OPERATING SYSTEMS

PROCESS SYNCHRONIZATION

Jerry Breecher

OPERATING SYSTEM Synchronization

What Is In This Chapter?

- This is about getting processes to coordinate with each other.
- How do processes work with resources that must be shared between them?
- How do we go about acquiring locks to protect regions of memory?
- How is synchronization really used?

OPERATING SYSTEM Synchronization

Topics Covered

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples
- Atomic Transactions

PROCESS SYNCHRONIZATION

The Producer Consumer Problem

A **producer** process "produces" information "consumed" by a **consumer** process.
Here are the variables needed to define the problem:

```
#define BUFFER_SIZE 10
typedef struct {
    DATA            data;
} item;
item   buffer[BUFFER_SIZE];
int    in = 0;              // Location of next input to buffer
int    out = 0;             // Location of next removal from buffer
int    counter = 0;         // Number of buffers currently full
```

Consider the code segments on the next page:

- Does it work?
- Are all buffers utilized?

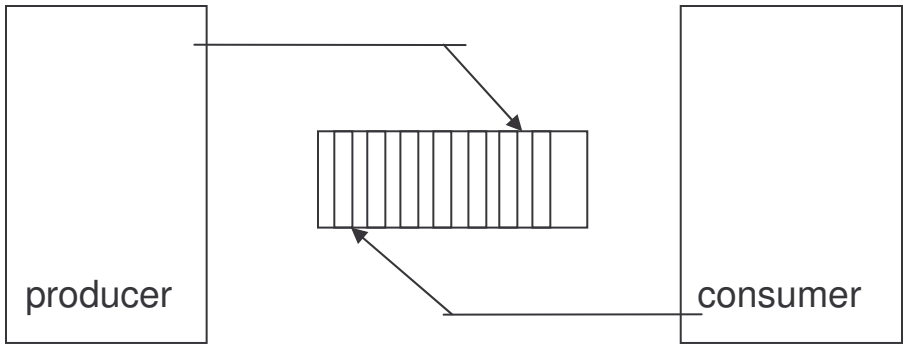
PROCESS SYNCHRONIZATION

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

```
item   nextProduced;

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

PRODUCER



The Producer Consumer Problem

```
#define BUFFER_SIZE 10
typedef struct {
    DATA        data;
} item;
item    buffer[BUFFER_SIZE];
int     in = 0;
int     out = 0;
int     counter = 0;
```

```
item  nextConsumed;

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) %
    BUFFER_SIZE;
    counter--;
}
```

CONSUMER

PROCESS SYNCHRONIZATION

The Producer Consumer Problem

Note that **counter++;** **← 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**

PROCESS SYNCHRONIZATION

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

PROCESS SYNCHRONIZATION

Critical Sections

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.

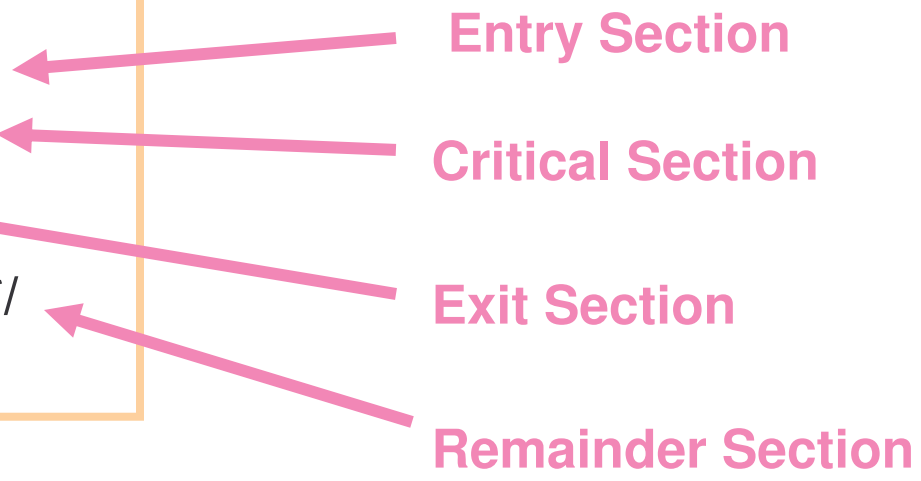
Bounded Wait: All requesters must eventually be let into the critical section.

PROCESS SYNCHRONIZATION

Two Processes Software

Here's an example of a simple piece of code containing the components required in a critical section.

