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);
```

Using Hardware Test_and_set.

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:

```
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?}

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:

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

```
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:

```
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;
```

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

```
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;
      }
}
```

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

- 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.)