Process Synchronization and Critical Section

On the basis of synchronization, processes are categorized as one of the following two types:

1. **Independent Process:** Execution of one process does not affect the execution of other processes.

2. **Cooperative Process:** Execution of one process affects the execution of other processes.

So far we have discussed Independent processes where each process had there own address space, and they did not require to communicate with other processes. But the problem arises in the case of Cooperative process because resources are shared in Cooperative processes and various processes communicate with each other.
In this respect, we here will be discussing the problems in process synchronization, in the case of single PC inter-process communication, due to concurrent execution.

Let's look at the below example

List of Global Variable:

```


int SIZE = 10;
char buffer[SIZE];
int in = 0, out = 0;
int count = 0


```

Run

The Producer:

```

void producer()
{
    while(true)
    {
        while(count == size)
        {
            ;
        }
        buffer[in] = produceItem();
        in =(in+1) % SIZE;
        count++;
    }
}

```

Run

The Consumer:

```


void consumer()
{
    while(true)
    {
        while(count == 0)
        {
            ;
        }
        consumeItem(buffer[out]);
        out =(out+1) % SIZE;
        count--;
    }
}


```

Run

Here we have written codes for Producer and Consumer. We have a few global variables which are shared among both the Producer and Consumer The Producer keeps on adding elements to by increasing the count by 1, each time it adds an element. The consumer, on the other hand, decreases the value of count by 1 each time an item is taken out. This process goes on and is enitled to be preempted at any point of time when the Consumer and producer are needed to be switched between.

Let's delete the unnecessary part of the code and focus on the locus of understanding, count++ and count--. These are not any simple instructions but are converted to the following internal assembly instructions.

count in Producer:

```


reg = count;
reg = reg + 1;
count = reg;


```

Run

count in Consumer:

```


reg = count;
reg = reg - 1;
count = reg;


```

Run

Let's analyze it: Let the value of count be 8 in the Producer part. If in the producer part, just after the register is incremented and before the count gets updated, the process is preempted to some other process, the problem arises. Now if the control flows to Consumer, the reg of the consumer will get value 8 instead of 9. On completion the count = 7 in Consumer part. When the control flows back to the Producer, the process resumes from where it had stopped and count in Producer gets 9. Therefore we have inconsistent values for Producer and Consumer.
This inconsistency among the interdependent stream of execution where each step is related to one another leads to the Race Condition.

Race Condition: Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Let's look at another example:

```


// Global Variable
x = 2

// first function
void fun1()
{
    .
    .
    .
    if (x == 2) // Preemption occurs
        y = x + 5;
    .
    .
    .
}
```

```
// second function
void fun2()
{
     .
     .
     .
     x++;
     .
     .
     .
}

```

Run

If there is no preemption in this program, then the execution goes smoothly and y becomes 7. But suppose just after the if condition in fun1(), preemption occurs, then the control flows to fun2(), where x is incremented to 3. When the flows go back to fun1() the if condition will never satisfy and we won't get the desired value of y as 7. This happens because of the Race Condition.

Critical Section: Critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables which need to be synchronized to maintain consistency of data variables.
Let's look at the following example:

```


// Global variable
int balance = 100;

void Deposit(int x)
{
    // Preemption occurs
    balance = balance + x;  // Critical section
}

void Withdraw(int x)
{
    balance = balance - x;  // Critical section
}

```

Run

Due to preemption, we get the value of balance as 100 in Deposit() and as 90 in Withdraw(). This inconsistency is called the Race Condition and it occurs due to the preemption along the critical section of the problem. Therefore, we need a mechanism to deal with such cases. The mechanism includes putting up conditions in the entry or exit section.

Entry Section: It is part of the process which decides the entry of a particular process in the Critical Section, out of many other processes.
Exit Section: This process allows the other process that is waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical Section can be removed through this Exit Section.
Remainder Section: The other parts of the Code other than Entry Section, Critical Section and Exit Section is known as Remainder Section.

Any solution to the critical section problem must satisfy these four requirements or we can say that there are four goals of synchronization algorithm:

1. **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

2. **Progress:** If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will

enter in the critical section next, and the selection can not be postponed indefinitely.

3. **Bounded Waiting(Fair):** Abound must exist 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.

4. **Performance:** The mechanism that allows only one process to enter the critical section should be fast. This mechanism is referred to as the locking mechanism and can be implemented in two ways, hardware or software mechanism. The hardware-based mechanism is generally faster since it only involves the registers.

Note: Mutual Exclusion and Progress are mandatory goals for synchronization mechanism.