```
do {
        while  ( turn  ^=  i );
        /* critical section  */
        turn  =  j;
        /* remainder section */
 } while(TRUE);
```



PROCESS SYNCHRONIZATION

Two Processes Software

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:

```
do {
        while  ( turn  ^=  i );
        /* critical section  */
        turn  =  j;
        /* remainder section */
} while(TRUE);
```

Algorithm 1

Are the three Critical Section Requirements Met?

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** $\Rightarrow$ 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** $\Rightarrow$ 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.

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.)

PROCESS SYNCHRONIZATION

Semaphores

DEADLOCKS:

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

P1:

```
wait(S);
wait(Q);
.....
signal(S);
signal(Q);
```

P2:

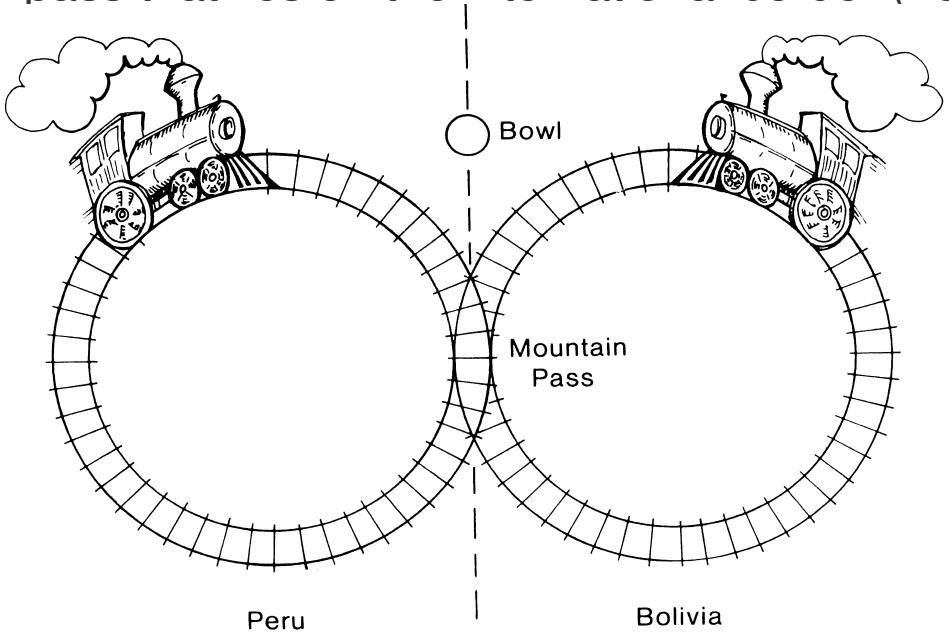
```
wait(Q);
wait(S);
.....
signal(Q);
signal(S);
```

- How can this be fixed?

PROCESS SYNCHRONIZATION

Railways in the Andes; A Practical Problem

High in the Andes mountains, there are two circular railway lines. One line is in Peru, the other in Bolivia. They share a common section of track where the lines cross a mountain pass that lies on the international border (near Lake Titicaca?).



Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the common section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains are both **blind** and **deaf**, so they can neither see nor hear each other.

The two drivers agreed on the following method of preventing collisions. They set up a large bowl at the entrance to the pass. Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see it it contains a rock. If the bowl is empty, the driver finds a rock and drops it in the bowl, indicating that his train is entering the pass; once his train has cleared the pass, he must walk back to the bowl and remove his rock, indicating that the pass in no longer being used. Finally, he walks back to the train and continues down the line.

If a driver arriving at the pass finds a rock in the bowl, he leaves the rock there; he repeatedly takes a siesta and rechecks the bowl until he finds it empty. Then he drops a rock in the bowl and drives his train into the pass. A smart graduate from the University of La Paz (Bolivia) claimed that subversive train schedules made up by Peruvian officials could block the train forever.

Explain

The Bolivian driver just laughed and said that could not be true because it never happened.

Explain

Unfortunately, one day the two trains crashed.

Explain

Following the crash, the graduate was called in as a consultant to ensure that no more crashes would occur. He explained that the bowl was being used in the wrong way. The Bolivian driver must wait at the entry to the pass until the bowl is empty, drive through the pass and walk back to put a rock in the bowl. The Peruvian driver must wait at the entry until the bowl contains a rock, drive through the pass and walk back to remove the rock from the bowl. Sure enough, his method prevented crashes.

Prior to this arrangement, the Peruvian train ran twice a day and the Bolivian train ran once a day. The Peruvians were very unhappy with the new arrangement.

Explain

The graduate was called in again and was told to prevent crashes while avoiding the problem of his previous method. He suggested that two bowls be used, one for each driver. When a driver reaches the entry, he first drops a rock in his bowl, then checks the other bowl to see if it is empty. If so, he drives his train through the pass. Stops and walks back to remove his rock. But if he finds a rock in the other bowl, he goes back to his bowl and removes his rock. Then he takes a siesta, again drops a rock in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty. This method worked fine until late in May, when the two trains were simultaneously blocked at the entry for many siestas.

Explain

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.

