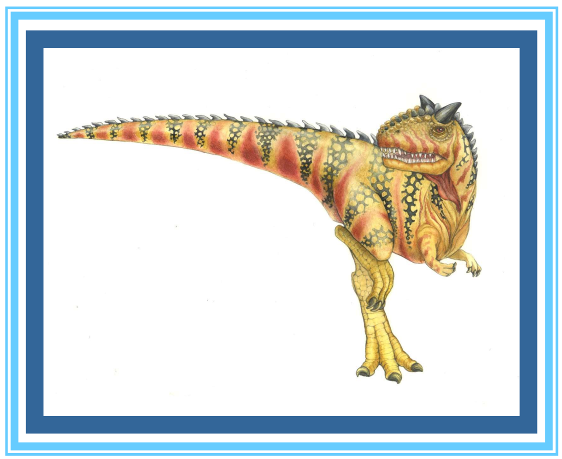
Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

Process Management

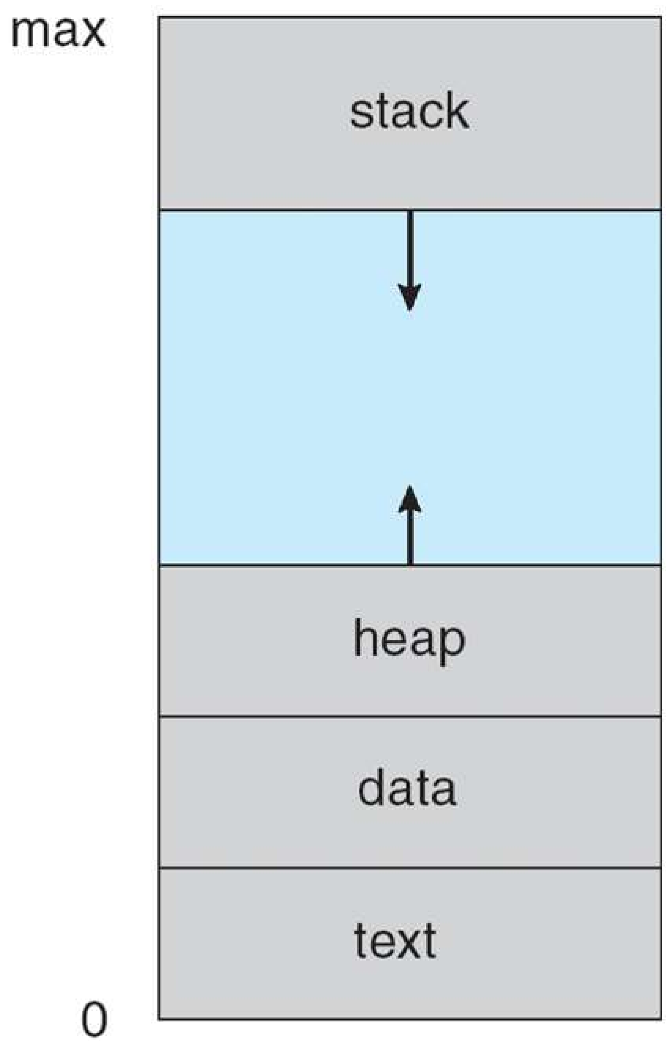
- A process can be thought of as a program in execution.
- A process will **need certain resources**—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.
- A process is the unit of work in most systems.
- Two types of Processes
  - **operating-system processes** execute system code, and
  - **user processes** execute user code.
- All these processes may **execute concurrently**.
- The operating system is **responsible for**
  - the **creation and deletion** of both user and system processes;
  - The **scheduling** of processes; and
  - the **provision of mechanisms** for synchronization,
  - **communication**, and
  - **deadlock handling for processes**.

Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- A program by itself is not a process.
- **A program is a passive entity**, such as a file containing a list of instructions stored on disk (often called an executable file).
- In contrast, **a process is an active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.
- **A program becomes a process** when an executable file is loaded into memory.

- Multiple parts (of a process)
  - 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 in Memory



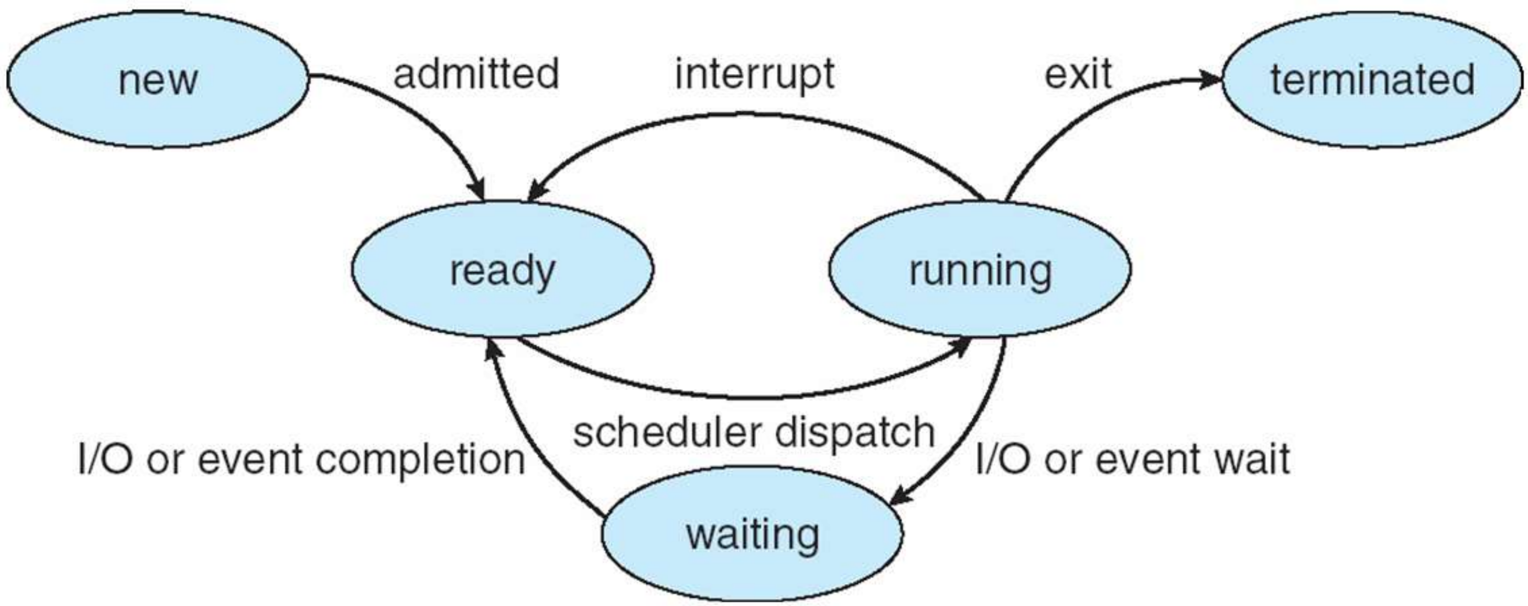
Process Concept (Cont.)

