OPERATING SYSTEMS

Module-2 Part1: Process Management

Anuradha Pai
Assistant Prof
BTI College of Engineering

Table Of Contents

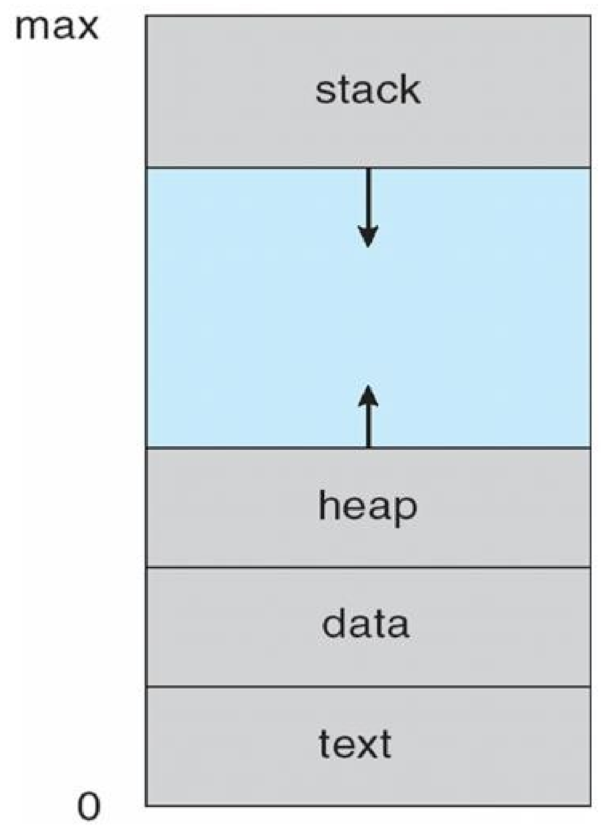
1. Process Concept

- 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
- A process is a program under execution.
- Its current activity is indicated by PC (Program Counter) and the contents of the processor's registers.

1.1 Process in Memory

- Process memory is divided into four sections as shown in the figure below:
  - The **stack** is used to store temporary data such as local variables, functio parameters, function return values, return address etc.
  - The **heap** which is memory that is dynamically allocated during process run time
  - The **data** section stores global variables.
  - The **text** section comprises the compiled program code.
- Note that, there is a free space between the stack and the heap. When the stack is full, it grows downwards and when the heap is full, it grows upwards.

1.1 Fig: Process in Memory

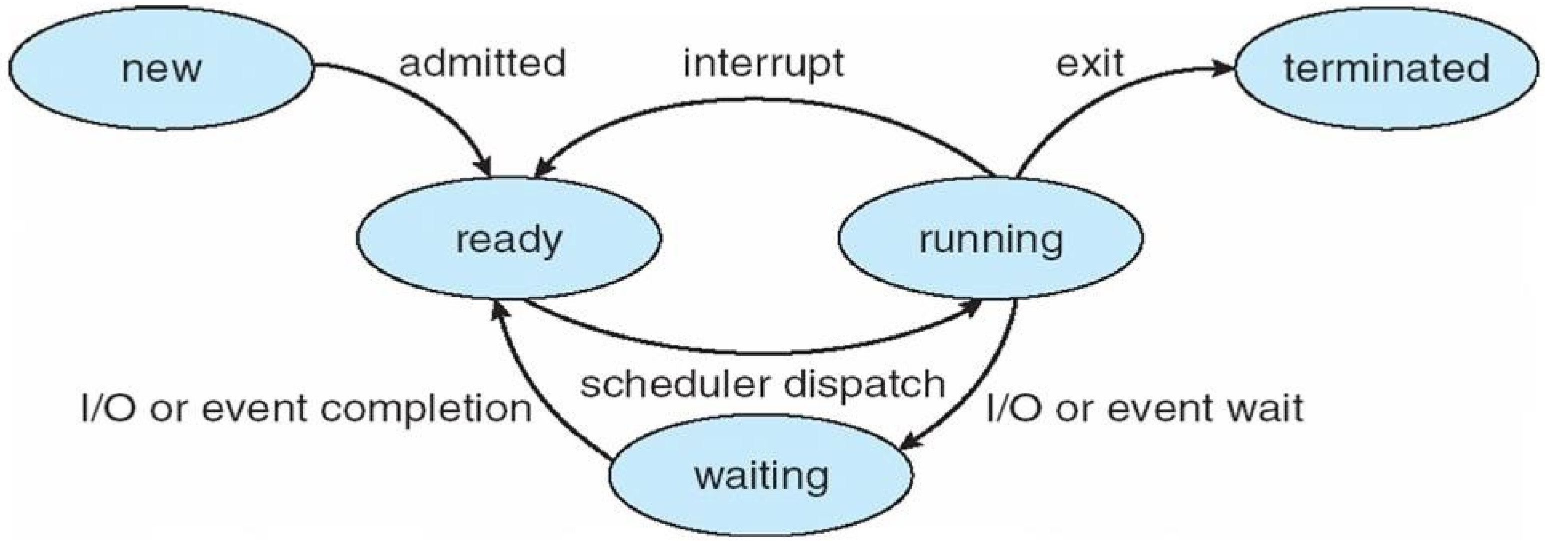


1.2 Process State

A Process has 5 states. Each process may be in one of the following states –

1. **New** - The process is in the stage of being created.
2. **Ready** - The process has all the resources it needs to run. It is waiting to be assigned to the processor.
3. **Running** – Instructions are being executed.
4. **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.
5. **Terminated** - The process has completed its execution.