− Overview of Process Synchronization

Overview of Synchronization Mechanism:

1. **Disabling Interrupts:** In this, a process tells the processor that it is undergoing a critical section job and it must not be interrupted before its completion. Now, this might sound good but it comes with various serious problems. This sort of mechanism works only with a single processor system and not in multiprocessor systems. The second and more serious problem is that allowing user processes to stop interruption might lead to system failure or security issues. Therefore, this is not feasible.

2. **Locks or Mutex:** A mutex is a binary variable whose purpose is to provide a locking mechanism. It is used to provide mutual exclusion to a section of code, which means only one process can work on a particular code section at a time. This can be implemented in both software and hardware. It is the building block of all other mechanisms.

3. **Semaphores:** A semaphore is simply a variable. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

4. **Monitors:** Monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

Few applications of Synchronization are:

- Producer-Consumer Problem
- Dining Philosopher Problem
- Bounded Buffer Problem
- Reader-Writer Problem

− Process Synchronization - Atomic transaction

In the database, operations frequently need to carry out atomic transactions, in which the entire transaction must either complete or not occur at all. The classic example is a transfer of funds, which involves withdrawing funds from one account and depositing them into another - Either both halves of the transaction must complete, or neither must complete. Operating Systems can be viewed as a database in terms of needs and problems, in that, an OS can be said to manage a small database of process-related information. As such, OSes can benefit from emulating some of the techniques originally developed for databases. Here we first look at some of those techniques, and then how they can be used to benefit OS development.

System Model:

- A transaction is a series of actions that must either complete in its entirety or must be rolled-back as if it had never commenced.
- The system is considered to have three types of storage:
    - When system crashed volatile storage usually gets lost.
    - When a system crashed non-volatile storage usually survives system crashes, but may still get lost.
    - Stable storage "never" gets lost or damaged. In practice, this is implemented using multiple copies stored on different media with different failure modes.

Log-Based Recovery:

- Before each step of a transaction is conducted, an entry is written to a log on stable storage:
    - Each transaction has a unique serial number.
    - The first entry is a "start"
    - Transaction number is mentioned for every data changing entry specifically(the old value, and the new value).
    - The final entry is a "commit".
    - All transactions are idempotent - The can be repeated any number of times and the effect is the same as doing them once. Likewise, they can be undone any number of times and the effect is the same as undoing them once. (I.e. "change x from 3 to 4", rather than "add 0 to x").
- Any transaction which has "commit" recorded in the log can be redone from the log information, after a crash. Any transaction can be undone if it has "start" but not "commit".

Checkpoints:

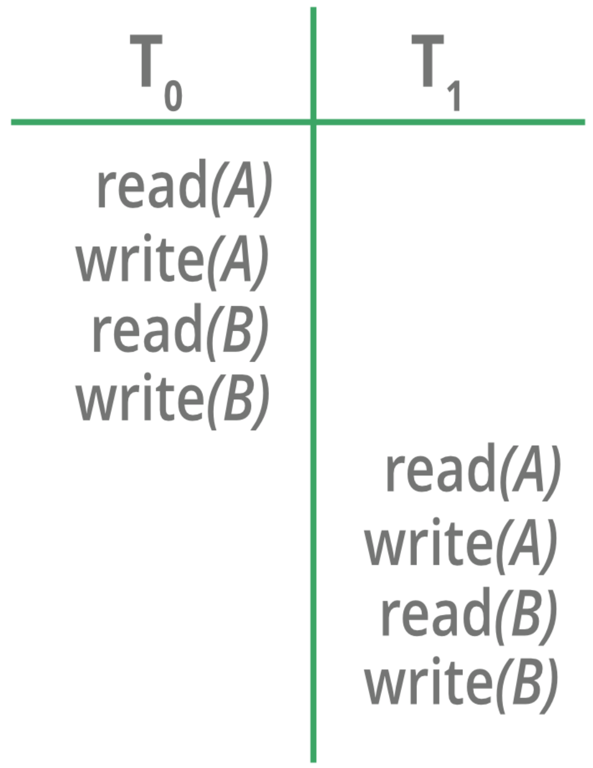
- When system got crashed, all data can be recovered using the system described above, by going through the entire log and performing either redo or undo operations on all the transactions listed there.
- Unfortunately this approach can be slow and wasteful because many transactions are repeated that were not lost in the crash.
- Alternatively, one can periodically establish a checkpoint, as follows:
    - Use stable storage to Write all data that has been affected by recent transactions (since the last checkpoint).
    - Write a entry to the log.
- One only needs to find transactions that did not commit to recovering crash prior to the most recent checkpoint. Specifically one looks back from the end of the log for the last record and then looks backward from there for the most recent transaction that started before the checkpoint. Only that transaction and the ones more recent need to be redone or undone.

Concurrent Atomic Transactions:

- As discussed above log-based recovery assumed that only one transaction could be conducted at a time. For now, we can relax that restriction, and allow multiple transactions to occur concurrently, while still keeping each individual transaction atomic.