- A program is loaded and 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 (multiple copies)
  - Or the same user may invoke many copies of the web browser program
  - Each of these is a separate process; and
  - although the text sections are equivalent, the data, heap, and stack sections vary.
- It is also common to have **a process that spawns many processes** as it runs.

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



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- priorities, scheduling queue pointers (chap 6)
- Memory-management information – memory allocated to the process (chap 8)
- 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

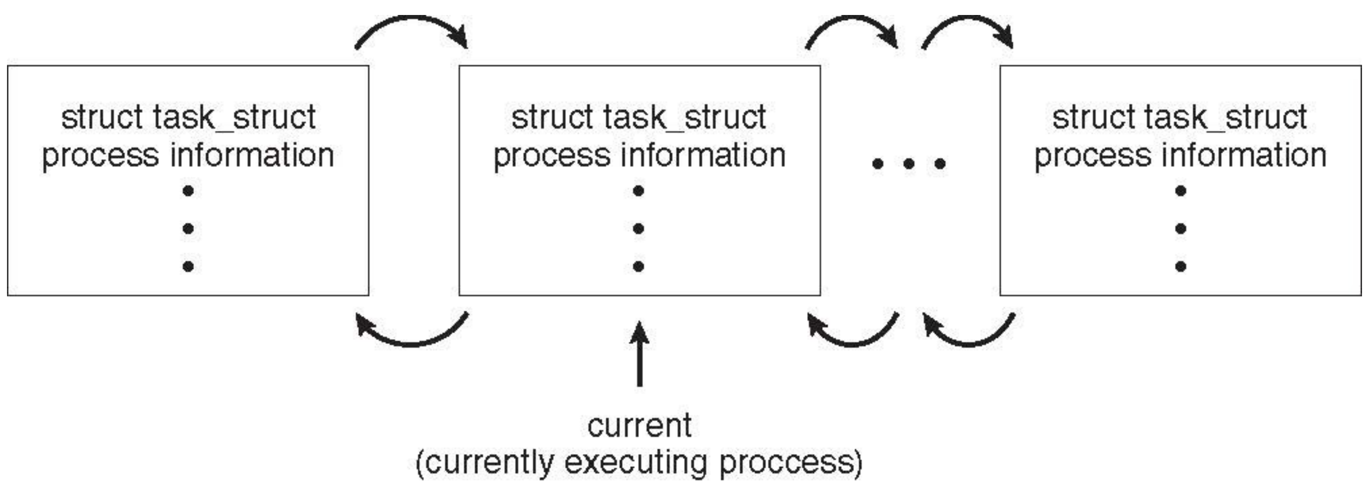
Threads

- So far, process has a single thread of execution
- The user cannot simultaneously type in characters and run the spell checker within the same process, for example.
- Modern operating systems extended the process concept to **allow a process to have multiple threads** of execution and thus to perform more than one task at a time.
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, **multiple program counters in PCB**
- **Other changes are also needed** to support threads.
- See next chapter (chap 4)

