Unit 1.1
Process Management

Chapter 2 of Text Book

Objectives

- **After studying this unit, you should be able to**
  - Explain process state
  - Describe the process control block(PCB)
  - Discuss the process scheduling
  - Describe threads in Operating system

Process Concept

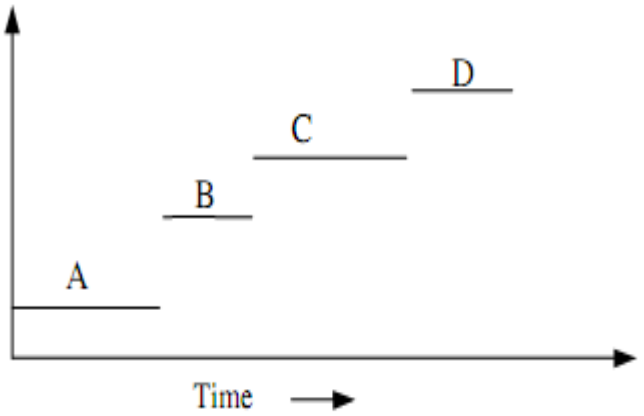
- An operating system executes a variety of programs
  - batch systems - jobs
  - time-shared systems - user programs or tasks
  - job and program used interchangeably
- Process - a program in execution
  - process execution proceeds in a sequential fashion
- A process is an activity of some kind, it has program, input, output and state.

Process Model

- Uni-programming
- Multiprogramming
- Multiprocessing

Uni-programming Model

Only one process at a time

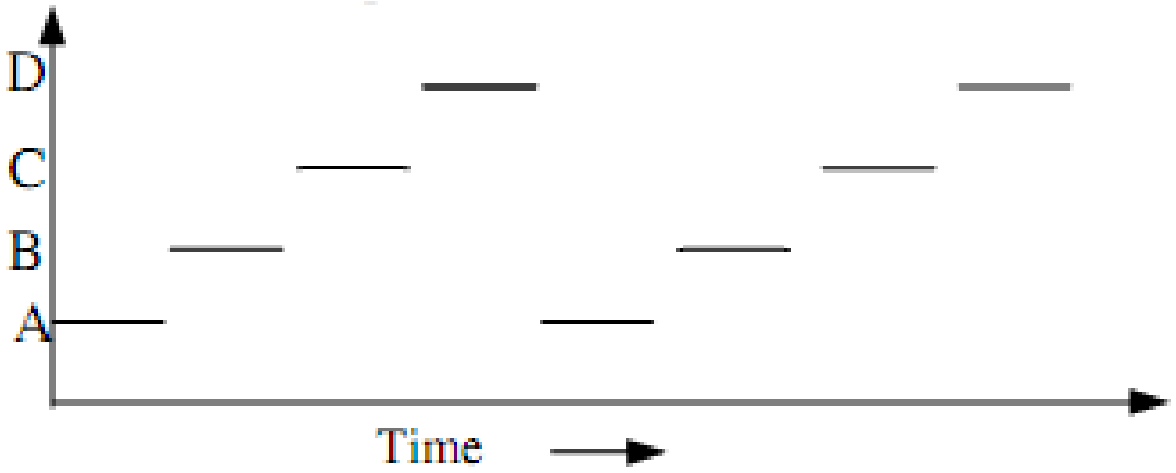


- Examples: Older systems
- Advantages: Easier for OS designer
- Disadvantages: Not convenient for user and poor performance

Multiprogramming Mode

multiple process at a time

- OS requirements for multiprogramming:
- Policy: to determine which process is to schedule.
- Mechanism: to switch between the processes.

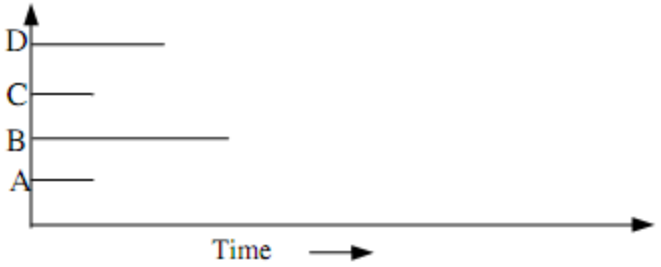


- Examples: Unix, WindowsNT
- Advantages: Better system performance and user convenience.
- Disadvantages: Complexity in OS

