Chapter 5 - Mutual exclusion and synchronization

Interprocess Synchronization

A ***critical section*** is a section code in which a process(or thread) competes in a potentially destructive way with another process(thread) for access to a shared data item or file. We discussed this particular example in the previous section:

```
for (i = 0; i < loopcount; i++)
{
      get_lock(id);
      val = shared;
/*    if ((i % 1000) == 0)
         printf("%d - %d - %d  \n", id, i, shared); */
      val += 1;
      shared = val;
      free_lock(id);
}
```

If two processes are allowed to concurrently be in competing critical sections, then incorrect results may be computed.

The process of ensuring that this destructive interaction does no occur is called ***mutual exclusion*** (mutex). Failure to ensure that mutex is provided when required can have literally *fatal* consequences is medical and military systems

Objective

proc 1	proc 2
while(1)	while(1)
{	{
magic bullet	other processing
c.s.	magic bullet
other processing	c.s.
	other processing
}	}

--> The magic bullet
ensures only 1 process in the critical section at any time

Properties Desired in a mutex algorithm:

- **Safe**: both processes can't be in critical section at the same time.
- **Deadlock Free** (Definite infinite postponement)
- **Starvation Free** (Indefinite postponement)
- **Doesn't require strict alternation** (if other process doesn't need access to c.s., then a process should be able to enter immediately)

Possible application level approaches to Mutual Exclusion that do and don't work:

1. Have a single turn variable to control access.

```
int turn = 1;        /* must be in shared memory */
p1:
while (1)
{
     while(turn == 2);    /* must be in shared memory   */
                          /* this is a LOOP        */
     c.s.;
     turn = 1;
     otherstuff ;
}
p2:
while (1)
{
     while(turn == 1);    /* must be in shared memory   */
                          /* this is a LOOP        */
     c.s.;
     turn = 1;
     otherstuff ;
}
```

Problems:

Requires alternating access to c.s. Suppose p1 spends 5 hours in "otherstuff" versus 5 seconds for p2.)

Fails even if a process dies outside of the c.s.

2. Use 2 variables to control access with *test then set*:

```
int p1using, p2using;    /* must be in shared memory */
p1:
while(1)
{
     while( p2using == 1 );   /*  loop...busy-wait */
     /* IF PREEMPTED HERE--BOTH  PROCESSES CAN GET INTO CS */
     p1using = 1;
          c.s.;
     p1using = 0 ;
     otherstuff ;
}

p2:
while(1)
{
     while( p1using == 1 );   /*  loop...busy-wait */
     /* IF PREEMPTED HERE--BOTH  PROCESSES CAN GET INTO CS */
     p2using = 1;
          c.s. ;
     p2using = 0 ;
     otherstuff ;
}
```

This "solution"

Doesn't require alternating access.
Process failure outside the c.s. is not a problem.
But is **UNSAFE...** Both processes can be in c.s. at the same time.

Even though the solution is UNSAFE it may appear to operate correctly for months or even years, depending upon the relative size of *cs* and *otherstuff*.

For failure to occur the following sequence of events is necessary:

p1 preempted after testing p2using but before setting p1using
p2 dispatched and *then preempted in the cs*
p1 redispatched and enters the C.S.

3. Use 2 variables to control access *with set then test*:

```
     int p1using, p2using;         /* must be in shared memory */

     p1:
     while(1)
     {
          p1using = 1;
/* IF PREEMPTED HERE--BOTH  PROCESSES CAN DEADLOCK */
          while( p2using == 1 );   /*  loop...busy-wait */
               c.s.;
          p1using = 0 ;
          otherstuff ;
     }

     p2:
     while(1)
     {
          p2using = 1 ;
/* IF PREEMPTED HERE--BOTH  PROCESSES CAN DEADLOCK */
          while( p1using == 1 );   /* loop...busy-wait */
               c.s. ;
          p2using = 0 ;
          otherstuff ;
     }
```

This "solution"

Doesn't require alternating access.
Solves problem of other process fails.
Is safe
But can cause *deadlock.*

For deadlock to occur:

p1 must be preempted after setting p1using but before testing p2using
p2 must then be dispatched and try to enter the *cs.*

This "solution" is more likely to fail than the unsafe one.

