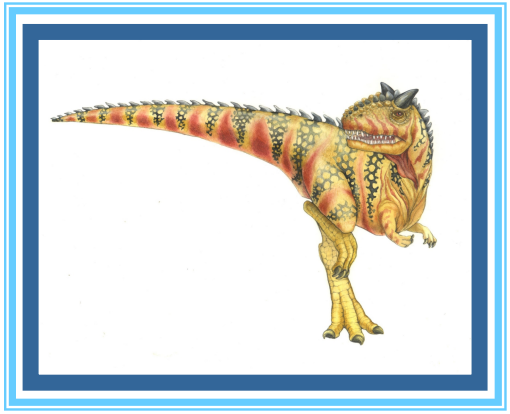
Chapter 3: Processes



Chapter 3: Processes

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

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

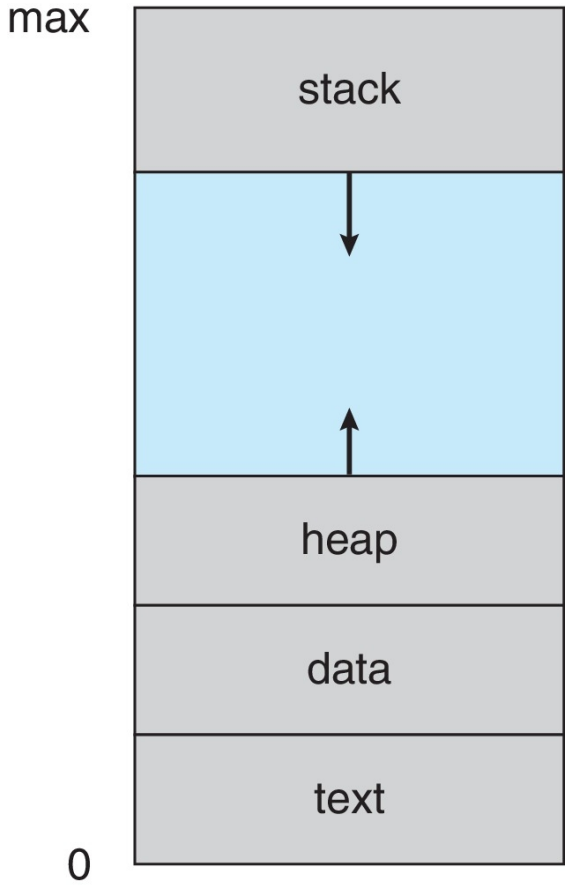
Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Process memory is divided into following sections:
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Heap** containing memory dynamically allocated during run time
  - The program code, also called **text section**
  - **Data section** containing global variables
  - Current activity including **program counter**, processor registers

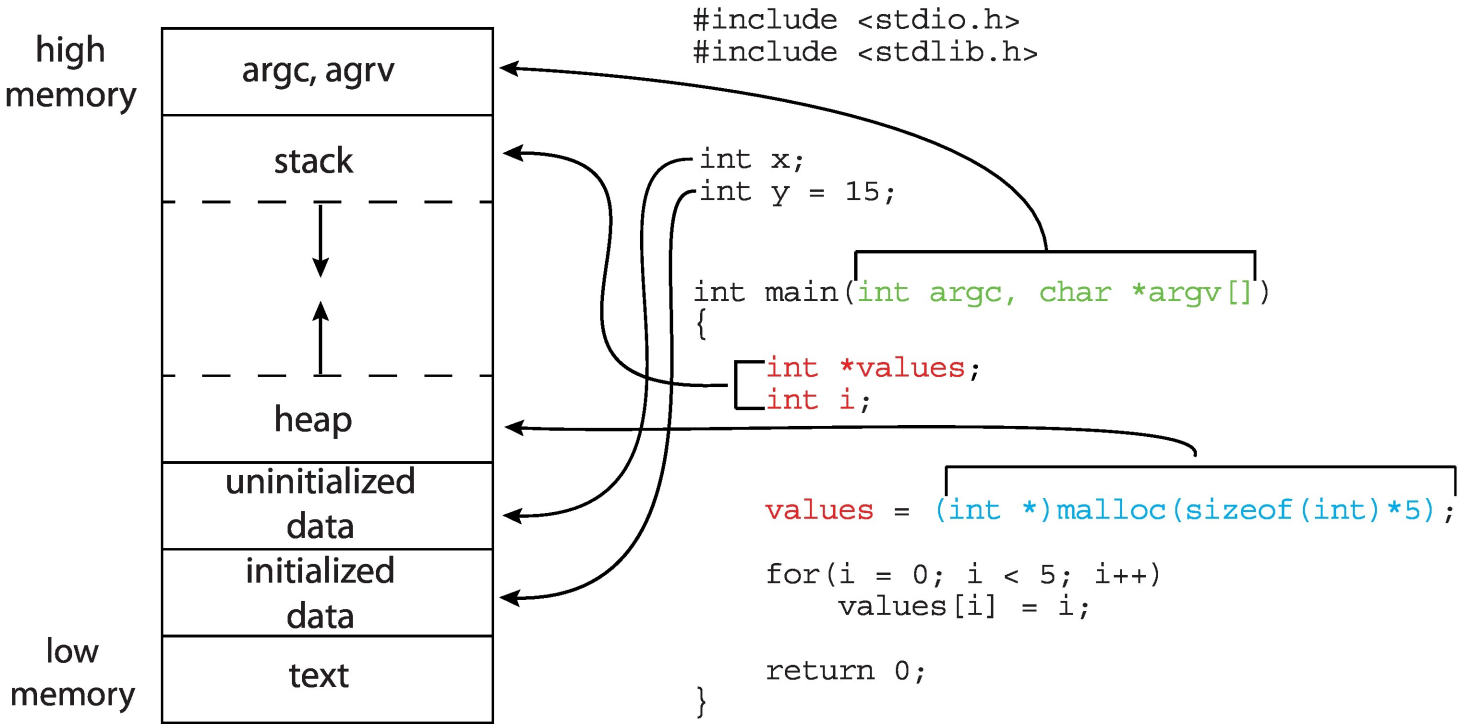
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- 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
    - Compiler
    - Text editor

