Operating Systems

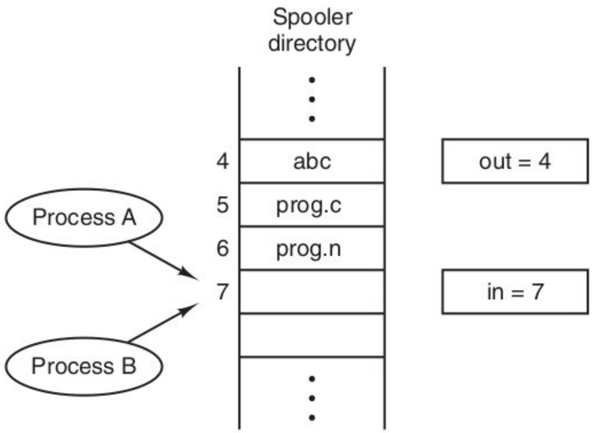
Synchronization

Carlos Alberto Llano R.

10 de septiembre de 2015

Contents

Background - Producer and Consumer example 1



Background - Producer and Consumer example 2

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Background - Producer and Consumer example

- cont = cont + 1

```
mov  [cont],AX
add  AX,1
mov  AX,[cont]
```

- cont = cont – 1

```
mov  [cont],BX
add  BX,-1
mov  BX,[cont]
```

Background - Producer and Consumer example

Time	Process	Instruction	Register
T0	Producer	AX = cont	AX = 9
T1	Producer	AX = AX + 1	AX = 10
T2	Consumer	BX = cont	BX = 9
T3	Consumer	BX = BX - 1	BX = 8
T4	Producer	cont = AX	cont = 10
T5	Consumer	cont = BX	cont = 8

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

Race condition

When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Race condition

Examples in C and C++

Critical Section or Critical Regions

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time.

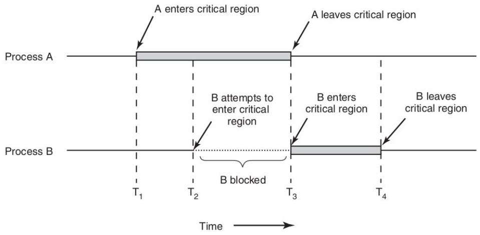
We need a mutual exclusion, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

That part of the program where the shared memory is accessed is called the critical region or critical section.

Critical Section or Critical Regions

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.



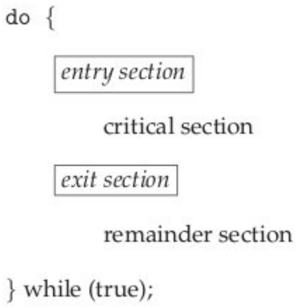
Critical Section Problem, in other words...

A solution to the critical section problem must satisfy the following three requirements:

- Mutual exclusion
- Progress
- Bounded waiting

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

Critical Section Solution - Struture



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Proposals for achieving mutual exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson's Solution
- The TSL Instruction
- Sleep and Wakeup
- Mutex Lock
- Semaphores

Disabling Interrupts

With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.

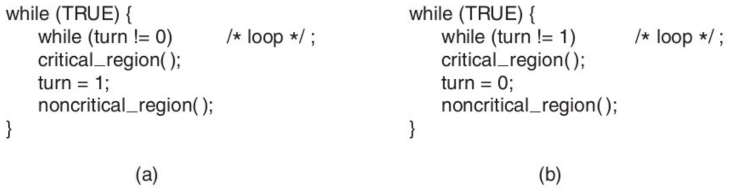
Lock Variables

When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.

Strict Alternation



```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

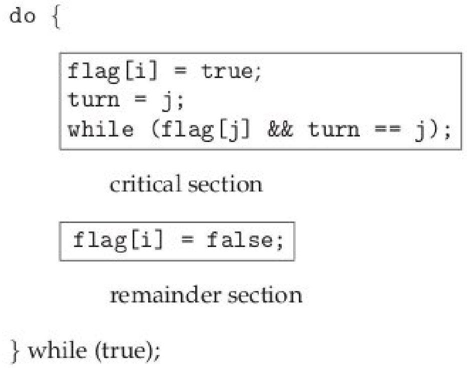
(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a spin lock.

Peterson's Solution



```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = false;

            remainder section

} while (true);
```

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The TSL Instruction

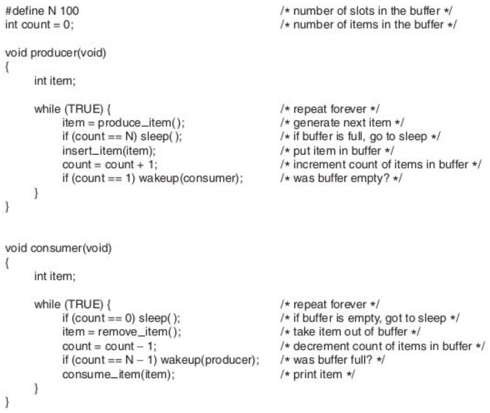
To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

```
enter_region:
        TSL REGISTER,LOCK               | copy lock to register and set lock to 1
        CMP REGISTER,#0                 | was lock zero?
        JNE enter_region                | if it was not zero, lock was set, so loop
        RET                             | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                    | store a 0 in lock
        RET                             | return to caller
```

Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring busy waiting (waste CPU time).



```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item();          /* generate next item */
                if (count == N) sleep();        /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);   /* was buffer empty? */
        }
}

void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep();        /* if buffer is empty, got to sleep */
                item = remove_item();           /* take item out of buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Mutex Locks

We use the mutex lock to protect critical regions and thus prevent race conditions.

That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock.

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
   available = true;
}
```

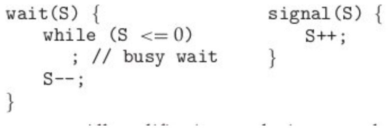
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Semaphores - E. W. Dijkstra (1965)

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

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

Semaphores - E. W. Dijkstra (1965)



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

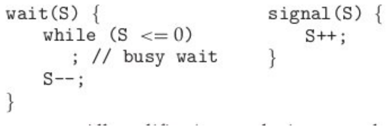
The wait operation on a semaphore checks to see if the value is greater than 0.

If so, it decrements the value and just continues.

If the value is 0, the process is put to sleep without completing the down for the moment.

Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action.

Semaphores - E. W. Dijkstra (1965)



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

The signal operation increments the value of the semaphore addressed.

If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its wait.

Poxis Semaphores

Operations:

- semaphore.h
- int sem_init(sem_t sem, int pshared, unsigned int value)
- int sem_wait(sem_t sem)
- int sem_post(sem_t sem)
- int sem_getvalue(sem_t sem, int valp)

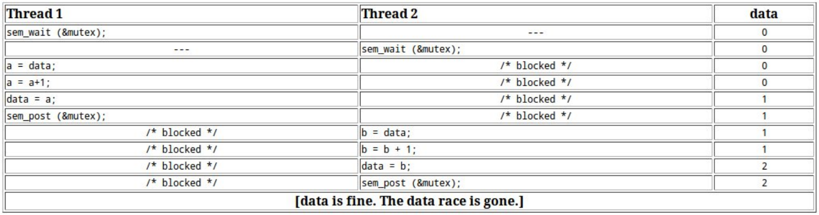
Semaphore use

Declare the semaphore global (outside of any funcion):

```
    sem_t mutex;
```

Initialize the semaphore in the main function:

```
    sem_init(&mutex, 0, 1);
```



Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b + 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2
[data is fine. The data race is gone.]		

Semaphore examples

Examples