```
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);
```

PROCESS SYNCHRONIZATION

Some Interesting Problems

THE READERS/WRITERS PROBLEM:

This is the same as the Producer / Consumer problem except - we now can have many concurrent readers and one exclusive writer.

Locks: are **shared** (for the readers) and **exclusive** (for the writer).

Two possible (contradictory) guidelines can be used:

- No reader is kept waiting unless a writer holds the lock (the readers have precedence).
- If a writer is waiting for access, no new reader gains access (writer has precedence).

(NOTE: starvation can occur on either of these rules if they are followed rigorously.)

PROCESS SYNCHRONIZATION

Some Interesting Problems

THE READERS/WRITERS PROBLEM:

```
BINARY_SEMAPHORE   wrt       = 1;
BINARY_SEMAPHORE   mutex     = 1;
int                readcount = 0;
```

```
Writer:
do {
        wait( wrt );
        /*   writing is performed    */
        signal(  wrt  );
} while(TRUE);
```

```
Reader:
do {
        wait( mutex );                          /* Allow 1 reader in entry*/
        readcount = readcount + 1;
        if readcount == 1  then  wait(wrt );    /* 1st reader locks writer */
        signal( mutex );
                /*   reading is performed  */
         wait( mutex );
         readcount = readcount - 1;
         if readcount == 0  then  signal(wrt );   /*last reader frees writer */
         signal( mutex );
} while(TRUE);
```

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

PROCESS SYNCHRONIZATION

Some Interesting Problems

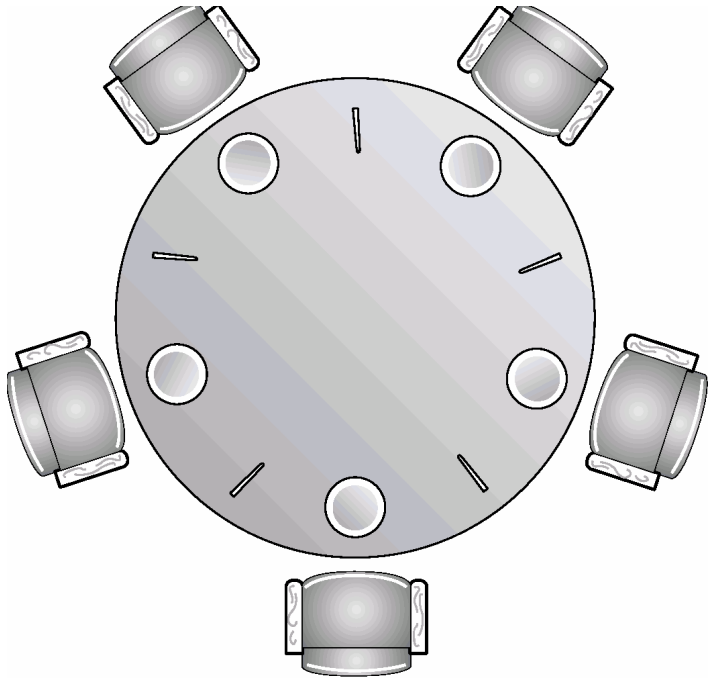
THE DINING PHILOSOPHERS PROBLEM:

5 philosophers with 5 chopsticks sit around a circular table. They each want to eat at random times and must pick up the chopsticks on their right and on their left.

Clearly deadlock is rampant (and starvation possible.)

Several solutions are possible:

- Allow only 4 philosophers to be hungry at a time.