Process Representation in Linux

Represented by the C structure `task_struct`

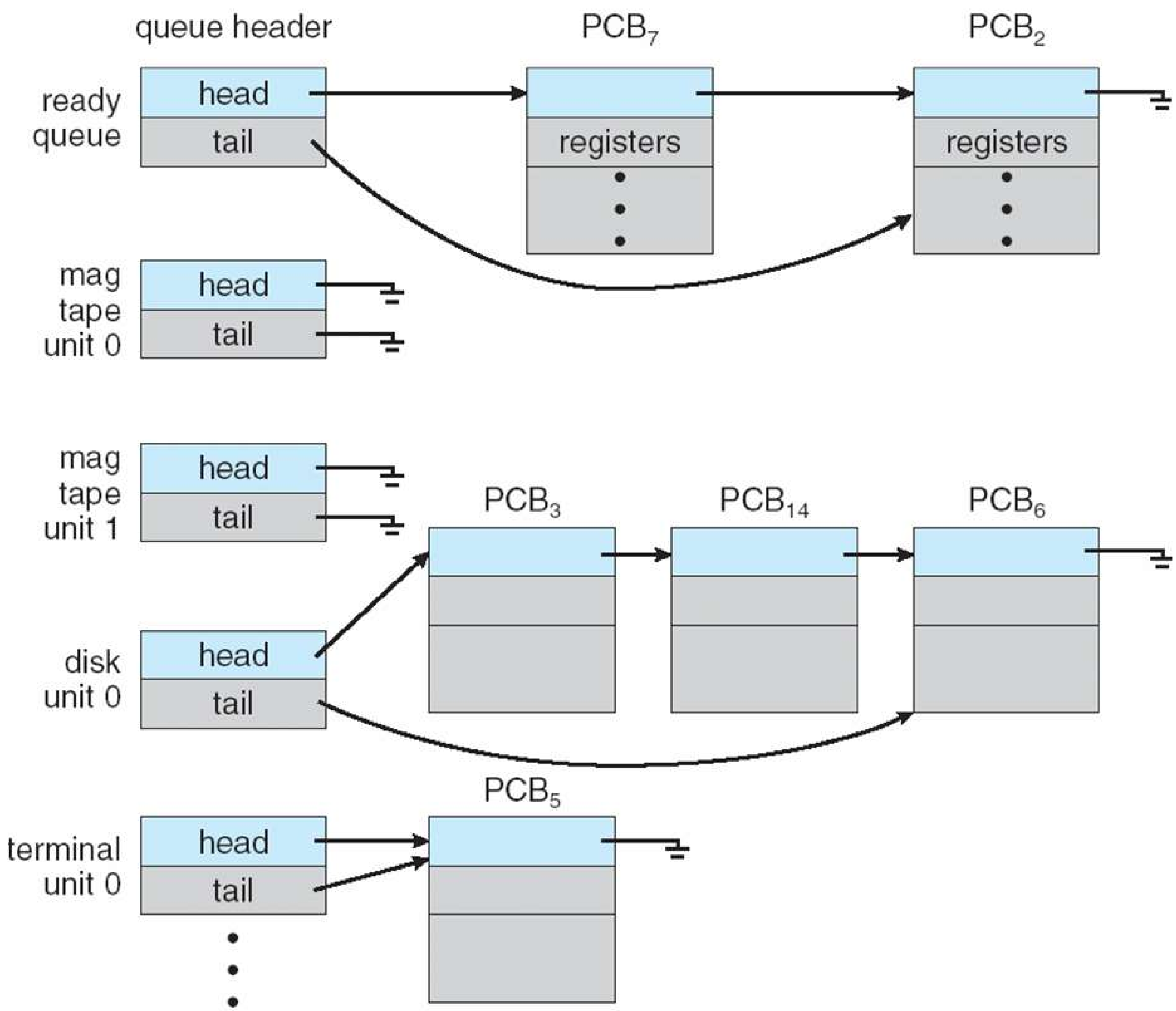
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

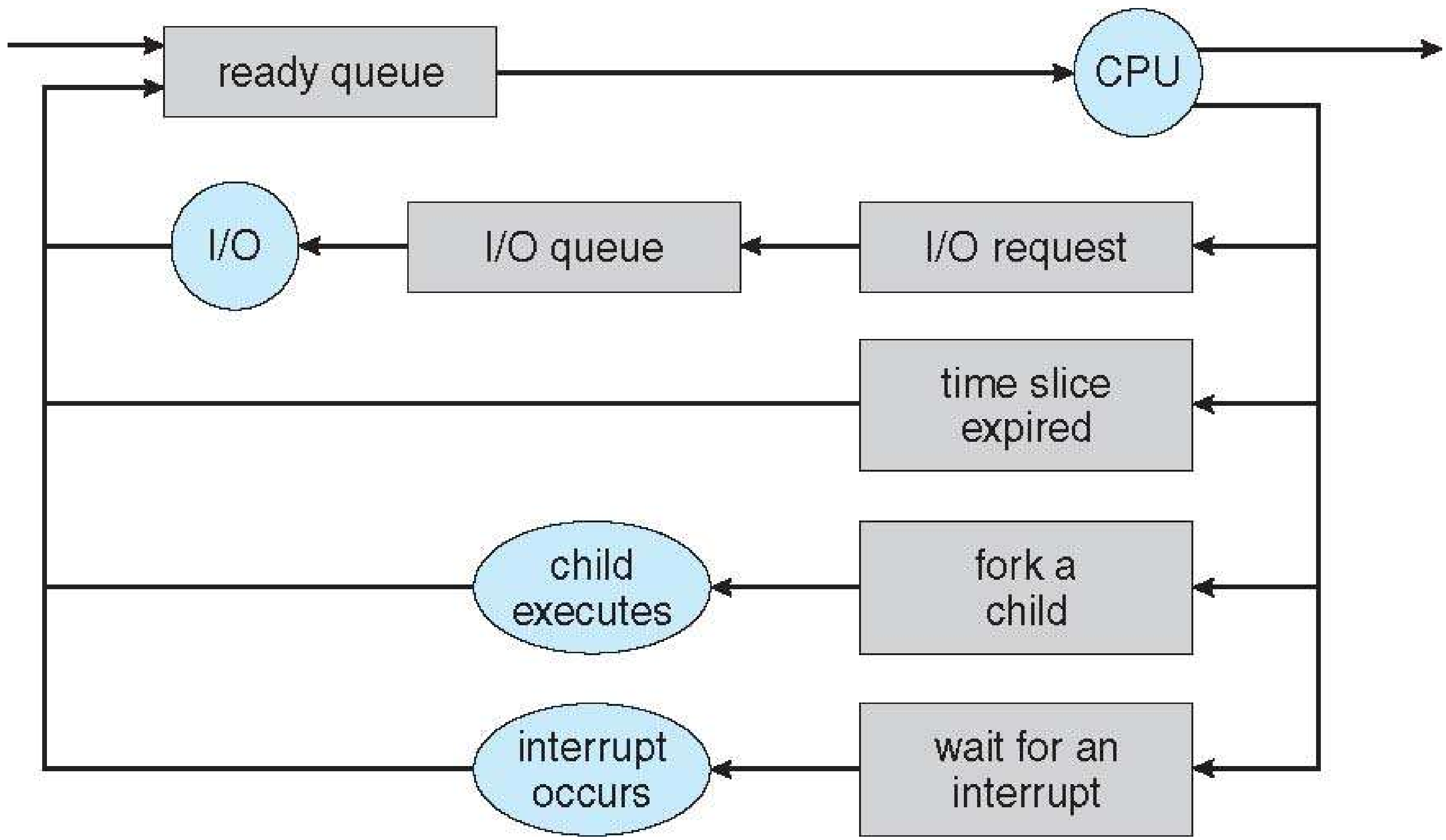
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device (Each device has its own device queue)
  - **Processes migrate** among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