Multiprocessing Model

System with multiple processor

- Distributed OS and Network OS



Process States diagram

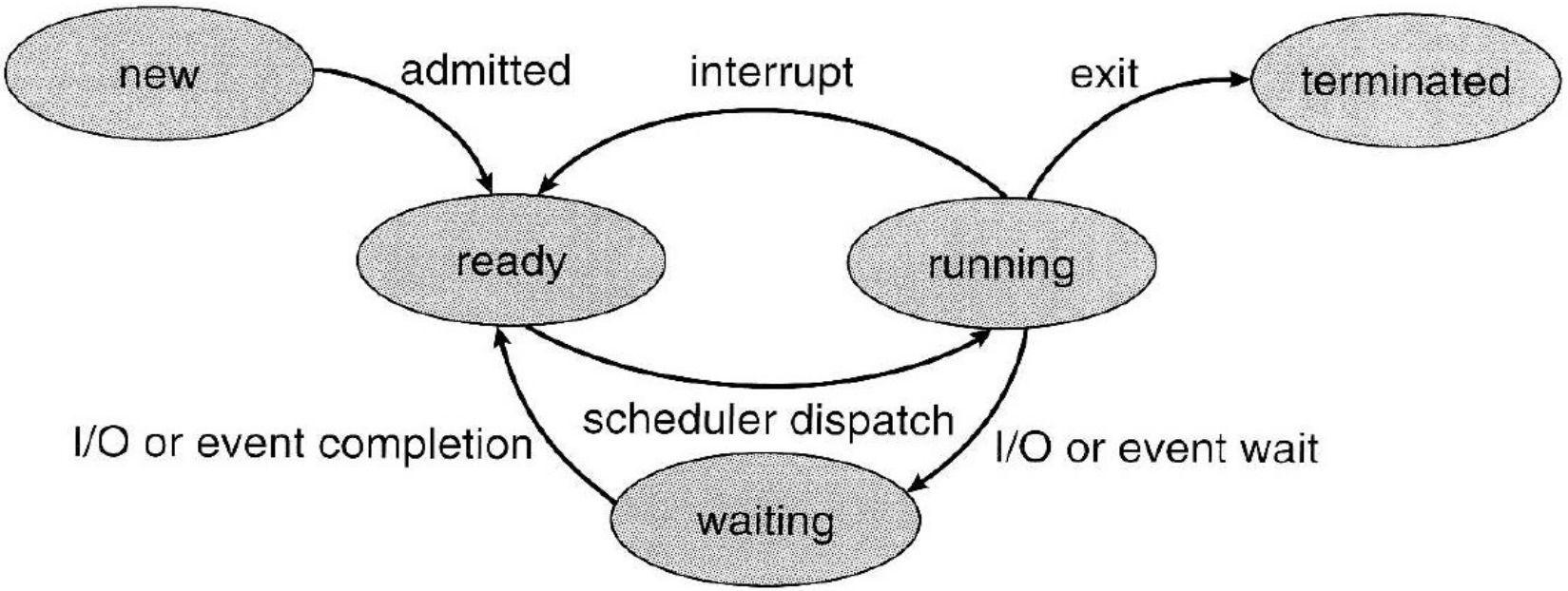


Figure 3.2 Diagram of process state.

Process States

- New - The process is being created.
- Running - Instructions are being executed.
- Waiting - Waiting for some event to occur.
- Ready - Waiting to be assigned to a processor.
- Terminated - Process has finished execution.

Process Control Block

- Process must be saved when the process is switched from one state to another so that it can be restarted later as it had never been stopped.
- The PCB is the data structure containing certain important information about the process -also called process table or processor descriptor.
  - Process state: running, ready, blocked.
  - Program counter: Address of next instruction for the process.
  - Registers: Stack pointer, accumulator, PSW etc.
  - Scheduling information: Process priority, pointer to scheduling queue etc.
  - Memory-allocation: value of base and limit register, page table, segment table etc.
  - Accounting information: time limit, process numbers etc.
  - Status information: list of I/O devices, list of open files etc.

PCB

- The PCB simply serves as repository of any information that may vary from process to process.