4. Two variables, set then test, with yielding

```
p1:
while(1)
{
RESTART:
        p1using = 1 ;
        if( p2using == 1 )
        {
                p1using = 0 ;
                goto RESTART ;
        }
        c.s. ;
        p1using = 0 ;
        otherstuff ;
}
```

```
p2:
while(1)
{
RESTART2:
        p2using = 1 ;
        if (p1using == 1 )
        {
                p2using = 0 ;
                goto RESTART2 ;
        }
        c.s. ;
        p2using = 0 ;
        otherstuff ;
}
```

Can result in **LiveLock.**

If only one process yields, then the other thread has priority and the solution becomes livelock free.

4. Dekker's Algorithm (Correct) Two variables, set then test with alternating yielding.

```
P1:
while(1)
{
        p1using = 1 ;
        while (p2using == 1)
        {
                if (turn == 2)
                {
                        p1using = 0;
                        while (turn == 2);
                        p1using = 1;
                }
        }
        critical-section
        turn = 2;
        p1using = 0;
        otherstuff ;
}

P2:
while(1)
{
        p2using = 1 ;
        while (p1using == 1)
        {
                if (turn == 1)
                {
                        p2using = 0;
                        while (turn == 1);
                        p2using = 1;
                }
        }
        critical-section
        turn = 1;
        p2using = 0;
        otherstuff ;
}
```

If not my turn then I yield

I unset my variable and wait until my turn

Peterson's Algorithm

```
P1:
while (1)
{
        p1using = 1 ;
        turn = 1;
        while( ( p2using == 1 ) && ( turn == 1 ) ) ;

        c.s;
        p1using = 0 ;
        other-stuff;
}
```

```
P2:
while (1)
{
        p2using = 1;
        turn = 2;
        while( ( p1using == 1 ) &&
                (turn == 2 ) ) ;
        c.s
        p2using = 0;
        other-stuff
}
```

Notes:

If only one process is trying to access the critical section the *turn* variable is irrelevant because the other processes using variable will be 0.

If both processes are *racing* to enter the critical section, both using variables will be 1, but the *turn* variable can only have a single value: either 1 or 2.

Thus the process that sets *turn* last will get stuck in the loop.

The order of the setting of the "using" and the turn variable is critical.

If the order is reversed and a process is preempted after setting the turn variable but before setting the using variable, then both processes can get into the critical section!!

In summary:

Preemption can occur anywhere.
The OS doesn't interrupt a process... the hardware does.
The OS could decide not to preempt a process..
This would require a system call like "OSEnterCriticalSection"
Would also require processes not to abuse this privilege.

Dekker's & Peterson's Algorithms are:

Safe (as long as the SMP hardware enforces strict read after write mem access order)
Deadlock free
Starvation free
Don't require strict alternation

Disadvantages of application level mutual exclusion

However, they have several *disadvantages* :

Each employs **busy-waiting**
One process loops while other process is in critical section

Busy-waits can be *fatal* on a UP with strict priority scheduling.
Low priority process enters C.S.
High priority process gets unblocked and tries to enter C.S.
High priority process will loop for every

Work for only two processes (modulo westall's hack 2)

May not work at all on some MP's with write buffering (modulo westall's hack 1)

Therefore, according to most texts, these are of little use the the real-world.

(Though I actually found a real world use after 15 years of searching)

Westall's hack #1 (overcoming the SMP write buffering problem)

In a multiprocessor system with write buffering, this is not only susceptible do deadlock, it can actually be *unsafe*.

```
p1using = 1;                    p2using = 1;
while (p2using == 1);           while (p1using = 1);
c.s.                            c.s.
p1using = 0;                    p2using = 0;
```

Westall's hack will defeat this problem (assuming you can determine FIFO).

