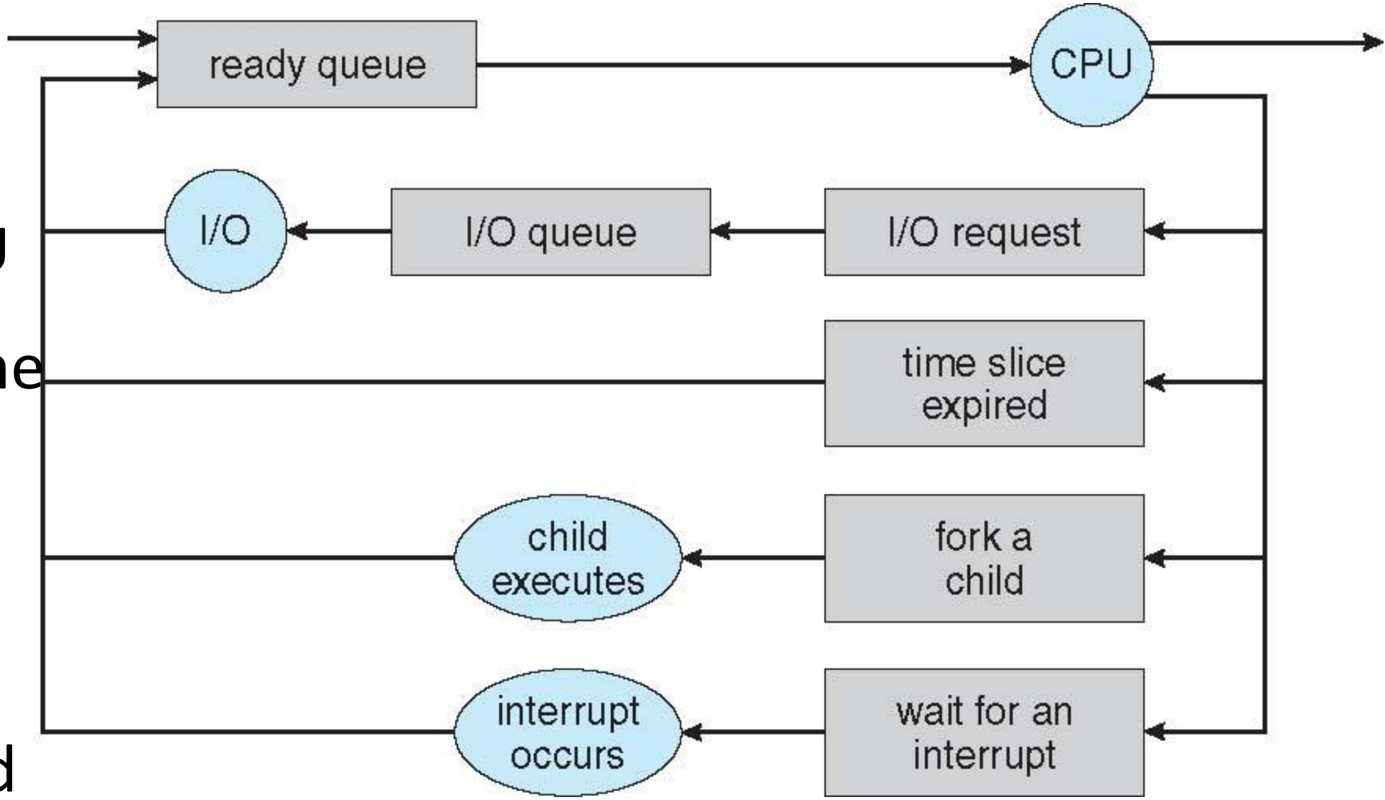
Where is the PCB for process C ?