Process in Memory



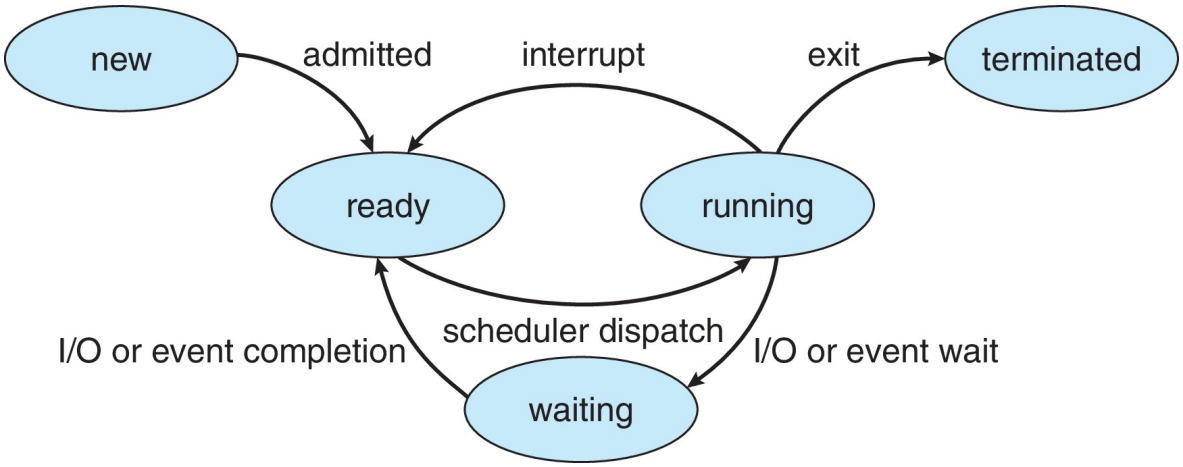
Memory Layout of a C Program



Process State

- A Process has 5 states. Each process may be in one of the these states
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Ready**: The process has all the resources it needs to run.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. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
  - **Terminated**: The process has finished execution

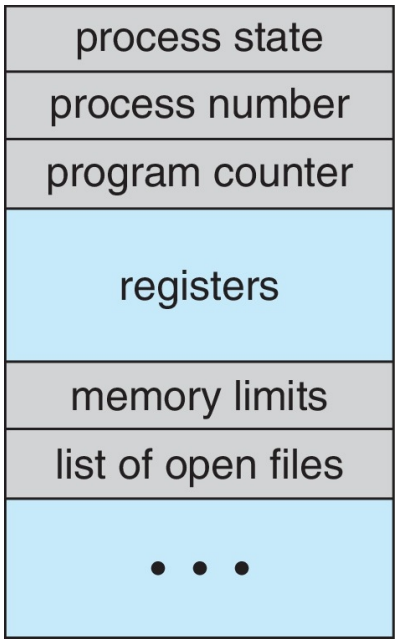
Diagram of Process State



Process Control Block (PCB)

Information associated with each process(also called **task control block**)

- Process state – state of the process. Eg. running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers. Eg. Accumulator, index regs., SP, GPRs
- CPU scheduling information- priorities, scheduling queue pointers, other scheduling parameters
- Memory-management information – memory allocated to the process, page/segment tables
- 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

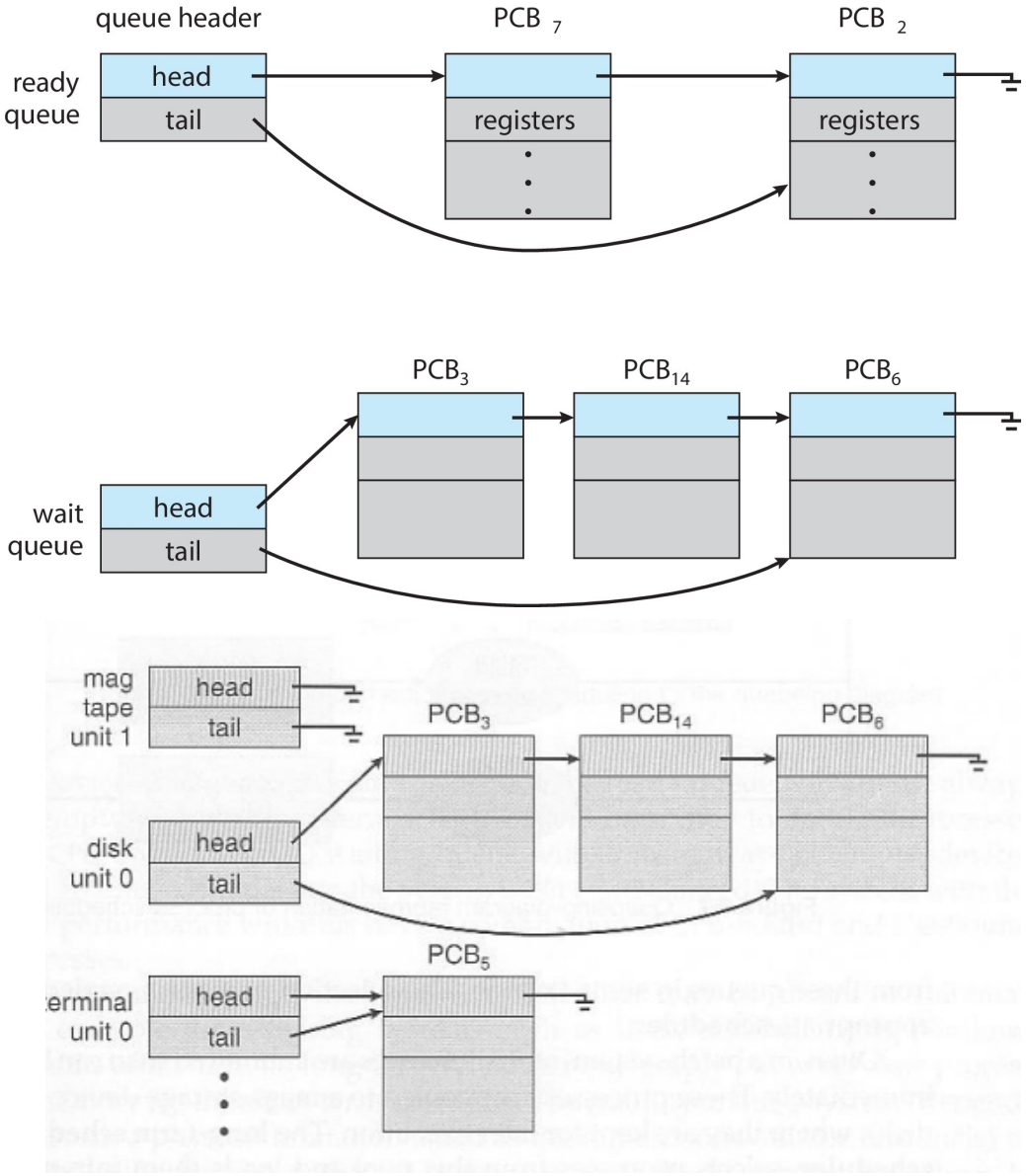
- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core.
- Goal -- Maximize CPU use by keeping CPU busy all the time, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Job queue**– set of processes admitted to the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queuesore

