KTU B.TECH CSE S4 NOTE
OPERATING SYSTEMS
CS204
MODULE - **2**

By

Ms Jasheeda P

CSE Department

MEA Engineering College Peinthalnanna

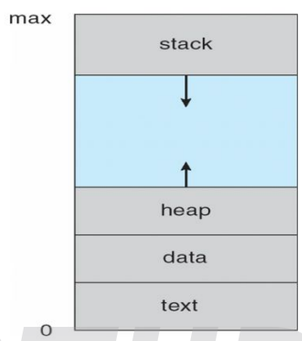
MODULE II

PROCESS CONCEPT

- Process is a program in execution. A process is the unit of work in a modern time-sharing system.
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks

The Process

- A process is more than the program code, it includes
  - 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
- program is a *passive* entity(executable files), whereas a process is an *active* entity
- A program becomes a process when an executable file is loaded into memory.



Process in memory

Process State

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:
  - 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
- Only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*.

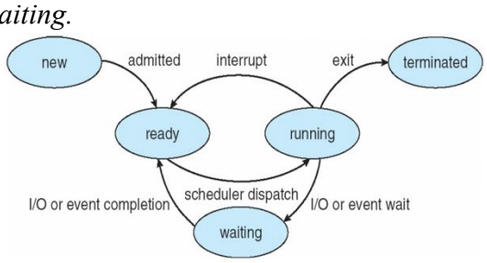


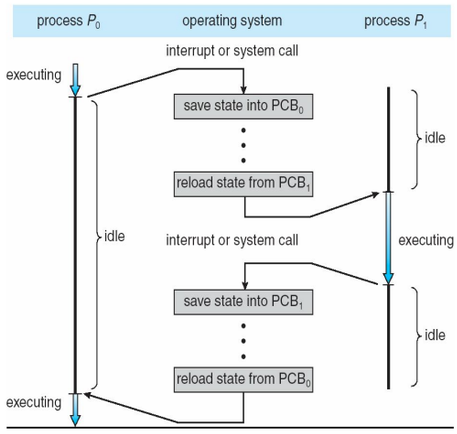
Diagram of process state

Process Control Block

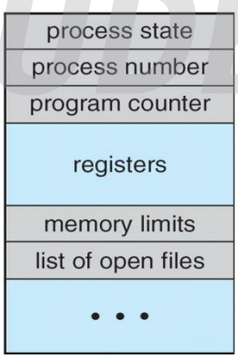
- Each process is represented on the OS by a Process Control Block(PCB) also called a *task control block*..
- It includes
  - Process state – The state may be new, ready running, waiting, halted etc
  - Program counter – The counter indicates the address of the next

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, plus any condition-code information. 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- includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information -include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting 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 – I/O devices allocated to process, list of open files



Process Control Block

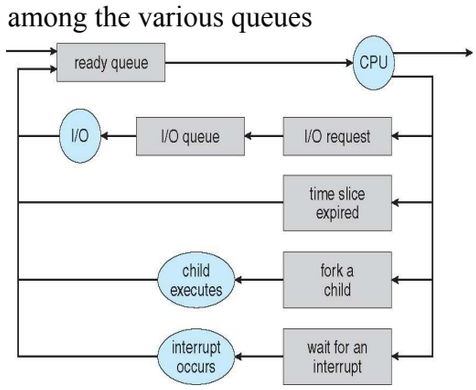
Threads

- Process is a program that performs single thread of execution.
- Single thread of control allows the process to perform only one task at one time.
- Modern Operating Systems have 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.

PROCESS SCHEDULING

- 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.
- The process scheduler selects an available process for program execution on the 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
- Processes migrate among the various queues

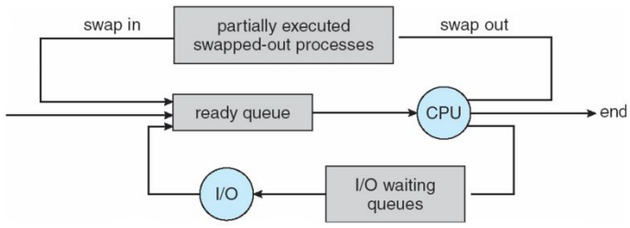


Queuing diagram representation of process scheduling

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:
- The process could issue an I/0 request and then be placed in an I/0 queue.
- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler
- The long-term scheduler, or job scheduler, selects processes from pool and loads them into memory for execution. Long-term scheduler is invoked infrequently (seconds, minutes).
- The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them. Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- 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.
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
- **Swapping** -Remove process from memory, store on disk, bring back in from disk to continue execution



Addition of medium-term scheduling to the queuing diagram

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 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.