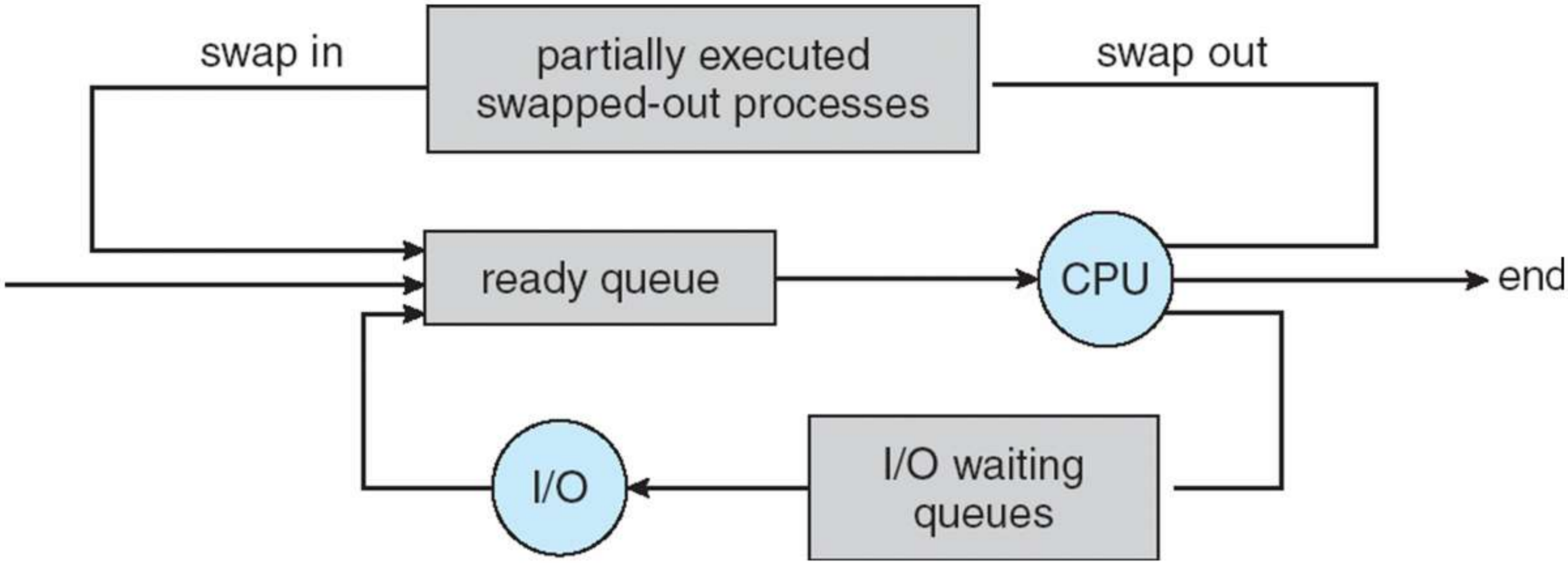


Schedulers

- **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)
  - The long-term scheduler controls the **degree of multiprogramming (the number of processes in memory)**
- 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***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
- Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**
- Swapping may be necessary
  - to improve the process mix (of CPU bound and I/O bound)
  - or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.


swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- 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 with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
  - Apple probably limits multitasking due to battery life and memory use concerns.
- Android does not place such constraints on the types of applications that can run in the background
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks

  (a separate application component that runs on behalf of the background process.)

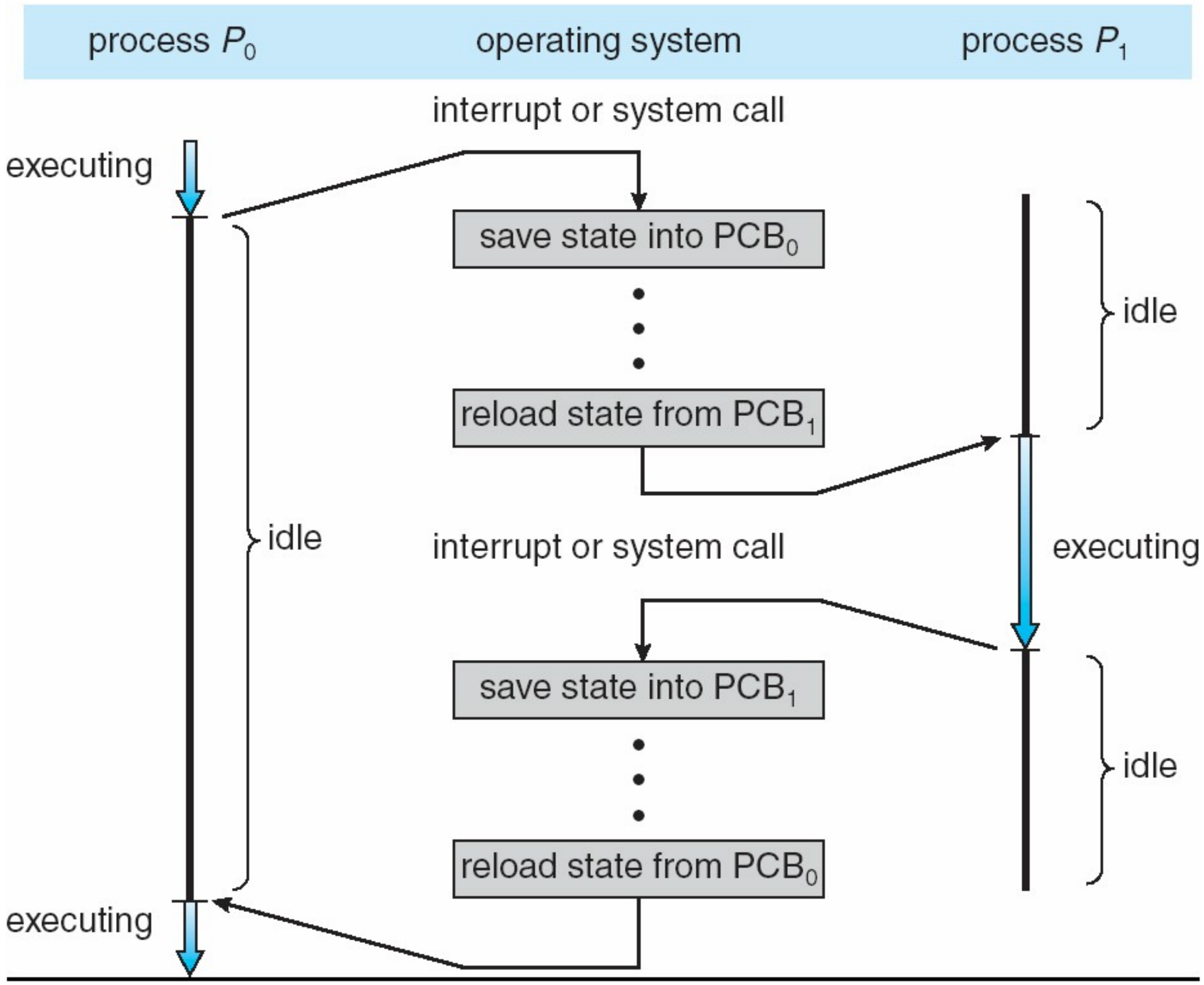
Multitasking in Mobile Systems cont.