- Queues are stored as linked list of PCBs. Queue header contains 2 pointers- head pointer pointing to first PCB and tail pinting to the last PCB in the queue.

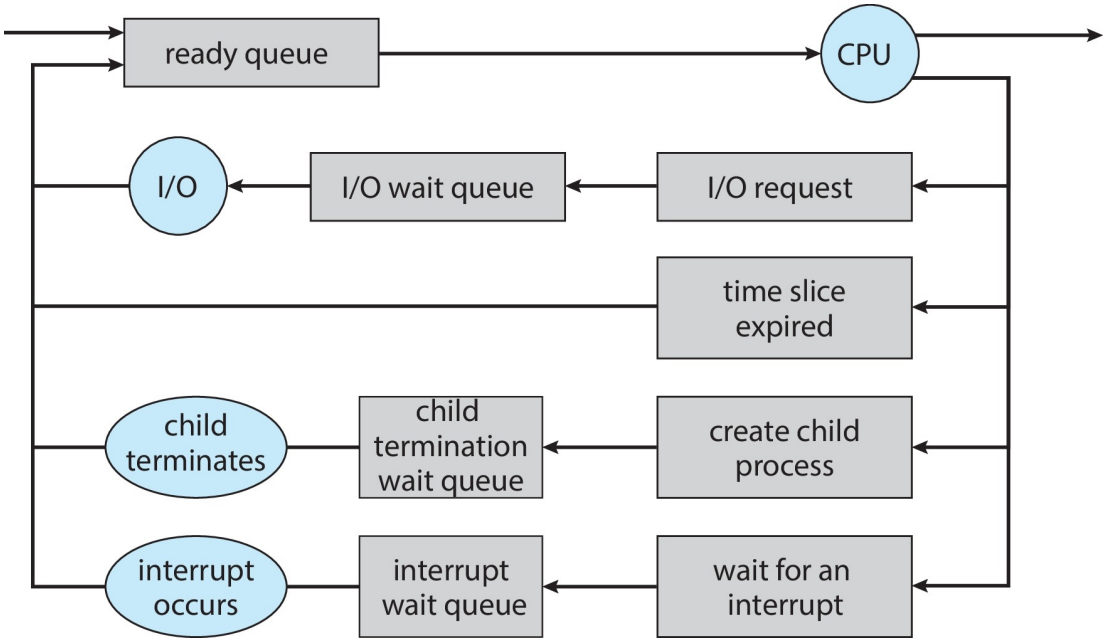
Ready and Wait Queues



Representation of Process Scheduling

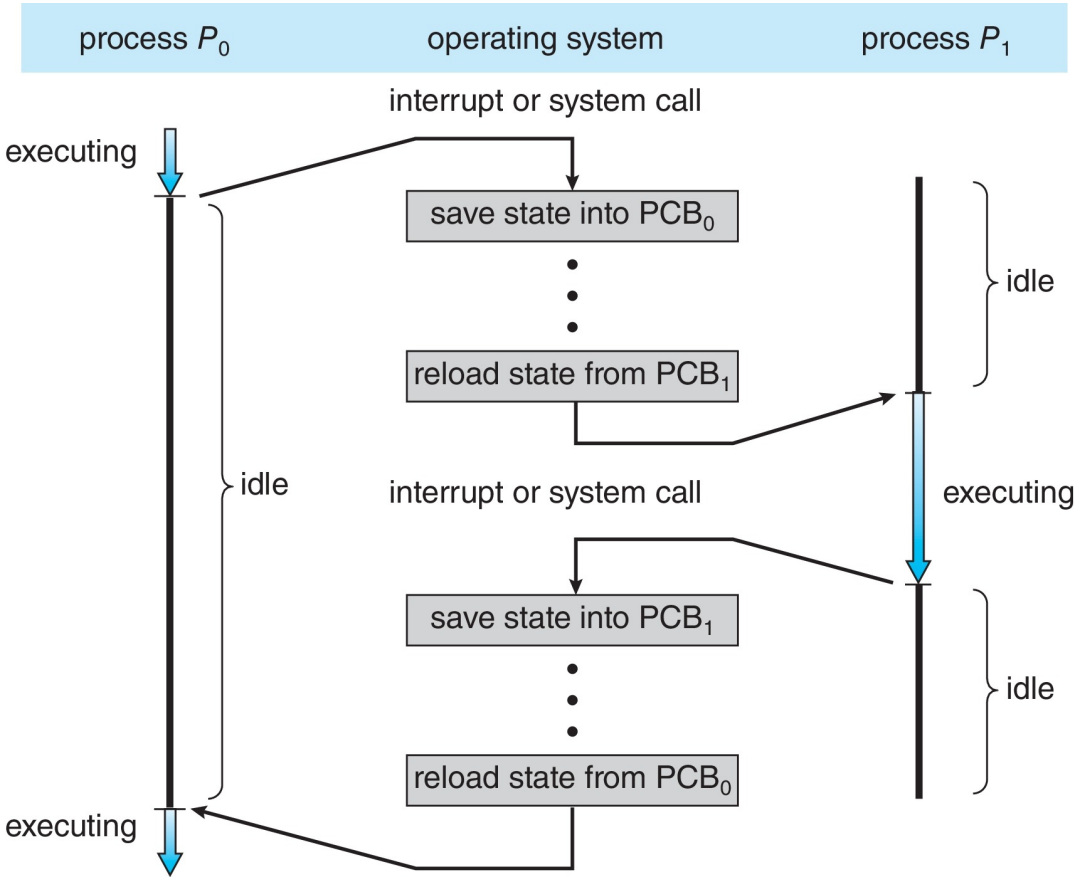
- A common representation of process scheduling is a queueing diagram.
- Each rectangular box in the diagram represents a queue. Two types of queues are present: the ready queue and a set of device queues.
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution and is given the CPU. Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request, and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Representation of Process Scheduling



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



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**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Schedulers

Schedulers are software which selects an available program to be assigned to CPU.

- **Long term scheduler or Job scheduler**: selects processes from the pool (of secondary memory disk) and loads them into the memory for execution.
  - It runs infrequently
  - will be invoked only when a process leaves the system
  - can take time to select the next process because of the longer time between the executions.
  - controls **degree of multiprogramming** - number of processes in memory
