OPERATING SYSTEM (BISOS304)

MODULE-3

Process Synchronization

1. Background

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Processes can execute concurrently or in parallel. The CPU scheduler switches rapidly between processes to provide concurrent execution.
Since processes frequently needs to communicate with other processes therefore, there is aneed for a well- structured communication, without using interrupts, among processes.

In the bounded buffer our original solution allowed atmost BUFFER SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

The code for the producer process can be modified as follows:

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

The code for the consumer process can be modified as follows:

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

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the
statements “counter++” and “counter--”. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

The statement “counter++” may be implemented in machine language as follows:

```
register1 = counter
```

register1 = register1 + 1
counter = register1
where register1 is one of the local CPU registers. Similarly, the statement "counter--" is implemented as follows:
register2 = counter
register2 = register2 − 1
counter = register2
where again register2 is one of the local CPU registers. Even though register1 and register2 may be the same physical register.

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:
T0: producer execute register1 = counter {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = counter {register2 = 5}
T3: consumer execute register2 = register2 − 1 {register2 = 4}
T4: producer execute counter = register1 {counter = 6}
T5: consumer execute counter = register2 {counter = 4}

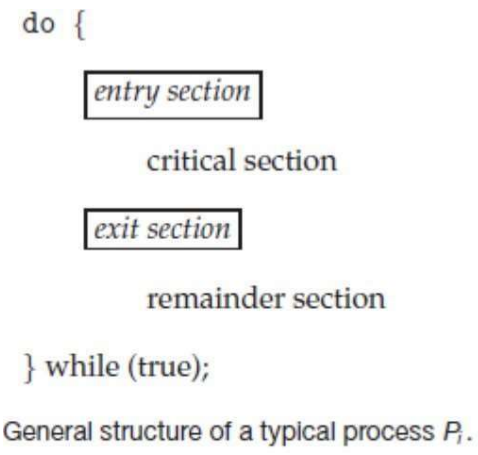
Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at *T*4 and *T*5, we would arrive at the incorrect state "counter == 6".

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place,is called a race condition.

To guard against the race condition ensure only one process at a time can be manipulating the variable or data. To make such a guarantee processes need to be synchronized in some way

2. The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segmentof code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. when one process is executing in its criticalsection, no other process is allowed to execute in its critical section. The critical-sectionproblem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this requestis the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure



General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems:

- **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- **Nonpreemptive kernels**.. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

3. Peterson's Solution

A classic software-based solution to the critical-section problem known as **Peterson's solution**. It addresses the requirements of mutual exclusion, progress, and bounded waiting.

It Is two process solution.

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

The two processes share two variables:

```
int turn;
Boolean flag[2]
```

The variable turn indicates whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready.

- The structure of process *Pi* in Peterson's solution:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

To prove property 1, we note that each *Pi* enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that *P*0 and *P*1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

To prove properties 2 and 3,we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section. once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section. If *Pj* resets flag[j] to true, it must also set turn to i. Thus, since *Pi* does not change the value of the variable turn while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting)

- It proves that
    1. Mutual exclusion is preserved
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

4. Semaphores

The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. So operating-systems designers build software tools to solve the critical-section problem, and this synchronization tool called as Semaphore.

- Semaphore *S is an* integer variable
- Two standard operations modify S: wait() and signal()
  Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S <= 0
          ; // no-op
    S--;
}
signal (S) {
    S++;
}
```

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.

Usage:

Semaphore classified into:

- Counting semaphore: Value can range over an unrestricted domain.
- Binary semaphore (Mutex locks): Value can range only between from 0 & 1. It provides mutual exclusion.

```
Semaphore mutex;    // initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

- Consider 2 concurrently running processes: *S*1; signal(synch);

  In process *P*1, and the statements

  wait(synch);

  *S*2;

  Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Implementation:

The disadvantage of the semaphore is **busy waiting** i.e While a process is in critical section, any other process that tries to enter its critical section must loop continuouslyin the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process spins while waiting for the lock.