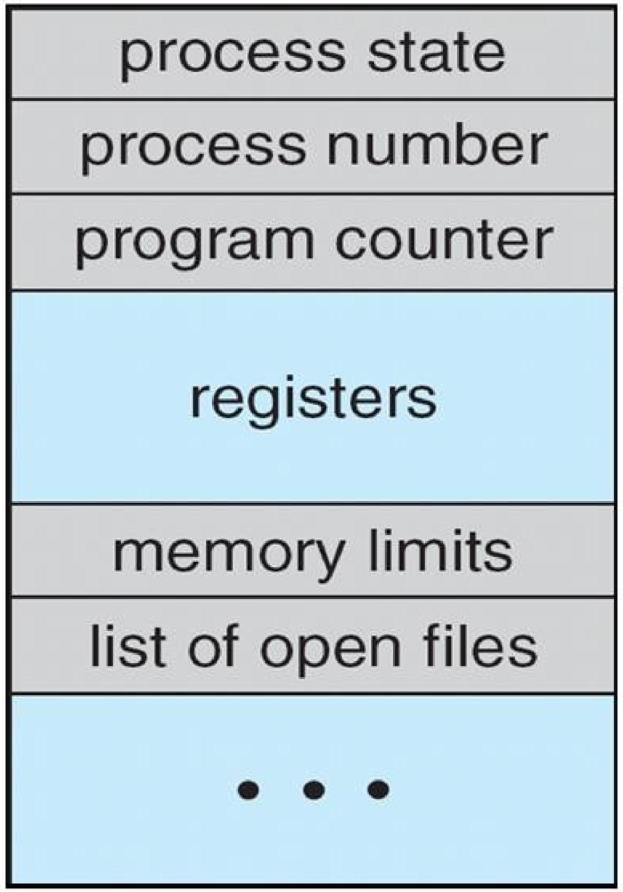
1.2 Fig: Diagram of Process State



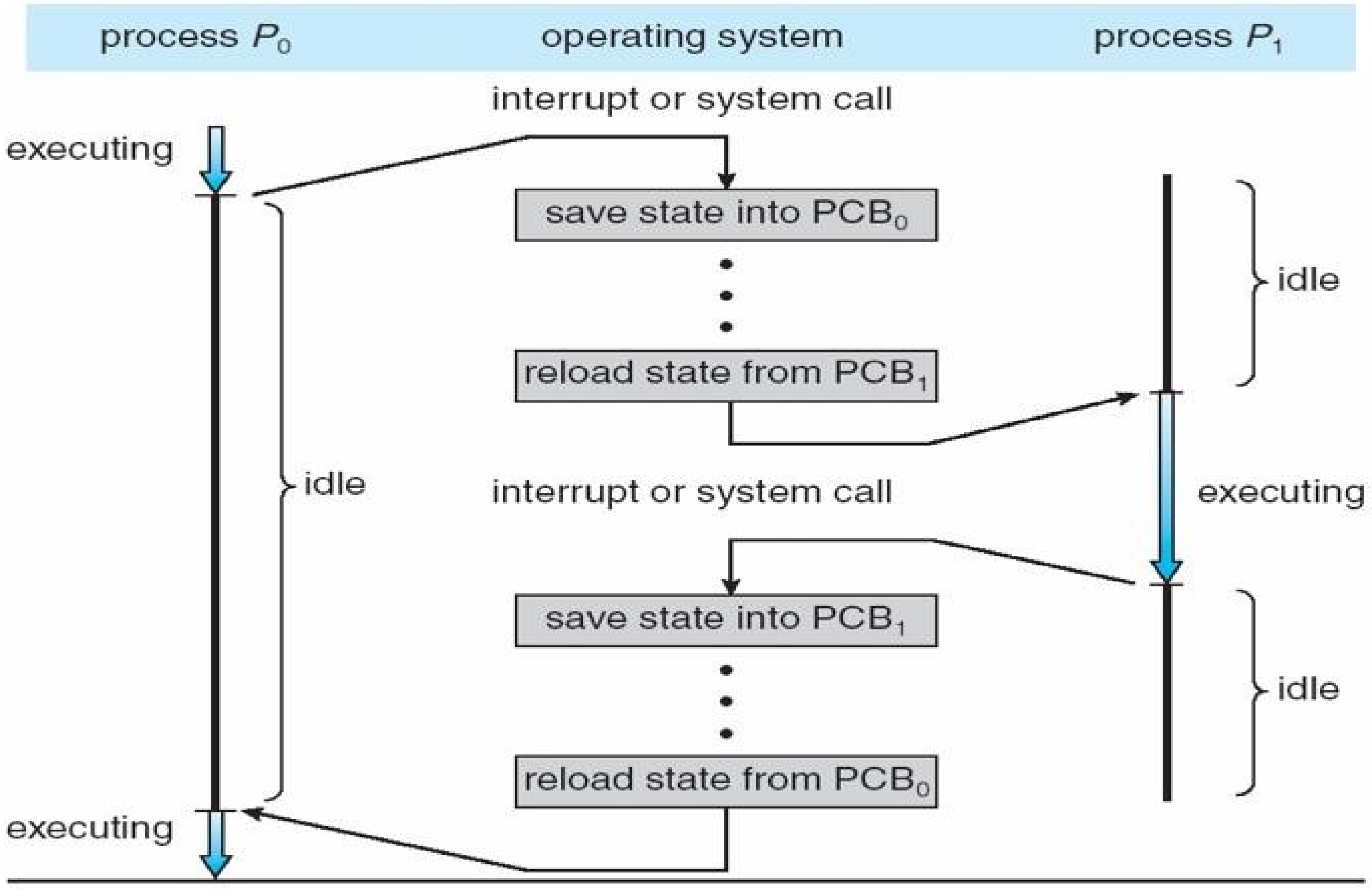
1.3 Process Control Block

- For each process there is a Process Control Block (PCB), which stores the process-specific information as shown below –
- **Process State** – The state of the process may be new, ready, running, waiting, and so on.
- **Program counter** – The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU scheduling information-** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – This includes information such as the value of the base and limit registers, the page tables, or the segment tables.
- **Accounting information** – This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information** – This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

1.3 Fig: Process Control Block(PCB)



1.4 CPU Switch From Process to Process



- When a CPU switches from one process to another, it saves the current process's state information in a structure called the process control block (PCB). This includes the program counter, CPU registers, and memory management details.
- After saving the current state, the CPU retrieves the new process's state from its PCB and continues execution from the point it was paused

1.5 Threads

- Traditional processes operate on a single thread of execution, meaning they can handle only one task at a time. In contrast, multithreaded processes can execute multiple threads simultaneously, allowing for concurrent execution of tasks (e.g., typing while running a spell checker).
- To support multithreading, the process control block (PCB) is expanded to include information for each thread.

2. Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled

2.1 Scheduling Queues

- **Scheduling Queues**
    - As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
    - The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list.
        - A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
    - **Device queues** – set of processes waiting for an I/O device
    - Processes migrate among the various queues

Fig: Ready Queue And Various I/O Device Queues

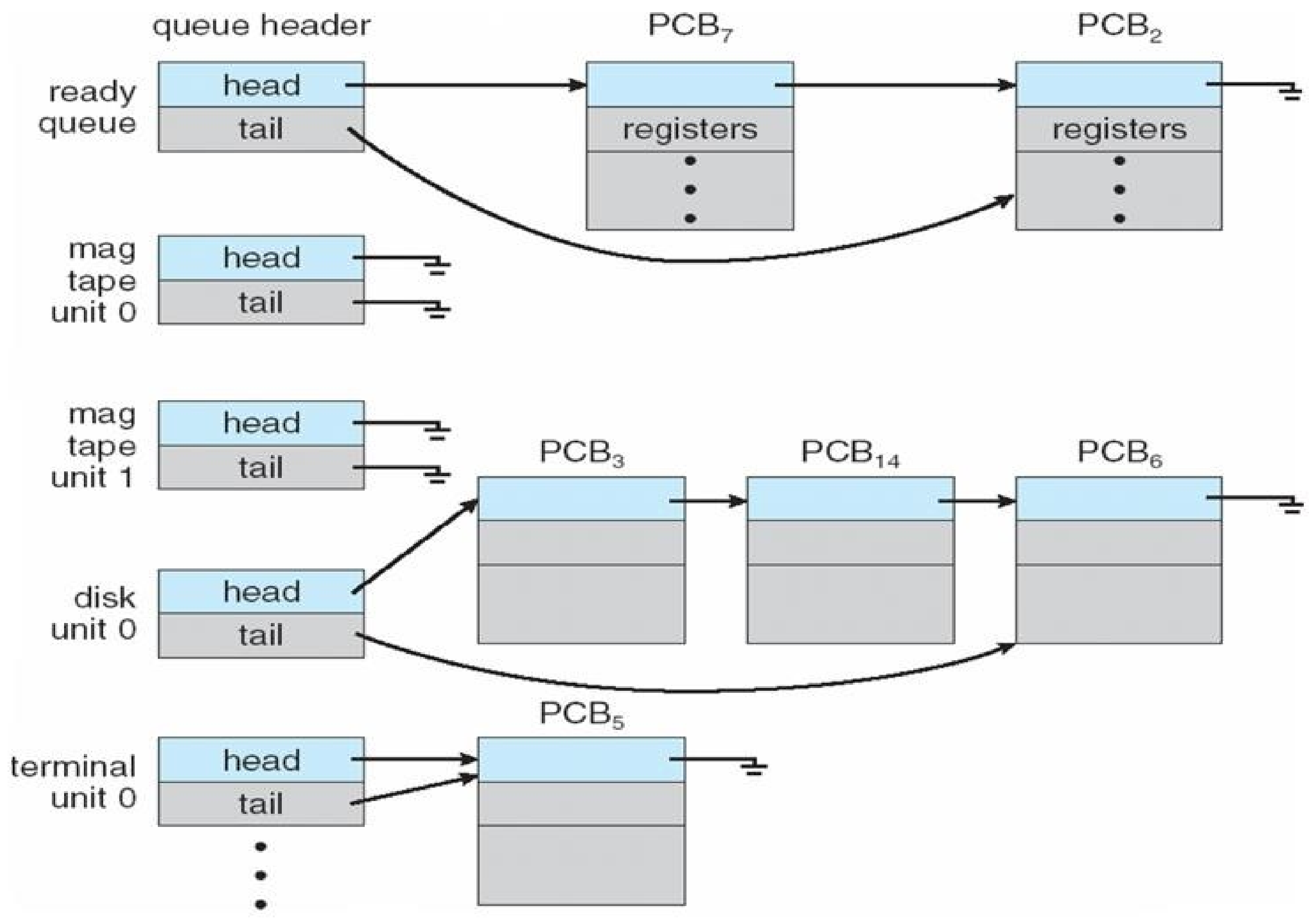
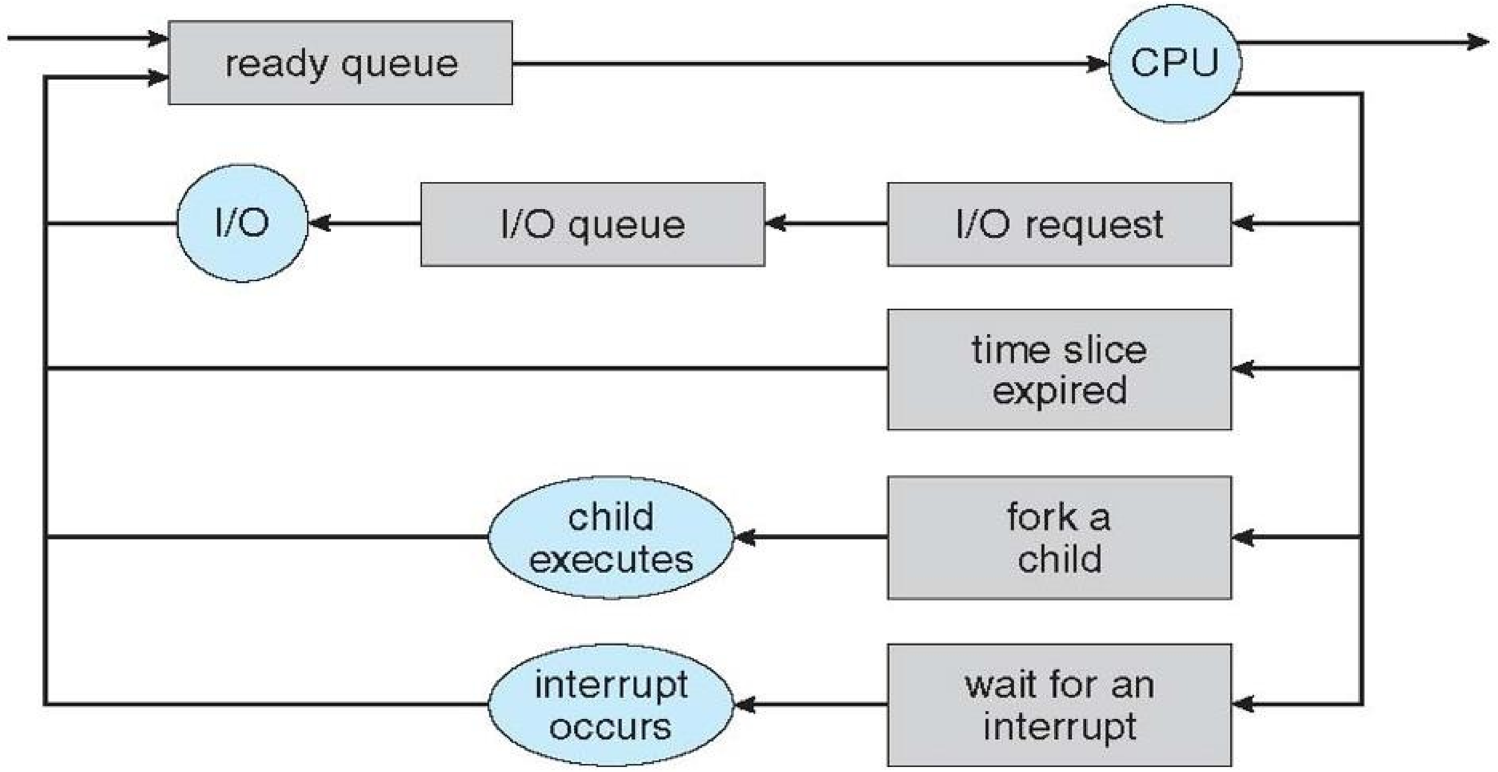


