Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** is an integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`** *//Dutch words Probeer (try) and Verhoog (increment)//*

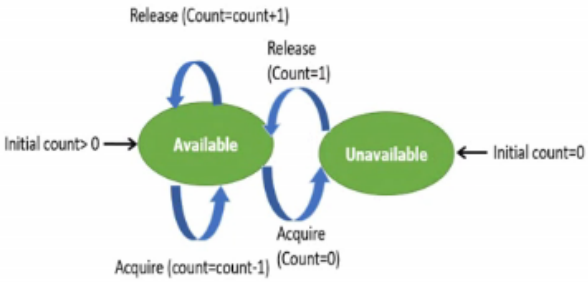
Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

Definition of the **`signal()` operation**

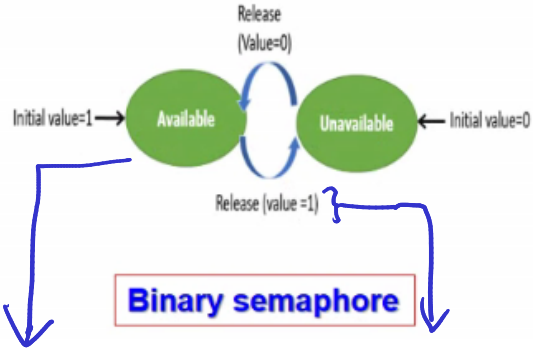
```
signal(S) {
    S++;
}
```

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can only be 0 or 1
  - Same as a **mutex lock**
- We can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems



0, 1, 2 --- n

Counting semaphore



Binary semaphore

- A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal()
- The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S)
{ while(S<=0); // busy waiting
S--;
}
signal(S)
{
S++;
}
```

Semaphores are of two types:

- ✓ **Binary Semaphore –** This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
- ✓ **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Binary Semaphore vs mutex

Data Type	Semaphore is an integer variable.	Mutex is just an object.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource management	If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads.	You can have multiple program threads in mutex but not simultaneously.
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.
Types	Types of Semaphore are counting semaphore and binary semaphore and	Mutex has no subtypes.
Operation	Semaphore value is modified using wait () and signal () operation.	Mutex object is locked or unlocked.
Resources Occupancy	It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes >1.	In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released.

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – INITIALIZED TO 1; integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks** (but are not mutex)
- We can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

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

Counting semaphores

innovate achieve lead

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

- The semaphore is initialized to the number of resources available
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count)
- When a process releases a resource, it performs a signal() operation (incrementing the count)
- When the count for the semaphore goes to 0, all resources are being used
- After that, processes that wish to use a resource will block until the count becomes greater than 0

↳ will go to waiting

→ No two processes can execute wait() or signal() on the same resource at the same time()

Busy Waiting

Process waiting for cond. to be satisfied in a tight loop without relinquishing the processor

You need to relinquish the processor

Solution to the CS Problem

- Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

↳ confusing, but is not a mutex!!

Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2

- Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

No busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list

} for every process!

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Non busy waiting code

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

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

What is busy waiting?

- ✓ *Busy waiting* means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor
- ✓ Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future
- ✓ Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached

Monitors

Process synchronization: Monitors

(@bstract data type)

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

Pseudocode syntax of a monitor:

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

  initialization code (…) { … }

}
```

BITS Pilani, Hyderabad Cam

Variables

semaphore mutex

mutex = 1

Each procedure P is replaced by

wait(mutex);

...

body of P;

...

signal(mutex);

Mutual exclusion satisfied

condition x, y;

x.wait() -

x.signal()

conditional variables

2 processes P_1 & P_2 execute S_1 & S_2 S_1 first

Create a monitor F_1 & F_2 invoked by P_1 & P_2 resp.

Conditional $X = 0$

boolean done;

F_1:

```
S1;
done = true;
x.signal();
```

F_2:

```
if (done == false)
    x.wait()
S2;
```

For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

The operation **x.wait()** can 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--;
}
```

if x.signal is executed, which process should be resumed?

So FCFS can halt later process

So use conditional wait
x.wait(c);

[Priority based approach]

c – is an int → priority of process
The process with lowest number (highest priority) will be scheduled next

stored for later use to determine highest priority

Single Resource allocation (Slides pls)

Liveness :

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Priority Inheritance Protocol

$P_1 > P_2 > P_3$, assume P_3 holds semaphore

(P_1 high, P_3 low)

Assume P_2 preempts P_3 → P_3 is still holding semaphore

This is priority Inversion ← P_2 prevents P_3 from gaining access to resource C (indirectly atleast)

We need Priority Inheritance Protocol to fix it.

(Not in syllabus currently)

Monitor – Same as lock block , Limited to single process single thread

Mutex – Heavy
Shared across processes
One thread by process

Semaphore multiple threads configurable ,
shared across processes.

→ We need to decide what to use