Solution for Busy Waiting problem:

Modify the definition of the wait() and signal()operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler,which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, define a semaphore as follows:

```
typedef struct {int
  value;
  struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process. Now, the wait() semaphoreoperation can be defined as:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

and the signal() semaphore operation can be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes, these processes are said to be deadlocked.

Consider below example: a system consisting of two processes, $P0$ and $P1$, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
   .                   .
   .                   .
   .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that $P0$ executes wait(S) and then $P1$ executes wait(Q).When $P0$ executes wait(Q), it must wait until $P1$ executes signal(Q). Similarly, when $P1$ executes wait(S), it must wait until $P0$ executes signal(S). Since these signal() operations cannot be executed, $P0$ and $P1$ are deadlocked.

Another problem related to deadlocks is **indefinite blocking** or **starvation**.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes.

As an example, assume we have three processes—L, M, and H—whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby pre-empting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource R.

This problem is known as priority inversion. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically, these systems solve the problem by implementing a priority-inheritance protocol. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

5. Classic Problems of Synchronization

a) The Bounded-Buffer Problem:

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. In our problem, the producer and consumer processes share the following data structures:

int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

- Code for producer is given below:

```
do {
   . . .
   /* produce an item in next_produced */
   . . .
   wait(empty);
   wait(mutex);
   . . .
   /* add next_produced to the buffer */
   . . .
   signal(mutex);
   signal(full);
} while (true);
```

Code for consumer is given below:

```
do {
   wait(full);
   wait(mutex);
   . . .
   /* remove an item from buffer to next_consumed */
   . . .
   signal(mutex);
   signal(empty);
   . . .
   /* consume the item in next_consumed */
   . . .
} while (true);
```

b) The Readers–Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers– can both read and write
- Problem – allow multiple readers to read at the same time

  Only one single writer can access the shared data at the same time

Several variations of how readers and writers are treated – all involve priorities.

First variation – no reader kept waiting unless writer has permission to useshared object.

Second variation- Once writer is ready, it performs asap.

The structure of writer process:

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Shared Data

- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

The structure of reader process:

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

c) The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.
It is a simple representation of the need to allocate several resources among several processesin a deadlock-free and starvation-free manner.

Solution: One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher *i* isshown in Figure

Several possible remedies to the deadlock problem are replaced by:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

6. Monitors

Incorrect use of semaphore operations:

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed,resulting in the following execution:

```
signal(mutex);
  ...
critical section
  ...
wait(mutex);
```

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
  ...
critical section
  ...
wait(mutex);
```

In this case, a deadlock will occur.

Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Solution:

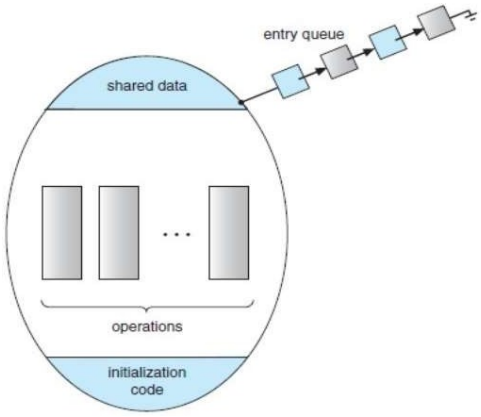
Monitor: An abstract data type—or ADT—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

The syntax of a monitor type is shown in Figure:

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Schematic view of a monitor:



To have a powerful Synchronization schemes a *condition* construct is added to the Monitor.So synchronization scheme can be defined with one or more variables of type *condition.*

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). Theoperation

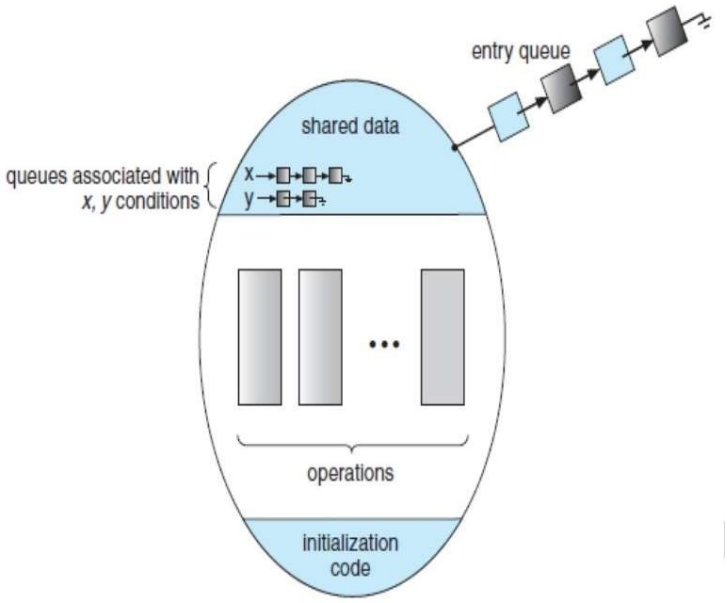
x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation

had never been executed. Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.