- Consider a streaming audio application:
- if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application.
- Service can keep running even if background process is suspended
- Service has no user interface, small memory use

Context Switch

- When CPU switches to another process, the system must **save the state (in its PCB)** 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 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 (e.g. Sun UltraSPARC) 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



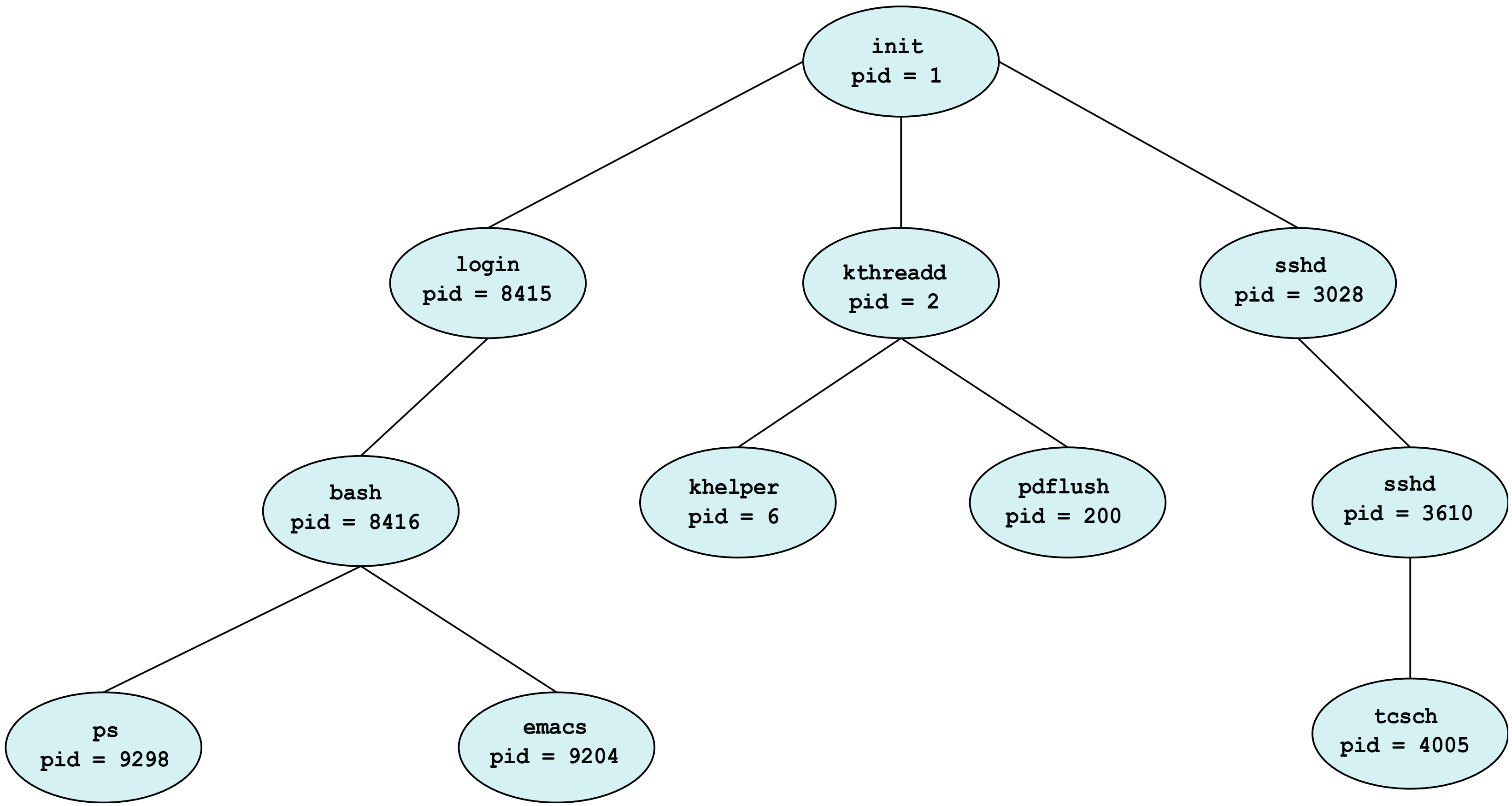
Operations on Processes

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

Process Creation

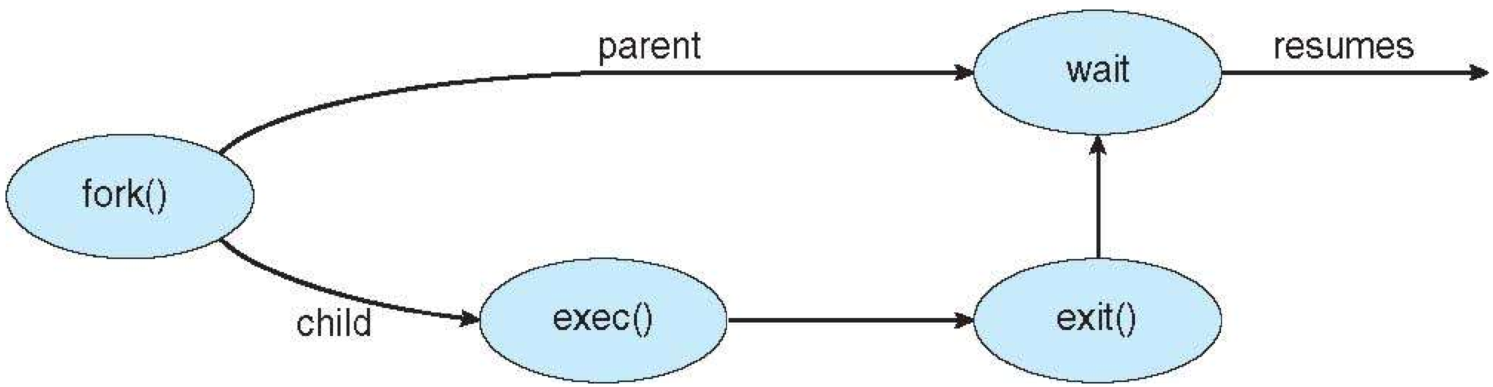
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- A child process will need certain resources (CPU time, memory, files, I/O devices)
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources (the child process obtains its resources directly from the operating system)
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux



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



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;