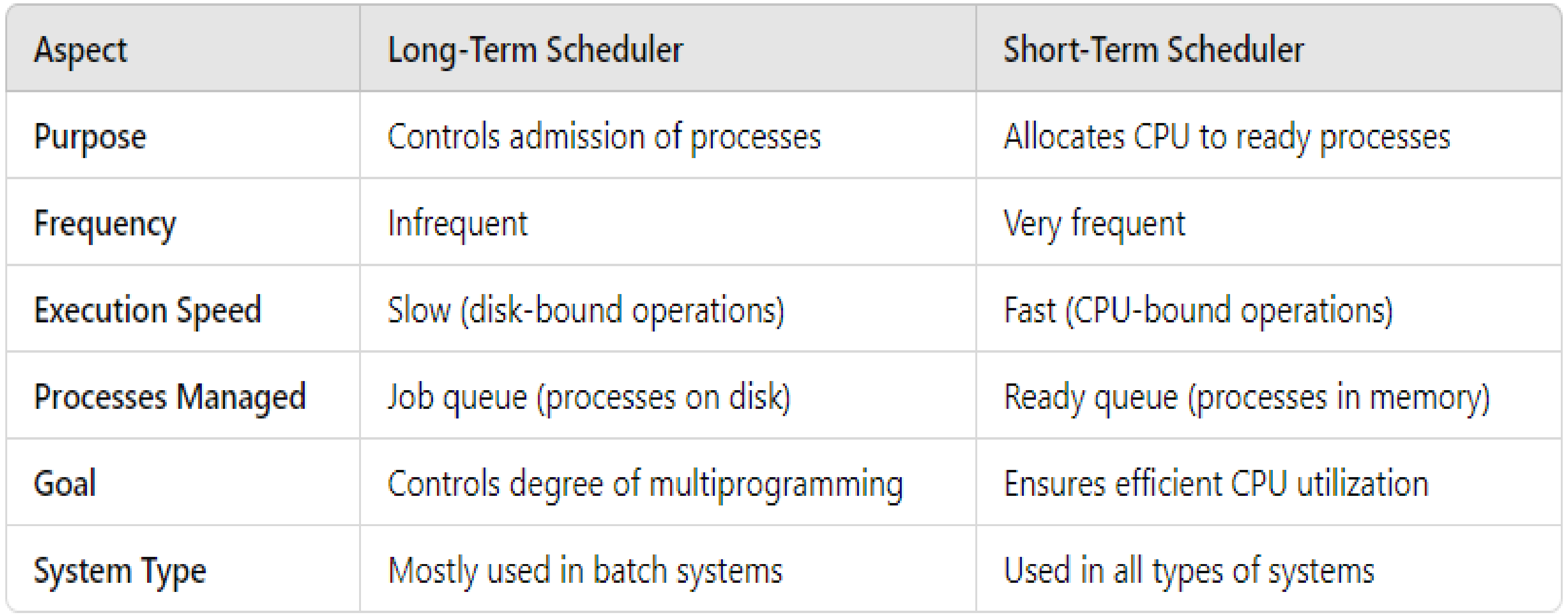
Fig: Representation of Process Scheduling