main memory

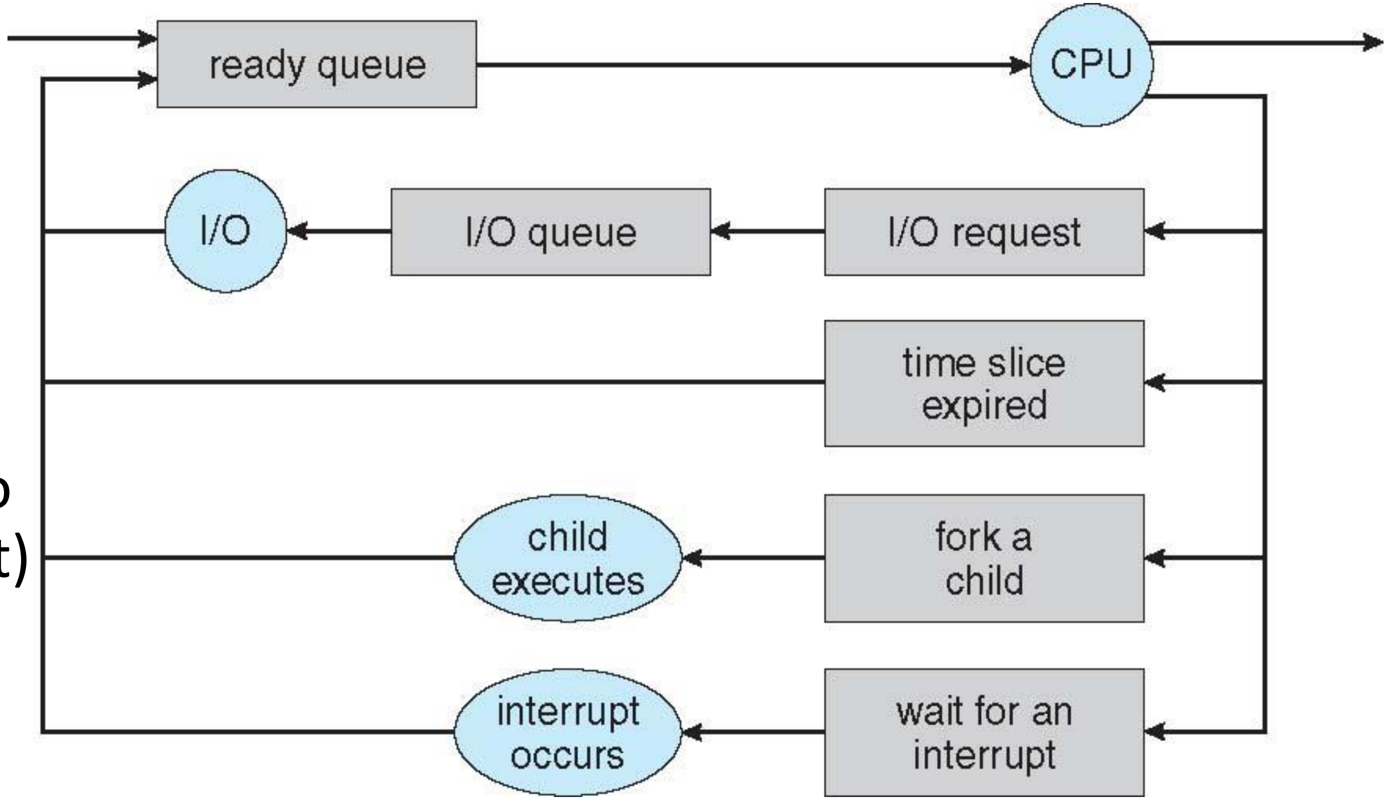
3.2 Process Scheduling

- 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



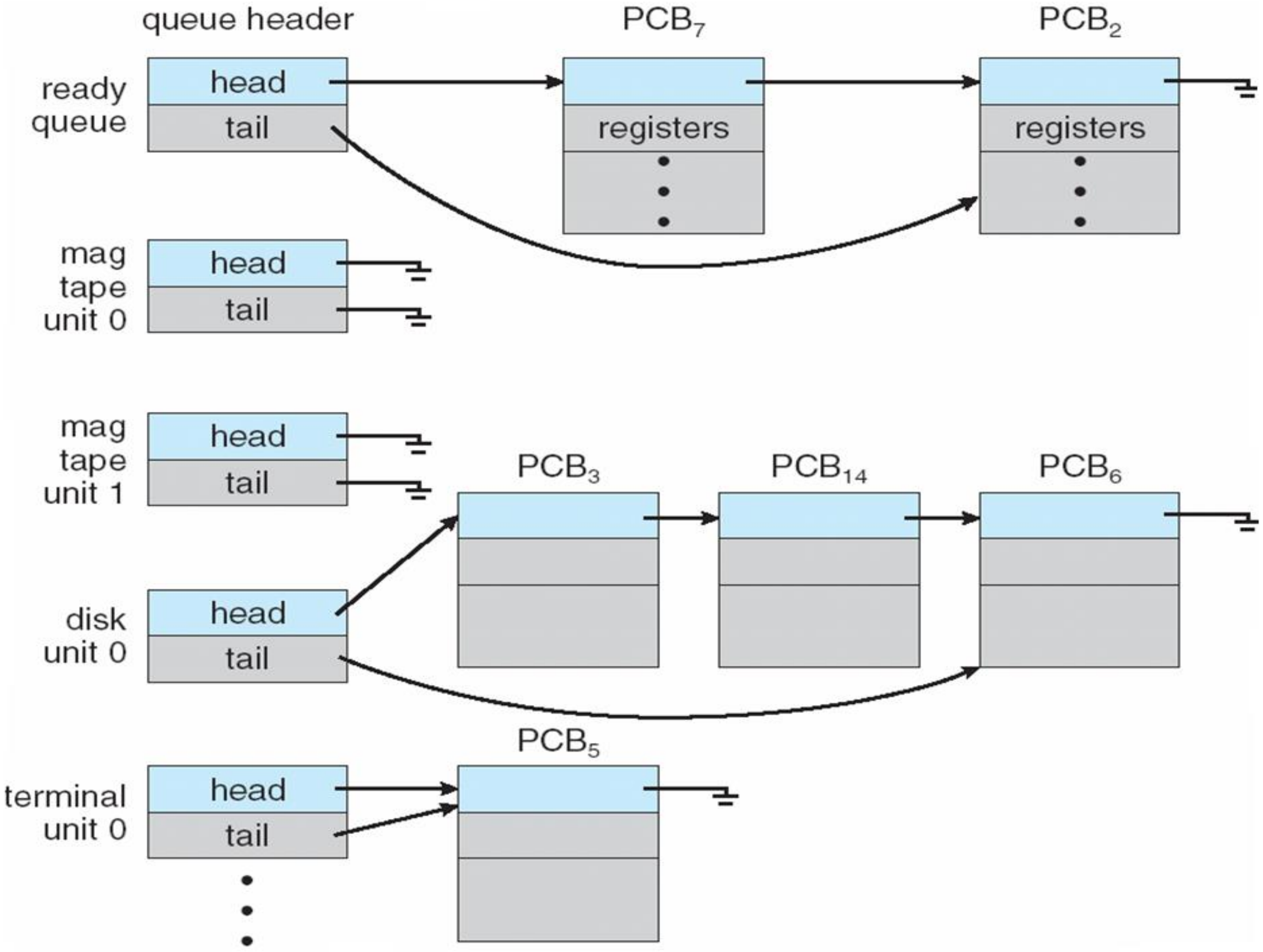
Queueing diagram represents queues, resources, flows