- Allow pickup only if both chopsticks are available. (Done in critical section)
- Odd # philosopher always picks up left chopstick 1st, even # philosopher always picks up right chopstick 1st.



PROCESS SYNCHRONIZATION

Critical Regions

High Level synchronization construct implemented in a programming language.

A shared variable v of type T, is declared as:
var v; **shared** T

Variable v is accessed only inside a statement:
region v **when** B **do** S

where B is a Boolean expression.

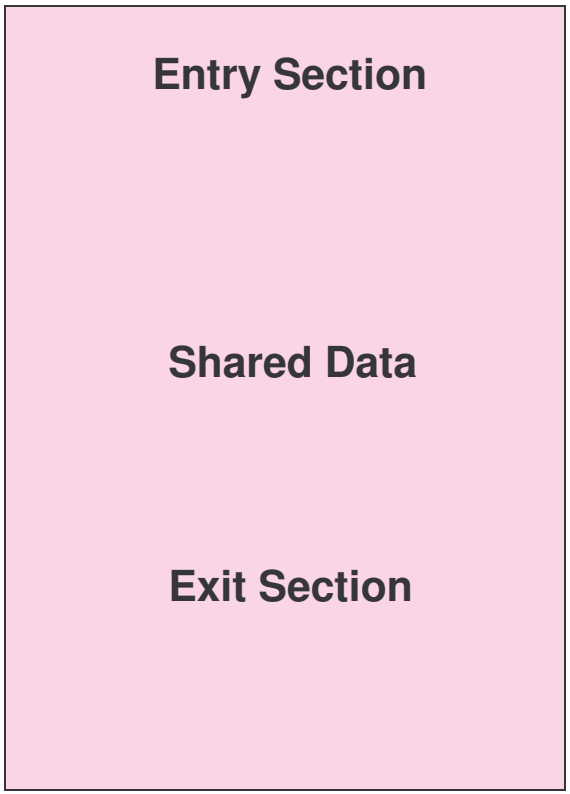
While statement S is being executed, no other process can access variable v.

Regions referring to the same shared variable exclude each other in time.

When a process tries to execute the region statement, the Boolean expression B is evaluated.

If B is true, statement S is executed.

If it is false, the process is delayed until B is true and no other process is in the region associated with v.



Critical Region

PROCESS SYNCHRONIZATION

Critical Regions

EXAMPLE: Bounded Buffer:

Shared variables declared as:

```
struct buffer {
    int     pool[n];
    int     count, in, out;
}
```

Producer process inserts **nextp** into the shared buffer:

```
region   buffer  when( count < n) {
            pool[in] = nextp;
            in:= (in+1) % n;
            count++;
}
```

Consumer process removes an item from the shared buffer and puts it in nextc.

```
Region  buffer  when (count > 0) {
            nextc = pool[out];
            out = (out+1) % n;
            count--;
}
```

PROCESS SYNCHRONIZATION

Monitors

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (...) {
            . . .
        }
        procedure body P2 (...) {
            . . .
        }
        procedure body Pn (...) {
            . . .
        }
        {
            initialization code
        }
}
```

PROCESS SYNCHRONIZATION

Monitors

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

    means that the process invoking this operation is suspended until another process invokes