2.2 Schedulers

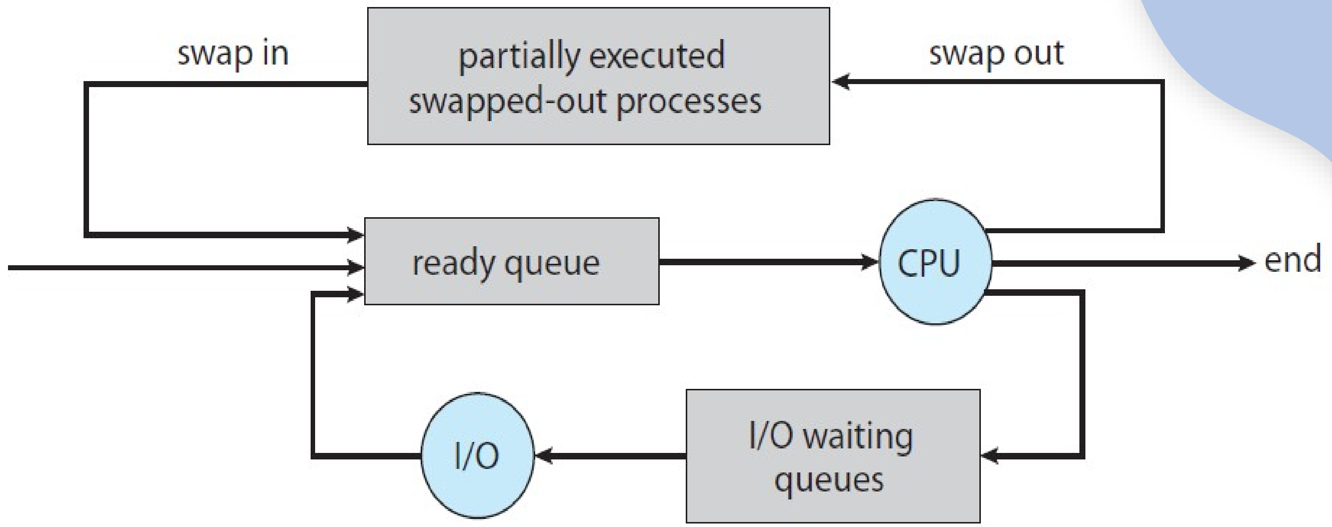
- **Long-term scheduler** (or job scheduler) – selects which processes should b brought into the ready queue. Invoked very infrequently.
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU. Invoked frequently.
  - Sometimes the only scheduler in a system
- Long Term Scheduler is responsible for selecting which jobs from the job pool (typically on disk) are brought into the ready queue (in memory) for execution.
- Today's **interactive and time-sharing systems**, processes are admitted into memory almost immediately after they are created, which reduces the need for a distinct long-term scheduler.

2.2 Schedulers



Aspect	Long-Term Scheduler	Short-Term Scheduler
Purpose	Controls admission of processes	Allocates CPU to ready processes
Frequency	Infrequent	Very frequent
Execution Speed	Slow (disk-bound operations)	Fast (CPU-bound operations)
Processes Managed	Job queue (processes on disk)	Ready queue (processes in memory)
Goal	Controls degree of multiprogramming	Ensures efficient CPU utilization
System Type	Mostly used in batch systems	Used in all types of systems

2.2 Medium Term Scheduler



- The medium-term scheduler temporarily removes processes from memory to control the level of multiprogramming.
- This process of removal is called swapping, where processes are moved from memory to disk.
- The purpose is to reduce the number of processes actively competing for CPU resources.
- Swapped-out processes are reintroduced into memory later, resuming their execution from where they left off.

2.3 Context Switching

- The task of switching a CPU from one process to another process is called context switching.
- **Context** of a process represented in the PCB. Whenever an interrupt occurs (hardware or software interrupt), the state of the currently running process is saved into the PCB and the state of another process is restored from the PCB to the CPU.
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

3.1 Operations on Processes: Process Creation

- A process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes. Every process has a unique process ID.
- **Execution**
  - Wait for the child process to terminate and then continue execution. The parent makes a wait() system call.
  - Run concurrently with the child, continuing to execute without waiting.

3.1 Resource Sharing

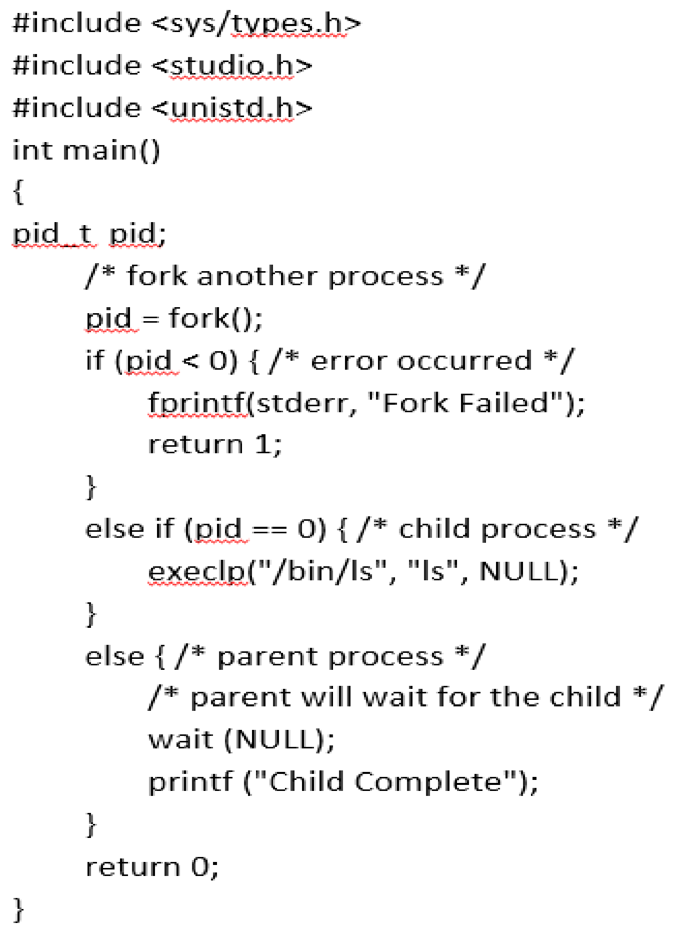
- A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways:
  - directly from the operating system
  - Subprocess may take the resources of the parent process. The resource can be taken from parent in two ways
    - The parent may have to partition its resources among its children
    - Share the resources among several children.