}
```

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);
}
```

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 a process terminates but has no parent waiting (did not invoke **`wait()`**) process is a **zombie (because its entry in the process table entry continues to exist).**
- If parent terminated without invoking **`wait`**, process is an **orphan (in Linux and Unix the init process becomes the parent which periodically invokes wait(), to allow the exit status of the orphan to be collected and thus to release the orphan's process table entry)**

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
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

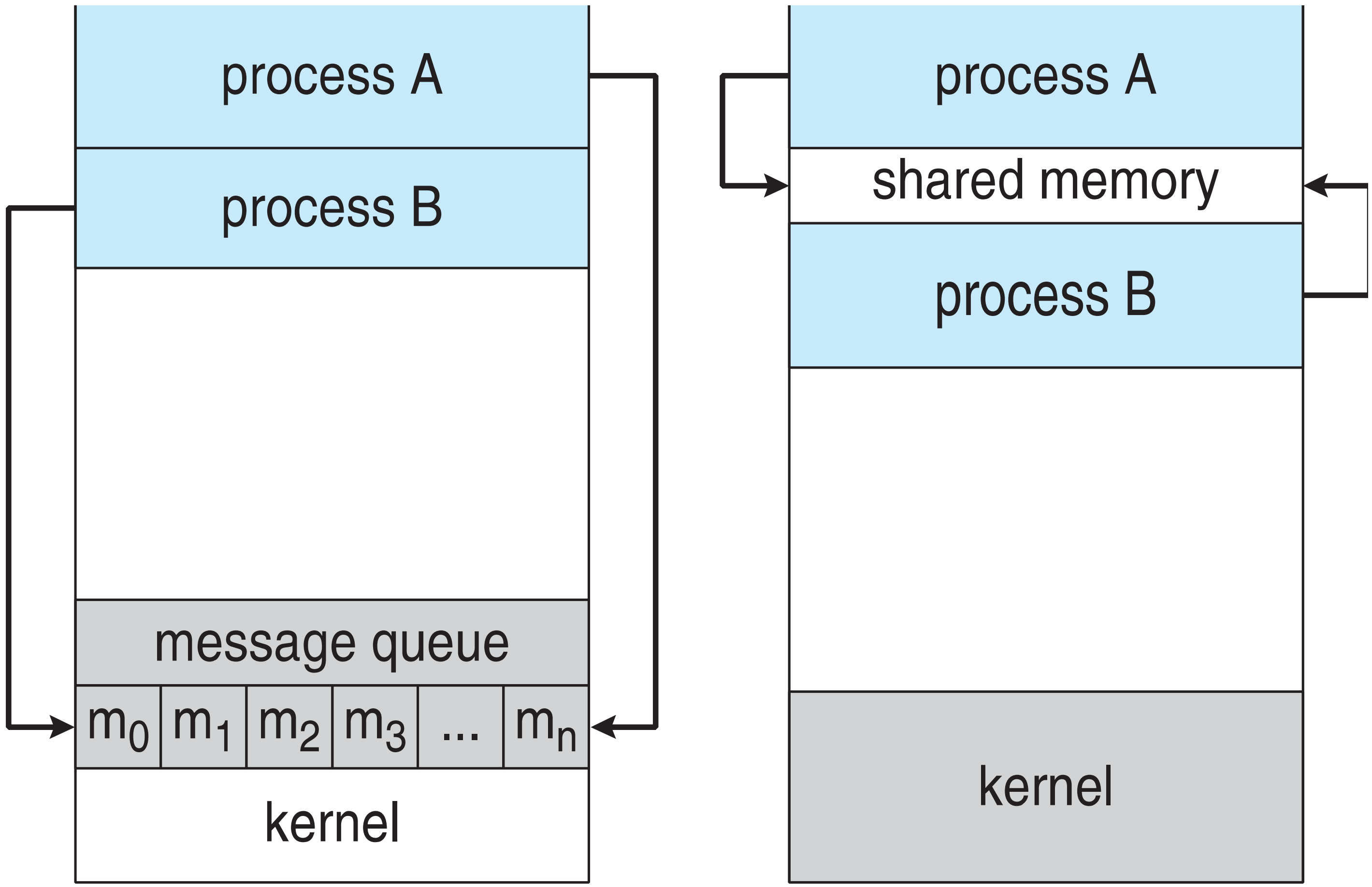
Interprocess Communication

- 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. a shared file between several users)
  - Computation speedup (to break a task into several tasks to run them n parallel)
  - Modularity (dividing the systems functions into separate processes)
  - Convenience (For instance, a user may be editing, listening to music, and compiling in parallel)