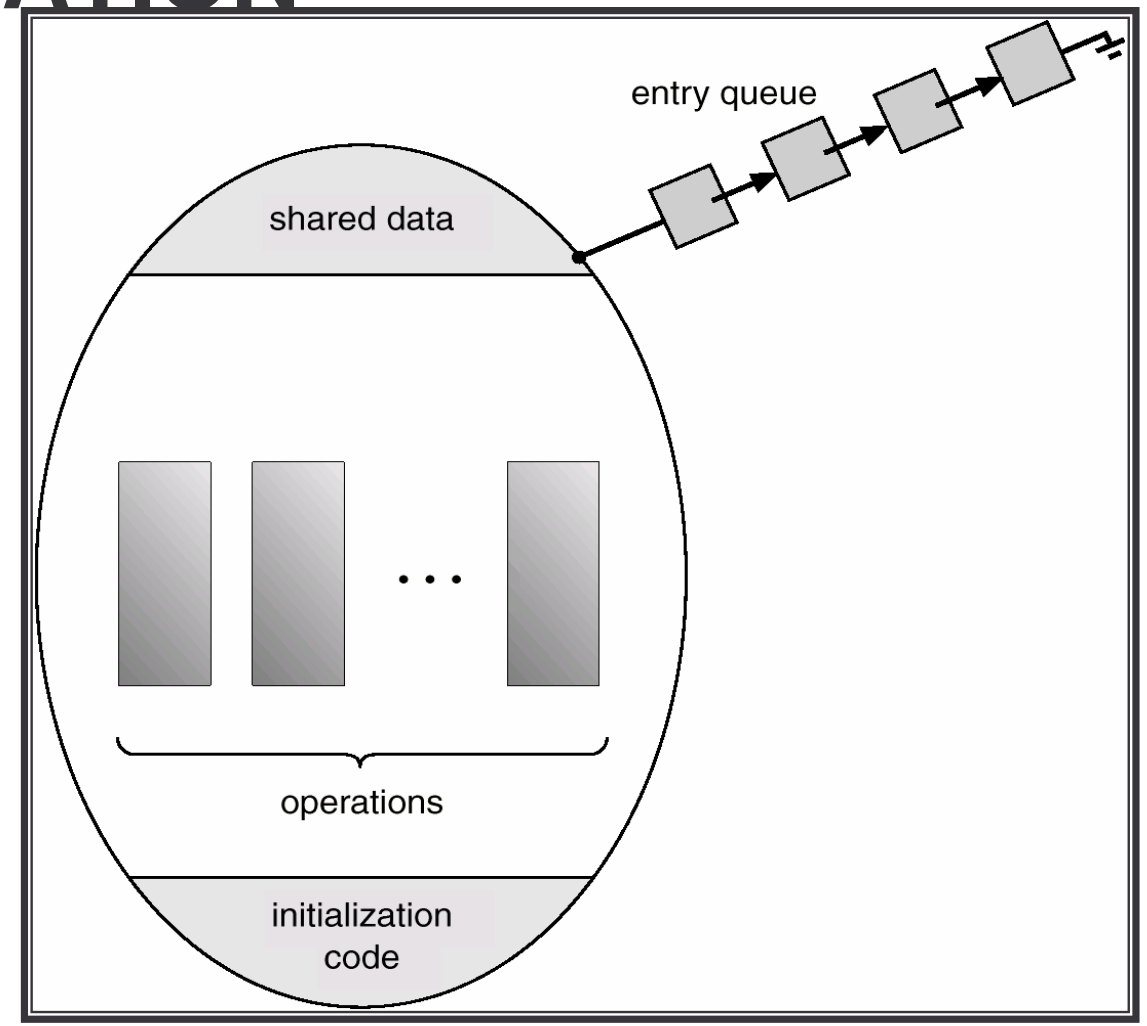
    x.signal();

