PROCESS SYNCHRONIZATION

Two Processes Software

FLAG FOR EACH PROCESS GIVES STATE:

Each process maintains a flag indicating that it wants to get into the critical section. It checks the flag of the other process and doesn't enter the critical section if that other process wants to get in.

Shared variables

☞**boolean flag[2]**;
initially **flag [0] = flag [1] = false.**

☞**flag [i] = true** ⇒ P_i ready to enter its critical section

Algorithm 2

```
do {
        flag[i] := true;
        while (flag[j]) ;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

Are the three Critical Section Requirements Met?

PROCESS SYNCHRONIZATION

Two Processes Software

FLAG TO REQUEST ENTRY:

- Each processes sets a flag to request entry. Then each process toggles a bit to allow the other in first.
- This code is executed for each process i.

Algorithm 3

Shared variables

☞**boolean flag[2]**;
initially **flag [0] = flag [1] = false.**

☞**flag [i] = true** ⇒ P_i ready to enter its critical section

```
 do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn == j) ;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

Are the three Critical Section Requirements Met?

This is Peterson's Solution

PROCESS SYNCHRONIZATION

Critical Sections

The hardware required to support critical sections must have (minimally):

- Indivisible instructions (what are they?)
- Atomic load, store, test instruction. For instance, if a store and test occur simultaneously, the test gets EITHER the old or the new, but not some combination.
- Two atomic instructions, if executed simultaneously, behave as if executed sequentially.

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.

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.