Solution 4a:

Idea: Use two flags, both FALSE initially; flag0 is true when Process 0 is in the critical section, flag 1 is true when Process 1 is in the critical section.

```
while ( TRUE )                     while ( TRUE )
   {                                  {
   while ( flag1 );                   while ( flag0 );
   flag0 = TRUE;                      flag1 = TRUE;
   critical_section();                critical_section();
   flag0 = FALSE;                     flag1 = FALSE;
   }                                  }
```

Process 0 Process 1

NO, mutual exclusion fails ! (violates property 1) e.g.

Process 0 does while test, finds flag1 is FALSE.
Process 1 does while test, finds flag0 is FALSE.
Process 0 sets flag0 to TRUE and enters critical section (CS)
Process 1 sets flag1 to TRUE and enters critical section (ouch !)

Solution 4b:

Idea: Use two flags for ‘want to enter’ (similar to previous example).

```
while ( TRUE )                     while ( TRUE )
   {                                  {
   flag0 = TRUE;                      flag1 = TRUE;
   while ( flag1 );                   while ( flag0 );
   critical_section();                critical_section();
   flag0 = FALSE;                     flag1 = FALSE;
   }                                  }
```

Process 0 Process 1

NO, progress fails ! (violates property 3) e.g.

Process 0 sets flag0 to TRUE
Process 1 sets flag1 to TRUE
Both loop forever !

Solution 4c: Peterson’s Solution

A solution that satisfies all properties for two processes.

```
P0:                                  P1:
while (TRUE)                         while (TRUE)
   {                                    {
   flag0 = TRUE;                        flag1 = TRUE;
   turn = 1;                            turn = 0;
   while(flag1 && turn == 1);           while(flag0 && turn == 0);
   critical_section();                  critical_section();
   flag0 = FALSE;                       flag1 = FALSE;
   }                                    }
```

Uses both **turn** and **flags** variables with both **flags** initially FALSE.

Partial analysis

- Suppose P0 wants critical section and P1 is not in the critical section and does not want it. Then P0 enters critical section, as flag1 is FALSE. (part of progress proof).
- Suppose P0 wants critical section and P1 is in the critical section. Then P0 waits for flag1 to go FALSE (mutual exclusion).
- Suppose both P0 and P1 want critical section. Then the *first* one to set turn is the one to get in (mutual exclusion and progress).

```
P0:                              P1:                     time
flag0 = TRUE;
                                 flag1 = TRUE;
turn = 1;
                                 turn = 0;
while(flag1&&turn==1);
                                 while(flag0&&turn==0);
/* Enter critical                /* Loop and wait for
   section - set                    flag0 to go FALSE. */
   flag0 to FALSE
   when done. */
```

Check bounded waiting:

- Suppose both P0 and P1 want to enter the critical section.
- P0 sets turn = 1, P1 sets turn = 0 and P0 enters critical section.
- P1 is waiting in the while loop for either flag0 to go FALSE or for turn to be 1.
- Suppose P0 exits critical section.
- P0 sets flag0 to FALSE.
- Suppose P1 is unlucky and P0 sets flag0 to TRUE before P1 runs again.
- P0 sets turn to 1.
- P1 enters critical section on 'turn = 1' test.

Can Peterson's solution provide mutual exclusion if the flag0/1 statements are reversed with the turn0/1 statements ?

Test and Set Lock Instruction (TSL)

TSL instruction commonly found on multiprocessors.

- TSL reads a value of a memory word into a register and then stores a non-zero value at the memory address.
- Reading and then writing are indivisible operations, e.g. in a multiprocessor system, the memory bus is locked between the read and write.

Example use:

Assume a memory variable **lock** is used to control access to shared memory.

When set to 0, any process can read and write shared memory by setting **lock** to 1 using **TSL** instruction.

When process is finished, it sets **lock** back to 0 using the **move** instruction.


```
enter_region:
  tsl register, lock
  cmp register, #0
  jne enter_region
  ret

leave_region:
  move lock, #0
  ret
```

Example usage from C program

```
enter_region();
critical_section();
leave_region();
```

Suppose **lock** is FALSE

- Then TSL sets lock to TRUE and we enter critical section.
- On leaving critical section, lock is set back to FALSE.

Suppose **lock** is TRUE for process P1

- Then some other process is in the critical section, process P2
- While loop runs until P2 process sets lock to FALSE
- Then P1 sets lock back to TRUE and enters critical section

Suppose both P1 and P2 call enter_region at the same time, with **lock** FALSE

- Then the first process to execute TSL gets in

Suppose both P1 and P2 call enter_region at the same time, with a third process, P3, in the critical section

- First process to execute TSL after P3 sets lock to FALSE gets in.