- **Short-term scheduler, or CPU Scheduler:** selects from among the processes that are ready to execute and assigns the CPU to it.
  - It must select the new process for CPU frequently.
  - Must execute atleast once every 100ms
  - Must execute very fast.

Schedulers

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations,
- CPU-bound process – spends more time doing computations and few I/O operations.

An efficient scheduling system will select a good mix of CPU-bound processes and I/O bound processes. If the scheduler selects more I/O bound process, then I/O queue will be full and ready queue will be empty.

- On some systems (time sharing UNIX), long term scheduler may be absent, all new processes are submitted to memory for the short term scheduler.

➢ the system stabilty depends on physical limitations (no. of terminals) or adjusting nature of human users

- Time sharing systems employ a **Medium-term scheduler:** It swaps out the process from ready queue and swap in the process to ready queue.

➢ When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Schedulers

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce the degree of multiprogramming (number of processes in memory).
- To make a proper mix of processes(CPU bound and I/O bound)

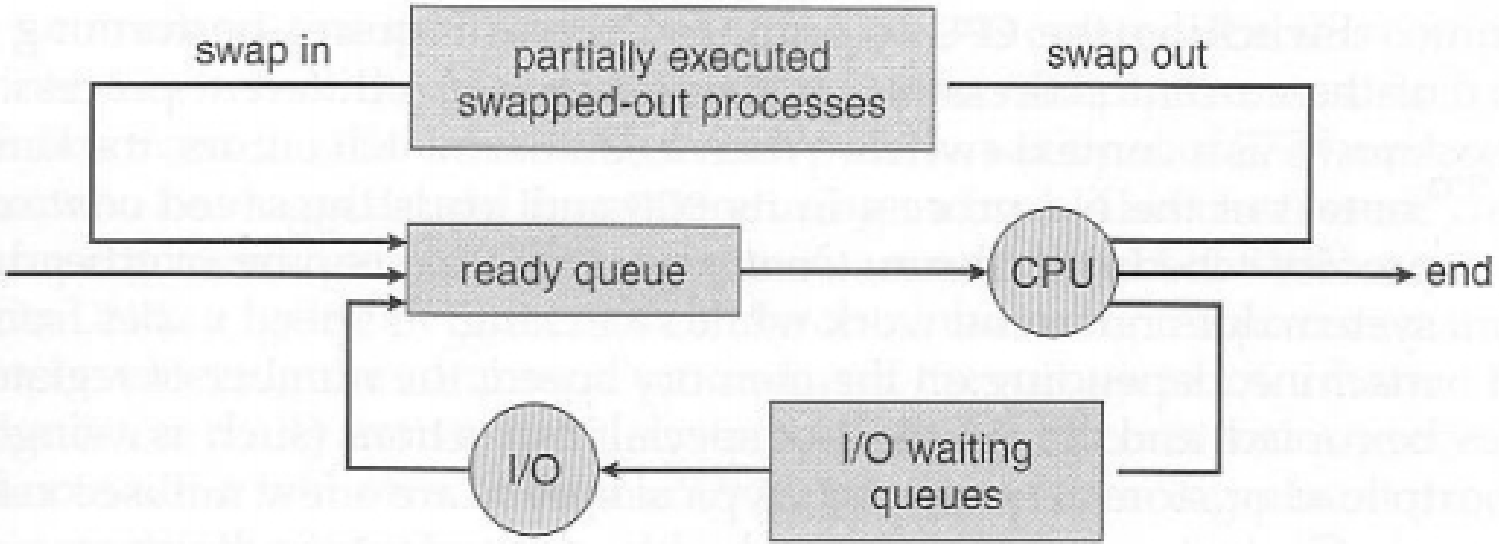


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

- **Parent** process create **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