Pointer	Process state
Process number	
Program counter	
Registers	
Memory Limits	
List of open files	
…….	

Operation on process

- The processes in the system can execute concurrently, and they must be created and deleted dynamically. OS provide the mechanism for process creation and termination.
  - Process Creation.
  - Process Termination

Process Creation

- There are four principal events that cause the process to be created:
  - » System initialization.
  - » Execution of a process creation system call.
  - » User request to create a new process.
  - » Initiation of a batch job.
- A process may create several new processes during the course of execution. The creating process is called a parent process, where as the new processes are called children of that process.

Process Termination

- Process are terminated on the following conditions
  - Normal exit.
  - Error exit.
  - Fatal error.
  - Killed by another process.
- Example:
  - In Unix the normal exit is done by calling a exit system call. The process return data (output) to its parent process via the wait system call.
  - kill system call is used to kill other process.

Process Scheduling

- Main objective of multiprogramming is to maximize the CPU utilization
- At any time, a process is in any one of the new, ready, running or waiting state.
- The OS module known as scheduler schedules processes from ready queue for execution by CPU
- Scheduler select one of the many process from ready queue on certain criteria.
- (Will discuss it in detail in next chapter)

Context Switch

- CPU switching form one process to another requires saving the state of the current process and loading the latest state of next process.
- This is known as context switch.
- The time require to context switch is a overhead since it does not produce any useful work in aspect of process execution time.

Thread

- It is a single sequence stream which allows a program to split itself into two or more simultaneously running task.
- It is the basic unit of CPU utilization and consists of program counter, a register set and a stack space.
- Sometime called light weight process.

Thread and Process

- Threads are not independent as process.
- Thread shares with other thread: code section, data section, Other resource such as opens files.
- Similarities:
  - Like process, thread share CPU and only one thread is running at a time.
  - Like process, thread within processes execute sequentially
  - Like process if one thread is blocked, another can run.

Thread and process

- Dissimilarities
  - Unlike process, threads are not independent of each other.
  - Unlike process, all threads can access every address space.
  - Unlike process, threads are designed to assist one other.

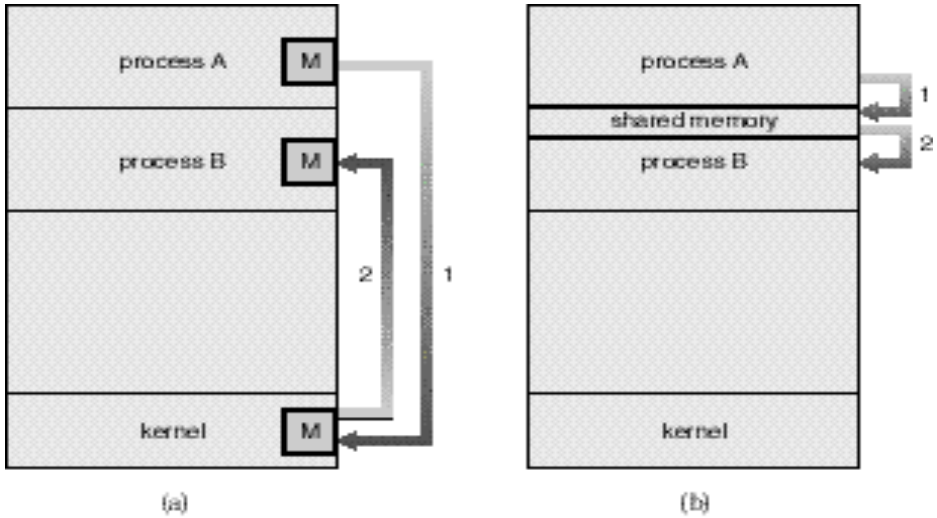
IPC(Inter Process Communication)

- How one process can pass the information to the another?
- How to make sure two or more processes do not get into each other's way when engaging in critical activities?
- How to maintain the proper sequence when dependencies are presents?

- Thus, Inter Process Communication (IPC) and synchronization
  - IPC provide the mechanism to allow the processes to communicate and to synchronize their actions.

IPC models

- There are two fundamental models of inter-process communication:
  - **Shared Memory:**
  - **Message Passing**.



Massage Passing

- Communication takes place by means of messages exchanged between the cooperating processes.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than shared memory for inter-computer communication.