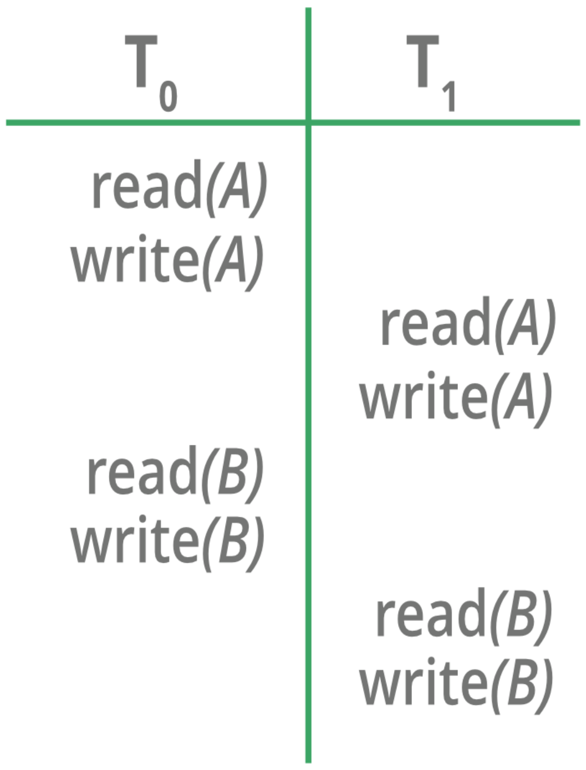
  Serializability:

    - **Figure 1** below shows a schedule in which transaction 0 reads and writes data items A and B, followed by transaction 1 which also reads and writes A and B. The two transactions are conducted serially this is termed a serial schedule. There are N! possible serial schedules for any N transactions.
    - A nonserial schedule is one in which the steps of the transactions are not completely serial, i.e. they interleave in some manner. These schedules are not necessarily bad or wrong, so long as they can be shown to be equivalent to some serial schedule. A nonserial schedule that can be converted to a serial one is said to be conflict serializable, such as that shown in **Figure 2** below. Legal steps in the conversion are as follows:
        - Two operations from different transactions are said to be conflicting if they involve the same data item and at least one of the operations is a write operation. If operations from two transactions are either involve different data items or do not involve any write operations then these transactions are non-conflicting.
        - If two operations in a schedule are from two different transactions and if they are non-conflicting, can be swapped.
        - A schedule is conflict serializable if there exists a series of valid swap that converts the schedule into a serial schedule.



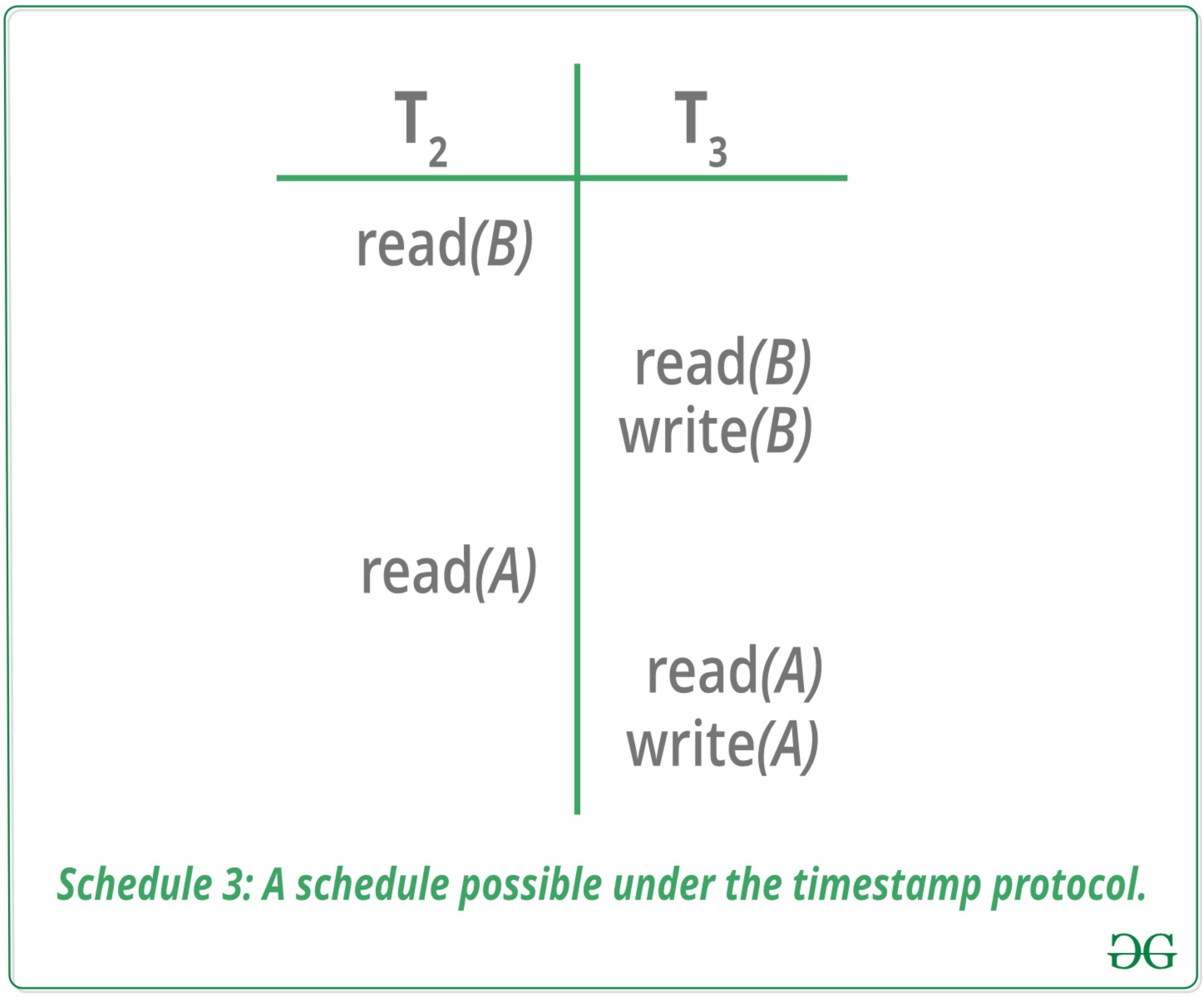
T_0	T_1
read(*A*)	
write(*A*)	
read(*B*)	
write(*B*)	
	read(*A*)
	write(*A*)
	read(*B*)
	write(*B*)