Process Creation

- Process tree on Solaris system

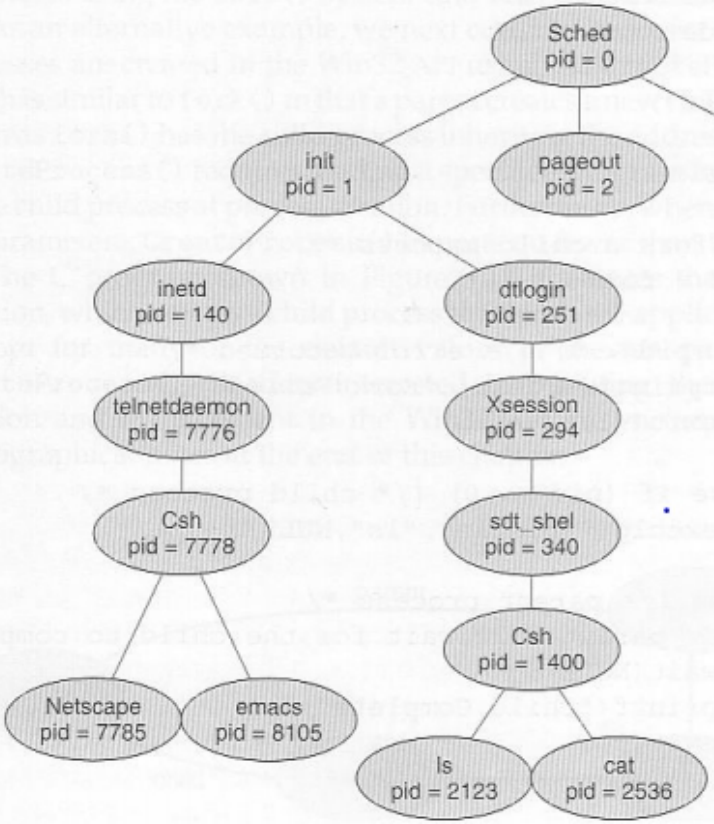
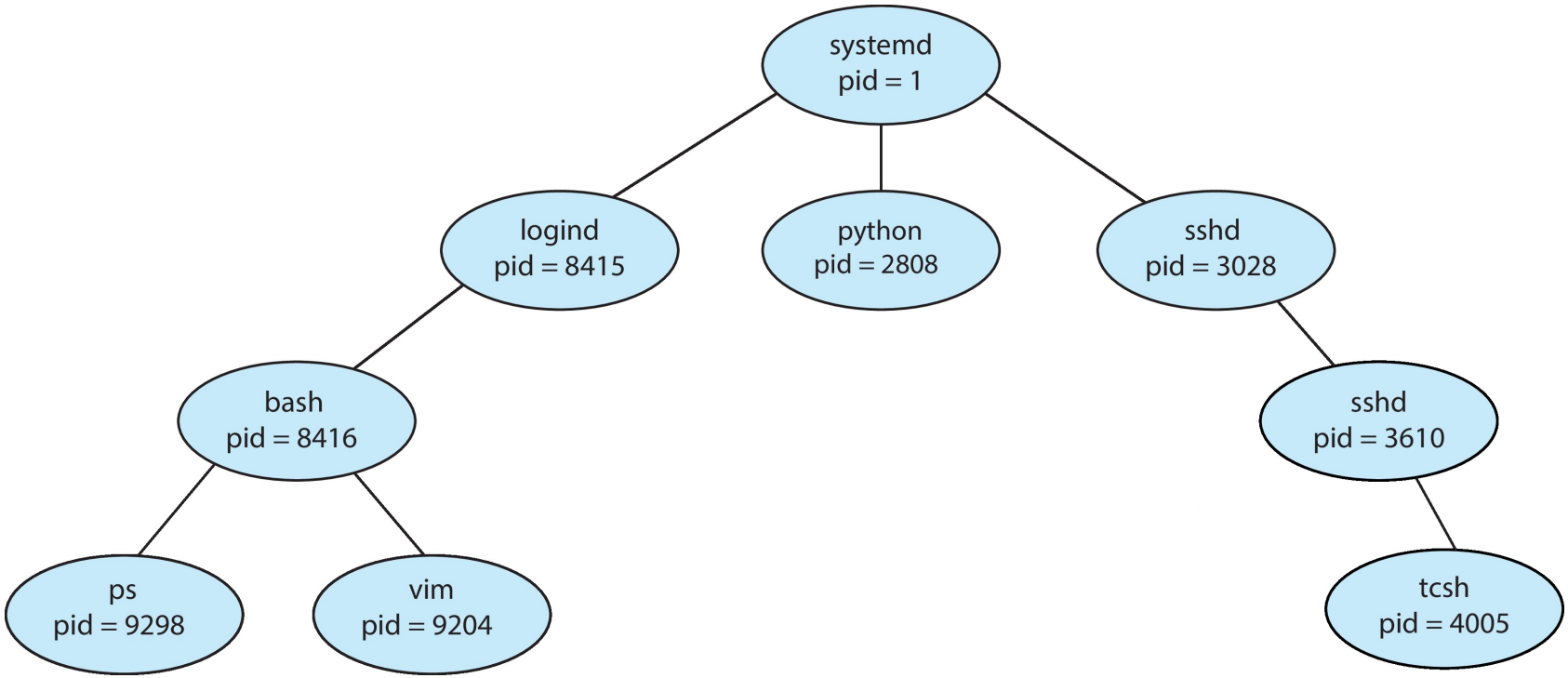


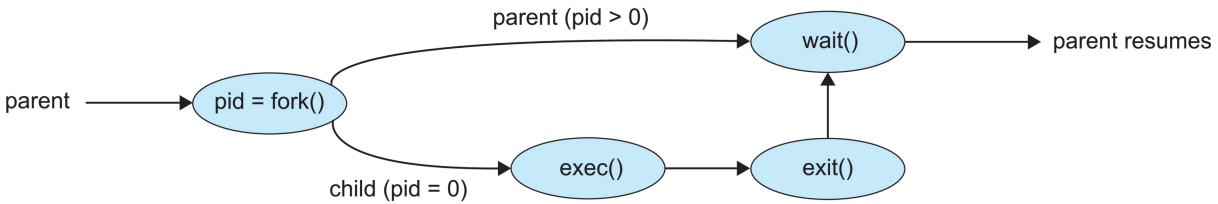
Figure 3.9 A tree of processes on a typical Solaris system.