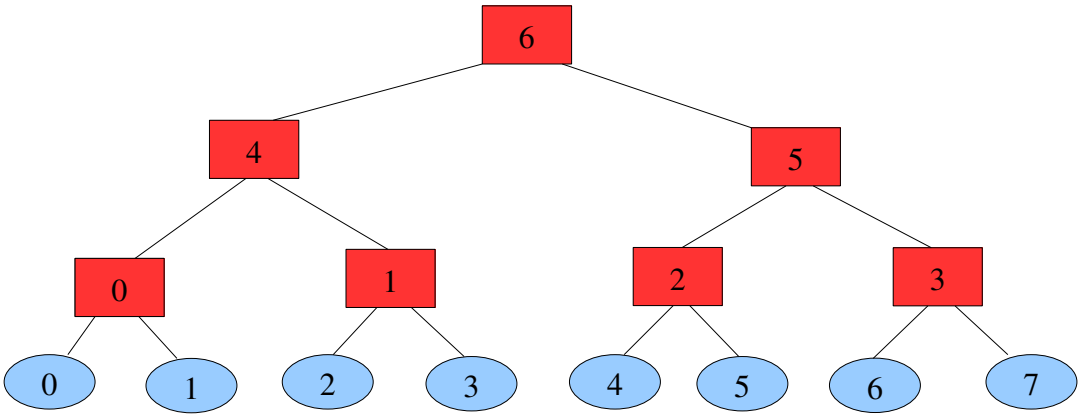
```
p1using = 1                     p2using = 1;
for (i = 0; i < FIFO; i++)      for (i = 0; i < FIFO; i++)
   dummy[i] = i;                        dummy[i] = i;
while (p2using == 1);           while (p1using = 1);
        c.s.                            c.s.
p1using = 0;                    p2using = 0;
```

Westall's hack #2

Can be used to provide mutex to any number of competing threads or processes. We will assume the number is a power of 2. A process wishing to enter the critical section must traverse a binary tree of Peterson type lock obstacles and win the competition at the root of the tree.

When it completes the critical section it must unlock each Peterson type lock it held in the reverse order it locked them.

If there are N competing threads or processes there must be N-1 lock structures arranged in log2(N) tree levels



The lock structures look like:

```
struct pete_lock_type
{
  int using[2];         /* 0 = left 1 = right */
  int  turn;            /*  Ditto             */
} locks[7];
```

Each thread has a path to the root:

```
struct pete_path_type
{
  int lockid;
  int side;       /* = left 1 = right */
}  paths[8][3] =
{{0, 0}, {4, 0}, {6, 0},     // thread 0 path
 {0, 1}, {4, 0}, {6, 0},     // thread 1 path

 {3, 1}, {5, 1}, {6, 1} }  // thread 7 path
```

In general Dekker and Peterson should be *last resorts*

OS based mutex/synchronization mechanisms are preferred for application code.

- They don't use busy waiting
- They don't require the application programmer to understand the subtle ways in which *ad hack* mechanisms may fail.
- They (should) support any number of competing threads / processes

Semaphores

Invented by Djkstra in late '60s
Support any number of processes
Eliminate busy-waiting

The semaphore is an abstract data type that supports two operations,

wait(s)
signal(s)

Operation of wait & signal *system calls*..

```
wait( s )
{
     if ( s.value == 0 )
          suspend( self ) ;
     else
          s.value = s.value -1 ;
}


signal( s )
{
     if (processes are suspended on s)
          unsuspend(exactly one of them) ;
     else
          s.value = s.value + 1 ;
}
```