Schedule 1: A serial schedule in which T_0 is followed by T1



T_0	T_1
read(*A*)	
write(*A*)	
	read(*A*)
	write(*A*)
read(*B*)	
write(*B*)	
	read(*B*)
	write(*B*)

Schedule 2: A concurrent serializable schedule.


T_2
T_3
read(B)
read(B)
write(B)
read(A)
read(A)
write(A)
Schedule 3: A schedule possible under the timestamp protocol.

Locking Protocol:

- Serializability can be assured by using locks on data items during atomic transactions.
- Exclusive and shared locks correspond to the Readers and Writers problem discussed above.
    - The two-phase locking protocol operates in two phases:
    - During the growing phase, the transaction continues to gain additional locks on data items as it needs them, and has not yet relinquished any of its locks.
    - In the shrinking phase, it relinquishes locks. After releasing any lock a transaction then is in the shrinking phase and cannot acquire any more locks.
- The two-phase locking protocol can be proven to yield serializable schedules, but it does not guarantee avoidance of deadlock. There may also be perfectly valid serializable schedules that are unobtainable with this protocol.

Timestamp-Based Protocols:

- Each transaction is issued a unique timestamp entry before it begins execution under the timestamp protocol. The system clock is accessed by all processes on the system for the system time or some non-decreasing serial number. TS(T_i) is the timestamp for transaction.
- All generated schedules are equivalent to a serial schedule generated in timestamp order.
- There are two timestamps the W-timestamp and the R-timestamp associated with each data. The timestamp of the last transaction to successfully write the data item is the W-timestamp, and the stamp of the last transaction to successfully read from it the R-timestamp. (Note: These are the respective transaction's timestamps of the, not the time at which the read or write occurred.)

The timestamps are used in the following manner:

Suppose transaction T^i issues a read on data item Q:

- If TS(T_i) < the W-timestamp for Q, then it is attempting to read data that has been changed. T_i is rolled back and restarted with a new timestamp.
- If TS(T_i) > the W-timestamp for Q, then the R-timestamp for Q is updated to the later of its current value and TS(T_i).

Suppose T_i issues a write on Q:

- If TS(T_i) < the R-timestamp for Q, then it is attempting to change data that has already been read in its unaltered state. T_i is rolled back and restarted with a new timestamp.
- If TS(T_i) < the W-timestamp for Q it is also rolled back and restarted, for similar reasons.
  Else, the operation proceeds and the W-timestamp for Q is updated to TS(T_i).

Inter-process Communication

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
Though one can think that those processes, which are running independently, will execute very efficiently but in practice, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity.
Interprocess communication (IPC) is a set of interfaces, which is usually programmed in order for the programs to communicate between a series of processes. This allows running f program concurrently in an Operating System. These are methods in IPC:

1. **Pipes (Same Process) -** This allows the flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until the input process receives it which must have a common origin.