A Tree of Processes in Linux

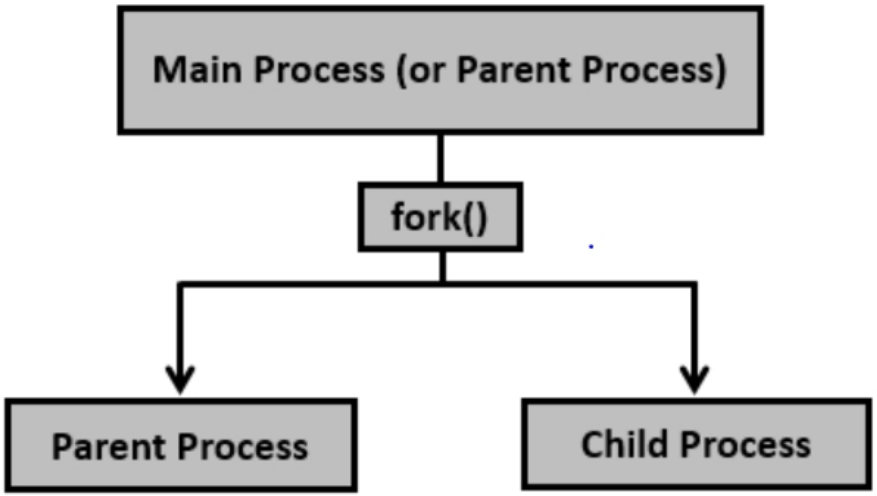


Process Creation (Cont.)

- 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
  - Parent process calls **`wait()`** waiting for the child to terminate



Process Creation (Cont.)



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 exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- 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 no parent waiting (did not invoke **`wait()`**) process is a **zombie.** It has completed but it still has an entry in the process table. They do not use any resources but retain their process id.
- If parent terminated without invoking **`wait()`**, process is an **orphan.** These are the processes that are still running even though their parent process has terminated.

Interprocess Communication

- Processes executing may be either **co-operative** or **independent** processes.
  - Independent Processes – processes that cannot affect other processes or be affected by other processes executing in the system.
  - Cooperating Processes – processes that can affect other processes or be affected by other processes executing in the system; including sharing data
- Reasons for cooperating processes:
  - Information sharing: There may be several processes which need to access the same file. So the information must be accessible at the same time to all users.
  - Computation speedup: Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks, which are solved simultaneously (particularly when multiple processors are involved)

Interprocess Communication

- Modularity: A system can be divided into cooperating modules and executed by sending information among one another. A system can be constructed in a modular approach divng the sysstem functions into separate processes or threads.
- Convenience: Even a single user can work on multiple task by information sharing. Eg: A user may be editing, compiling and printing at the same time.

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory** (under the control of users): share a common buffer pool, and the code for implementing buffer is explicitly written by user.
  - **Message passing** (under the control of OS): OS provides means of communication between cooperating processes.