5.8.2 Dining-Philosophers Solution Using Monitors

A deadlock-free solution to the dining-philosophers problem using monitor concepts. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Consider following data structure:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

And also declare:

Condition self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain thechopsticks she needs.

A monitor solution to the dining-philosopher problem:

```
monitor DiningPhilosophers
  {
    enum { THINKING; HUNGRY, EATING) state [5]
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
          state[i] = THINKING;
              // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
     }
      void test (int i) {
            if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
             self[i].signal () ;
             }
      }

      initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
      }
 }
```

5.8.3 Implementing a Monitor Using Semaphores

For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
   ...
   body of F
   ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured.

For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
        signal(next);
   else
        signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

5.8.2 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process

that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

The ResourceAllocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type ResourceAllocator.

The monitor concept cannot guarantee that the preceding access sequence will be observed.In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to theresource.
- A process might never release a resource once it has been granted access to theresource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource)

MEMORY MANAGEMENT

Background:

Memory management is concerned with managing the primary memory. Memory consists of array of bytes or words each with their own address. The instructions are fetched from the memory by the cpu based on the value program counter.

Functions of memory management:-

Address Binding:-

- Keeping track of status of each memory location..
- Determining the allocation policy. Memory allocation technique. De-allocation technique.

Programs are stored on the secondary storage disks as binary executable files.

When the programs are to be executed they are brought in to the main memory and placed within a process.

The collection of processes on the disk waiting to enter the main memory forms the input queue.

One of the processes which are to be executed is fetched from the queue and placed in the main memory.

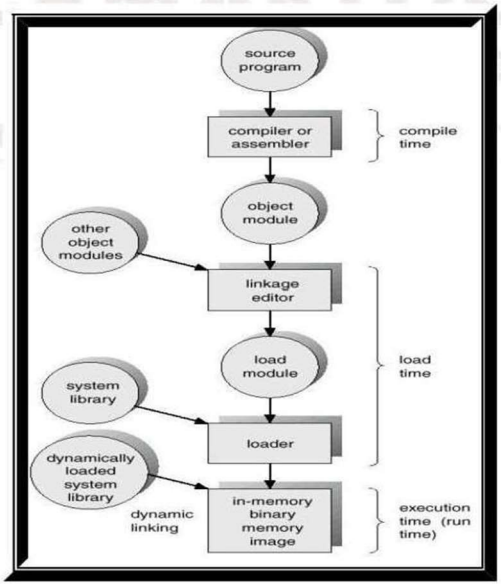
During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space.

During execution the process will go through different steps and in each step the address is represented in different ways.

In source program the address is symbolic.

The compiler converts the symbolic address to re-locatable address.

The loader will convert this re-locatable address to absolute address.



Binding of instructions and data can be done at any step along the way:-

1. Compile time:- If we know whether the process resides in memory then absolute code can be generated. If the static address changes then it is necessary to re-compile the code from the beginning.

2. Load time:-If the compiler doesn't know whether the process resides in memory then it generates the re-locatable code. In this the binding is delayed until the load time.

3. Execution time:- If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.

Logical versus physical address:-

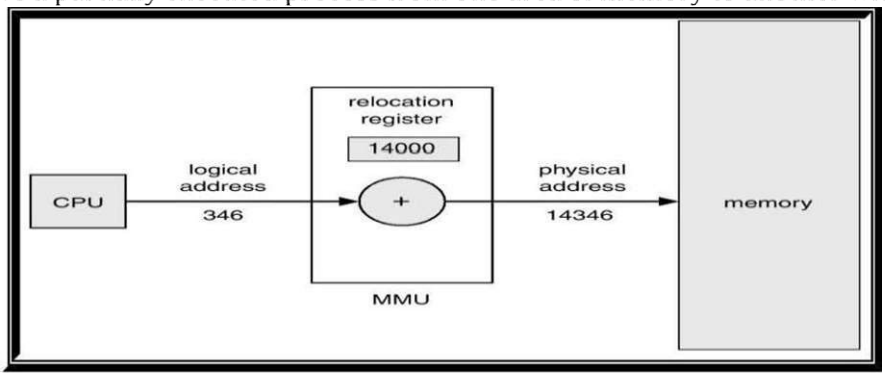
The address generated by the CPU is called logical address or virtual address.