2. **Names Pipes (Different Processes) -** This is a pipe with a specific name it can be used in processes that don't have a shared common process origin. E.g. in FIFO, where the data is written to a pipe, is first named.

3. **Message Queuing -** This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.

4. **Semaphores -** This is used in solving problems associated with synchronization and to avoid race condition. These are integer values which are greater than or equal to 0.

5. **Shared memory -** This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.

6. **Sockets -** This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent.

Inter-Process Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, FIFO and message queue – is that for two processes to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, FIFO or message queue.

- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.

- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 reads and 2 writes). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

```
ftok(): is use to generate a unique key.

shmget():  int shmget(key_t,size_tsize,intshmflg); upon successful completion,
shmget() returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat()
shmid is a shared memory id. shmaddr specifies a specific address to use but we should set
it to zero and OS will automatically choose the address.

shmdt(): When you're done with the shared memory segment, your program should
detach itself from it using shmdt(). int shmdt(void *shmaddr);

shmctl(): when you detach from shared memory,it is not destroyed. So, to destroy
shmctl() is used.  shmctl(int shmid,IPC_RMID,NULL);
```

Interprocess Communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

- **Shared Memory** Communication between processes using shared memory requires processes to share some variable and it completely depends on how the programmer will implement it.
  The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

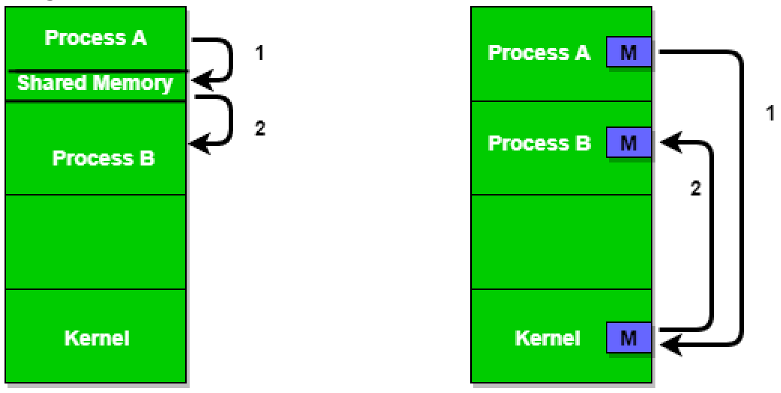
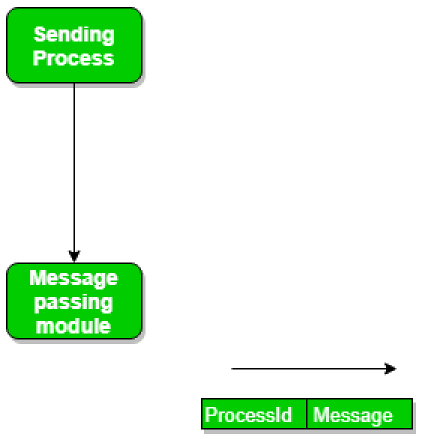
  

  Figure 1 - Shared Memory and Message Passing

  Example of shared memory: Producer consumer problem

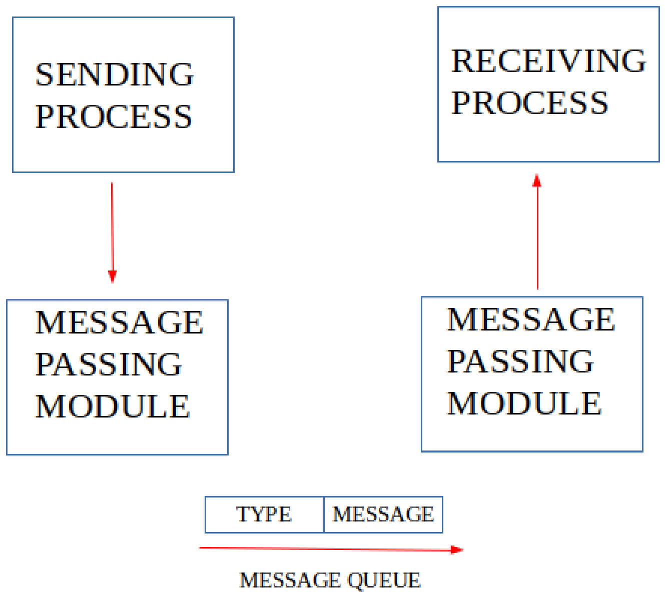
- **Message passing:** In message passing method processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

  - Establish a communication link (if a link already exists, no need to establish it again.)
  - Start exchanging messages using basic primitives.
    We need at least two primitives:
    – **send**(message, destinaion) or **send**(message)
    – **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for the programmer and if it is of variable size then it is easy for the programmer but complicated for the OS designer. A standard message can have two parts: **header and body.** The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, the priority. Generally, the message is sent using the FIFO style.
A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.
New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type of field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue that can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. The process must share a common key in order to gain access to the queue in the first place.