Communications Models

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

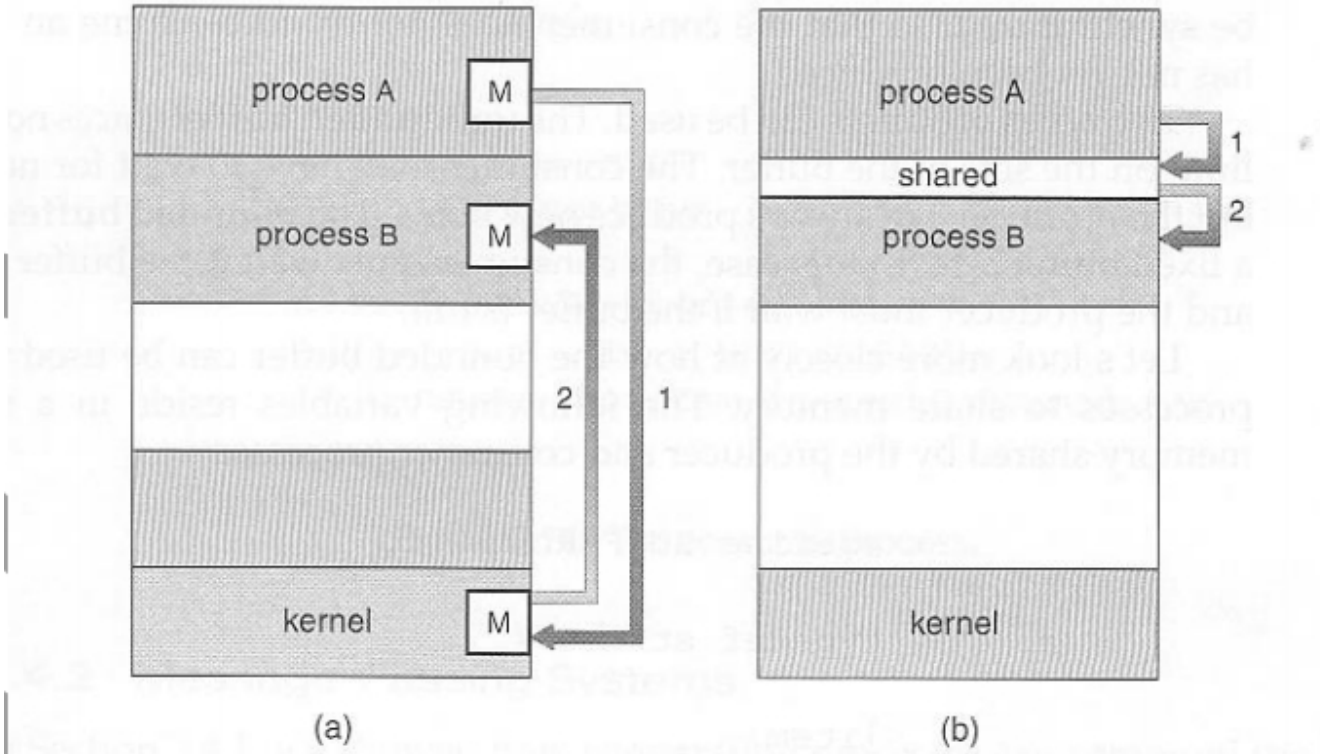
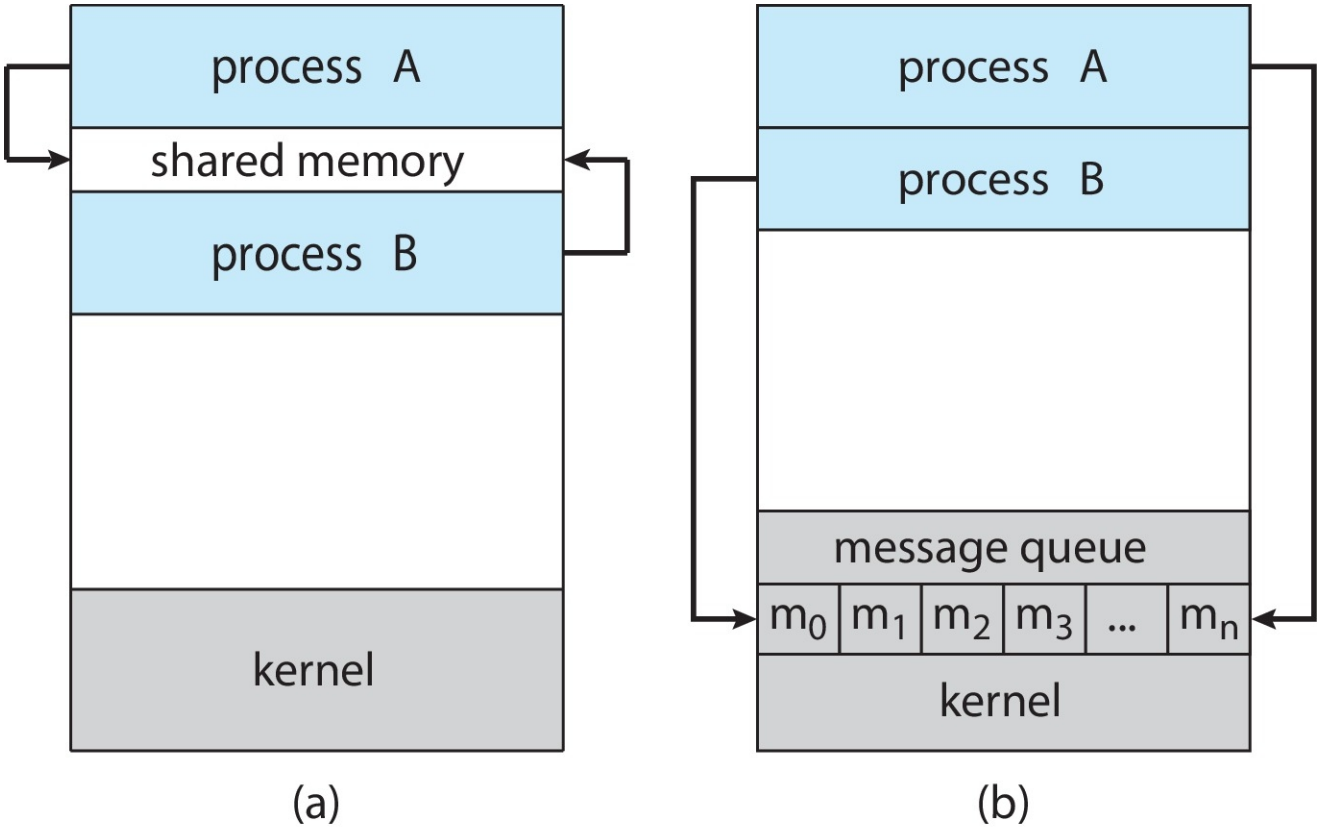


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Communications Models

(a) Shared memory.

(b) Message passing.



Shared Memory Vs. Message passing

Shared Memory

- A region of memory is shared by communicating processes, into which the information is written and read
- Useful for sending large block of data
- System call is used only to create shared memory
- Message is sent faster, as there are no system calls

Message passing

- Message exchange is done among the processes by using objects.
- Useful for sending small data.
- System call is used during every read and write operation.
- Message is communicated slowly.

Shared Memory

- An area of memory shared among the processes that wish to communicate
- A region of shared-memory is created within the address space of a process, which needs to communicate.
- Other processes that needs to communicate uses this shared memory.
- The form of data and position of creating shared memory area is decided by the process.
- 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.

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - The data is passed via an intermediary buffer (shared memory).
  - ***producer*** process produces information that is consumed by a ***consumer*** process.
  - A producer can produce one item while the consumer is consuming another item.
  - The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer may have to wait for new items.
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no item to consume

Bounded-Buffer – Shared-Memory Solution

- This example uses shared memory as a circular queue. The in and out are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".
- Set up Shared Memory

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

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

Producer Process – Shared Memory

- Note that the buffer is full when [(in+1)%BUFFER_SIZE == out]

```
item next_produced;

while (true)
{
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

Note that the buffer is empty when [in == out]

```
item next_consumed;

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

        /* consume the item in next consumed */
}
```

This scheme allows at most BUFFER_SIZE-1 i.e., 9 items in the buffer at the same time .

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The integer **`counter`** is and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)
{
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;
        /* consume the item in next consumed */
}
```

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- Particularly useful in distributed environment where the communicating processes may reside on different computers connected with a network.

