Synchronization Primitives

Semaphore

Software-based synchronization mechanism/tool (originally proposed by Dijkstra)

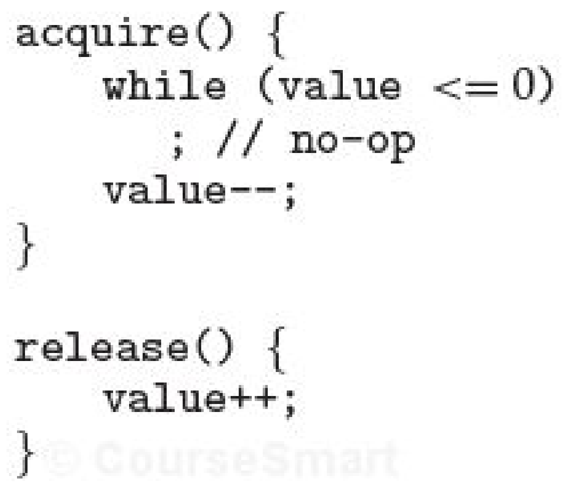
It avoids busy waiting i.e. wasting of CPU time

A semaphore S is an integer variable

Two indivisible operations can modify **S**:

P(S) & V(S) or acquire() & **release() or wait() & signal()**

Indivisible testing & modification of value



```
acquire() {
    while (value <= 0)
       ; // no-op
    value--;
}

release() {
    value++;
}
```

```
WAIT ( S ):
    while   ( S  <=  0 );
    S  =  S  -  1;
```

```
SIGNAL ( S ):
    S  =  S  +  1;
```

FORMAT:

```
        wait ( mutex );                    <-- Mutual exclusion: mutex init to 1.
         CRITICAL SECTION
        signal( mutex );
        REMAINDER SECTION
```

Instead of loop on busy, suspend can be used:

Block on semaphore == False,

Wakeup on signal (semaphore becomes True),

There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,

Wakeup one of the blocked processes upon getting a signal (choice of who depends on strategy).

To PREVENT looping, we redefine the semaphore structure as:

Semaphore *S* – system object

With each semaphore there is an associated waiting queue.

Each entry in the waiting queue has two data items (object properties):

- **value** (of type integer)
- pointer to next record in the queue

P(S) and V(S) operations

```
P(S) : If S ≥ 1 then S :=S-1
    else block the process on the semaphore queue;
V(S) : If some processes are blocked on the semaphore S
        then unblock a process
        else S = S+1;
```

```
typedef struct {
    int                 value;
    struct process   *list;      /*  linked list of  process id waiting on  S  */
} SEMAPHORE
```

```
SEMAPHORE s;
wait(s) {
     s.value  =  s.value  -  1;
     if (  s.value  <  0 ) {
               add this process to s.L;
               block;
     }
}
```

```
SEMAPHORE s;
signal(s)  {
     s.value  =  s.value  +  1;
     if (  s.value  <=  0 )  {
               remove a process P from s.L;
               wakeup(P);
     }
}
```

Counting semaphore (General semaphore)

- An integer value used for signalling among processes and the integer value can range over an unrestricted domain
- Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
- The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.
- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
- Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting

Binary Semaphore

- A semaphore that takes on only the values 0 and 1 on which processes can perform two indivisible operations i.e. changing the integer value from 0 to 1 or 1 to 0

- Often known as mutex lock. A key difference between mutex and binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