System calls used for message queues:

```
ftok(): is use to generate a unique key.
```

```
msgget(): either returns the message queue identifier for a newly created message
queue or returns the identifiers for a queue that exists with the same key value.

msgsnd(): Data is placed on to a message queue by calling msgsnd().

msgrcv(): messages are retrieved from a queue.

msgctl(): It performs various operations on a queue. Generally it is use to
destroy message queue.
```

Locks for Synchronization

Let's understand the working of Lock mechanism using the following example:

```


// Global variables
int amount = 100;
bool lock = false;

void Deposit(int x)
{
    // Critical section
    amount = amount + x;
}

void Withdraw(int x)
{
    // Critical section
    amount = amount - x;
}

```

Run

Previously we saw that this problem provided inconsistent results. Now, let's add the lock mechanism into this piece of code:

```

// Global variables
int amount = 100;
bool lock = false;

void Deposit(int x){
    // entry section
    while(lock == true){ ; }
    // Preempted
    lock = true;

    // Critical section
    amount = amount + x;

    // exit section
    lock = false;
}

void Withdraw(int x){
    // entry section
    while(lock == true){ ; }
    lock = true;

    // Critical section
    amount = amount - x;

    // exit section
    lock = false;
}
```

Run

Let's analyze the mechanism. Although we have applied the locking mechanism, still the code doesn't guarantee mutual exclusion that is a mandatory requirement. Suppose in the deposit part, the process gets preempted just after the while statement and the control flows to the Withdraw part. After the execution, when the control flows back to the Deposit part, it enters the critical section. Therefore, the principle of mutual exclusion is violated, as both the processes might enter the critical section and cause a race condition.
To avoid this we use a hardware-based implementation called Test and Set or the TSL lock mechanism.
Deposit and Withdraw mechanism using TSL lock:

```
// Global variables
int amount = 100;
bool lock = false;

void Deposit(int x){
    while(test_and_set(lock)){ ; }
    // Critical section
    amount = amount + x;
    lock = false;
}
void Withdraw(int x){
    // entry section
    while(test_and_set(lock)){ ; }
    // Preempted
    lock = true;
    // Critical section
    amount = amount - x;
    // exit section
    lock = false;
}
// TSL Lock mechanism
bool test_and_set(bool *ptr){
    bool old = *ptr;
    *ptr = true;
    return old;
}
```

Run

This TSL lock work both ways, it waits while the lock is true and also the lock is updated as true if it is false. This function doing both the things has to be atomic. So this is an atomic operation to acquire the lock and once it is acquired nobody else could acquire the lock. This solves the basic problem of Process Synchronization.

- Process synchronization - Semaphore and MUTEX

Problem with Locking Mechanism: In the previous lectures, we have seen how the locking mechanism facilitates process execution by applying locks when a process enters the critical section and releasing it when the job is done. While doing this other processes had to wait and they went into a waiting loop until the process inside completes its execution. This is a major problem and can be avoided using Semaphores.

Semaphore: Semaphore is simply a variable. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.

1. **Counting Semaphore:** Let's see how this works using a restroom analogy. Suppose there is a restroom(critical section) which is being used by a person(process). When another process comes, he had to wait until the previous process comes out and releases the lock. All the upcoming processes had to go into a waiting loop. This problem can be avoided using a semaphore. Consider a semaphore is like a guard who maintains a record and as long as a process is inside the restroom, it tells all other processes to sleep or do any other task and not to go into a waiting loop. When the process inside completes its execution, the guard informs the process in the queue and it can enter the restroom.
   To perform this complex process, the guard or the semaphore maintains a record. It maintains two functions namely, wait and signal.

wait(): The wait keeps a record of the count. The count is initially assigned a value equal to the number of available resources. Every time a process is allocated to a resource, the value of count is decreased by one. Once the value of count is negative, it indicates that all the resource is underuse and the address of the PCB of the upcoming processes are stored in a queue. This way, the processes are not kept waiting. The negative value now indicates the number of processes outside who need to use the resource.

signal(): The signal function increments the value of count by one every time a process finish using a resource which can be made available to some other process. The guard wakes or signals a process in the queue when the count is decremented, that a resource is free to use and releases a process in the FCFS basis.

