# Process

## Communication / Synchronization

# 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.
    - **Atomic = non-interruptable**
- Two atomic instructions, if executed simultaneously, behave as if executed sequentially.

# PROCESS SYNCHRONIZATION

## Hardware Solutions

**Disabling Interrupts**:

Works for the Uniprocessor case only.
Needs a modified approach for multiprocessor / multi-core processors.

**Disable interrupts (for e.g., DI)**
**/* critical region */**
**Enable interrupts (for e.g., EI)**

**Atomic test and set (Use of TSL instruction)**

Returns parameter & sets parameter to true atomically.

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

# The TSL Instruction ...(1)

```
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 nonzero, 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
```

**Entering and leaving a critical region using the TSL instruction.**

# The XCHG Instruction ...(2)

```
enter_region:
      MOVE REGISTER,#1          | put a 1 in the register
      XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
      CMP REGISTER,#0           | was lock zero?
      JNE enter_region          | if it was non 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
```

**Entering and leaving a critical region using the XCHG instruction.**

# Can we find a solution to busy waiting?

Can we have a mechanism were a process is not constantly checking for the availability of CR, rather is being informed about the availability of CR as and when that scenario arises?

# Can we find a solution to busy waiting?

Can we have a mechanism were a process is not constantly checking for the availability of CR, rather is being informed about the availability of CR as and when that scenario arises?

SLEEP and WAKEUP operations

A call to **SLEEP** blocks the calling process.

A call to **WAKEUP** unblocks a process that is passed as an argument in the call.

# Producer-Consumer Problem

Assume there are two special operations – sleep and wakeup.

**Write a pseudocode for the producer-consumer problem using these above operations.**

**Analyze your pseudocode.**

# Pseudocode for Producer-Consumer Problem

```
#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 */
        }
}
```

Are there any issues with the above code?

# Semaphores

- Need to generalize critical section problems
- Need to ensure ATOMIC access to shared variables.
- Semaphore provides an integer variable that is only accessible through semaphore operations:

P

```
WAIT ( S ):
     while   ( S  <=  0 ); /* empty while loop */
     S  =  S  -  1;
```

V

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

**Typical Usage Format:**

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

# Understanding Semaphore
## Implementation

We don't want to loop on busy, so will block the process instead:

- Block on semaphore == False (or on a value of 0)
- 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 need to redefine the semaphore structure and operations wait / signal.**

# Counting Semaphore Implementation

```
struct semaphore {
    int  value;
    int  L[size];
} s ;
```

Different semaphores will have different queues.

Assumes two internal operations: *block;* ***and*** *wakeup(p);*

block – place process invoking the operation on an appropriate waiting queue.

wakeup – remove one of processes in the waiting queue and place it in the ready queue.

```
wait(s)
  s.value--;
  if (s.value < 0)
     add to s.L
     block;
```

```
signal(s)
  s.value++;
  if (s.value <= 0)
     remove P from s.L;
     wakeup(P)
```

# PROCESS SYNCHRONIZATION

# Some Interesting Problems

## THE BOUNDED BUFFER ( PRODUCER / CONSUMER ) PROBLEM:

This is the same producer / consumer problem as before. But now we'll do it with signals and waits. Remember: a **wait decreases** its argument and a **signal increases** its argument.

**HINT**

**BINARY_SEMAPHORE mutex = 1; // Can only be 0 or 1**

**COUNTING_SEMAPHORE empty = n; full = 0; // Can take on any integer value**

# Some Interesting Problems

## THE BOUNDED BUFFER ( PRODUCER / CONSUMER ) PROBLEM:

This is the same producer / consumer problem as before. But now we'll do it with signals and waits. Remember: a **wait decreases** its argument and a **signal increases** its argument.

**BINARY_SEMAPHORE mutex = 1; // Can only be 0 or 1**

**COUNTING_SEMAPHORE empty = n; full = 0; // Can take on any integer value**

```
producer:
do {
     /* produce an item in nextp */
     wait (empty);       /* Do action     */
     wait (mutex);       /* Buffer guard*/
     /* add nextp to buffer  */
     signal (mutex);
     signal (full);
  } while(TRUE);
```

```
consumer:
do {
     wait (full);
     wait (mutex);
     /* remove an item from buffer to nextc */
     signal (mutex);
     signal (empty);
     /* consume an item in nextc */
  } while(TRUE);
```