BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Dr. Shubhangi

Synchronization

PROCESS SYNCHRONIZATION

Hardware Solutions

Disabling Interrupts: Works for the Uni Processor case only. WHY?

Atomic test and set: Returns parameter and sets parameter to true atomically.

```
while ( test_and_set ( lock ) );
/* critical section */
lock = false;
```

Example of Assembler code:

```
GET_LOCK:   IF_CLEAR_THEN_SET_BIT_AND_SKIP <bit_address>
            BRANCH   GET_LOCK                    /* set failed */
                     -------                     /* set succeeded */
```

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin - requires code built around the lock instructions.

PROCESS SYNCHRONIZATION

Hardware Solutions

```
Boolean          waiting[N];
int              j;            /* Takes on values from  0  to  N - 1  */
Boolean          key;
do  {
waiting[i]      = TRUE;
key             = TRUE;
while( waiting[i]  &&  key  )
                key  =  test_and_set( lock );   /*  Spin lock        */
waiting[ i ] = FALSE;
            /******   CRITICAL SECTION   ********/
j  =  ( i + 1 ) mod  N;
while (  ( j != i )  &&  ( ! waiting[ j ] ) )
            j  =  ( j + 1 ) % N;
if  ( j  ==  i )
            lock = FALSE;
else
            waiting[ j ] = FALSE;
            /******* REMAINDER SECTION *******/
} while (TRUE);
```

key useful for the most case, first BUSY WAIT.

ENTRY CODE

if it is willing to enter and it has the key.

key is set to false

now no longer interested in entering the critical section

checks in cyclic order

if in one cycle it could not find any P_i such that $P[j]$ = false and $j != i$.

if it found any waiting[j] = false before j == i, let lock = true; and set that process's waiting as false.

Using Hardware Test_and_set.

P_0	P_1	P_2	. . . . .	P_n
waiting[0]	waiting[1]	waiting[2]		waiting[n]
false	false	false		false

initial values

indicating that those processes will not be willing to enter the critical section

} local variables

false
key

false
lock

} global variables

→ initial value.

Whenever they want to enter, they will make their waiting value and key value to TRUE.

⇓

EXCHANGE VALUE OF KEY AND LOCK.

Assume that P_2 wants to enter.

	P_0	P_1	P_2	. . . . .	P_n
waiting[i]	F	F	T	. . . . .	F

T	F
key	lock

∴ key && waiting[2] = true.

key = TSL(&lock)

rv = *(lock)
rv = false.
set lock = true.
return rv == return false

∴ before P_2 enters CS,

	P_0	P_1	P_2	. . . . .	P_n
waiting[i]	F	F	F	. . . . .	F

F	T
key	lock

∴ while of P_2 is exit ∴ can enter CS.

Assume that P_1 tries to enter when P_2 in CS.

P_0	P_1	. . . .	P_n
F	T	. . . .	F

T	T
key	lock

∴ while (key && waiting[i])

key = TSL(&lock)

rv = *(lock) = True.

Lock = true

return rv == return True.

∴

T	T
key	lock

∴ stuck in while loop since (key && waiting[1]) is still true

∴ P_1 cannot enter CS when P_2 is already there.

∴ mutually exclusive.

CHECKING FOR PROGRESS

Case 1:

waiting = T, key = T ⇒ process wants to enter CS

Case 2:

waiting = T, key = F ⇒ for the very first process. (exactly before it enters Critical section)

Case 3:

waiting = F, key = T/F → in this case also it can enter CS.

P_0	P_1	. . . . .	P_n	key	lock
F	F	. . . .	F	F	F

P_2 enters :-

P_0	P_1	P_2	. . . .	P_n	key	lock
F	F	T	. . . .	F	T	F

while (key && waiting[2]) → true ⇒ key = TSL(&lock)

P_0	P_1	P_2	. . . .	P_n	key	lock
F	F	T	. . . .	F	F	T

∴ while breaks, and waiting[2] = F

∴

P_0	P_1	P_2	. . .	P_n	key	lock
F	F	F	. . .	F	F	T

Assume that n = 4, i = 2

j = (i+1) % 4 = 3 % 4 = 3

∴ j = 3

while(i ≠ j && !waiting[j])

∴ waiting[3] = false

∴ it gives turn to P_3 first.

but P_3 is not in waiting state

∴ it gives turn to P_0

⇒ P_0 is trying to enter CS ∴ waiting[0] = True

waiting[0]	waiting[1]	waiting[2]	waiting[3]
T	F	F	F

key	lock
T	T

key = true and waiting[0] = T and lock = T

∴ P_0 is stuck in while loop.

If waiting[0] = T, then j = 0; i = 2

∴ waiting[0] = F.