- Cooperating processes need **interprocess communication** (**IPC**) mechanism that will allow them to exchange data and information.
- Two models of IPC
  - **Shared memory**
    - **A region of memory that is shared by cooperating processes is established.**
    - **Processes can then exchange information by reading and writing data to the shared region**
  - **Message passing**
    - **communication takes place by means of messages exchanged between the cooperating processes**
    - **Useful for exchanging smaller amounts of data, because no conflicts need be avoided and**
    - **easier to implement.**
- **Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls (time consuming)**
- **However recent research show that in systems with several cores message passing is faster (due to caches coherency issues)**

Communications Models

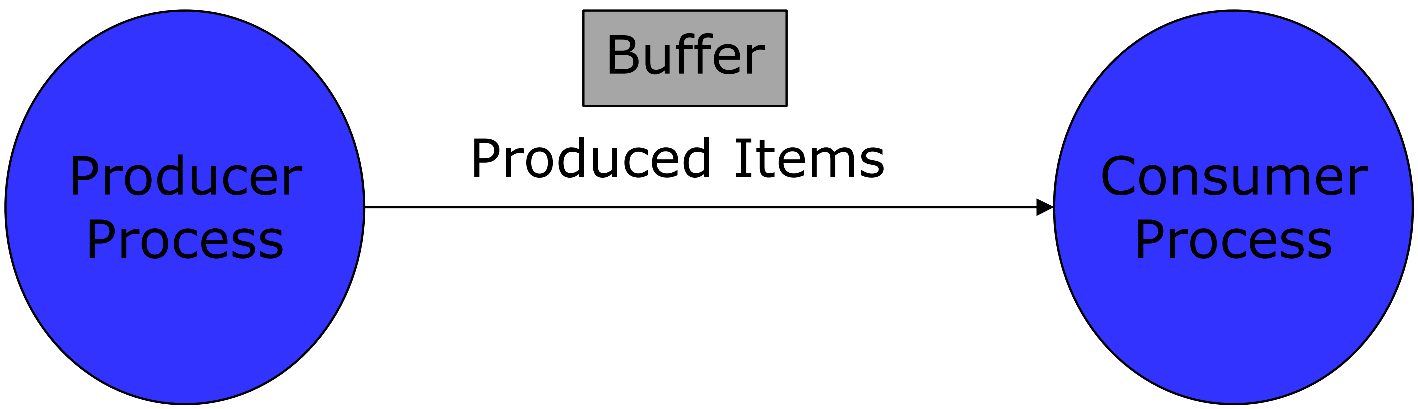
(a) Message passing. (b) shared memory.



(a) (b)

Producer-Consumer Problem

- To illustrate use of the **shared memory IPC mechanism**, a general model problem, called producer-consumer problem, cam be used.
- *producer* process produces information that is consumed by a *consumer* process, e.g., a compiler produces code and the assembler consumes it



- Client-server metaphor
  - **unbounded-buffer** places no practical limit on the size of the buffer (the consumer may have to wait but the producer does not have to wait)
  - **bounded-buffer** assumes that there is a fixed buffer size (both may have to wait; the producer has to wait if the buffer is full, while the consumer has to wait if the buffer is empty)

- The shared buffer **is implemented as a circular array** with
- two logical pointers: **in** and **out**.
- The variable **in points to the next free position** in the buffer;
- **out points to the first full position** in the buffer.
- The buffer is empty when in == out;
- the buffer is full when ((in + 1)% BUFFER SIZE) ==out.