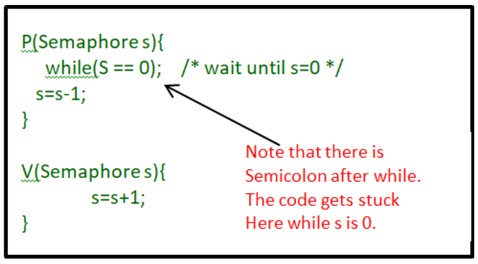
Let's write a pseudo code for the above process. For start we assume there are 3 available resources. So count will be initialized as 3. We have a semaphore containing a count and queue, to keep track of the processes.

PseudoCode:

```
// A semaphore structure having count and queue
struct Sem{
    int count; Queue q;
}
// Creating variable s of semaphore where count = 3 and q is empty
Sem s(3, empty);

// Used while entering into the washroom or the critical section
void wait(){
    // process before entering CS
    s.count--;

    // No resource is available
    if(s.count < 0){
    // No resource is available
        1. Add the caller process P to q;
        2. sleep(P);
    }
}
// Used while exiting from the washroom or the critical section
void signal(){
    // process after exiting CS
    s.count++;
    // Checking for process in queue
    if(s.count <= 0)
    {
        1. Remove a process P from q
        2. Wakeup(P);
    }
}
```

Run

Now let's look at two operations that can be used to access and change the value of the semaphore variable and how this was originally implemented by Dijkstra:



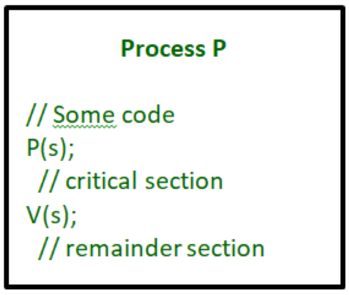
```
P(Semaphore s){
   while(S == 0);   /* wait until s=0 */
  s=s-1;
}

V(Semaphore s){
        s=s+1;
}
```

Note that there is Semicolon after while. The code gets stuck Here while s is 0.

Some point regarding P and V operation

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.

3. A critical section is surrounded by both operations to implement process synchronization. See below image critical section of Process P is in between P and V operation.



```
Process P

// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

This PV operation comes with few problems like busy waiting, as the while loop keeps running when the process does not get a resource or critical section. This way the CPU cycle gets wasted.
Another problem is that of bounded waiting when a process because of preemption might not get a resource for a very long time. These problems have been solved in the previous discussion where we have used count, queue and sleep and wake up concept to make efficient use of CPU ans see that no CPU cycle is wasted.

2. **Binary Semaphore:** In a binary semaphore, the counter logically goes between 0 and 1. You can think of it as being similar to a lock with two values: open/closed. In binary semaphore, only two processes can enter into the critical section at any time and mutual exclusion is also guaranteed. Let's look a the pseudo-code below to understand the logic behind the implementation of binary semaphore. This Binary Semaphore can be used as mutex too with the additional functionalities like sleep and wake concept etc. Let's look at the pseudo-code:

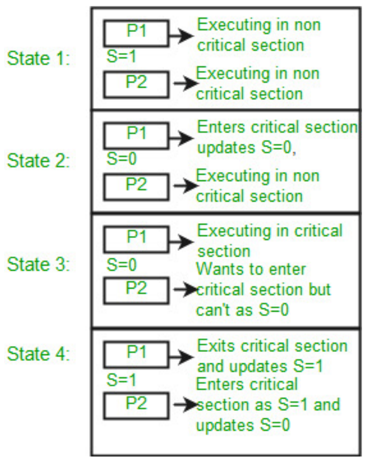
```


// Binary Semaphore containing
// a boolean value and queue q
// to keep track of processes
// entering critical section
struct BinSem
{
    bool val;
    Queue q;
};

// Global initialization
BinSem s(true, empty);

// wait() is called before critical section
// during entry
void wait()
{
    // Checking if critical section is
    // available or not
    if (s.val == 1)
        // acquiring the critical section
        s.val = 0;

    // if not available
    else
    {
        1. Put this process P in q;
        2. sleep(P);
```

Run

```
// Binary Semaphore containing
// a boolean value and queue q
// to keep track of processes
// entering critical section
struct BinSem
{
    bool val;
    Queue q;
};
```

```
// Global initialization
BinSem s(true, empty);

// wait() is called before critical section
// during entry
void wait()
{
    // Checking if critical section is
    // available or not
    if (s.val == 1)
        // acquiring the critical section
        s.val = 0;

    // if not available
    else
    {
        1. Put this process P in q;
        2. sleep(P);
    }
}

// signal() is called after critical section
// during exit
void signal
{
    if (q is empty)
        s.val = 1;
    else
    {
        1. Pick a process P from q;
        2. Wakeup(P);
    }
}
```

Implementation of Mutual Exclusion: Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details.



State 1:
P1
Executing in non critical section
S=1
P2
Executing in non critical section
State 2:
P1
Enters critical section updates S=0,
S=0
P2
Executing in non critical section
State 3:
P1
Executing in critical section
S=0
P2
Wants to enter critical section but can't as S=0
State 4:
P1
Exits critical section and updates S=1
S=1
P2
Enters critical section as S=1 and updates S=0

The description above is for binary semaphore which can take only two values 0 and 1. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instances is 4. Now we initialize S = 4 and rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or wait() function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls signal function and the value of semaphore becomes positive.

Problem in this implementation of semaphore
Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle. To avoid this another implementation is provided below.