  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

PROCESS SYNCHRONIZATION

Monitors

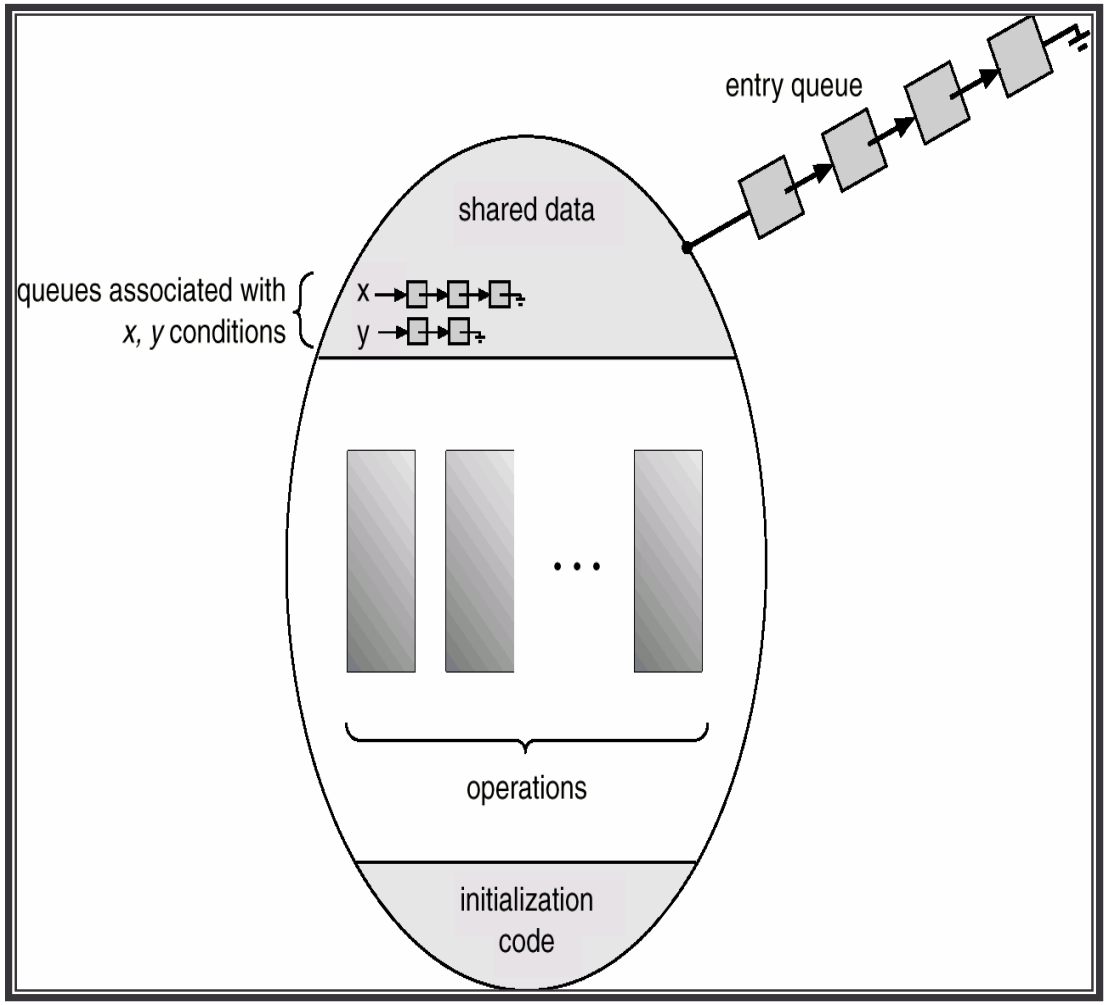
Schematic View of a Monitor



PROCESS SYNCHRONIZATION

Monitors

Monitor With Condition Variables



PROCESS SYNCHRONIZATION

Monitors

Dining Philosophers Example

```
monitor dp {
      enum {thinking, hungry, eating} state[5];
      condition          self[5];
}
```

```
initializationCode() {
    for ( int  i = 0;  i < 5;  i++ )
              state[i] = thinking;
}
```

```
void test(int i) {
      if ( (state[(I + 4) % 5] != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
                state[i] = eating;
                self[i].signal();
       }
}
```

```
void putdown(int i) {
      state[i] = thinking;
      // test left & right neighbors
      test((i+4) % 5);
      test((i+1) % 5);
}
```

```
void pickup(int i) {
      state[i] = hungry;
      test[i];
      if (state[i] != eating)
                self[i].wait();
}
```

PROCESS SYNCHRONIZATION

How Is This Really Used?

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Also provides *dispatcher objects* which may act as either mutexes or semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

PROCESS SYNCHRONIZATION

Wrap up

In this chapter we have:

Looked at many incarnations of the producer consumer problem.

Understood how to use critical sections and their use in semaphores.

Synchronization IS used in real life. Generally programmers don't use the really primitive hardware locks, but use higher level mechanisms as we've demonstrated.