- **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

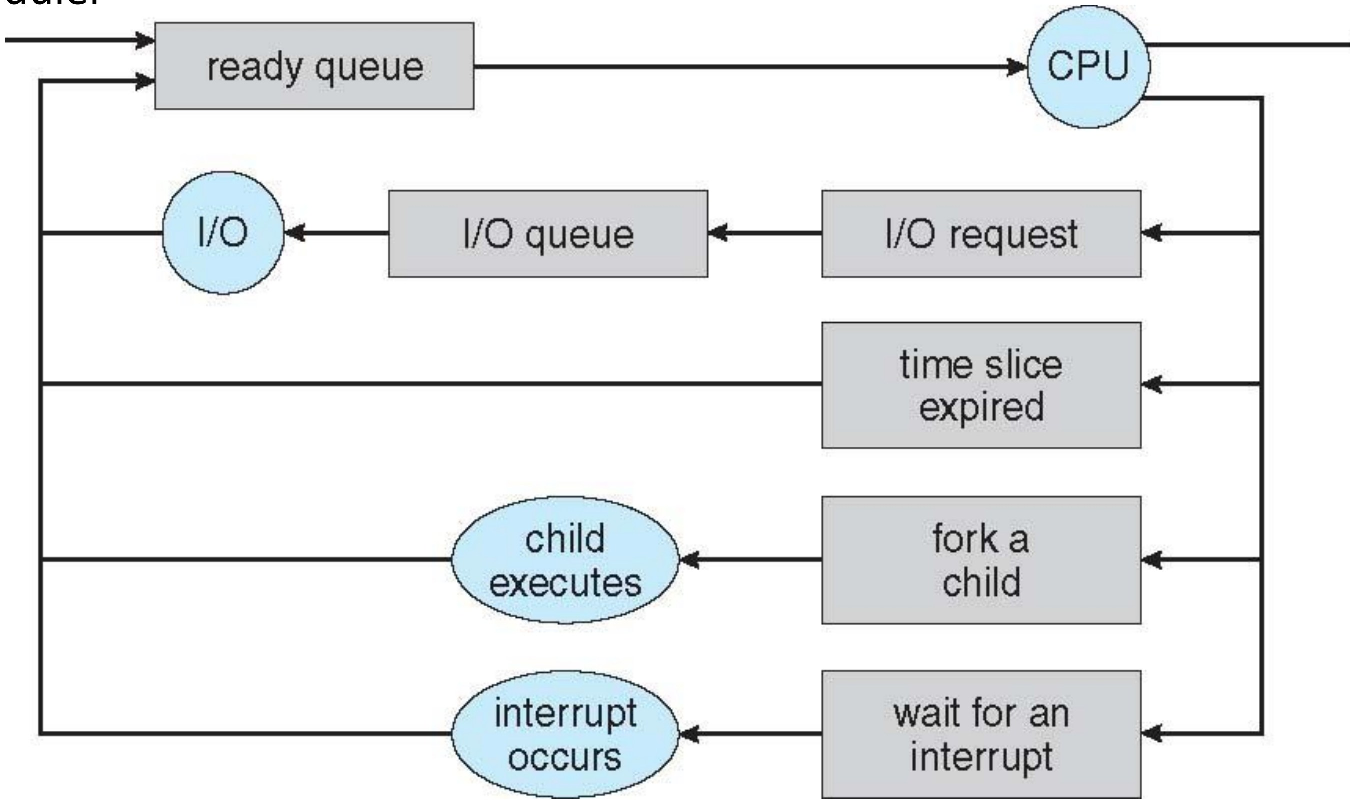


Schedulers

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**

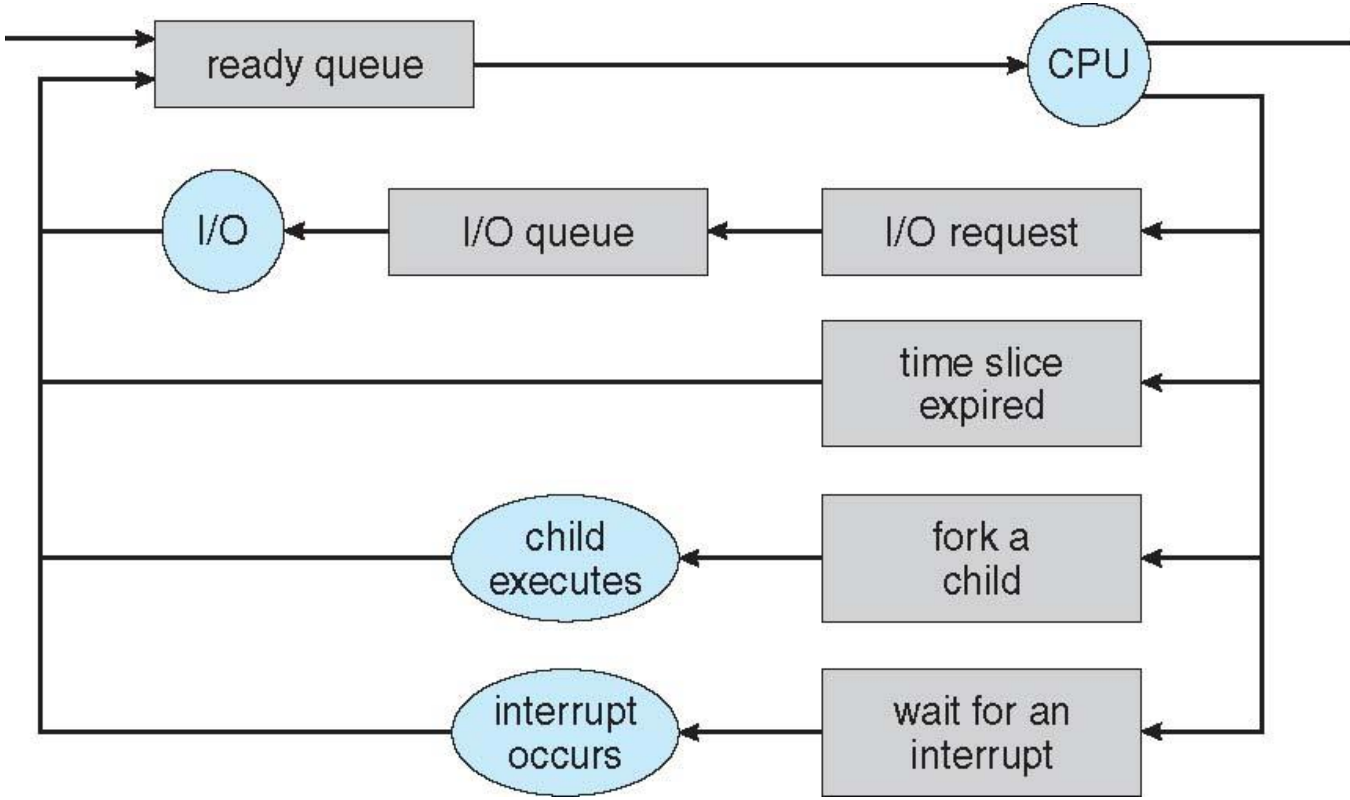
from long term scheduler



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.



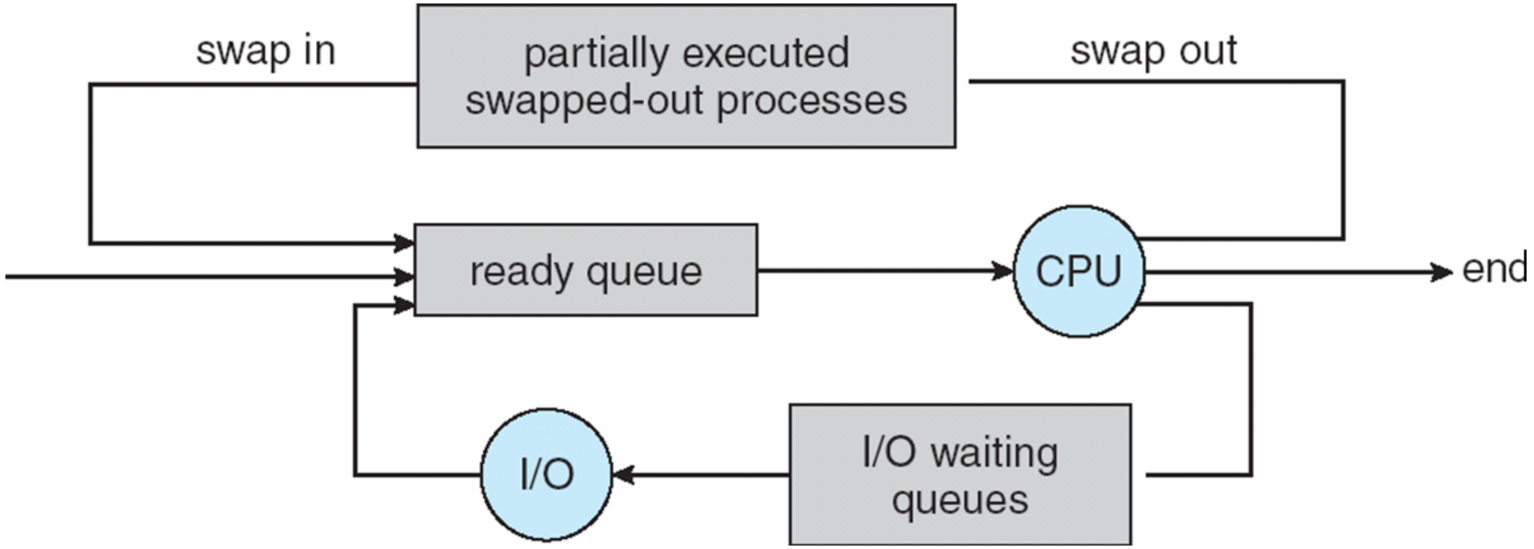
Queueing diagram represents queues, resources, flows