Binary Semaphore

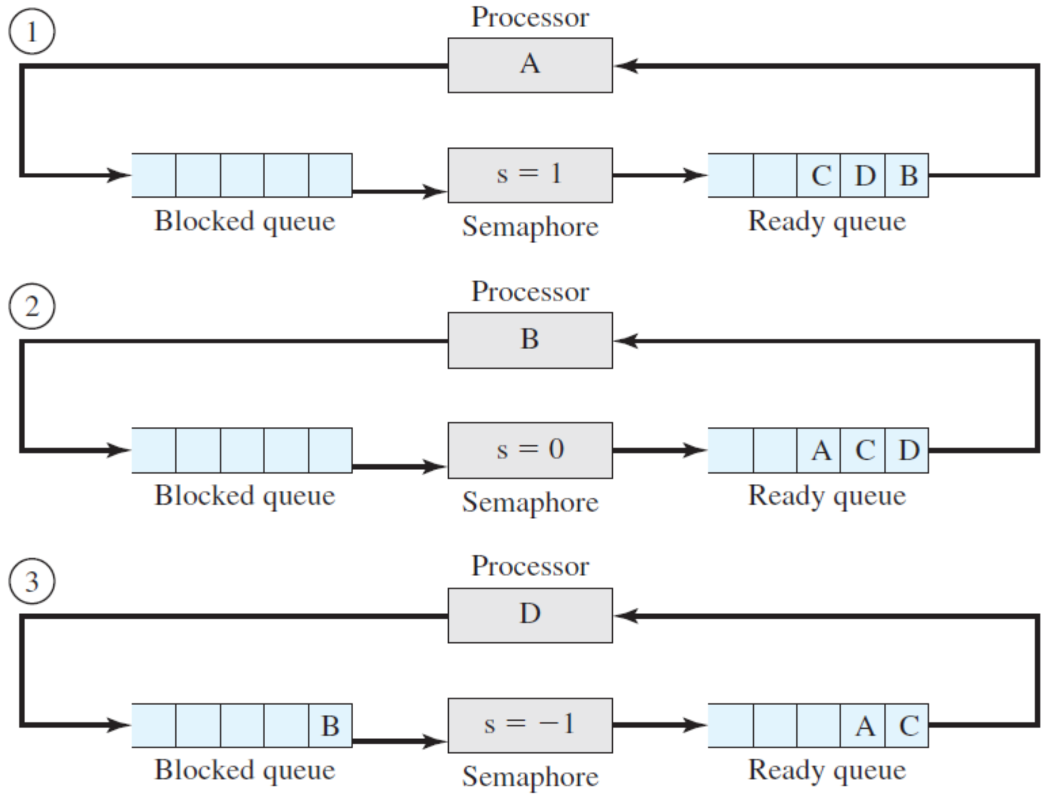
```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
        };
void semWaitB(binary_semaphore s)
        {
        if (s.value == one)
                s.value = zero;
        else {
            /* place this process in s.queue */;
            /* block this process */;
        }
        }
```

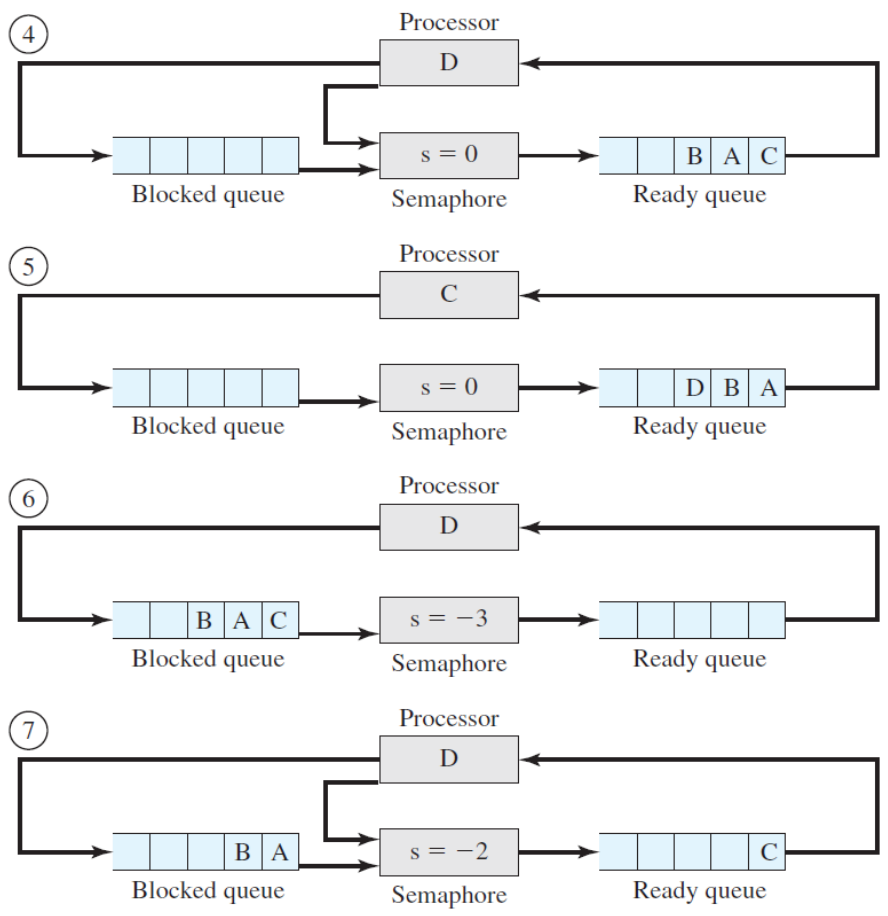
```
void semSignalB(semaphore s)
    {
        if (s.queue is empty())
            s.value = one;
        else {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
          }
    }
```