Problem: Bounded waiting may be violated (property 4). How ?

Assume we have a C library call **TSL(lock)** that sets value of **lock** and then returns old value.

(Initially lock is FALSE)

```
P0: while TSL(lock);        P0 enters CS
P1: while TSL(lock);        P1 waits
P0: critical_section();     P0 does stuff in CS
P1: while TSL(lock);        P1 waits
P0: lock = FALSE;

*Now suppose Schedular does not run P1 for a bit*

P0: while TSL(lock);        P0 enters CS AGAIN !
P1: while TSL(lock);        P1 is still waiting.

*P1 can be bypassed any number of times*
```

2.2.4 Sleep(Wait) and Wakeup(Signal)

Both TSL and Pererson's solution require busy waiting.

Problem in OS systems where processes have different priorities:

- Process A has lower priority than process B.
- Process A obtains lock and is preempted by process B.
- Process B tries to enter critical section and waits forever (violates property 3, progress).

Called **priority inversion** problem.

Solution: IPC primitives that **block** instead of busy waiting.

- **Sleep**: A system call that causes caller to block until another process wakes it up.
- **Wakeup**: A system call with a parameter -> the process to wake up.

Producer-Consumer Problem (bounded buffer problem)

An application of Sleep and Wakeup.

Assume one produce and one consumer.

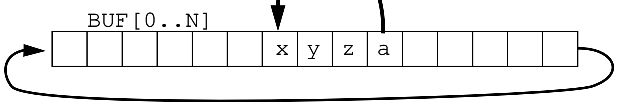
Assume count, in and out are all 0.

Producer

```
while (1)
   {
   item = read(file);
   if ( count == N )
      Sleep();
   BUF[in] = item;
   in = (in+1)%N;
   count++;
   if ( count == 1 )
      Wakeup(consumer);
```

Consumer

```
while (1)
   {
   if ( count == 0 )
      Sleep();
   item = BUF[out];
   out = (out+1)%N;
   count--;
   if ( count == N-1 )
      Wakeup(producer);
   write(printer, item);
```



Consider the following possible sequence of events (or trace).

- Suppose BUF is empty and count is 0.
- Consumer sees count == 0.
- There is a context switch.
- Producer adds item to BUF, increments count to 1, and tries to wake the Consumer (executes Wakeup(consumer)).
- Consumer executes Sleep() AFTER it received the wakeup message from Producer - wakeup message is LOST !
- When Producer fills the buffer, it too will sleep. Since the Consumer will never consume another item, both processes sleep forever.

Problem: Wakeup message was sent to a process that was not yet sleeping !

Question: Can we treat the test of count and the Sleep() together, as a critical section ?

One solution is to keep a queue of wake-up calls, or simply a count of pending wake-ups (e.g. **Semaphores**).

2.2.5 Semaphores

```
typedef struct {
   int val;
   process_list L;
   } sema;
```

```
Wait(sema *S)
  {
  S->val = S->val - 1;
  if ( S->val < 0 )
     {
     Enqueue(CURRENT, S->L);
     ChangeState(CURRENT,
        waiting);
     }
  }
```

```
Signal(sema *S)
  {
  S->val = S->val + 1;
  if ( S->val <= 0 )
     {
     resume_proc =
        Dequeue(S->L);
     ChangeState(
        resume_proc,ready);
     }
  }
```

ChangeState() in Wait() MUST move process to Wait queue and call Schedular - process can NOT continue, otherwise it enters critical section !

ChangeState() in Signal() may or may not call Schedular - moves a process from Wait queue to Ready queue.

If the list L is a FIFO queue, we have bounded waiting.

Semaphores MUST execute atomically.

- On a single processor system, we can disable interrupts.
- On a multi-processor system, we need a **lock** shared variable updated using the TSL instruction, that ensures calls to Wait and Signal are done atomically.

Given mutual exclusion, we can implement semaphores, and given semaphores we can implement mutual exclusion.

If S->val is negative, then abs(S->val) is the number of waiting processes.