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.
- Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.
- Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

Massage passing Vs Sharing Memory

- **In contrast**, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Process Synchronization

- Synchronization is often necessary when processes communicate.
- Processes are executed with unpredictable speeds. One can view synchronization as a set of constraints on the ordering of events.
- The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.
- As an illustrative example, consider the batch operating system again. A shared buffer is used for communication between the reader process and the writer process. These processes must be synchronized so that, for example, the reader process never attempts to read data from the input if the buffer is empty.

Race Condition

Situation where two or more processes are reading or writing some shared data and the final result depends on who run precisely, are called race conditions

Race Condition?

- **Possibilities of race:**
  - one process reading and another process writing same data.
  - two or more process writing same data.
- Solution: prohibiting more than one process from reading /writing the same data at the same time- **Mutual Exclusion or (next slide)**.

Solution to Race condition

- Make an operating system not to perform several tasks in parallel. This is called serialization. Two strategies to serializing processes in a multitasking environment:
  - **The Scheduler can be disabled**
  - **A Protocol can be introduced**

- **Scheduler** can be disabled for a short period of time, to prevent control being given to another process during a critical action like modifying shared data. This will be inefficient on multiprocessor machines, since all other processors have to be halted every time one wishes to execute a critical section.

- A **protocol** can be introduced which all programs sharing data must obey. The protocol ensures that processes have to queue up to gain access to shared data.

Then what is Mutual Exclusion really?

- Some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.

Critical-Section Problem

- Code executed by the process can be grouped into sections, some of which require access to shared resources, and other that do not.
- The section of the code that require access to shared resources are called critical section.
- ***Facts:*** *The part of the time, process is busy doing internal computations and other things that do not lead to the race condition. Non critical section*
- ***Helpful to avoid race condition.***

Critical-Section Problem

General structure of process Pi (other process Pj)

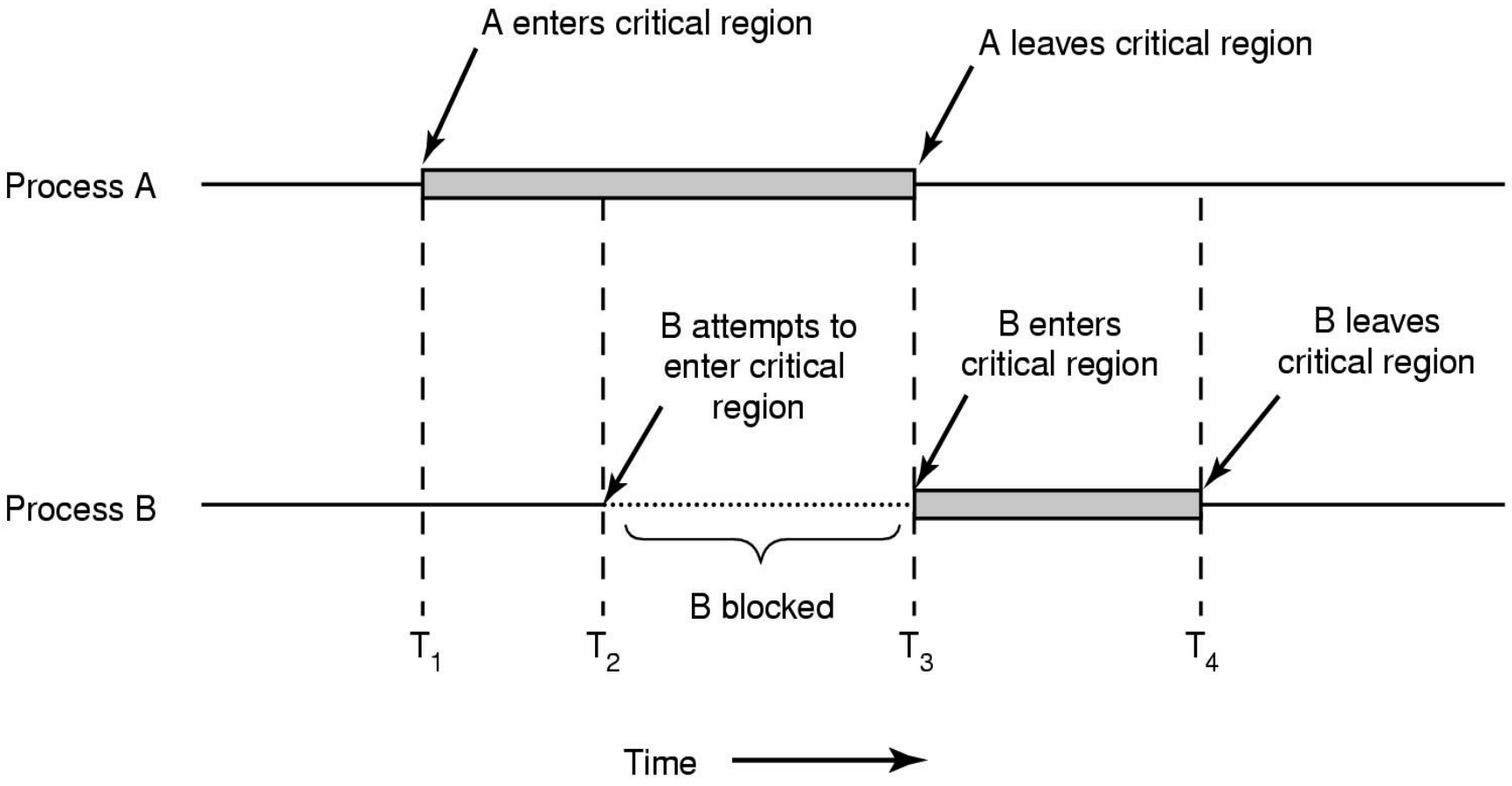
```
do{
        entry_section
                critical_section(CR)
        exit_section
                reminder_section
} while(true)
```