Semaphore example

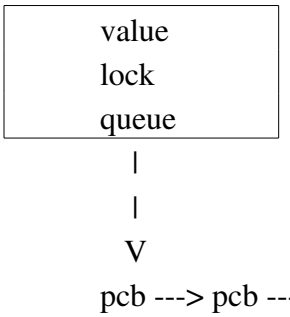
For n processes to synchronize, all they have to do is create a semaphore and then use it.

```
semaphore : sem;          /* value defaults 1 */

process 1:                process 2:                process 3:
while( 1 )                while( 1 )                while( 1 )
{                         {                         {
        wait( sem );              wait( sem );              wait( sem );
        cs;                       cs;                       cs;
        signal( sem );            signal( sem );            signal( sem );
        otherstuff;               otherstuff;               otherstuff;
}                         }                         }
```

Implementation of Semaphores

Semaphore structure:




```
struct semtype
{
        int value;                      /* A non-negative value  */
        unsigned char lock;             /* Used to serialize access to value */
        struct pcbtype *queue ;         /* Queue of blocked(waiting) pcbs */
}
```

A (broken) implementation of *wait()*

```
wait( struct semtype *sem )          /* *sem, a pointer to the semaphore stucture */
{
        if( sem->value  >  0 )

                sem->value  =  sem->value – 1;
        else
                suspend-self( sem->queue);    /* suspend me on the semaphore queue */
}
```

This implementation can fail if:

Two processes attempt to wait on the same semaphore.
The value of the semaphore is 1 and
process 1 is preempted after test of sem->value but before it is decremented.

==> Testing and resetting of the semaphore value is itself a critical section

Possible solutions:

Disable interrupts before entering a critical section
Reenable interrupts after exiting the critical section.

This guarantees no preemption withing the critical section.
This solution is correct for a single CPU system.

Setting a global “don't preempt” flag will also work.

Using a global “don't preempt” policy in kernel mode as is the case with traditional Unix implementations will also work.

ALL these “solutions” will fail on a multiprocessor system

because two processors could be running the semaphore code in lock step.

The Test and Set Mechanism

When Computer Science "hits the wall" its common to call upon Computer Engineering to provide a hardware mechanism that provides the solution.

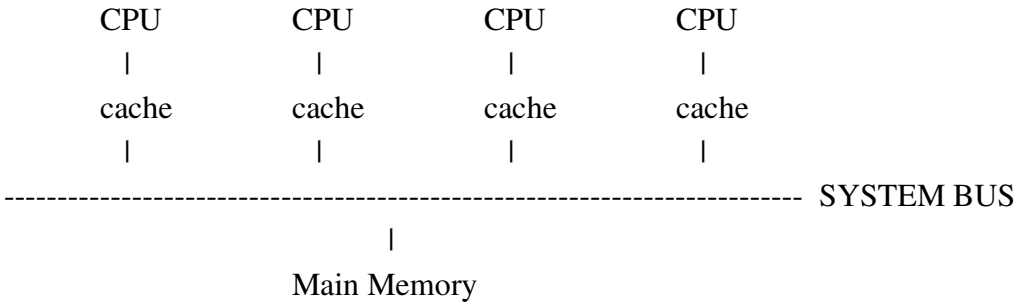
The *test and set* instruction provides a hardware locking mechanism that can be use to safely implement semaphores in a SMP system.

```
void  setlock(
int *lock-var)
{
        CLI                 /* Disable interrupts.                          */
RETRY:
        CMPI  lock-var, 0   /* read the lock-var and see how it looks.      */
        JNZ   retry         /* if not zero retry it.                        */
        TS    lock-var      /* it will unconditionally set lock-var to 1.  it will set a condition
                               flag to indicate the value ( 0 or 1 ) before TS is executed */
        JNZ   retry         /* jump not zero; 1 => locked, 0 => available */
        RET
}


void  rlselock(
int *lock-var)
{
        MOVI  lock-var, 0   /* Unlock the lock */
        STI                 /* Enable interrupts */
        RET
}
```

TS hardware support serializes access to shared memory by all processors.

CPU memory in a multiprocessor system



Considerations in the use of TS:

Interrupts *must be disabled* while holding a TS lock because

on a single- CPU system with priority dispatching

a low-priority process is preempted holding the lock,
a high-priority process tries to obtain the lock...
then the two processes are **deadlocked.**

on a 2 CPU process, the deadlock is not guaranteed but can occur if.
number of processes contending for a lock = number of processors

When releasing the lock, the enabling of interrupts MUST FOLLOW, NOT PRECEDE, the release of the lock.