- For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.
- The fairest removal policy is first-in-first-out (FIFO):The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore**.
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**.

Example

- Processes A, B, and C depend on a result from process D.
- Initially (1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D’s results is available.
- When A issues a semWait instruction on semaphore *s*, the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.
- Then B runs (2), eventually issues a semWait instruction, and is blocked, allowing D to run (3).When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue (4).
- D rejoins the ready queue and C begins to run (5) but is blocked when it issues a semWait instruction.
- Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6).When D has a result, it issues a semSignal, which transfers C to the ready queue.
- Later cycles of D will release A and B from the Blocked state.


1
Processor
A
s = 1
C D B
Blocked queue
Semaphore
Ready queue
2
Processor
B
s = 0
A C D
Blocked queue
Semaphore
Ready queue
3
Processor
D
B
s = −1
A C
Blocked queue
Semaphore
Ready queue


4
Processor
D
s = 0
Semaphore
Blocked queue
B A C
Ready queue
5
Processor
C
s = 0
Semaphore
Blocked queue
D B A
Ready queue
6
Processor
D
B A C
Blocked queue
s = −3
Semaphore
Ready queue
7
Processor
D
B A
Blocked queue
s = −2
Semaphore
C
Ready queue

Semaphores can be used to force synchronization (precedence) if the preceding process does a signal at the end, and the follower does wait at beginning. For example, here we want P1 to execute before P2.

```
P1:                         P2:
   statement 1;                wait ( synch );
   signal ( synch );           statement 2;
```

DEADLOCKS:

May occur when two or more processes try to get the same multiple resources at the same time.

```
P1:                      P2:
    wait(S);                 wait(Q);
    wait(Q);                 wait(S);
    .....                     .....
    signal(S);               signal(Q);
    signal(Q);               signal(S);
```

Bounded-Buffer Problem using Semaphores

Three semaphores

- mutex – for mutually exclusive access to the buffer – initialized to 1
- used – counting semaphore indicating item count in buffer – initialized to 0
- free – number of free items – initialized to BUF_SZ

```
void producer() {
   while (1) {  /* Generate new item into nextProduced */
                   wait(free);
                   wait(mutex);
                   buffer[in] =  nextProduced; in = (in + 1) % BUF_SZ;
                   signal(mutex);
                   signal(used);
   }
}
```

```
void consumer() {
  while (1) { wait(used);
              wait(mutex);
              nextConsumed = buffer[out];  out = (out + 1) % BUF_SZ;
              signal(mutex);
              signal(free);
              /* Process the item from nextConsumed */
  }
}
```