The address seen by the memory unit i.e., the one loaded in to the memory register is called the physical address.
Compile time and load time address binding methods generate some logical and physical address. The execution time addressing binding generate different logical and physical address. Set of logical address space generated by the programs is the logical address space. Set of physical address corresponding to these logical addresses is the physical address space.

The mapping of virtual address to physical address during run time is done by the hardware device called memory management unit (MMU).
The base register is also called re-location register. Value of the re-location register is added to every address generated by the user process at the time it is sent to memory. User program never sees the real physical address.

The figure below shows the dynamic relation. Dynamic relation implies mapping from virtual address space to physical address space and is performed at run time usually with some hardware assistance.

Relocation is performed by the hardware and is invisible to the user. Dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.



Dynamic re-location using a re-location registers The above figure shows that dynamic re-location which implies mapping from virtual addresses space to physical address space and is performed by the hardware at run time.

Re-location is performed by the hardware and is invisible to the user dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.

Dynamic Loading:-

For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory.

Dynamic loading is used to obtain better memory utilization.

In dynamic loading the routine or procedure will not be loaded until it is called.

Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.

Advantage:-

- Gives better memory utilization.
- Unused routine is never loaded.
- Do not need special operating system support.
- This method is useful when large amount of codes are needed to handle in frequently occurring cases.

Dynamic linking and Shared libraries:-

Some operating system supports only the static linking.

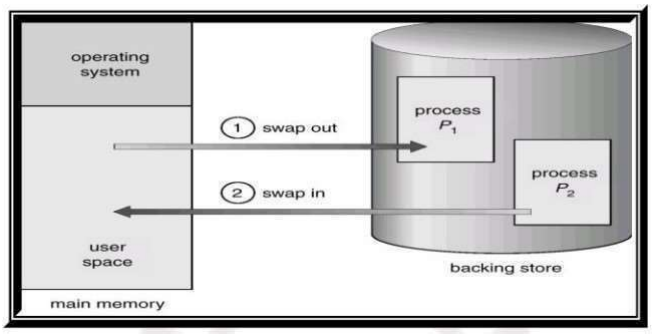
In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time.

With dynamic linking a "stub" is used in the image of each library referenced routine. A "stub" is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present.

When "stub" is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory.
This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library.

More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the program that are compiled with the new version are affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called "shared library".

Swapping:-



Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping.

Eg:- In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as Roll out and Roll in.
Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding.
If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time.

Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run.

Swapping is constant by other factors:-

- To swap a process, it should be completely idle.
- A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped.

Contiguous Memory Allocation:-

Memory allocation

One of the simplest method for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions. In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory.
When process terminates, the memory partition becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied.

When the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory.
When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available.
If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes.

The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:-
First fit:- Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.
Best fit:- It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.
Worst fit:- It allocates the largest hole to the process request. It searches for the largest hole in the entire list.

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes.