On a single processor system, the semaphore implementation...

Is safe without the lock.. causes no harm if the locking code is included.

The cost is a few extra instructions.
On a single CPU, the TS will always succeed on the first try
(Or else you have what is commonly called a system crash)

The complete implementation of wait:

```
wait( struct semtype *sem )             /* *sem, a pointer to the semaphore stucture */
{
        setlock(&sem->lock);
        if( sem->value > 0 )
        {
                sem->value = ( sem - 1 ) ;
                rlselock(&sem->lock);
        }
        else
                suspend-self( sem->queue);   /* suspend me on the semaphore queue */
                                             /* must release lock after PCB on sem queue */
}
```

As an exercise, implement signal().

Comparison of Semaphores and Locks:

Locks
- Use busy waiting (loop while waiting)
- Held only inside OS code
- Holder must not be preempted
- Holding time must be short.

Semaphores
- Use blocked waiting (suspended while waiting)
- Application and (some OS) code may use them
- Holder may be preempted
- Holding time can be arbitrarily long.

Binary vs. Counting Semaphores

- Binary sems only take on values of 0 or 1
- Signaling a binary semaphore with value 1 is typically a sign of a logic error.
- Counting semaphores can have any positive value.
- Binary semaphores are totally sufficient for mutex.
- Counting semaphores provide extra capability useful in other synchronization problems.

Simulating Semaphores in Unix

Original Unix had no semaphore support.
Recent Unix versions support both shared memory and semaphores via a painful API

Its easy to emulate a counting semaphore using a *pipe*.

Creating a semaphore corresponds to:

int pipedef[2];
pipe(pipedef); /* Create a single pipe and get back read and write handles */

Signaling a semaphore corresponds to:

write(pipedef[1], "T", 1);

pipedef[1] is the write handle
The "T" represents a 1 byte “token”
The 1 says write 1 byte into the pipe.

Waiting on the semaphore corresponds to:

read(pipedef[0], buff, 1);

pipedef[0] is the read handle
buff[0] will recieve the "T"

Classical Cooperating Process Problems

The Producer / Consumer Problem

a.k.a (Bounded buffer || Circular buffer)
Problem involves both
mutex and
general synchronization

The buffer is a circular array of “slots”

Two indices/pointers are used to manage buffer access

in_slot	identifies the next slot in the buffer in which a new item will be placed
out_slot	identifies the next slot in the buffer from which an item will be taken

When implemented in hardware such buffers are commonly called FIFOs

Synchronization problems

When the buffer is empty consumers *must be blocked until an item is produced*

When the buffer is full producers mu*st be blocked until space is available in the buffer*.

Counting semaphores that count both free slots and available items can accomplish both of these missions.

Mutex problems

If there are multiple competing producers, two or more of them *must not be allowed to produce into the same slot.*

If there are multiple competing consumers, two or more of them *must not be allowed to consume the same item.*

If there is only a single producer and a single consumer, the mutex problems don't exist but the syncrhonization problem remains critical.

The producer solution

```
#define SLOTS 32   // or whatever

sem  cmutex;
sem  pmutex;
sem  slot_free;
sem  item_avail;
int  in_slot;
int  out_slot;

producer()   /* This thread contains the producer code */
{

/* Initialize slot_free semaphore */

   for( i = 0 ; i < SLOTS ; i++ )
     signal(slot_free) ;

   in_slot = 0 ;
   while( 1 )
   {
     wait(slot_free) ;
     wait(pmutex ) ;

     buffer[in_slot] = produce_item();
     in_slot = (in_slot + 1)  %  SLOTS ;

     signal(pmutex ) ;
     signal(item_avail) ;
   }
}
```

The consumer solution

```
consumer()   /* This thread contains the consumer code */
{
   out_slot = 0 ;
   while( 1 )
   {
     wait( item_avail) ;
     wait(cmutex ) ;

     consumed_item =  buffer[out_slot] ;
     out_slot = (out_slot + 1) % SLOTS ;

     signal(cmutex ) ;
     signal( slot_free ) ;
   }
}
```

Mutex Considerations

Single producer and single consumer:

Mutexes not needed for a correct solution.