- IPC provides message passing with two operations:
  - **send**(*message*)
  - **receive**(*message*)

- The *message* size is either fixed or variable
- fixed size message : the s/m level implementation is simple but the task of programming becomes difficult.
- variable sized message : more complex s/m level implementation but programming task becomes simpler.

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via send/receive
- Implementation issues:
  - 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?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical: methods to logically implement a link and send/receive operations are
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Naming: The processes that want to communicate should have a way to refer eachother. (using some identity)
- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- 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 may be unidirectional, but is usually bi-directional

Direct Communication

- Types of addressing in direct communication –
- Symmetric addressing – the above described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender name is mentioned, but the receiving data can be from any system.

  send(P, message) --- Send a message to process P

  receive(id, message). Receive a message from any process
- Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system(sender and receiver), where the messages are sent and received.

Indirect Communication

- uses shared mailboxes or ports
- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Mailbox is an object into which messages can be sent and received. It has a unique ID. Using this identifier messages are sent and received.
- Two processes can communicate only if they have a shared mailbox.
- The send and receive functions are –

  send(A, message) – send a message to mailbox A

  receive(A, message) – receive a message from mailbox A

Indirect Communication (Cont.)

- Properties of communication link
  - A link is established between a pair of processes only if they have a shared 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
- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox

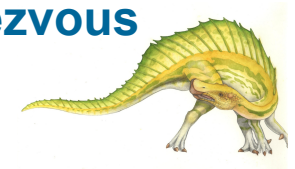
Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 each execute *receive* from A
  - 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 (either P2 or P3). Sender is notified who the receiver was.

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 resumes opeeration
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true) {
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed)

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

Buffering

- When messages are passed, a temporary queue is created. Such queue can be of three capacities:
- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.
  2. Bounded capacity – The queue is of fixed size(n). Senders must block if the queue is full. After sending ‘n’ bytes the sender is blocked.
  3. Unbounded capacity – The queue is of infinite capacity. The sender never blocks.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```

    - O_CREAT | O_RDWR : the shared-memory object is to be created if it does not yet exist (O_CREAT) and that the object is open for reading and writing (O_RDWR).
    - O_RDONLY: Open for read access only.
    - O_RDWR: Open for read or write access.
    - The last parameter establishes the directory permissions of the shared-memory object.
  - Set the size of the object

    ```
    ftruncate(shm_fd, 4096);
    ```

Examples of IPC Systems - POSIX

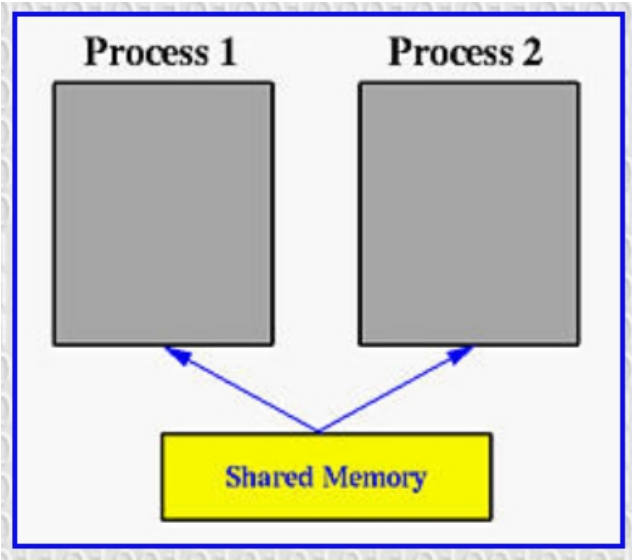
- Mapping a Shared Memory Object
- Use `mmap()` to memory-map a file pointer to the shared memory object
- Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

  pa=mmap(addr, len, prot, flags, fildes, off)
- The mmap() - establishes a mapping between the address space of the process at an address “pa” for “len” bytes to the memory object represented by the file descriptor “fildes” at offset “off” for “len” bytes.

Eg: psm = mmap(0, 1024, PROT_WRITE, MAP_SHARED, shmid1, 0);

Examples of IPC Systems - POSIX

- Suppose process 1 and process 2 have successfully attached the shared memory segment.
- This shared memory segment will be part of their address space, although the actual address could be different (i.e., the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).



IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

shmget()

- **shmget()** is used to obtain a shared memory identifier
- #include<sys/types.h>
- #include<sys/ipc.h>
- #include<sys/shm.h>
- int shmget(key_t key, int size, int flag);
- shmget() returns a shared memory ID if OK, -1 on error
- Key is typically the constant “IPC_PRIVATE”, which lets the kernel choose a new key – keys are non-negative integer identifier, but unlike fds they are system-wide, and their value continually increases to a maximum value, where it then wraps around to zero
- Size is the size of shared memory segment in bytes
- Flag can be “SHM_R”, “SHM_W” or “SHM_R | SHM_W”

shmat()

- **shmat()**
- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat():
- void *shmat(int shmid, void* addr, int flag);
- shmat() returns a pointer to shared memory segment if OK, -1 on error
- **shmdt()**
- shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process.
- **shmctl()** performs the control operation specified by cmd on the shared memory segment whose identifier is given in shmid.
- if cmd is IPC_RMID then Mark the segment to be destroyed. The segment will actually be destroyed only after the last process detaches it. The caller must be the owner or creator of the segment, or be privileged.

End of Chapter 3