Example traces:

```
P0, P1: Wait(&SEMA);
        CS();
        Signal(&SEMA);
```

One Process

```
Initially              S->val = 1
P0: Wait(&SEMA)        S->val = 0
P0: CS()               S->val = 0
P0: Signal(&SEMA)      S->val = 1
```

Two Processes

```
Initially              S->val = 1
P0: Wait(&SEMA)        S->val = 0
P0: CS()               S->val = 0
P1: Wait(&SEMA)        S->val = -1    P1 waits
P0: Signal(&SEMA)      S->val = 0     P1 is awakened
P1: CS()               S->val = 0
P1: Signal(&SEMA)      S->val = 1
```


Two Processes with reentry of P0:

```
Initially              S->val = 1
P0: Wait(S&EMA)        S->val = 0
P0: CS()               S->val = 0
P1: Wait(&SEMA)        S->val = -1    P1 waits
P0: Signal(&SEMA)      S->val = 0     P1 is awakened
P0: Wait(&SEMA)        S->val = -1    P0 waits
P1: CS()               S->val = -1
P1: Signal(&SEMA)      S->val = 0     P0 is awakened
P0: CS()               S->val = 0
P0: Signal(&SEMA)      S->val = 1
```

Three Processes

```
Initially              S->val = 1
P0: Wait(&SEMA)        S->val = 0
P0: CS()               S->val = 0
P1: Wait(&SEMA)        S->val = -1    P1 waits
P2: Wait(&SEMA)        S->val = -2    P2 waits
P0: Signal(&SEMA)      S->val = -1    P1 is awakened
P1: CS()               S->val = -1
P1: Signal(&SEMA)      S->val = 0     P2 is awakened
P2: CS()               S->val = 0
P2: Signal(&SEMA)      S->val = 1
```


Semaphore applications -> Synchronization

Suppose S1 in P1 must always happen before S2 in P2 (e.g. S1 is a write operation and S2 is a read operation).

Initially, synch is 0

```
      P0:                 P1:
-------------------------------------
       .                   .
       .                   .
       .                   .
      S1;            Wait(&synch);
Signal(&synch);           S2;
       .                   .
       .                   .
       .                   .
```

Deadlock

Two or more processes waiting on each other.

Example:

Initially, s = q = 0

```
P0:                      P1:
 :                        :
Wait(&s);                Wait(&q);
 :                        :
Signal(&q);              Signal(&s);
 :                        :
```

P0 waits for P1 and P1 waits for P0.

This can easily happen if Waits and Signals get out of order.

We will see more examples of this in the section on deadlocks.

Solving Producer-Consumer (Bounded Buffer) problem using Semaphores

Solution #1:

The first solution uses three semaphores: full, empty and mutex.

Initially, full=N, empty=0, mutex=1

Producer

```
while (1)
   {
   produce V;
/* full = full-1;
   if full < 0, wait. */
   Wait(&full);
   Wait(&mutex);
   add V to buffer;
   Signal(&mutex);
/* empty = empty+1;
   if empty <=0, then
   wake process waiting
   on empty. */
   Signal(&empty);
   }
```

Consumer

```
while (1)
   {
/* empty = empty-1;
   if empty < 0, wait. */
   Wait(&empty);
   Wait(&mutex);
   remove V from buffer;
   Signal(&mutex);
/* full = full+1;
   if full <=0, then
   wake process waiting
   on full. */
   Signal(&full);
   use V;
   }
```


full and **empty** count the number of empty and full buffers, respectively.

Semaphores with initial values other than 1 are known as **counting semaphores**.

Note: mutex is used for mutual exclusion while full and empty are used for synchronization.

When the buffer is empty, full = N, empty = 0 and the consumer waits on empty.

When the buffer is full, full = 0, empty = N and producer waits on full.

	full	empty	
Initially	N	0	
producer:Wait(&empty)	N-1	0	
producer:Signal(&full)	N-1	1	
consumer:Wait(&full)	N-1	0	
consumer:Signal(&empty)	N	0	
Initially	N	0	
consumer:Wait(&full)	N	-1	Consumer waits
producer:Wait(&empty)	N-1	-1	
producer:Signal(&full)	N-1	0	Wakes consumer
consumer:Signal(&empty)	N	0	

Solution #2:

In practice, we might do something simpler, with a single semaphore mutex, initially 1, for mutual exclusion.

Initially, mutex=1

Producer

```
while (1)
   {
   produce V;
/* Wait crit section. */
   Wait(&mutex);
/* Wait on buf. */
   while(full(buf))
      {
      Signal(&mutex);;
      sleep(1);
      Wait(&mutex);
      }
   Add V to buf;
   Signal(&mutex);;
   }
```

Consumer

```
while (1)
   {
/* Wait crit section. */
   Wait(&mutex);
/* Wait on buf. */
   while(empty(buf))
      {
      Signal(&mutex);;
      sleep(1);
      Wait(&mutex);
      }
   Remove V from buf;
   Signal(&mutex);
   use V;
   }
```