- When a process is accessing a shared modifiable data, the process is said to be in critical section.
- All other processes (those access the same data) are excluded from their own critical region.
- All other processes may continue executing outside their CR.
- When a process leaves its critical section, then another processes waiting to enter its own CR should be allowed to proceed.

Critical-Section Problem

- The critical section problem is to design a **protocol** that the processes can use to co-operate.
- A good solution to critical section problem must satisfy the following three requirements.
  - No two processes may be simultaneously inside their CRs (mutual exclusion).
  - No process running outside its CR may block other process.(Progress)
  - No process should have to wait forever to enter its CR.(Bounded Waiting)

Critical Section diagram (with mutual exclusion)



Solution 1(Busy waiting) Interrupt Disabling

- *Each process disable all interrupts just after entering its CR and re-enable them just before leaving it.*
- No clock interrupt, no other interrupt, no CPU switching to other process until the process turn on (enable) the interrupt.
- The general structure of process may be as
- Do

```
{
    DisableInterrupt()
            // perform CR task
    EnableInterrupt()
            //perform Remiander section task
 }while(1);
```

Facts about Disabling Interrupts

- Advantages:
  - Mutual exclusion can be achieved by implementing OS primitives to disable and enable interrupt.
- Problems:
  - allow the power of interrupt handling to the user.
  - The chances of never turned on is a disaster.
  - it only works in single processor environment.

Solution 2(Busy waiting)
Lock Variables

- A single, shared (lock) variable, initially 0.
- When a process wants to enter its CR, it first test the lock. If the lock is 0, the process set it to 1 and enters the CR. If the lock is already 1, the process just waits until it becomes 0.
- The general structure of process is
- Do{

```
While(lock!=0); //busy waiting
  Lock=1;
        //perform the CR task
  Lock=0
        //perform Non CR task
}while(true)
```

Facts

- **Advantages:** seems no problems.
- Problems:
  - problem like spooler directory;
  - suppose that one process reads the lock and sees that it is 0, before it can set lock to 1, another process scheduled, enter the CR, set lock to 1 and can have two process at CR (violates mutual exclusion)

Strict Alternation

- Processes share a common integer variable *turn*.

If turn == i

Then process Pi is allowed to execute in its CR,

if turn == j

then process Pj is allowed to execute.

```
While (true){
   while(turn!=i); /*loop*/
       critical_section();
   turn = j;
     noncritical_section
}
```

Structure of Process Pi

```
While (true){
   while(turn!=j); /*loop*/
          critical_section();
    turn = i;
         noncritical_section
}
```

Structure of Process Pj

Facts

Ensures that only one process at a time can be in its CR.