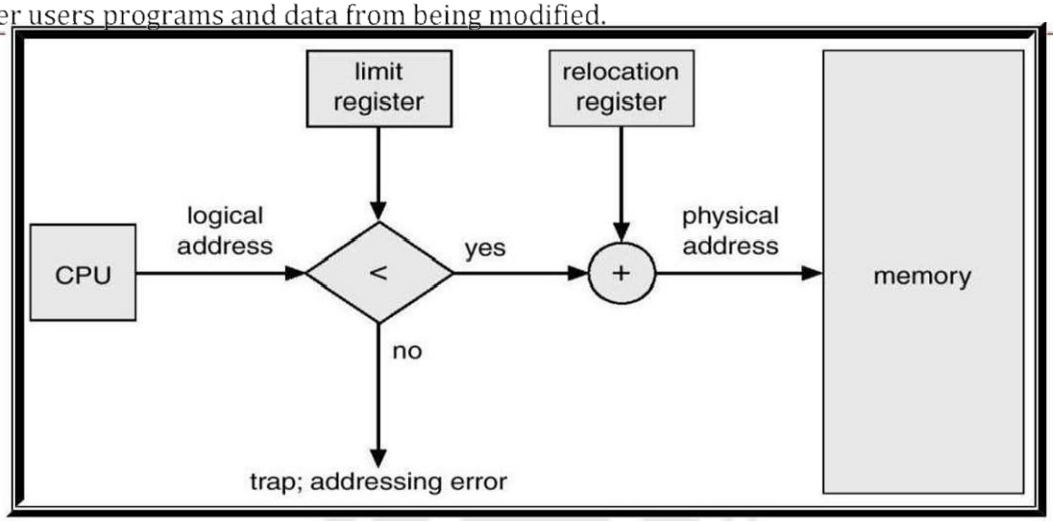
All these algorithms suffer from fragmentation.

Memory mapping and Protection:-

Memory protection means protecting the OS from user process and protecting process from one another. Memory protection is provided by using a re-location register, with a limit register.

Re-location register contains the values of smallest physical address and limit register contains range of logical addresses. (Re-location = 100040 and limit = 74600).
The logical address must be less than the limit register, the MMU maps the logical address dynamically by adding the value in re-location register. When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch.

Since every address generated by the CPU is checked against these register we can protect the OS and other users programs and data from being modified.



Fragmentation:-

Memory fragmentation can be of two types:-

- Internal Fragmentation
- External Fragmentation

In Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition.

Eg:- If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes. External Fragmentation may be either minor or a major problem.

One solution for over-coming external fragmentation is compaction. The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the re-location is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time.
Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing the process to be allocated physical memory whenever the latter is available.

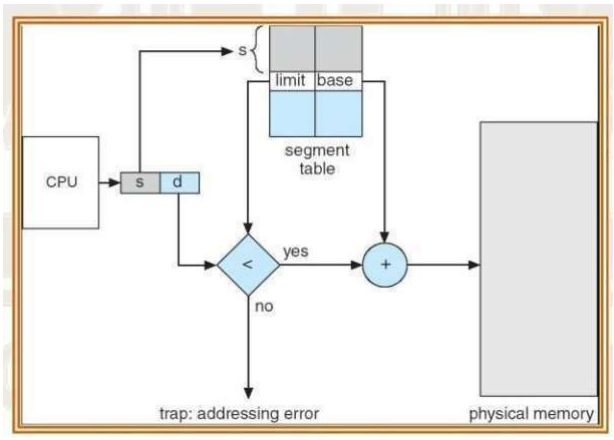
Segmentation:-

Basic method:-

Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack, heap etc.
A logical address is a collection of segments. Each segment has a name and length. The address specifies both the segment name and the offset within the segments.
The users specifies address by using two quantities: a segment name and an offset.
For simplicity the segments are numbered and referred by a segment number. So the logical address consists of <segment number, offset>.

Hardware support:-

We must define an implementation to map 2D user defined address in to 1D physical address.
This mapping is affected by a segment table. Each entry in the segment table has a segment base and segment limit.
The segment base contains the starting physical address where the segment resides and limit specifies the length of the segment.



The use of segment table is shown in the above figure:-

Logical address consists of two parts: segment number's' and an offset'd' to that segment.
The segment number is used as an index to segment table.
The offset 'd' must bi in between 0 and limit, if not an error is reported to OS.
If legal the offset is added to the base to generate the actual physical address.
The segment table is an array of base limit register pairs.

Protection and Sharing:-

A particular advantage of segmentation is the association of protection with the segments.
The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal access to memory like attempts to write in to read-only segment.

Another advantage of segmentation involves the sharing of code or data. Each process has a segmenttable associated with it. Segments are shared when the entries in the segment tables of two different processes points to same physical location.
Sharing occurs at the segment table. Any information can be shared at the segment level. Several segments can be shared so a program consisting of several segments can be shared. We can alsoshare parts of a program.

Advantages:-

- Eliminates fragmentation.
- Provides virtual growth.
- Allows dynamic segment growth.
- Assist dynamic linking.
- Segmentation is visible.