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**

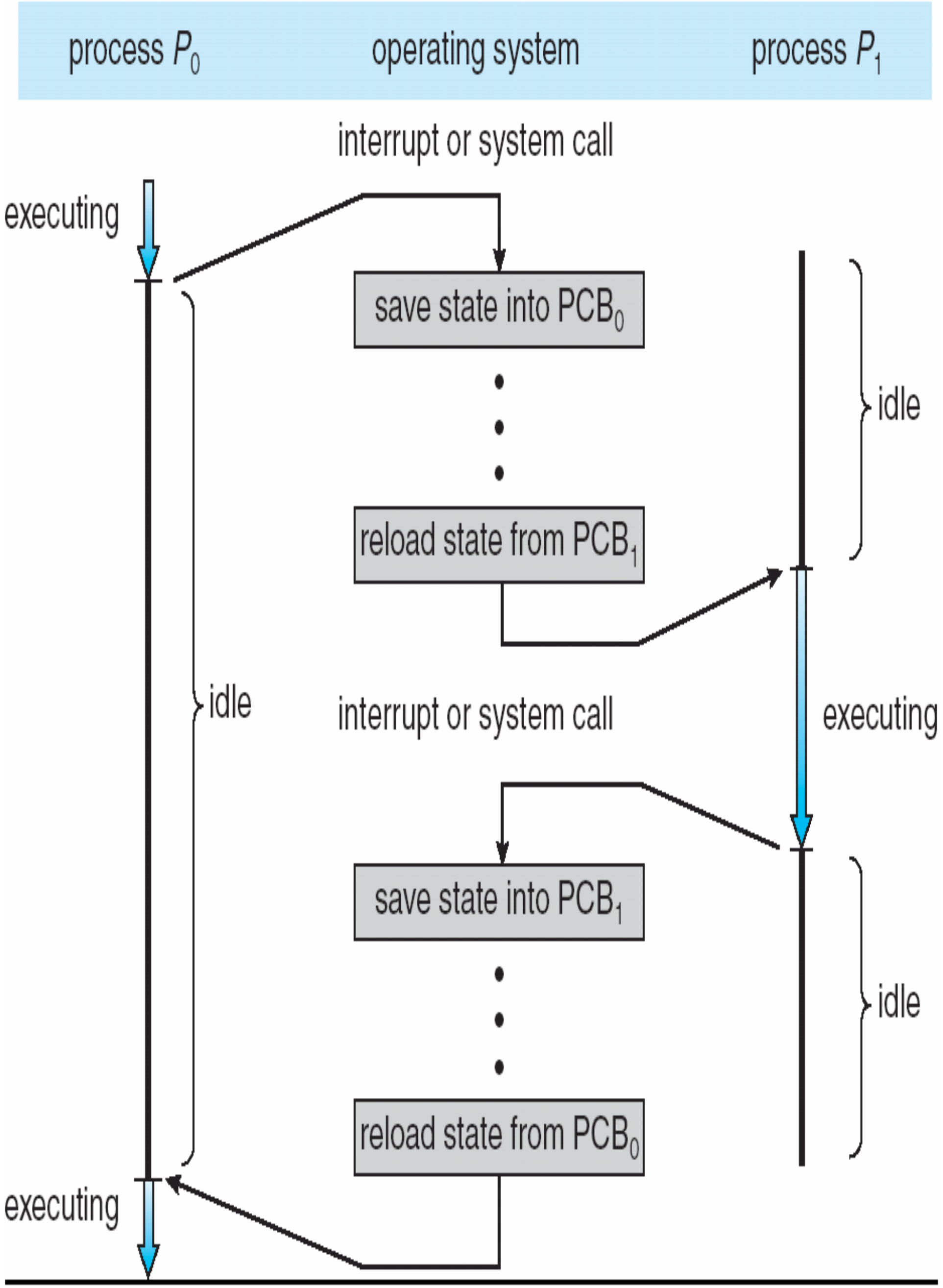


Multitasking in Mobile Systems

- 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

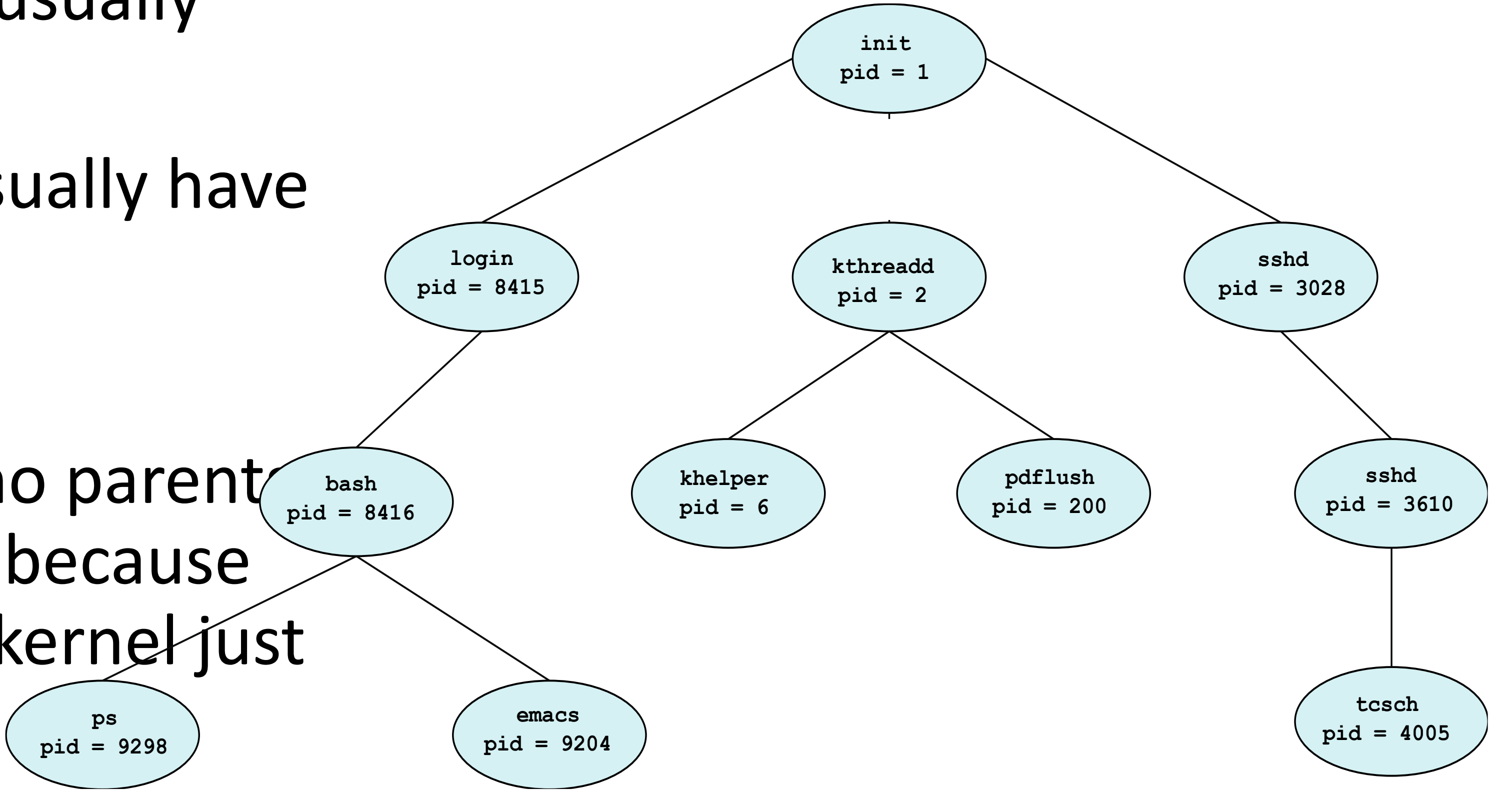
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

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

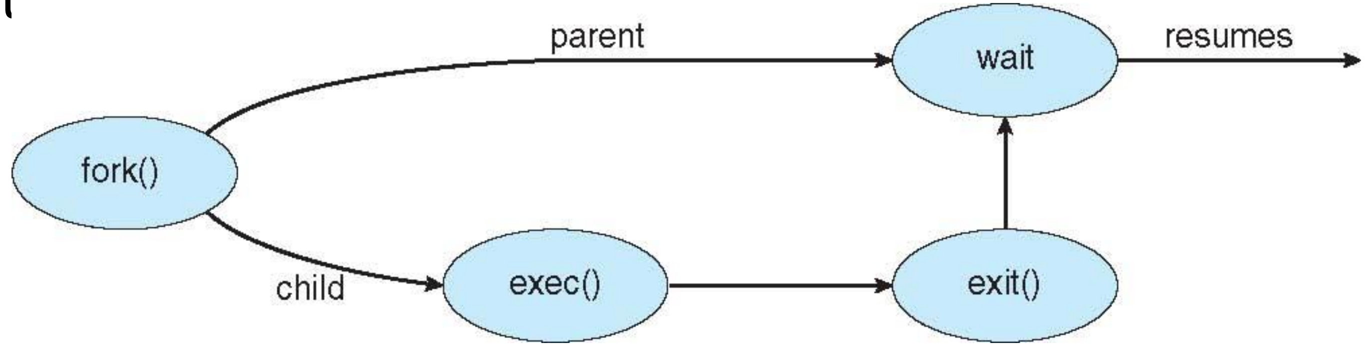
A Tree of Processes in Linux

- 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 parents (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