3.1 Address space of Child Process in relation to parent process

- The child may be an exact **duplicate** of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
- The child process may have a **new program loaded** into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows.

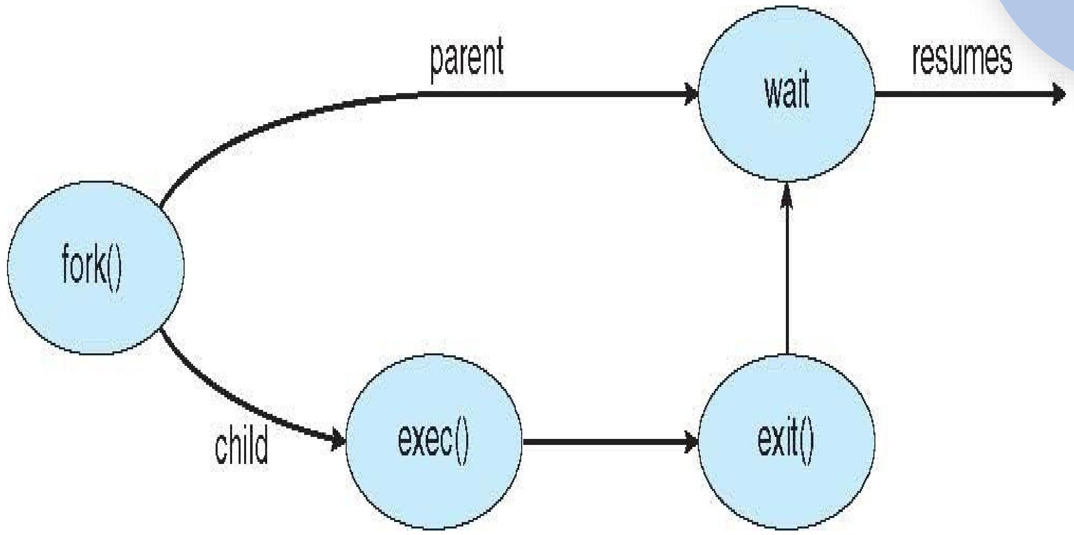
3.1 C Program forking Child Process

- In UNIX OS, a child process can be created by **fork()** system call. The **fork** system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent.
- Process IDs of current process or its direct parent can be accessed using the getpid( ) and getppid( ) system calls respectively.
- The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.
- In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.



```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t pid;
        /* fork another 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 */
                wait (NULL);
                printf ("Child Complete");
        }
        return 0;
}
```

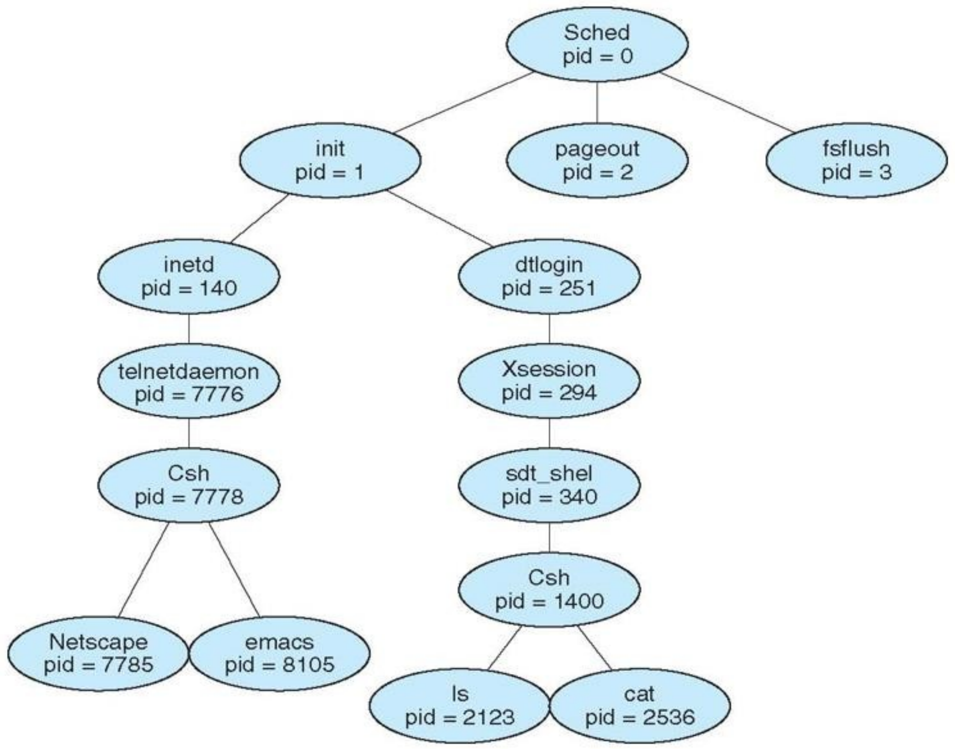
3.1 Process Creation



- After fork(), both the parent and child processes are running.
- **Parent Process Path**:
  - After the fork(), the parent process continues executing. It then calls wait() to wait for the child process to finish execution. During this time, the parent process is paused (blocked) until the child process terminates.
  - Once the child process completes and calls exit(), the wait() system call returns, allowing the parent process to resume its execution.
- **Child Process Path**:
  - After the fork(), the child process may call the exec() system call to replace its current program with a new one. This is typically done to run a different program.
  - When the child process finishes its execution, it calls the exit() system call to terminate itself and return its exit status to the parent process.