∴ P_0 breaks out of loop

since waiting[0] && key = false

(waiting[0]: False; key: True)

∴ P_0 enters critical section

it only allows P_2 to enter CS again no other process wants to enter.

but P_2 will eventually enter even if other processes exist that want to enter right after P_2 exits

∴ BOUNDED WAIT ACHIEVED.

∴ for the first iteration we have atleast 1 busy wait condition

Busy wait is not desirable because it leads to wastage in CPU cycles.

Swap Instruction

Definition:

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
}
```

Mutual exclusion using Swap

- Shared (global) Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key.
- Solution:

① Problem with Bounded Wait again

② Not portable: Architecture dependant (Hardware Solution)

```
while (true) {
   key = TRUE;
        while ( key == TRUE)
        Swap (&lock, &key );
                // critical section
        lock = FALSE;
                // remainder section
}
```

PROCESS SYNCHRONIZATION

Current Hardware Dilemmas

We first need to define, for multiprocessors:

caches,
shared memory (for storage of lock variables),
write through cache,
write pipes.

The last software solution we did (the one we thought was correct) may not work on a cached multiprocessor. Why? { Hint, is the write by one processor visible immediately to all other processors?}

↳ problems with reflecting value of lock/key updated.

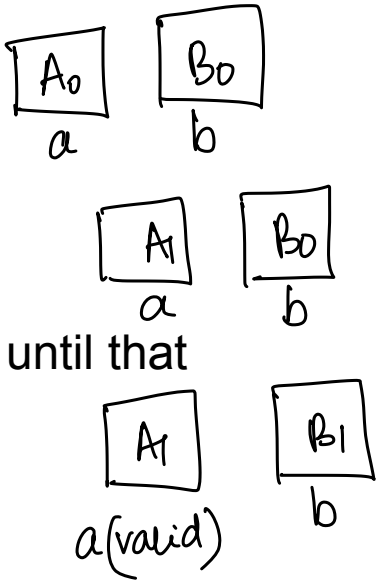
What changes must be made to the hardware for this program to work?

PROCESS SYNCHRONIZATION

Current Hardware Dilemmas

Does the sequence below work on a cached multiprocessor?

Initially, location **a** contains A0 and location **b** contains B0.



a) Processor 1 writes data A1 to location **a**.

b) Processor 1 sets **b** to B1 indicating data at **a** is valid.

c) Processor 2 waits for **b** to take on value B1 and loops until that change occurs.

d) Processor 2 reads the value from **a**.

What value is seen by Processor 2 when it reads **a**?

How must hardware be specified to guarantee the value seen?

a: A0 **b: B0**

PROCESS SYNCHRONIZATION

Current Hardware Dilemmas

We need to discuss:

Write Ordering: The first write by a processor will be visible before the second write is visible. This requires a write through cache.

Sequential Consistency: If Processor 1 writes to Location a "before" Processor 2 writes to Location b, then a is visible to ALL processors before b is. To do this requires NOT caching shared data.

The software solutions discussed earlier should be avoided since they require write ordering and/or sequential consistency.

PROCESS SYNCHRONIZATION

Current Hardware Dilemmas

Hardware test and set on a multiprocessor causes

- an explicit flush of the write to main memory and
- the update of all other processor's caches.

Imagine needing to write **all** shared data straight through the cache.

With test and set, **only** lock locations are written out explicitly.

In not too many years, hardware will no longer support software solutions because of the performance impact of doing so.

PROCESS SYNCHRONIZATION

Semaphores

PURPOSE:

We want to be able to write more complex constructs and so need a language to do so. We thus define semaphores which we assume are atomic operations:

P( ) variable: semaphore V( )

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

Busy Waiting

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

As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

FORMAT:

mutual exclusion.

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

PROCESS SYNCHRONIZATION

Semaphores

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

P1:

statement 1;

signal (synch);

P2:

wait (synch);

statement 2;

if we want $S_1(P_1)$ to occur before $S_2(P_2)$ then write it this way.

common semaphore.

synch: 1

At one time, only 1 process can enter the critical section.

synch: 0

P_1:
statement 1
signal(synch)

P_2
wait(synch)
statement 2

while(synch <= 0)
but synch = 1
∴ keeps on looping

P_1 ✓

signal(synch)
↳ >1 ∴ synch = 1

PROCESS SYNCHRONIZATION

Semaphores

We don't want to loop on busy, so will suspend instead:

- 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:

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

PROCESS SYNCHRONIZATION

Semaphores

2 → non binary
∴ "counting semaphore"

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

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

Add this process to blocked queue belongs to the semaphore:

no need to loop.

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

multiple semaphores → multiple blocked Queues.

- It's critical that these be atomic - in uniprocessors we can disable interrupts, but in multiprocessors other mechanisms for atomicity are needed.
- Popular incarnations of semaphores are as "event counts" and "lock managers". (We'll talk about these in the next chapter.)