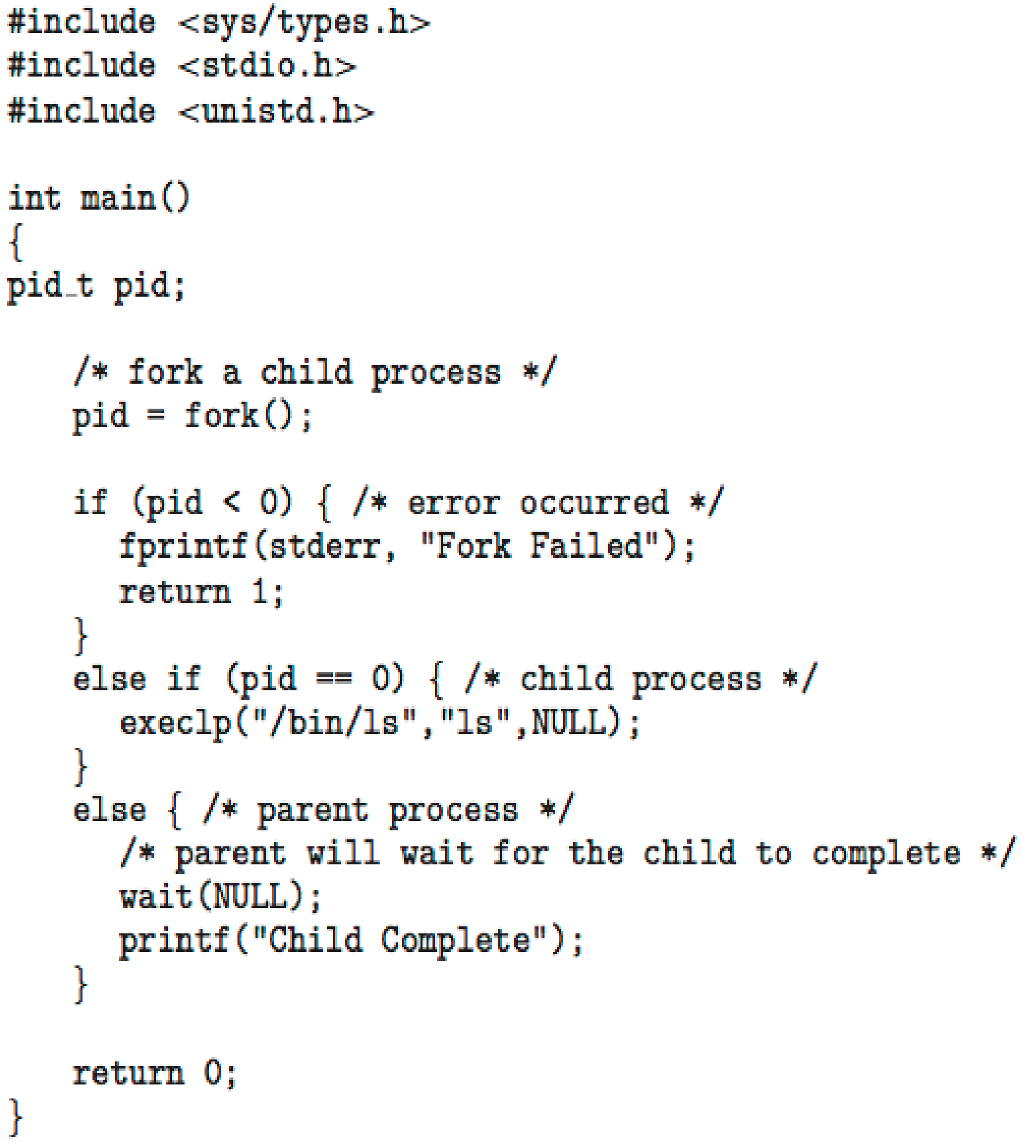


Process Creation (Cont.)

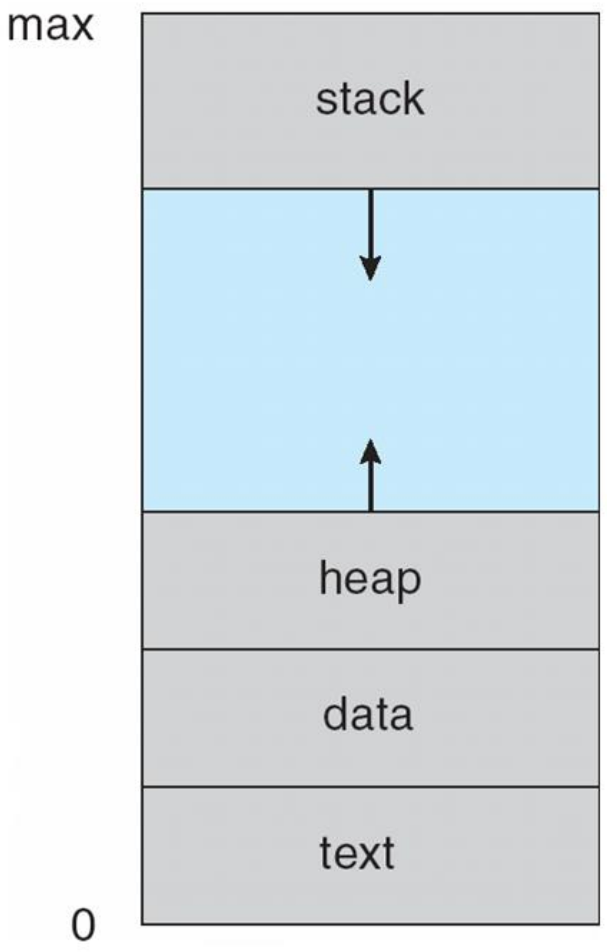
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates (aka spawns) a new process.
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program, i.e. loads new program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



