Semaphore

- Synchronization tool that does not require busy waiting if implemented along with an associated queue where process is queue is woken up when resoure is available.
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

Advantage of Semaphore

- It's simple and always have a non-negative starting Integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores. Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain. Example: indicate number of commodities in market.
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

Mutual Exclusion

Semaphore S=1;

```
{  wait (S);
   Critical Section
   signal (S);
}
```

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

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

Producer Consumer naïve - for one producer and one consumer

```
PRODUCER
    send(char c)
    {
       buf[in]=c
            in=(in+1)%N
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            c=buf[out];
            out=(out+1)%N;
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
```

Need to add below constraints
-Consumer consumes only if atleast one element in buffer
-Producer produces if buffer not full

consume when buffer not empty

```
PRODUCER
    send(char c)
    {
       buf[in]=c
            in=(in+1)%N
            signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];
    out=(out+1)%N;
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0;
```

Produce when buffer not full

```
PRODUCER
    send(char c)
    {
       wait(space);
       buf[in]=c
       in=(in+1)%N
       signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];
    out=(out+1)%N;
            signal(space);
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N;
```

consumer produces a space!

Multiple Producers and one consumer

Will two producers find single space left in line 1 and both write into the single space left?

```
PRODUCER
    send(char c)
    {
      wait(space);
      wait(lock);
       buf[in]=c
       in=(in+1)%N
       signal(lock);
       signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];
    out=(out+1)%N;
            signal(space);
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple Producers and consumers

Modify the code to now work for simultaneous working of producer and consumer

```
PRODUCER
    send(char c)
    {
       wait(space);
       wait(lock);
       buf[in]=c
       in=(in+1)%N
       signal(lock)
       signal(chars);
    }
```

```
CONSUMER
    char rcv()
    {     char c;
          wait(chars);
          wait(lock);
          c=buf[out];
          out=(out+1)%N;
          signal(lock);
          signal(space);
          return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1;
```

Counting Semaphores

Can Semaphores be used to allow processes to use multipe copies of an available resource. Eg: 5 printers available to a set of processes.

- Say you have a limited number of resources..eg:5 printers. How would you design a users code using semaphores.

- If all 5 in use then user cant print. If at least one available then user can use it.

- Say you have a limited number of resources..eg:5 printers. How do design a users code using semaphores.

```
Semaphore P=5;

user(..)
{wait(P);
  ......
 Signal(P);
}
```

Semaphore Implementation

The big problem with semaphores as described till now is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***

An alternative approach:

-Block a process if waiting and swap it out of the CPU.

-Each semaphore needs to maintain a list of processes that are blocked waiting for it

-One of the processes is woken up and swapped back when the semaphore becomes available.

Semaphore Implementation with 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

- 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.

Semaphore Implementation with no Busy waiting (Cont.)

Semaphore Structure:

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

- Implementation of wait:

Value=1//available
Value<=0//in use so wait in queue
The negative value indicates the number of processer waiting(exclude the one running)

```
wait (S){
        value--;
        if (value < 0) {
                //add this process to queue
                 block();  }
 }
```

- Implementation of signal:

```
Signal (S){
        value++;
         if (value <= 0) {
                //remove process P from the queue
                 wakeup(P);  }
}
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

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.

http://www.cis.upenn.edu/~lee/03cse380/lectures/ln2-ipc-v2.pdf

Constraints

- Only one writer writes at a time
- If even one reader is reading then, no writing should happen
- Multiple readers can read at the same time

Readers-Writers Problem (Cont.)

Writer Process	Reader Process
	wait(*mutex*); *readcount* := *readcount* + 1; **if** *readcount* = 1 **then** *wait*(*wrt*);
wait(*wrt*);	*signal*(*mutex*);
. . .	. . .
writing is performed	reading is performed
. . .	. . .
signal(*wrt*);	*wait*(*mutex*); *readcount* := *readcount* - 1; **if** *readcount* = 0 **then** *signal*(*wrt*);
	signal(*mutex*);

Second Readers-Writers Problem

- First-readers-writers-problem : Gives preference to readers –no readers will be kept waiting unless writer has got access
- Second-readers-writers-problem: Requires writers to be given priority where writers should not be kept waiting. That is, no new reader can gain access if a writer is waiting

Second readers writers

- Make sure multiple readers can read
  -You hence cant lock the reading part as it restricts another reader reading simultaneously
- Make sure more readers don't join the reading group if a writer expressed interest in writing
  -When writer interested lock readers from entering
- When a writer writes, no one reads/writes.
  -lock while writing
- If both reader and writer is waiting, give preference to writer
  A)writer should not release lock to reader if another writer is waiting.
  B)If first writer is writing, restrict the number of readers who can even show interest so that if a writer enters after first writer relinquishes lock then he does not have to wait for a queue of waiting readers to finish.

Second Readers-Writers Problem

Writer process

```
Repeat
        wait(mutex2);
        writers ← writers + 1;
        if writers = 1 then
                wait(read);
        end-if
        signal(mutex2);
        wait(write);

        // write the resource

        signal(write);
        wait(mutex2);
        writers ← writers - 1;
        if (writers = 0) then
                signal(read);
        end-if
        signal(mutex2);
until done
```

Reader process

```
repeat
        wait(mutex3);
        wait(read);
        wait(mutex1);
        readers ← readers + 1;
        if readers = 1 then
                wait(write);
        end-if
        signal(mutex1);
        signal(read);
        signal(mutex3);

        // read the resource

        wait(mutex1);
        readers ← readers - 1;
        if (readers = 0) then
                signal(write);
        end-if
        signal(mutex1);
until done
```

Mutex 1 and 2 makes sure that reader(writer) increment/decrement is not concurrently attempted by another reader(writer)

Semaphore read: makes sure that reading happens only when writer is not interested in writing. When writer interested then further readers are stopped by wait(read)

Semaphore write: makes sure only one writer writes and no writer writes when someone is still reading

Mutex 3: makes sure that only one reader accummulates when a writer is writing. Other readers even if interested can not join the read queue though they can join mutex3 queue.

Second reader writers problem says that a writer should not be kept waiting. Say a writer W1 is writing and readers R1,R2,R3,R4 arrive. If writer W2 arrives when W1 is still writing, then all readers wait until W2 is done(semaphore read). If W2 arrives after W1 is over, then W2 waits only for single reader R1 to complete as only R1 joins the read queue(mutex3).