3.1 A tree of processes on Solaris

- On typical Solaris systems, the process at the top of the tree is the '**sched**' process with PID of 0.
- The '**sched**' process creates several children processes – **init, pageout** and **fsflush**. Pageout and fsflush are responsible for managing memory and file systems.
- The init process with a PID of 1, serves as a parent process for all user processes.



3.2 Process Termination

- A process terminates when it finishes executing its last statement and asks th operating system to delete it, by using the **exit** () system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.
- A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.
- A parent may terminate the execution of children for a variety of reasons, such as:
  - The child has exceeded its usage of the resources, it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system terminates all the children. This is called **cascading termination**

4. Interprocess Communication(IPC)

- Processes within a system may be **independent** or **cooperating**
  - **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.
- Processes cooperate for several reasons:
  1. **Information Sharing**: Multiple processes may need access to the same file, so the information must be available to all users simultaneously.
  2. **Computation Speedup**: Problems can often be solved faster by dividing them into sub-tasks that run concurrently, especially when multiple processors are used.
  3. **Modularity**: A system can be organized into cooperating modules that communicate and share information with each other.
  4. **Convenience**: Even a single user can handle multiple tasks through information sharing between processes.

4. IPC: Two Models

Aspect	Shared Memory	Message Passing
Communication Method	A region of memory is shared by processes for reading/writing	Processes exchange messages using objects
Data Size Suitability	Suitable for sending large blocks of data	Ideal for sending small amounts of data
System Call Usage	Required only when creating shared memory	Required for every read and write operation
Speed	Faster, as there are no frequent system calls	Slower, due to system calls during communication