- Problems: strict alternation of processes in the execution of the CR.
  - What happens if process i just finished CR and again need to enter CR and the process j is still busy at non-CR work? (violate condition 2)

Peterson's solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Facts about Petersons

- Before entering its CR, each process call enter_region with its own process number, 0 or 1 as parameter.
  - Call will cause to wait, if need be, until it is safe to enter.
  - When leaving CR, the process calls leave_region to indicate that it is done and to allow other process to enter CR.
- Advantages: Preserves all conditions.
- Problems: difficult to program for n-processes system and less efficient.

Hardware Assistance(TSL)

- Checking for mutual exclusion is also possible through Hardware.
- Specific instruction called Test and Set Lock(TSL) is used for this purpose.
- The special properties of these instruction is that they cant be interrupted i.e. they are atomic instruction and carried out with out interruption.

TSL instruction

- Test and Set Lock (TSL) instruction reads the contents of the memory word lock (shared variable) into the register and then stores nonzero value at the memory address lock.
- This automatically as in a un-interruptible time unit.
- The CPU executing TSL locks the memory bus to prohibit other CPUs from accessing memory until it is done.
- When lock is 0, any process may set it to 1 using the TSL instruction.
- When it is done the process lock back to 0.

Alternate to Busy waiting

Busy waiting:

When a process want to enter its CR, it checks to see if the entry is allowed, if it is not, the process just sits in a tight loop waiting until it is.

Waste of CPU time for NOTHING!

- Possibility of Sleep and Wakeup pair instead of waiting.
- Sleep causes to caller to block, until another process wakes it up.

Sleep and week up
Consumer Producer problem

- Two process share a common, fixed sized buffer.
- Suppose one process, producer, is generating information that the second process, consumer, is using.
  - Their speed may be mismatched, if producer insert item rapidly the buffer will full and go to the sleep until consumer consumes some item, while consumer consumes rapidly, the buffer will empty and go to sleep until producer put something in the buffer.

Conti.

```
#define N = 100 /* number of slots in the buffer*/
int count = 0; /* number of item in the buffer*/

void producer(void)
        {
        int item;
        while(TRUE)
        { /* repeat forever*/
                item = produce_item(); /*generate next item*/
                if (count == N)
                    sleep(); /*if buffer is full go to sleep*/
                insert_item(item); /* put item in buffer */
                count = count +1; /*increment count */
                if (count == 1)
                    wakeup(consumer);
                /* was buffer empty*/
        }
```

Cont……

```
void consumer(void)
{
      int item;
      while(TRUE)
      { /* repeat forever*/
            if(count == 0)
               sleep(); /* if buffer is empty go to sleep*/
            item = remove_item(); /*take item out of buffer*/
            count = count -1; /*decrement count*/
            if (count == N-1)
                wakeup(producer); /*was buffer full ?*/
            consume_item(); /*print item*/
            }
}
```

Producer Consumer with weak up and sleep signals

- Problem:
  - leads to race as in Lock variable.
  - **What happen if**
    - When the buffer is empty, the consumer just reads count and quantum is expired, the producer inserts an item in the buffer, increments count and wake up consumer.
    - The consumer not yet asleep, so the wakeup signal is lost, the consumer has the count value 0 from the last read so go to the sleep.
    - Producer keeps on producing and fill the buffer and go to sleep, both will sleep forever.

Think: If we were able to save the wakeup signal that was lost.....(semaphores)

Semaphores

- E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups, called a semaphore.
- It could have the value 0, indicating no wakeups ware saved, or some positive value if one or more wakeups were pending.

- **Operations: Down** and **Up** (originally he proposed P and V in Dutch and sometimes known as wait and signal)
  - Down: checks if the value greater than 0
  - yes- decrement the value (i.e. Uses one stored wakeup) and continues.
  - No- process is put to sleep without completing down.
- **Checking value, changing it, and possibly going to sleep, is all done as single action.**
  - Up: increments the value; if one or more processes were sleeping, unable to complete earlier down operation, one of them is chosen and is allowed to complete its down.

