PROCESS SYNCHRONIZATION I

THE PRODUCER CONSUMER PROBLEM

A **producer** process "produces" information "consumed" by a **consumer** process.

Consider the following code segments:

- Does it work?
- Are all buffer locations utilized?

```
initialize: in = out = 0;

Producer

        Repeat forever
                ....
                produce an item in nextp
                while   in + 1  mod n  ==  out  do no-op;

                        buffer[in]  =  nextp;
                        in          =  in + 1 mod n;

                until false;


Consumer

        Repeat forever
                 ....
                while   in  ==  out   do no-op;
                        nextc  =  buffer[out];
                        out    =  out + 1 mod n;
                         ...
                        consume the item in nextc
                         ...

                until false;
```

- How about this implementation? Does it work?

```
initialize:  in = out = counter = 0;

Producer

    Repeat forever
        ....
        produce an item in nextp
        ....
        while  counter  ==  n  do no-op;

            buffer[in]  = nextp;
            in          = in + 1 mod n;
            counter     = counter + 1;

        until false;

Consumer

    Repeat forever
        while  counter  ==  0  do no-op;
            nextc     = buffer[out];
            out       = out + 1 mod n;
            counter   = counter - 1;
            ...
            consume the item in nextc
            ...
        until false;
```

Note that **counter = counter + 1;** ← **this line is NOT what it seems!!**

is really --> register = counter
register = register + 1
counter = register

At a micro level, the following scenario could occur using this code:

TO;	**Producer**	**Execute**	**register1 = counter**	**register1 = 5**
T1;	**Producer**	**Execute**	**register1 = register1 + 1**	**register1 = 6**
T2;	**Consumer**	**Execute**	**register2 = counter**	**register2 = 5**
T3;	**Consumer**	**Execute**	**register2 = register2 - 1**	**register2 = 4**
T4;	**Producer**	**Execute**	**counter = register1**	**counter = 6**
T5;	**Consumer**	**Execute**	**counter = register2**	**counter = 4**

CRITICAL SECTIONS:

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

Entry section	Code requesting entry into the critical region.
Critical Section	Code in which only one process can execute at any one time.
Exit section	The end of the critical region, releasing or allowing others in.
Remainder section	Rest of the code AFTER the critical region.

The critical section must ENFORCE ALL THREE of the following rules:

Mutual Exclusion	No more than one process can execute in its critical section at one time.
Progress	If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time who should go in.
Bounded Wait	All requesters must eventually be let into the critical section.

The hardware 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.

TWO PROCESSES SOFTWARE SOLUTIONS:

Here we try a succession of increasingly complicated solutions to the problem of creating valid entry sections.

NOTE: In all examples, **i** is the current process, **j** the "other" process. In these examples, envision the same code running on two processors at the same time.

TOGGLED ACCESS:

The code alternately allows each of the processes access in turn. But what happens, for example, if **i** wants to get in twice?

```
repeat
        while  ( turn  ^=  i )  do no-op;
        /* critical section  */
        turn  =  j;
        /* remainder section */
until ( false );
```

- Are the three Critical Section Requirements Met?

NOTE: The equivalent C code for this is:

```
while (TRUE)
      {
      while ( turn ^= i );
      /*  Critical Section  */
      turn = j;
}
```

FLAG FOR EACH PROCESS INDICATES STATE:

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

```
var flag: array [0..1] of Boolean;

repeat
        flag[i]  =  true;
        while ( flag[j] ) do no-op;
        /* critical section  */
        flag[i]  =  false;
        /* remainder section */
until ( false );
```

- Are the three Critical Section Requirements Met?

FLAG TO REQUEST ENTRY:

Each processes sets a flag to request entry. But each process toggles a bit to allow the other in first.

This code is executed for each process i.

```
var   flag:  array [0..1] of Boolean;
      turn:  0..1;

repeat
        flag[i]  =  true;
        turn     =  j;
        while ( flag[j]  and  turn  ==  j  ) do no-op;
        /* critical section  */
         flag[i]  =  false;
        /* remainder section */
until ( false );
```

- Are the three Critical Section Requirements Met?