Creating a separate process in Unix/Linux

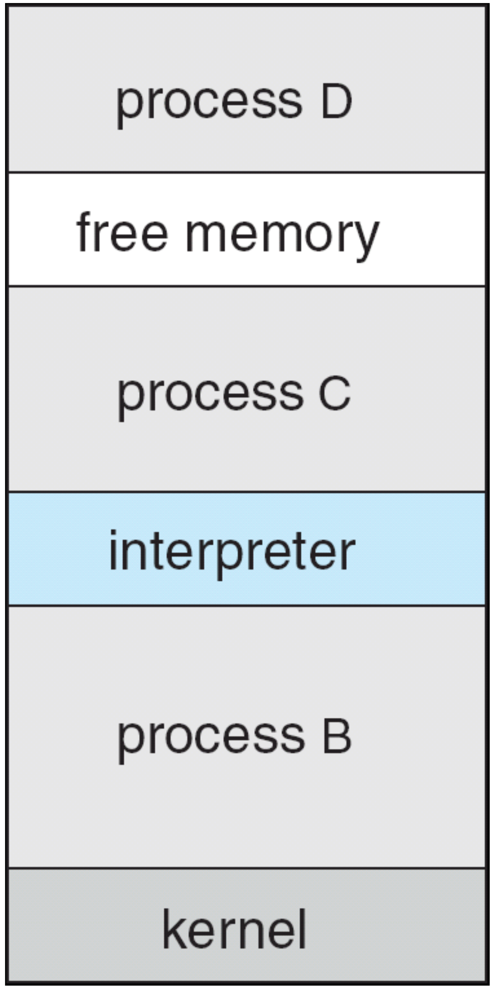


```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```

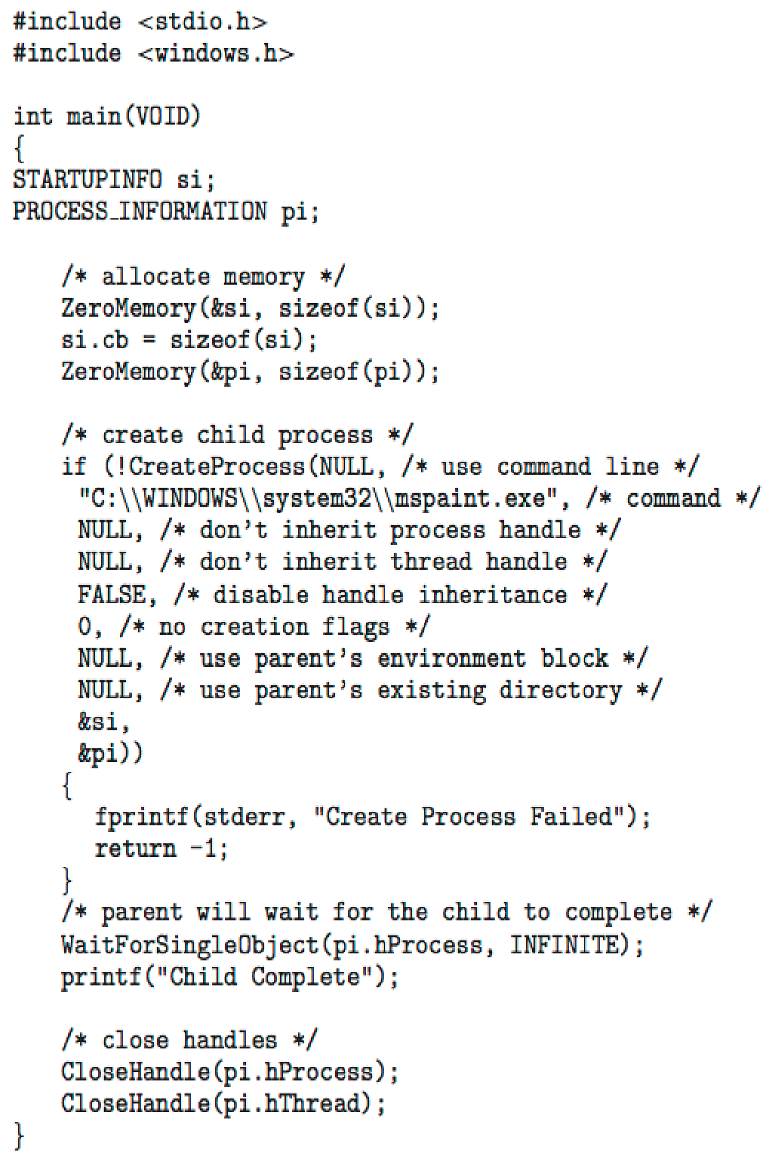


Creating a separate process in Unix/Linux - cont.

- 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

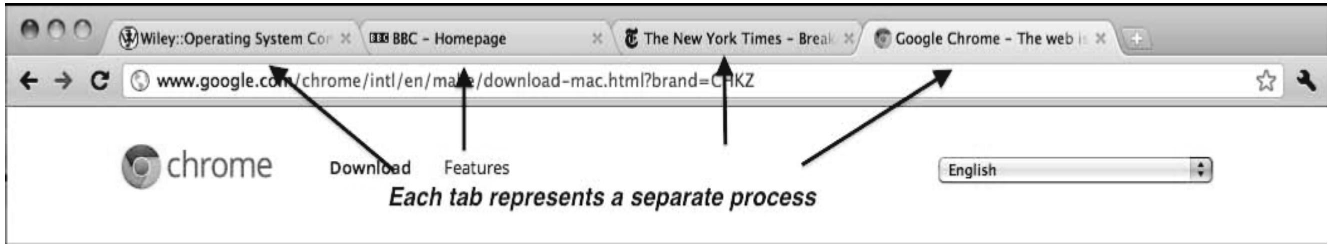
- Some operating systems do not allow a child to exist if its parent has terminated. Hence, in such systems, if a process terminates, then the OS terminates all its children.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating 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.

Multiprocess Architecture – Chrome Browser


Wiley::Operating System Cor
BBC - Homepage
The New York Times - Break
Google Chrome - The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English

Each tab represents a separate process

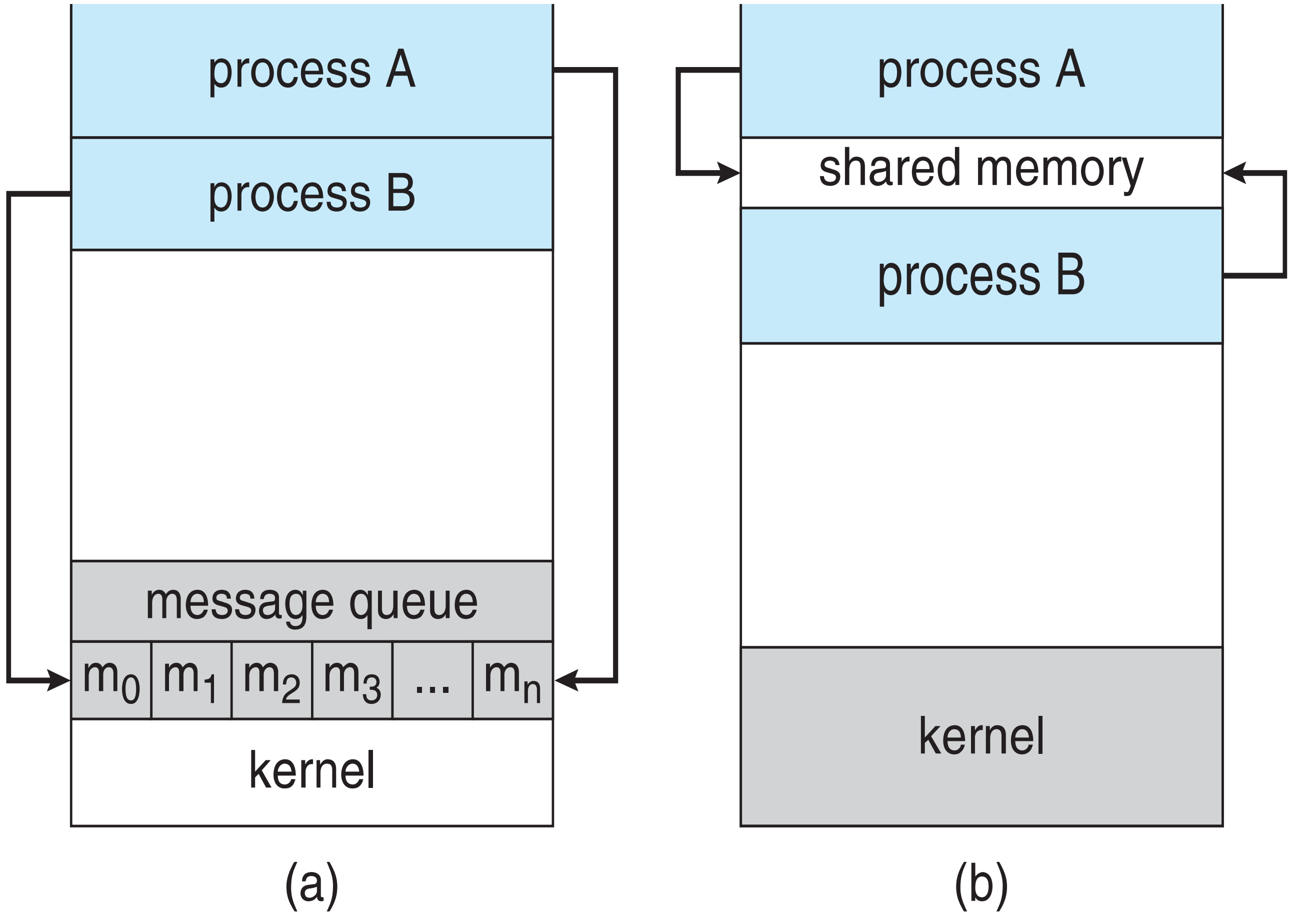
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 different types of (**communicating**) processes:
  - A **browser** process manages user interface, disk and network I/O
  - One 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

3.4 Inter-process Communication (IPC)

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing (e.g. shared file such as a database)
  - Computation speedup (if system has multiple CPU cores)
  - Modularity (may divide a program into tasks, may feed or use services of other tasks)
  - Convenience (e.g. a user may be editing and listening to music)
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

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



3.4.1 Shared memory systems

- An area of **memory shared among the processes** that wish to communicate (the creation of this area is facilitated by the OS kernel since each process normally has a separate address space)
- After the shared memory has been created by the OS, **the mechanism used for communication** between the user processes is administered by them, not the operating system.
- A major issue is to provide a mechanism that allows the user processes to **synchronize** their actions when they access shared memory locations.
  - The communicating processes may use synchronization functions provided by the OS kernel. This shall be discussed in great details in the next chapter.

Producer-Consumer Problem

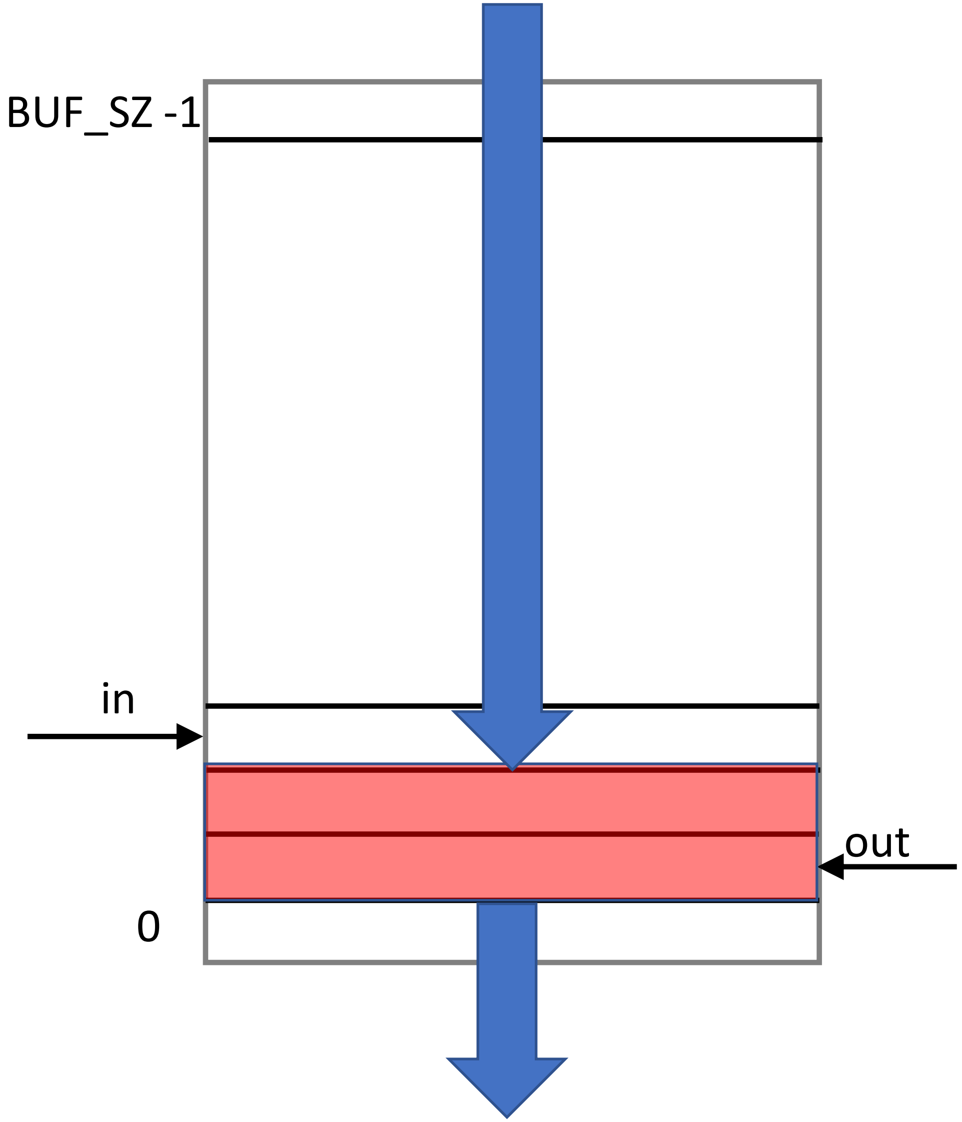
- The producer-consumer problem is a common paradigm for cooperating processes and may illustrate issues with shared memory systems.
- The *producer* process produces information that is consumed by a *consumer* process, e.g.
  - Multiple subtasks forming wider function such as a compiler producing an object file and a linker task consuming object files to produce the executable code.
  - A client producing window commands (e.g. draw a rectangle) and an X11 display server consuming window commands.
  - An X11 display server producing mouse/keyboard data and a client process receiving mouse clicks or keyboard keys.
  - These were general examples of processes producing and consuming information. X11 servers generally do not communicate using shared memory, they use TCP/IP sockets or pipes.
- Two types of buffers may be used
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data:

```
#define BUF_SZ 10
typedef struct {
  . . .
} item;

item buffer[BUF_SZ];
int in = 0;
int out = 0;
```

- Buffer needs to be administered and used as a FIFO or Queue

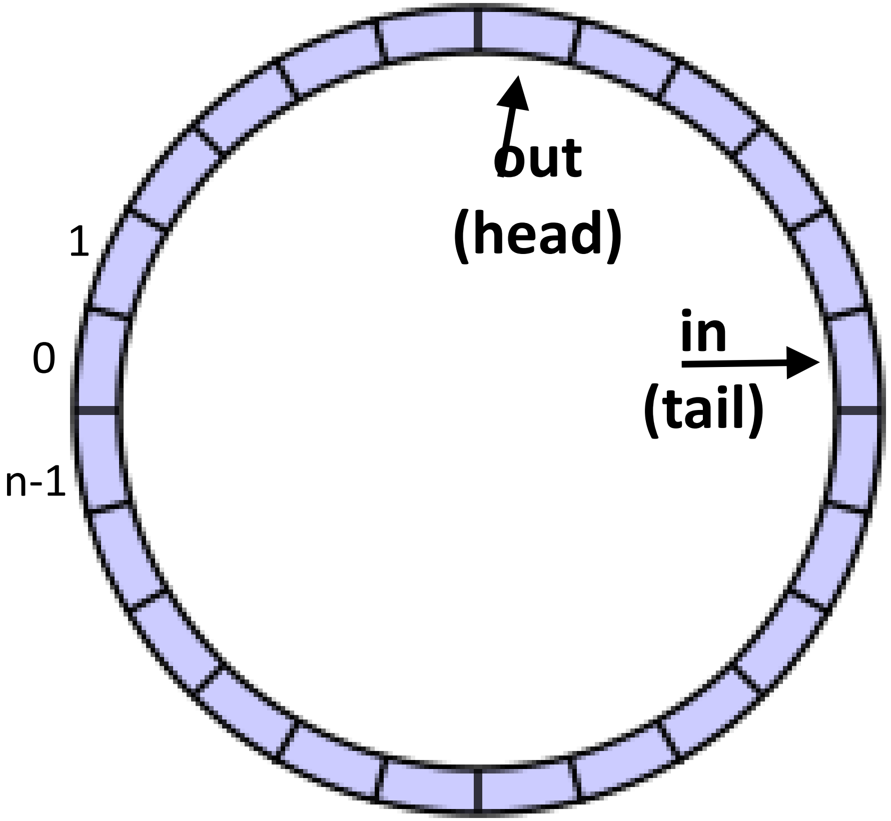


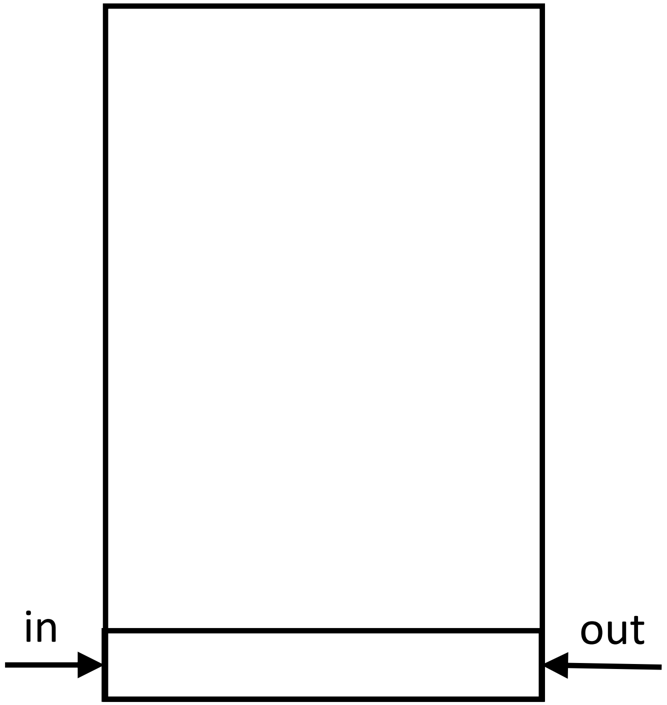
Bounded-Buffer – Shared-Memory Solution

- Shared data:

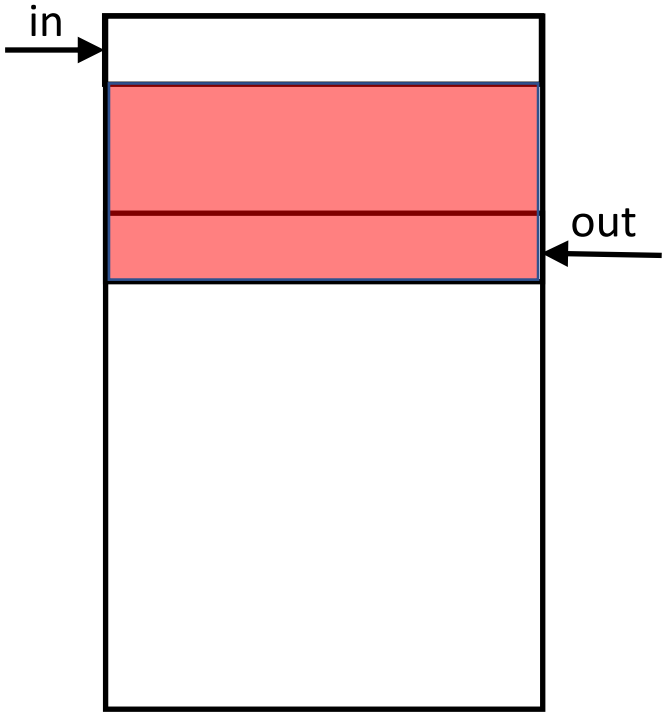
```
#define BUF_SZ 10
typedef struct {
  . . .
} item;