4. IPC: Two Models

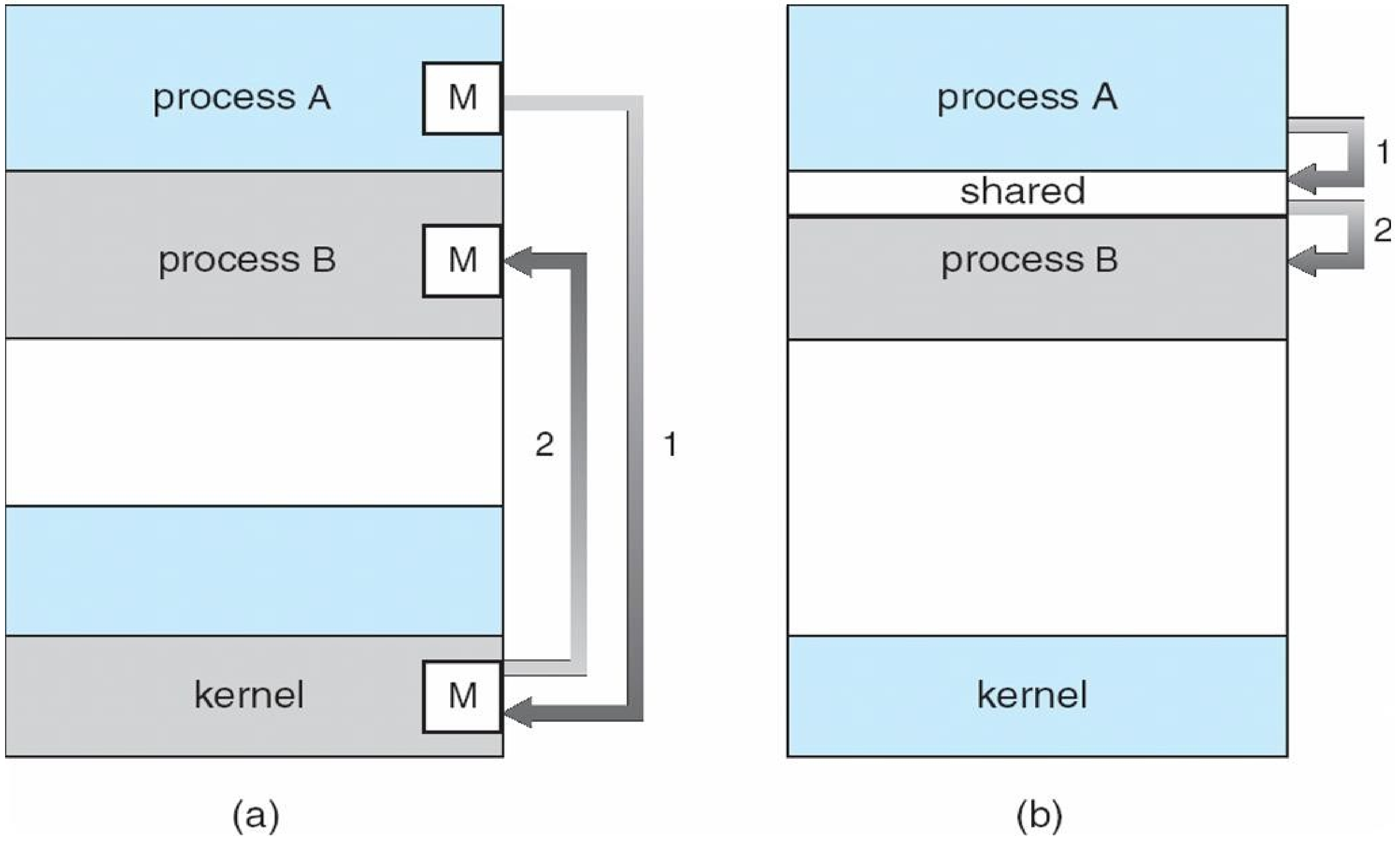


Fig: Communication Models (a)Message Passing (b) Shared Memory

4.1 IPC: Shared Memory

- A shared memory region is created within the address space of a process tha needs to communicate. Other processes involved in the communication also use this shared memory.
- The structure and location of the shared memory area are determined by the process. Typically, some initial messages must be exchanged between the cooperating processes to establish and coordinate access to the shared memory.
- Care must be taken to ensure that both processes do not write to the shared memory simultaneously.

4.2 Shared Memory: Producer-Consumer Problem

- This is a common scenario in inter-process communication where one process generates data (the producer) and another process retrieves and uses that data (the consumer)
- The data is passed through an intermediary buffer (shared memory). The producer adds data to the buffer, while the consumer retrieves it. The producer can generate new items while the consumer processes others. Synchronization is necessary to ensure the consumer doesn't attempt to retrieve data that hasn't been produced yet. In such cases, the consumer must wait until new data is available.
- There are two types of buffers used for data transfer:
  - **Unbounded buffer**: There is no size limit, allowing the producer to generate data without restriction, though the consumer may have to wait for new items.
  - **Bounded buffer**: The buffer size is fixed, so the producer must wait if the buffer is full, and the consumer must wait if it's empty.

4.2 Bounded Buffer: Shared-Memory Solution

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

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

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

4.2 Bounded Buffer: Producer

```
Item nextProduced;

while (true) {
        /* Produce an item */

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

            /* do nothing -- no free buffers */

        buffer[in] = item;

        in = (in + 1) % BUFFER SIZE;

    }
```

- Only Producer can change the “in” pointer
- The producer process has a local variable nextProduced in which the new item to be produced is stored

4.2 Bounded Buffer: Consumer

```
item nextConsumed;
while (true) {
         while (in == out)
               ; // do nothing -- nothing to consume
        // remove an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER SIZE;
    return item;
}
```

- Only Consumer can change the “out” pointer
- The consumer process has a local variable nextConsumed in which the item to be consumed is stored

4.2 Bounded Buffer: Issue

In the producer-consumer problem, both processes share a buffer where the producer adds items, and the consumer removes them. If both try to access the buffer simultaneously, it can lead to issues like race conditions or inconsistent data. Proper synchronization mechanisms, such as semaphores or mutex locks, are required to prevent this and ensure that only one process accesses the buffer at any given time.

4.3 IPC: Message Passing

- Message-passing is a mechanism that allows processes to communicate withou sharing an address space, commonly used in distributed systems.
- Here communication occurs via system calls, such as "send message" and "receive message."
- **Communication Link**: Before messages can be exchanged, a communication link must be established between the cooperating processes.
- There are three methods for establishing a communication link
  1. Direct or indirect communication (naming)
  2. Synchronous or asynchronous communication (Synchronization)
  3. Automatic or explicit buffering.

4.3.1 Message Passing: Direct Communication

Naming: Here processes that want to communicate must have a way to refer to each other.

- Direct Communication: The sender and receiver must explicitly know each other's names.
- The syntax for communication functions is:
  - send(P, message) – sends a message to process P.
  - receive(Q, message) – receives a message from process Q.
- Properties of Direct Communication:
  - A link is automatically established between every pair of processes that want to communicate, requiring only knowledge of each other's identity.
  - Each link is associated with exactly one pair of processes, and there is exactly one link between them.

4.3.1 Message Passing: Direct Communication

- Addressing in Direct Communication:
  - **Symmetric Addressing**: Both the sender and receiver must know each other's names to communicate.
  - **Asymmetric Addressing**: The sender specifies the receiver’s name, but the receiver can accept messages from any process. For example:
    - send(P, message) – sends a message to process P.
    - receive(id, message) – receives a message from any process.
- Disadvantages of Direct Communication: Changes in process identifiers require updates across the entire system, affecting both sender and receiver.

4.3.2 Message Passing: Indirect Communication

- In indirect communication, processes exchange messages through shared mailboxes or ports. A mailbox is an object with a unique ID where messages are sent and received.
  - send(A, message) – sends a message to mailbox A.
  - receive(A, message) – receives a message from mailbox A.
- Properties of Indirect Communication:
  - A communication link is established only if the processes share a mailbox.
  - A mailbox can be shared by more than two processes, with multiple links associated with one mailbox.
  - The operating system manages mailboxes, handling tasks like creating, deleting, and managing message transfers.

4.3.3 Message Passing: Synchronisatio

- Message sending and receiving can be either blocking (synchronous) or non-blocking (asynchronous):
  - **Blocking Send**: The sender waits until the message is received by the receiver or mailbox.
  - **Non-blocking Send:** The sender sends the message and continues without waiting.
  - **Blocking Receive:** The receiver waits until a message is available.
  - **Non-blocking Receive**: The receiver retrieves a message without waiting, which may be valid or null.

4.3.4 Message Passing: Buffering

- When messages are passed, a temporary queue (buffer) is created. Buffering can be categorized into three types:
    - **Zero Capacity**: No buffer exists. Senders must wait until the message is received.
    - **Bounded Capacity**: A fixed-size queue. The sender is blocked if the queue is full.
    - **Unbounded Capacity**: The queue has no size limit, and the sender never blocks.

Don’t let failure go to your heart, and don’t let success go to your head.

Will Smith

quotefancy