```


P(Semaphore s)
{
    s = s - 1;
    if (s <= 0) {

        // add process to queue
        block();
    }
}

V(Semaphore s)
{
    s = s + 1;
    if (s <= 0) {

        // remove process p from queue
        wakeup(p);
    }
}

```

Run

- In this implementation whenever process waits it is added to a waiting queue of processes associated with that semaphore. This is done through the system call block() on that process. When a process is completed it calls signal function and one process in the queue is resumed. It uses wakeup() system call.

    - A mutex is a binary variable whose purpose is to provide a locking mechanism. It is used to provide mutual exclusion to a section of code, which means only one process can work on a particular code section at a time.

    - There is misconception that binary semaphore is same as mutex variable but both are different in the sense that binary semaphore apart from providing locking mechanism also provides two atomic operation signal and wait, means after releasing resource semaphore will provide signaling mechanism for the processes who are waiting for the resource.

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore?

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as *synchronization primitives*). *Why do we need such synchronization primitives? Won't be only one sufficient?* To answer these questions, we need to understand a few keywords. Please read the posts on the critical section. We will illustrate with examples to understand these concepts well, rather than following the usual OS textual description.

The producer-consumer problem:

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

- Using Mutex:
  A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.

  At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using a semaphore.

- Using Semaphore:
  A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

- There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is a binary semaphore. *But they are not*! The purpose of mutex and semaphore is different. Maybe, due to the similarity in their implementation a mutex would be referred to as a binary semaphore.

- Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

- Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

Note: The content is generalized explanation. Practical details vary with implementation.

General Questions:

1. *Can a thread acquire more than one lock (Mutex)?*
   Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. *Can a mutex be locked more than once?*
   A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. *What happens if a non-recursive mutex is locked more than once.*
   Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in a deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of the mutex and return if it is already locked by the same thread to prevent deadlocks.

4. *Are binary semaphore and mutex same?*
   No. We suggest to treat them separately, as it is explained by signaling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with a mutex. We will cover these in a later article.

   A programmer can prefer mutex rather than creating a semaphore with count 1.

5. *What are a mutex and critical section?*
   Some operating systems use the same word *critical section* in the API. Usually, a mutex is a costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

6. *What are events?*
   The semantics of mutex, semaphore, event, critical section, etc... are the same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for the exact details.

7. *Can we acquire mutex/semaphore in an Interrupt Service Routine?*
   An ISR will run asynchronously in the context of the current running thread. It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR. The ISR is meant to be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

8. *What we mean by "thread blocking on mutex/semaphore" when they are not available?*
   Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of the processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

9. *Is it necessary that a thread must block always when the resource is not available?*
   Not necessary. If the design is sure '*what has to be done when the resource is not available*', the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

   For example POSIX pthread_mutex_trylock() API. When the mutex is not available the function returns immediately whereas the API pthread_mutex_lock() blocks the thread till resource is available.

Monitors

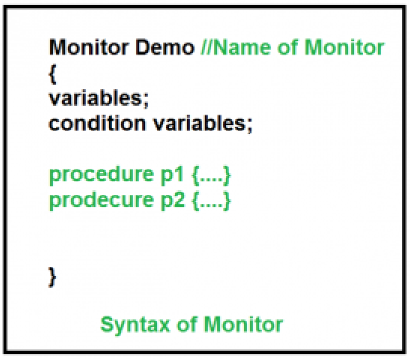
The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.

2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

3. Only one process at a time can execute code inside monitors.

Syntax of Monitor



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

The idea of implementing monitors is that instead of going into the complexities of acquiring, lease, wait for the signal, we use a class and put various functions inside it and synchronize the whole mechanism.

Example:

```java


// Illustrating Monitor concept in Java
// higher level of synchronization
class AccountUpdate
{
    // shared variable
    private int bal;
    // synchronized method
    void synhronized deposit(int x)
    {
        bal = bal + x;
    }

    // synchronized method
    void synhronized withdraw(int x)
    {
        bal = bal - x;
    }
}

```

Run

Condition Variables

Two different operations are performed on the condition variables of the monitor.

```
Wait.

signal.
```

let say we have 2 condition variables

condition x, y; //Declaring variable

Wait operation x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)

  // Ignore signal

else

  // Resume a process from block queue.
```