Multiple producers or consumers:

Separate consumer mutex and producer mutex as shown allows most parallelism

Use of a single mutex can work but more care must be taken.
Order of waits (but not signals) becomes critical.

If a single mutex used by both the producer and the consumer:
If `wait(mutex)` occurs before
`wait(slot_free)` or `wait(item_avail)`:

If the consumer gets the waits in the wrong order, the system can deadlock when the the buffer is empty. If the consumer starts before the producer starts.
The consumer will block on `item_avail`while *holding* `mutex.`

The producer then starts up and *blocks permanently on* `mutex.`

If the producer gets the waits in the wrong order deadlock can occur only when the buffer is full. In this case the producer will block on `slot_free` while *holding* `mutex.` The consumer will then *block permanently on* `mutex` when it tries to consume an item.

With pmutex for producers and cmutex for consumers, the order of waits is irrelevant. The order of signals *is always irrelevant because signal never blocks*.

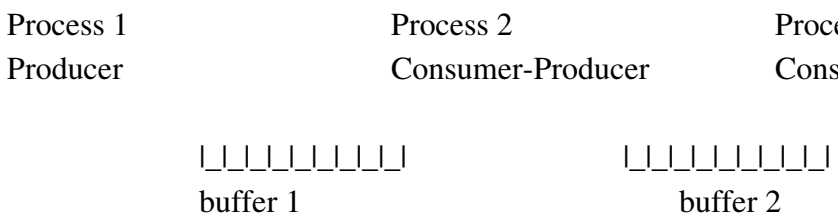
Shared mutex is undesirable because it prevents overlapped production and consumption.

Can each producer and each consumer have it's own semaphore?

NO, that would break the code completely.
By definitition for a mutex semaphore to work at all, *all processes requiring mutual exclusion must use the same mutex semaphore!*

Can a process be both a producer and a consumer?

Yes...
Child2 in mp1 was one



Process 2 will have the following organization:

```
while( 1 )
{
        usual consumer code for buffer 1 ;
        possibly perform some manner of operation on consumed data
        usual producer code for buffer 2 ;
        (producer code may or may not be called based on result of operation )
}
```

Can producer/consumer problems be solved without using semaphores?

Certainly! Your e-mail filter program is a classic example!

The Readers and Writers problem

Framework of the Problem:

A resource exists that can be read or can be written to.
Multiple processes that can safely read concurrently
Only one process can write at a time and only if nobody is reading.

There are 3 possible solution objectives:

Reader priority - Arriving readers may pass waiting writers to join existing readers. This may lead to *writer starvation.*

Writer priority – Arriving writers by pass waiting readers in the queue. This maximizes concurrent reading but may lead to *reader starvation.*

Strict FIFO – No passing by either readers or writers. This minimizes the level of concurrent reading, but *eliminates starvation... but with a potential nasty side effects if readers read for a VERY LONG TIME.*

There are semaphore based solutions to these but they are *very* tricky.

A safe (but defective) solution

```
writer()                    reader()
{                           {
        |                           |
        wait( wsem ) ;              wait(wsem);
        write() ;                   read();
        signal( wsem ) ;            signal(wsem);
        |                           |
}                           }
```

This “solution” doesn't allow concurrent reading.