item buffer[BUF_SZ];
int in = 0;
int out = 0;
```

- Buffer needs to be administered and used as a FIFO or Queue

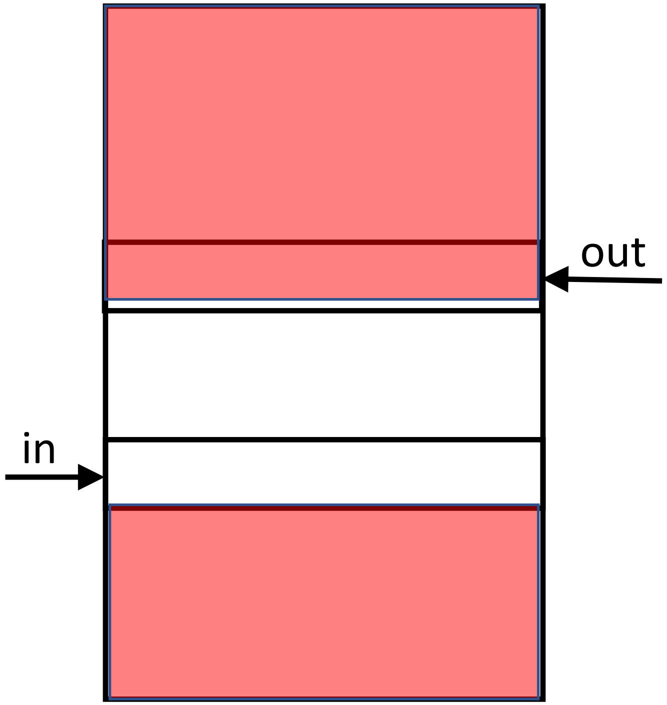




Initial state



After a few writes/reads - No wrapping OR Both ‘in” and “out” wrapped



After a few writes/reads - Only the “in” wrapped but not the “out” index

Bounded-Buffer – Producer

```
item next_produced;

while (true) {

  /* produce an item in next produced */


  /* wait till next in != out */

  while (((in + 1) % BUF_SZ) == out);


  buffer[in] = next_produced;

  in = (in + 1) % BUF_SZ;

}
```

Bounded Buffer – Consumer

```
item next_consumed;

while (true) {
  /* wait till in != out*/
  while (in == out);

  next_consumed = buffer[out];

  out = (out + 1) % BUF_SZ;


  /* consume the item in next consumed */

}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements (why?)
- In this solution, the producer and consumer do not access the same item simultaneously
- What happens if both need to access the same location concurrently (e.g. a shared variable or a shared counter?
    - Synchronization will then be needed (next chapter) → can’t do that for now

3.4.2 Message passing systems

- Another mechanism for processes to communicate and to synchronize their actions

- Message system – processes communicate with each other without resorting to shared variables

- IPC facility (implemented by OS) provides (two theoretical) operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via `send(message)` and `receive(message)`
- Design choices:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - Is the size of a message that the link can accommodate fixed or variable?
  - What is the capacity of a link?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - On the physical level:
    - Via shared memory
    - Hardware bus or a communication network.
  - On the logical level:
    - A. Direct or indirect
    - B. Synchronous or asynchronous
    - C. Automatic or explicit buffering

A . Naming: Direct Communication

- Symmetric schemes: Processes must specify each other explicitly (via process identifier):
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Asymmetric schemes: Only sender names the recipient
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*&ID, message*) – receive a message from any process and when you receive a message indicate the name of the sender (in the variable *`ID`*).
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link is usually bi-directional
- Process identifiers may be names or integer numbers.
- Disadvantage is that a process identifier needs to be hardcoded, which thus requires recompilation if the identifiers change.

Naming: Indirect Communication

- Messages are directed to, and received from **ports or mailboxes**. Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - Each mailbox has a unique identifier. (name or number)
  - Link established only if processes share a common mailbox
  - A link **may** be associated with many processes
  - Each pair of processes **may** share several communication links
  - Link may be unidirectional or bi-directional

Naming: Indirect Communication – cont.

- A port (or mailbox) may be **owned by a process**
  - The port/mailbox is attached to a process and implemented inside its address space.
  - If the process exits, the mailbox is destroyed.
  - Unidirectional:
    - Owner (server) can only receive messages via the mailbox
    - User (client) sends message to the mailbox.
- Alternatively, a port may be **owned by the operating systems**, and thus the OS may need to provide operations such as:
  - Create a new port/mailbox
  - Send and receive messages through the port
  - Delete the port.
  - Unix implements two such ports;
    - pipes (unidirectional) and
    - TCP/IP ports (bidirectional)

Naming: Indirect Communication – cont.

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

B. Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking and the link has zero buffering, we have a **rendezvous**
- NOTE: Throughout the course:

BLOCKING = SYNCHRONOUS

NON-BLOCKING = ASYNCHRONOUS

Synchronization (Cont.)

- Producer-consumer becomes trivial (i.e. no concern about how to manage a circular buffer

```
message next_produced;
while (true) {
  /* produce an item into
  next produced */
  send(next_produced);
}
```

```
message next_consumed;
while (true) {
 /* consume the item into
  next consumed */
  receive(next_consumed);
}
```

C. Buffering

- Queue of messages attached to the link.
- implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

Shared memory vs message passing

- Message passing may be advantageous for exchanging smaller amounts of data since the synchronization overhead is avoided.
- Shared memory can be faster than message passing, particularly for larger amounts of data since no copying is involved. This is true only in cases where the synchronization overhead can be minimized.
- However, in multi-processing (or multicore) systems, research has shown that message passing is more efficient, even for larger blocks of data, due to the cache coherency overhead.