Bounded-Buffer – Shared-Memory Solution

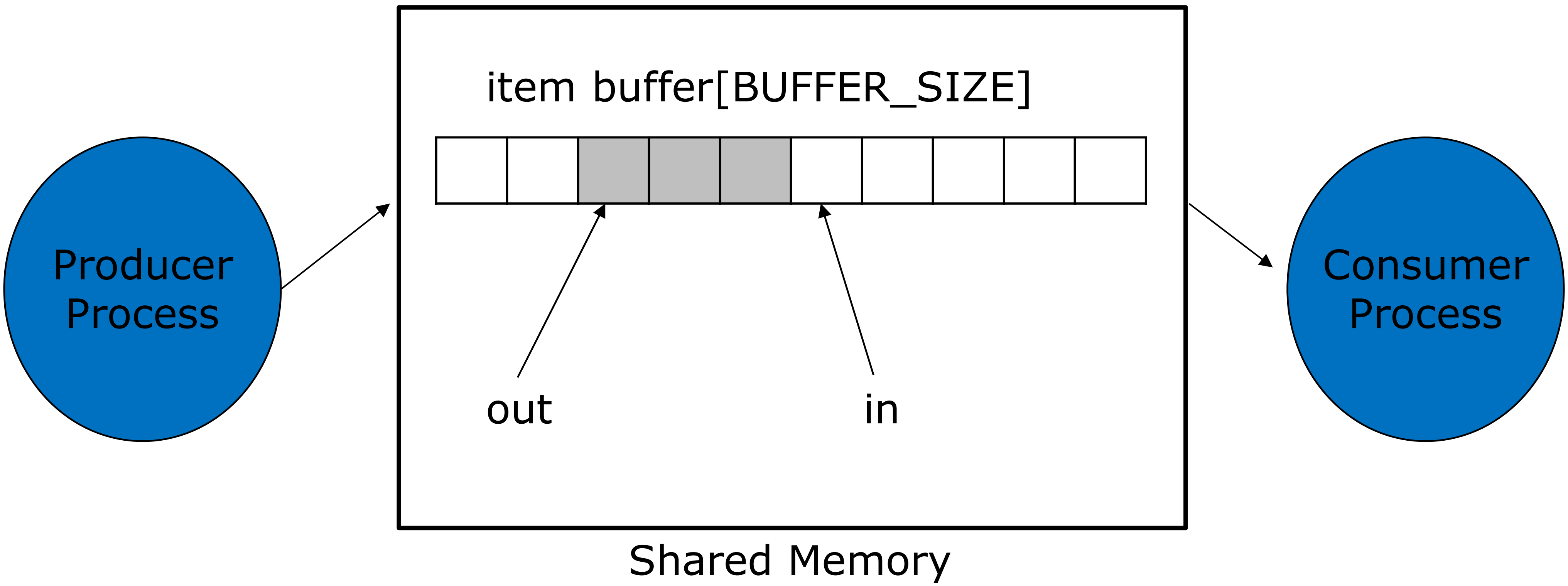
- Shared data

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

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

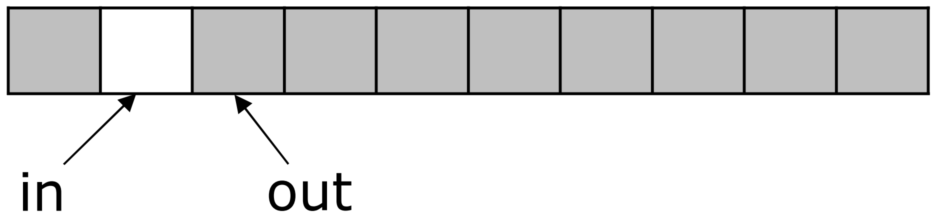
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Buffer State in Shared Memory



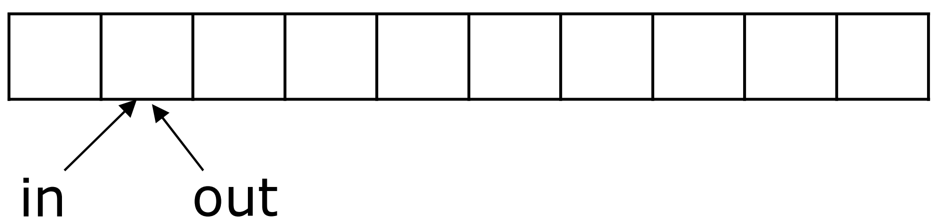
Buffer State in Shared Memory

Buffer Full



((in + 1) % BUFFER_SIZE) == out

Buffer Empty



in == out

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next_produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing - no free buffers */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing - nothing to consume */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next_consumed */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is **under the control of the users processes not the operating system**.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory (i.e. how can they access the shared buffer concurrently; solution in chap 5)
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

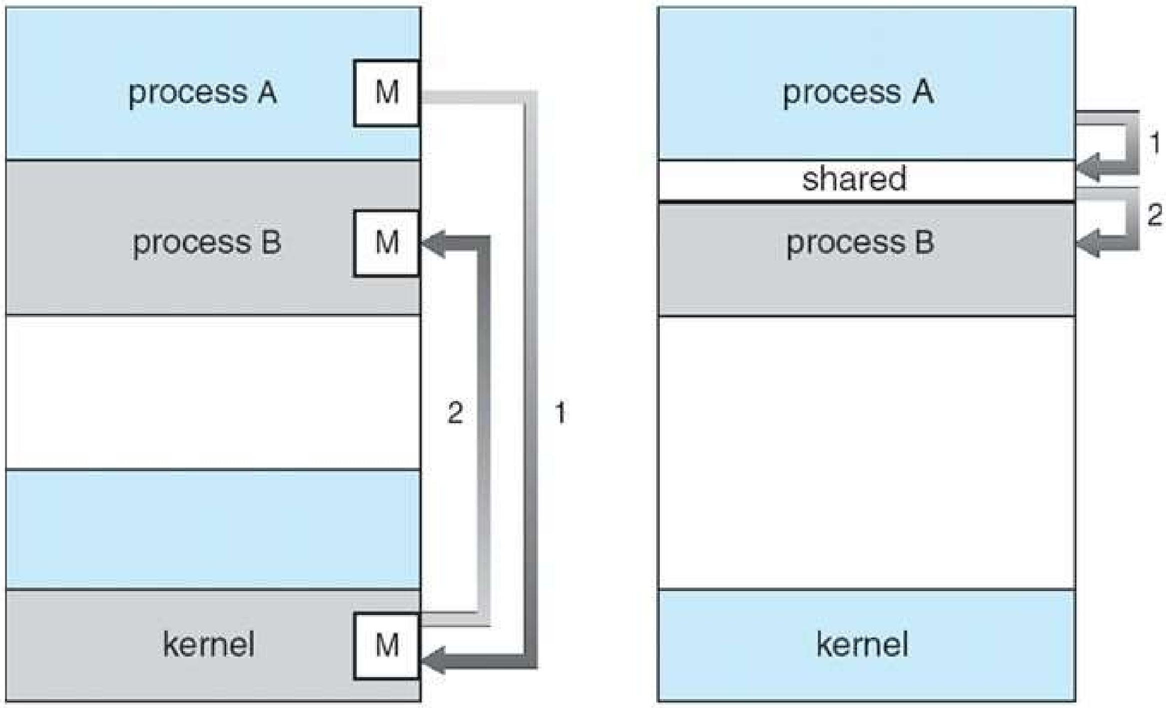
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions **without sharing the same address space.**
- It is particularly useful in **a distributed environment**,
- where the communicating processes may reside on different computers connected by a network.
- For example, **an Internet chat program**

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- Fixed-sized implementation is straightforward; however, it makes the task of programming more difficult
- Variable length messages require a more complex system level implementation, but the programming task becomes simpler
- A common tradeoff in OS design

Interprocess Communication – Summary

Communications Models



(a) Message Passing

(b) Shared Memory

End of Chapter 3