The Reader Priority Solution

readers bypass waiting writers to join existing readers.

```
writer()    // same as in the defective solution
{
        |
        wait( wsem ) ;
        write() ;
        signal( wsem ) ;
        |
}
```

```
reader()
{
        |
        wait( rmutex ) ;
        rcount += 1 ;           /* count of readers in system, increment */
        if( rcount == 1 )       /* no pre-existing readers, this is the first */
                wait( wsem ) ;
        signal( rmutex ) ;

        read() ;

        wait( rmutex ) ;
        rcount -= 1 ;           /* decrement count of readers */
        if( rcount == 0 )
                signal( wsem ) ;
        signal( rmutex ) ;
}
```

If writing is taking place when readers arrive:

First reader gets blocked on wsem...
Subsequent readers get blocked on rmutex
Subsequent writers get blocked on wsem

Whenever one reader is allowed to read, *all existing readers will join it.*

	last arriving ---------- first arriving									
Example:	Wtr	Wtr	Rdr	Rdr	Rdr	Wtr	Wtr	Wtr	Wtr	Wtr
blocked on:	wmtx	wmtx	rmtx	rmtx	wsem	wsem	wsem	wsem	wsem	writing

Never more than one reader blocked on wsem.

Starvation issues:

Writers can get blocked out forever (starvation) if there are enough readers.
Reader starvation is NEVER possible.

The Strict FIFO solution

With *Strict FIFO*, a reader can't bypass waiting writers to join existing readers.
Semaphore-based implementation exists and but is even more complex!

Rdr Rdr Wtr Rdr Rdr Rdr Wtr Wtr || Rdr/Rdr/Rdr

An easier approach is to just invent a new O/S system call designed specificially for this purpose (This mechanism is sometimes called a *monitor).*

This implementation is used to serialize readers and writers is the enqueue and dequeue mechanism provided by MVS--

Enqueue and dequeue are

higher-level primitives than wait and signal
They are implemented in the MVS kernel.

To gain access to the resource an application makes the system call:

enqueue(Resource ID, "r") or enqueue(Resource ID, "w")

When complete the process makes the system call:

dequeue(Resource ID)

A possible implementation:

The operating system is responsible for managing a FIFO queue of waiting PCB's associated with each resource.

```
16 struct qeltype
17 {
18    struct pcbtype *pcb;     /* Pointer to PCB        */
19    int            action;   /* Read or write         */
20    struct qeltype *qnext;   /* Next element in list  */
21 };
28 struct qcbtype
29 {
30    int state;               /* 0 Free, 1 Read, 2 write   */
31    int numusing;            /* Active of readers/writers */
32    int numwaiting;          /* Waiting readers/writers   */
33    int lock;                /* TS lock for serialization */
34    struct qeltype *qhead;   /* Head of wait queue        */
35    struct qeltype *qtail;   /* Tail of wait queue.       */
36 };
```

```
void enqueue(struct qcbtype *qcb, int action)
{
     get_lock(&qcb->lock);
     if (qcb->state == FREE)
     {
          qcb->numusing += 1;
          qcb->state = action;
          goto out;
     }

     if ((action == READ) && (qcb->state == READ) &&
          (qcb->numwaiting == 0))
     {
             qcb->numusing += 1;
             goto out;
     }

     qel = get_qel();
     qel->pcb = current;
     qel->action = action;
     qel_queue_tail(qcb, qel);
     rlse_lock(&qcb->lock);
     schedule();
     return;

out:
     rlse_lock(&qcb->lock);
}
```

Condition Variables

A synchronization primitive that is more general than the semaphore which can be used to solve the Readers and Writers Problem (and other problems).

Declaring:

```
pthread_cond_t      cv;
pthread_mutex_t     mtx;
```

Initializing:

```
pthread_cond_init(&cv, NULL);
pthread_mutex_init(&mtx, NULL);
```

Usage: One thread is waiting for some condition to become true (for example, there is an item in a buffer)

```
pthread_mutex_lock(&mtx);
while (some-condition-is-not-true)
     pthread_cond_wait(&cv, &mtx);
pthread_mutex_unlock(&mtx);
```

Another thread causes something to happen (places an item in the buffer)

```
pthread_mutex_lock(&mtx);
make-some-condition-become-true
pthread_cond_broadcast(&cv);
pthread_mutex_unlock(&mtx);
```