Semaphore Operations

```
void down(S)
{
    if(S> 0)
            S--;
    else
     sleep(); //wait until s>0
}
void up (S)
{
    if(one or more processes are sleeping on S)
            one of these process is proceed;
    else
        S++;
}
```

Semaphore: Discussions

- To avoid busy waiting, a semaphore may have an associated waiting queue of process.
- If process does a down operation on a semaphore which is zero, the process is added to the semaphore's queue.
- When another process increments the semaphore by doing up operation and there are task in the queue, one is taken off and resumed.

Consumer producer using Semaphore

```
#define N 100 /*number of slots in buffer*/
typedef int semaphore; /*defining semaphore*/
semaphore mutex = 1; /* control access to the CR */
semaphore empty = N /*counts empty buffer slots*/
semaphore full = 0; /counts full buffer slots*/
    void producer(void)
    {
        int item;
        while(TRUE){ /*repeat forever */
        item = produce_item(); /*generate something */
        down(empty); /*decrement empty count*/
        down(mutex); /*enter CR */
        insert_item(); /* put new item in buffer*/
        up(mutex); /* leave CR*/
        up(full); /*increment count of full slots*/
    }
```

Consumer producer using Semaphore

```
void consumer(void)
{
        int item;
        while(TRUE){ /*repeat forever*/
        down(full); /*decrement full count */
        down(mutex); /*enter CR*/
        item = remove_item(); /*take item from buffer*/
        up(mutex); /*leave CR*/
        up(empty); /*increment count of empty slots*/
        consume_item(); /*print item*/
        }
}
```

Semaphore's attraction

- The have the following attractive properties
  - Machine independent
  - Simple
  - Powerful: provide exclusion and waiting
  - Work with many process
  - Can ask for many resources simultaneously with multiple down operations

Message Passing

- With the trend of distributed operating system, many OS are used to communicate through Internet, intranet, remote data processing etc.
- Inter process communication based on two primitives operations:
  - send
  - and receive.
- The general syntax for these operations looks like
  - send(destination, &message);
  - receive(source, &message);
- The send and receive calls are normally implemented as operating system calls accessible from many programming language environments.

Producer-Consumer with Message Passing

```
#define N 100 /*number of slots in the buffer*/

        void producer(void)
        { int item;
        message m; /*message buffer*/
        while (TRUE){
        item = produce_item(); /*generate something */
        receive(consumer, &m); /*wait for an empty to arrive*/
        build_message(&m, item); /*construct a message to send*/
        send(consumer, &m); }
        }

    void consumer(void)
    { int item;
    message m;
    for(i = 0; i<N; i++) send(producer, &m); /*send N empties*/
    while(TRUE){
    receive(producer, &m); /* get message containing item*/
    item = extract_item(&m); /* extract item from message*/
    send(producer, &m); /* send back empty reply*/
    consume_item(item); /*do something with item*/
    }
```

Message passing

- No shared memory.

- Messages sent but not yet received are buffered automatically by OS, it can save N messages.
- The total number of messages in the system remains constant, so they can be stored in given amount of memory known in advance.

- Implementing Message Passing can be with:
  - Direct addressing: provide ID of destination.
  - Indirect addressing: Send to a mailbox.

- **Mail box**: It is a message queue that can share by multiple senders and receivers, senders sends the message to the mailbox while the receiver pick ups the message from the mailbox.

Monitors

- Higher level synchronization primitive.
- A monitor is a programming language construct that guarantees appropriate access to the CR.
- It is a collection of procedures, variables , and data structures that are all grouped together in a special kinds of module or package.
- Processes that wish to access the shared data, do through the execution of monitor functions.
- Only one process can be active in a monitor at any instant.

Monitors: discussion

- A procedure defined with in a monitor can access only those variables declared locally with in the monitors.
- A **monitor** is an object or module intended to be used safely by more than one process.
- The defining characteristic of a monitor is that its methods are executed with mutual exclusion.
- That is, at each point in time, at most one process may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

The Syntax of Monitors

```
Monitor monitor_name
{
    shared variable declarations;
    procedure p1(){ ..... }
    procedure p2(){ .... }
    .... .
    procedure pn(){ ...}

    {initialization code;}
}
```

Homework 3

- Discuss intercrosses communication in details
- What is critical section and critical section problem?
- What are semaphores? Explain.
- What are monitors? Explains
- Show the Peterson's algorithm preserve mutual exclusion, indefinite postponement and progress (deadlock)