HARDWARE APPROACHES:

- **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 ) ) do no-op;

        /* critical section */

lock = false;
```

Example of Assembler code:

```
GET_LOCK:   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.

```
Boolean     waiting[[N];
int         j;              /* Takes on values from  0   to   N - 1
  */
Boolean     key;

while
  {
  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 ) mod N;
  if  (  j  ==  i )
          lock = FALSE;
  else
          waiting[ j ] = FALSE;
  REMAINDER SECTION
}
```

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) fails utterly 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?

- 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?

- 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.

- 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.

- This is cheap relative to avoiding cache for ALL shared data. Here 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.

SEMAPHORES:

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 do no-op;
        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
```

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

```
P1:                                P2:
   statement 1;                            wait ( synch );
   signal ( synch );                  statement 2;
```

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

  a) Block on semaphore == False,
  b) Wakeup on signal (semaphore becomes True),
  c) There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,
  d) Wakeup one of the blocked processes upon getting a signal (choice of who depends on strategy).

To PREVENT looping, we redefine the semaphore operations as:

```
type semaphore = record
    value:      integer;
    L:          list of process;
end;

wait(s):
    s.value = s.value - 1;
    if ( s.value < 0 )
        then begin
        add this process to s.L;      /*  linked list of PTBL waiting on
S  */
        block;
    end;


signal(s):
    s.value = s.value + 1;
    if ( s.value <= 0 )
        then begin
        remove a process P from s.L;
        wakeup(P);                    /*  Make P ready to run  */
    end;
```

- 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 a few minutes.)

DEADLOCKS:

- May occur when two or more processes try to get the same multiple resources at the same time.

```
P1:                          P2:
    wait(S);                     wait(Q);
    wait(Q);                     wait(S);
    .....                             .....
    signal(S);                   signal(Q);
    signal(Q);                   signal(S);
```

- How can this be fixed?

SEQUENCERS AND EVENTCOUNTS:

Sequencer: A non-decreasing integer variable, initialized to 0, that can be used to totally order events. (Philosophically similar to the take-a-number machines in bakeries and other such places.) The operation of getting the next sequential number is given by

v = **ticket**(S);

The next sequential number is returned as "v" from sequencer "S".

Event Count: Is the stack of used sequence numbers. The top number on the stack represents the customer currently being served.

AWAIT:

Eventcounts can be operated on as follows:

await(E, v);

Await on eventcount/stack "E" until all the previous numbers/customers have been served. Wait until the sequence of numbers, or the eventcount, reaches the number v and then wake up the process. More formally,

```
await( E, v );
  if E .count < v then
        place the executing process in the waiting queue
        associated with E and invoke the CPU scheduler.
  endif;
```

ADVANCE:

Eventcounts can also be operated on as follows:

advance(E);

This corresponds to the clerk completing service of a customer and moving on to the next one. For a process, it causes that process to be awakened whose ticket has now been reached. The operation is defined as:

advance(E);
E .count = E.count + 1;
wake up the process(es) waiting for E's value to
reach the current value just attained;

READ:

Eventcounts can also be "read". This allows programs to find out how long it will be before they are serviced.

current_count = read(E);

THE PRODUCER CONSUMER PROBLEM USING EVENTCOUNTS:

```
/* Shared Variables */

var     Pticket, Cticket:       sequencer;
        in, out:                        eventcount;
        buffer: array[0..N-1]   of message;

/* Parallel programs for producer and consumer are shown here*/
```

PRODUCER I

```
var t:  integer;                        /* Variable t is local to each producer.*/

loop
 /*Create a new message m*/

        t:= ticket(Pticket);            /*One producer at a time*/
        await(in, t);

        await(out, t - N + 1 );         /*Await an empty cell */
        buffer[ t mod N ] := m;

        advance( in );                  /* Signal full buffer: allow other producers.*/
endloop
```

CONSUMER J

```
var u:  integer;                        /* Variable u is local to each producer. */

loop
        u:= ticket(Cticket);            /*One consumer at a time*/
        await(out, u);

        await(in, u + 1 );              /*Await a message */
        m := buffer[ u mod N ];

        advance( out );         /* Signal empty buffer: allow other consumers.*/
        /* Consume message m */
        ...
endloop
```

- The use of two "awaits" allows concurrently one producer and one consumer. A single await for each would have a shorter code path, but wouldn't be concurrent.
- The "in" eventcount assures only one producer at a time and guarantees a buffer location has been filled before enabling a consumer.