Synchronization: –

Mutex: –

Mutex m is an object. It has a state. State of mutex \in \{ free, lock \}

Mutex has two methods:
1) m.wait() for lock
2) m.signal() for unlock

Entering \rightarrow wait()

Leaving \leftarrow wait()

\| wait() \| - \| signal() \| \leq 1

At any point in time, there can be only one more wait than signal. It means only one thread can be in critical section at a one time.

A state between thread and mutex can be either null or waiting.

Set of threads = \{ threads \ | \ th \cdot m = waiting \}
Mutex can be implemented as follows:

lock:
loop:
    if !get-spinlock
        queue self on m.queue
        sleep
        goto loop

unlock: release spinlock
        wake first thread on m.queue (if any)

On SMP you may want to call spinlocks for some time before calling lock. The reason for this is that you may want to wait a little so that some process may
release the lock very soon and the process will not have to call lock.

There are two issues that may occur with mutex implementation.

1) What happens if you free a mutex that's already free?
2) What happens if you lock one thread and unlock another thread?

The implementation of mutex requires that programmers use it correctly. Only the thread that locks a mutex should have the right to unlock it.

**Recursive Mutex:**
If current mutex has locked a mutex then it is safe to lock if the same thread locks it again. It is called recursive mutex.
It is useful from code structure point of view. Mutex should be associated with data structure rather than code.
Producer Consumer Problem :-
It is also called bounded buffer problem.

producers(s) \rightarrow write \rightarrow \text{?} \rightarrow consumers(s) \leq N \rightarrow read

Circular Buffer

Conditions of Buffer:
empty : \quad H = T
full : \quad H = (T - 1) \mod N

Operations on Buffer :-
write : \quad \ast H ++ = \text{item}
read : \quad \text{item} = \ast T ++

In case of single producer and single consumer, we don't need any locking mechanism.

Another practical digression from this problem is if we use spin lock and busy waiting, busy waiting is more efficient on multiprocessors.
There can be two problems in producer consumer problem:

1. Multiple consumers, multiple producers. We will need some synchronization for multiple producers & consumers. Solution to this problem is use of mutex m.

   write:
   m.wait()
   *H++ = item
   m.signal()

   read:
   m.wait()
   item = *T++
   m.signal

This solution fixes the problem of multiple readers and multiple writers.
We would like the writer to lock the buffer until there is some space in buffer and reader to lock the buffer until there are some items in the buffer.

**Solution:**

- Use of semaphore:
  - Semaphore \( s = \text{sem}(N) \)
  - Has two operations: `wait()` & `signal()`

For any \( |\# \text{ of returns from wait}| - |\# \text{ of signals}| \leq N \)

We will introduce two semaphores `free` and `data`.

- Initial value of `free` = `sem(N)`
- Initial value of `data` = `sem(0)`

Free space starts at \( N \), it decreases by 1 for every write and increases by 1 for every read.

So, the solution to problem 2 using above 2 semaphores is as follows:

**Write:**
- `free.wait()`
- `m.wait()`
- \( \times H++ = \text{item} \)
- `m.signal()`
- `data.signal()`

**Read:**
- `data.wait()`
- `m.wait()`
- \( \times T++ = \text{item} \)
- `m.signal()`
- `free.signal()`
Other names for \texttt{wait()} & \texttt{signal}:

\begin{align*}
\texttt{wait()} & \quad \texttt{signal()} \\
\texttt{P()} & \quad \texttt{V()} \\
\texttt{down()} & \quad \texttt{up()}
\end{align*}

In case of mutex, which is binary semaphore, we call these lock and unlock.

\underline{Rendezvous:--}

A rendezvous is a technique we can use when we have two processes running and we want to synchronize their execution at a certain point. We may know that process 1 will reach point A before process 2 reaches point B, and wish them to synchronize at these points (following figure).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{rendezvous_diagram.png}
\caption{Rendezvous diagram}
\end{figure}
When process 1 reaches point A, it should wait there until process 2 reaches point B.

Solution to Rendezvous using Semaphore:

If we know that one process will reach the rendezvous (meeting point) ahead of the other one, then we can use a semaphore to do the synchronization. A semaphore is created with a count of zero, and the first process to reach the rendezvous calls wait() on the semaphore. Since there are no resources available, it will block.

When the second process subsequently reaches rendezvous, it calls signal() to unblock the first process. At this point the two processes have synchronized and both can proceed.

Implementation:

We can use two semaphores mutex and qv.

```
#include <semaphore.h>

int a, b, n = 0;

r. meet (arg) {
  mutex.lock();
  tmp = ++n
  mutex.unlock();
}

if tmp = 1
  a = arg
  qv.wait();
  return b
else
  b = arg
  qv.signal();
  return a
```
If you have a shared variable and lock at its state, then you can modify it and copy state in a local variable. In this way, you can drop mutex and don’t need locking since you have your own copy of variable.

```
meet (5) |
  \|
  n = 1
  m = 1
  a = 5
  q = wait ()
  n = 2
  b = 2
  signal
  signal
  return 2
  return 5
```

There is a better way to do this using Monitors:

Monitor is a user-defined object that makes use of many functions. It’s a class where each object has instance variables and condition variables.

```
class Monitor:
    class with:
    instance variables
    condition variables (c1, c2, ...)
    methods:
      condition.wait() block thread
      condition.signal() unblock thread
      condition.broadcast() unblock all waiting threads
```
A monitor has implicit mutex. Only one thread can be in mutex.

You enter monitor:
when you enter method
OR \rightarrow return from wait

You leave monitor:
when you leave ← method
OR ← block or wait

With monitors we can create synchronization objects that use our own synchronization lock.

People refer to two different types of monitor semantics:

- Hoar

  A thread checks a condition if it is true, not true, go to sleep, awake when some other thread signals it.

- Mesa

  When a thread awakes after getting a signal, it goes back and check if condition is still true.

Broadcasts concept
Monitor Implementation of Rendezvous:

```java
class rendezvous
{
    monitor {
        int a, b, n
        cond c
        meet (arg) {
            if ++n == 1
                a = arg
                c.wait()
            return b
        else
            n = 0
            b = arg
            c.signal()
            return a
        }
    }
}
```

Monitor implementation provides implicit mutex. Implicit mutex always provides mutual exclusion you want. It is used in POSIX Threads.