Operation:

```
pthread_cond_wait(&cv, &mtx);
```

Release the mutex and block thread
On wakeup reacquire the mutex

```
pthread_cond_broadcast(&cv);
```

Wake up *ALL THREADS* waiting on the mutex

Mutex requirements

Its critical that the waiting thread lock the mutex before testing the condition and potentially going to sleep. It is also critical that the causing thread lock the mutex ensuring that making-the-condition-true and broadcasting the condition being atomic.

```
pthread_mutex_lock(&mtx);
while (some-condition-is-not-true)
     pthread_cond_wait(&cv, &mtx);
pthread_mutex_unlock(&mtx);
```

Another thread causes something to happen:

```
pthread_mutex_lock(&mtx);
make-some-condition-become-true
pthread_cond_broadcast(&cv);
pthread_mutex_unlock(&mtx);
```

If not the following events can happen.

thread 0 – test the condition and find it false
thread 1 – make the condition become true
thread 1 – broadcast the condition variable (no threads are blocked so nothing happens)
thread 0 – block on the condition variable (possibly forever!!!)

Unlike the counting semaphore the CV does not have a "value" that allows it to "accumulate broadcast credits"

Semaphores can be readily constructed with condition variables

```
struct semtype
{
     int value;
     pthread_cond_t cv;
     pthread_mutex_t mtx;
};

void wait(struct semtype *s)
{
     pthread_mutex_lock(&s->mtx);

     while (s->value == 0)
     {
          pthread_cond_wait(&s->cv, &s->mtx);
     }
     s->value -= 1;
     pthread_mutex_unlock(&s->mtx);
}


void signal(struct semtype *s)
{
     pthread_mutex_lock(&s->mtx);

     s->value += 1;
     pthread_cond_broadcast(&s->cv);

     pthread_mutex_unlock(&s->mtx);
}
```

Exercise: Show how in the absence of the mutex in the signal code the semaphore would not work reliably. Would the failure be hard(every time signal called) or transient(much of the time it might appear to work correctly)?

The producer solution

```
#define SLOTS 32   // or whatever


int  slots_free;
int  items_avail;
int  in_slot;
int  out_slot;

pthread_cond_t  scv;
pthread_mutex_t smtx;

pthread_cond_t  icv;
pthread_mutex_t imtx;

producer()   /* This thread contains the producer code */
{

/* Initialize slots_free counter */

   slots_free = SLOTS;

   in_slot = 0 ;
   while( 1 )
   {

  /* “wait” until a slot is available */

     pthread_mutex_lock(smtx);

     while (slots_free == 0)
     {
          pthread_cond_wait(scv, smtx);
     }
     pthread_mutex_unlock(smtx);

  /* produce an item into it */

     buffer[in_slot] = produce_item();
     in_slot = (in_slot + 1)  %  SLOTS ;


  /* “signal” that an item is available */

     pthread_mutex_lock(imtx);
     items_avail += 1;
     pthread_cond_broadcast(icv);
     pthread_mutex_unlock(imtx);
   }
}
```

The consumer solution

```
consumer()   /* This thread contains the consumer code */
{
   out_slot = 0 ;
   while( 1 )
   {

  /* wait until an item is available */

     pthread_mutex_lock(imtx);

     while (items_avail == 0)
     {
          pthread_cond_wait(icv, imtx);
     }
     pthread_mutex_unlock(imtx);


  /* Consume the item */

     consumed_item =  buffer[out_slot] ;
     out_slot = (out_slot + 1) % SLOTS ;


   /* "Signal" a slot is now available */

     pthread_mutex_lock(smtx);
     slots_avail += 1;
     pthread_cond_broadcast(scv);
     pthread_mutex_unlock(smtx);

   }
}
```

Exercise: Will this solution work if: *scv = icv = cv and smtx = imtx = mtx?*

Exercise: How should the solution be changed if *two* producers are active??