Operations on processes

- System must provide mechanisms for:
- process creation,
- process termination,

Process Creation

- A process may create several new processes, via a create-process system call,during the course of execution.
- 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, 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

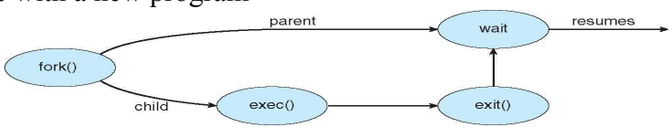
- Parent and children execute concurrently
- Parent waits until children terminate

Possibilities in terms of the address space of the new process

- The child process is a duplicate of the parent process
- The child process has a new 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




```
#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;
}
```

C program forking a separate process

Process Termination

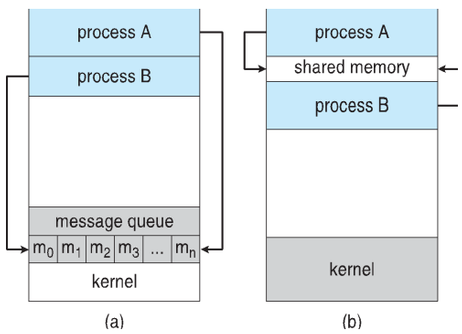
- Process executes last statement and then asks the operating system to delete it using the

exit() system call

- Returns status data from child to parent (via **wait()**)
- Process' resources are deal located by operating system
- A process can cause the termination of another process via an appropriate system call. Such a system call can be invoked only by the parent of the process that is to be terminated
- A parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
- 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
- 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. This phenomenon, referred to as **cascading termination.** 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

INTERPROCESS COMMUNICATION

- Inter process communication (IPC) refers to the coordination of activities among cooperating processes.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Two models of IPC
  - **Shared memory-**
  - **Message passing**



Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- A shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- They can then exchange information by reading and writing data in the shared areas.
- The communication is under the control of the users processes not the operating system.
- Example for cooperating processes.:-Producer-consumer problem.
  - A producer process produces information that is consumed by a consumer process.
  - One solution to the producer-consumer problem uses shared memory.
  - A buffer which reside in a region of memory that is shared by the producer and consumer

processes is used

- 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 types of buffers can 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 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.

➢ Producer

```
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

```
item next_consumed;
while(true){
        while (in == out)
                ;/*donothing*/
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in next consumed */
}
```

Message-Passing Systems

➢ Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

➢ A message-passing facility provides at least two operations: send (message) and receive (message).

➢ If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

➢ methods for logically implementing a link and the send() /receive() operations:

- Direct or indirect communication.
- Synchronous or asynchronous communication.

- Automatic or explicit buffering

Naming

- Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
  - send(P, message) -Send a message to process P.
  - receive (Q, message)-Receive a message from process Q.
- Symmetry in addressing- both the sender process and the receiver process must name the other to communicate.
- Asymmetry in addressing- Here, only the sender names the recipient; the recipient is not required to name the sender.
  - send (P, message) -Send a message to process P.
  - receive (id, message) -Receive a message from any process
- With indirect communication, the messages are sent to and received from mailboxes, or ports.
- Two processes can communicate only if the processes have a shared mailbox.
  - send (A, message) -Send a message to mailbox A.
  - receive (A, message)-Receive a message from mailbox A.
- The operating system must provide a mechanism that allows a process to do the following:
  - Create a new mailbox.
  - Send and receive messages through the mailbox.
  - Delete a mailbox.

Synchronization

- Message passing may be either blocking or nonblockingalso known as synchronous and asynchronous.
  - Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - Nonblocking send. The sending process sends the message and resumes operation.
  - Blocking receive. The receiver blocks until a message is available.
  - Nonblocking receive. The receiver retrieves either a valid message or a null.

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
  - **Zero capacity -**The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
  - **Bounded capacity**. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link's capacity is finite. If the link is full, the sender must block until space is available in the queue.
  - **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

PIPES

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems and typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

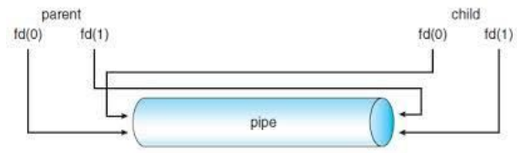
1. Does the pipe allow unidirectional communication or bidirectional communication?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as parent-child) exist between the communicating processes?

4 Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer consumer fashion; the producer writes to one end of the pipe(write end) and the consumer reads from the other end(read end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message Greetings to the pipe, while the other process reads this message front the pipe.

On UNIX systems, ordinary pipes are constructed using the function pipe (int fd []) This function creates a pipe that is accessed through the int fd [] file descriptors: fd [0] is the read-end of the pipe, and fd [1] is the write end.



File descriptors for an ordinary pipe.

Named Pipes

Ordinary pipes provide a simple communication mechanism between a pair of processes. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and terminated, the ordinary pipe ceases to exist. Named pipes provide a much more powerful communication tool; communication can be bidirectional, and no parent-child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes. Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. *A* FIFO is created with the mkfifo () system call and manipulated with the ordinary open(), read(), write(), and close () system calls. It will continue to exist m<til it is explicitly deleted from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used.