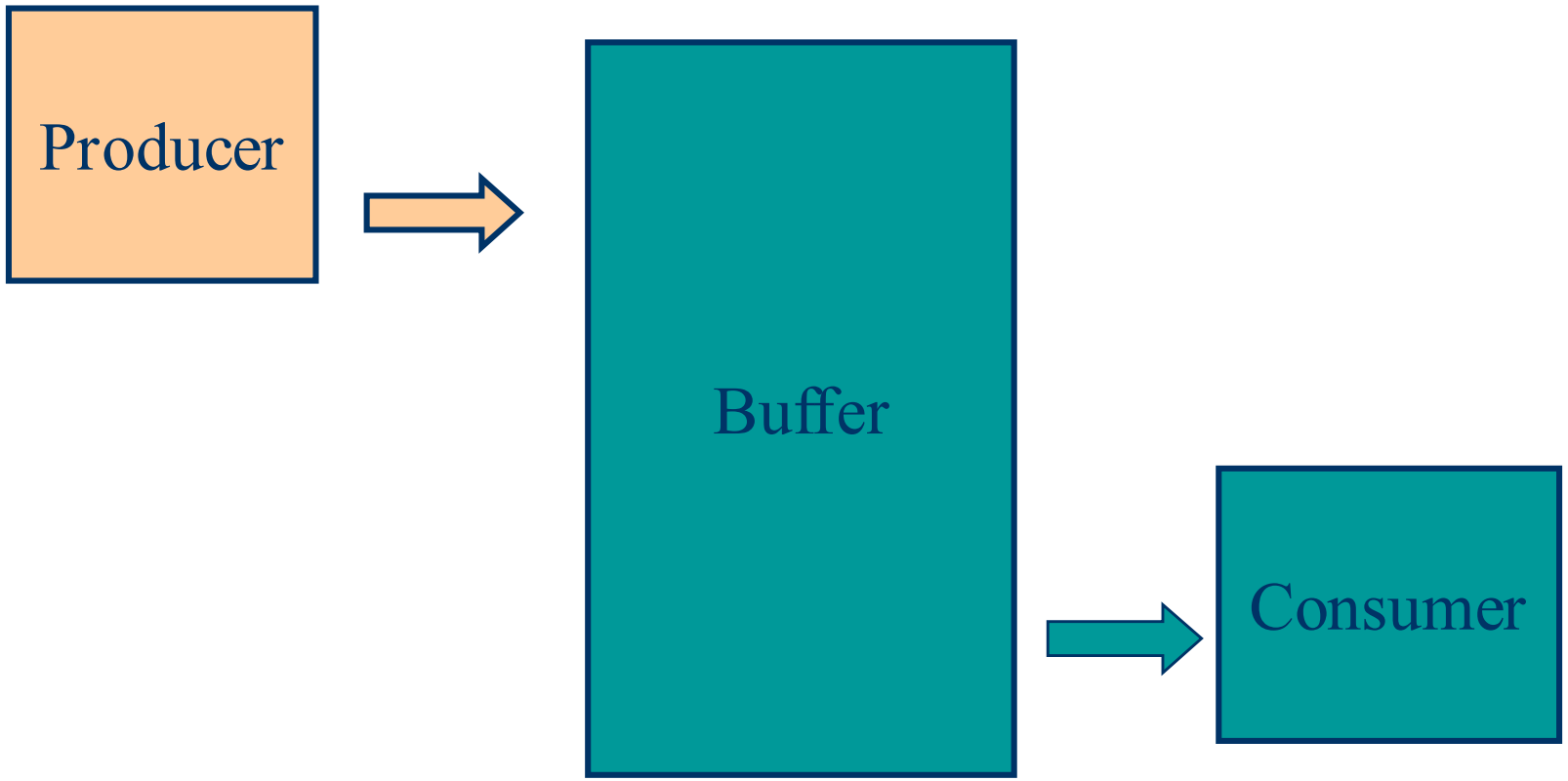
Inter-Process Communication

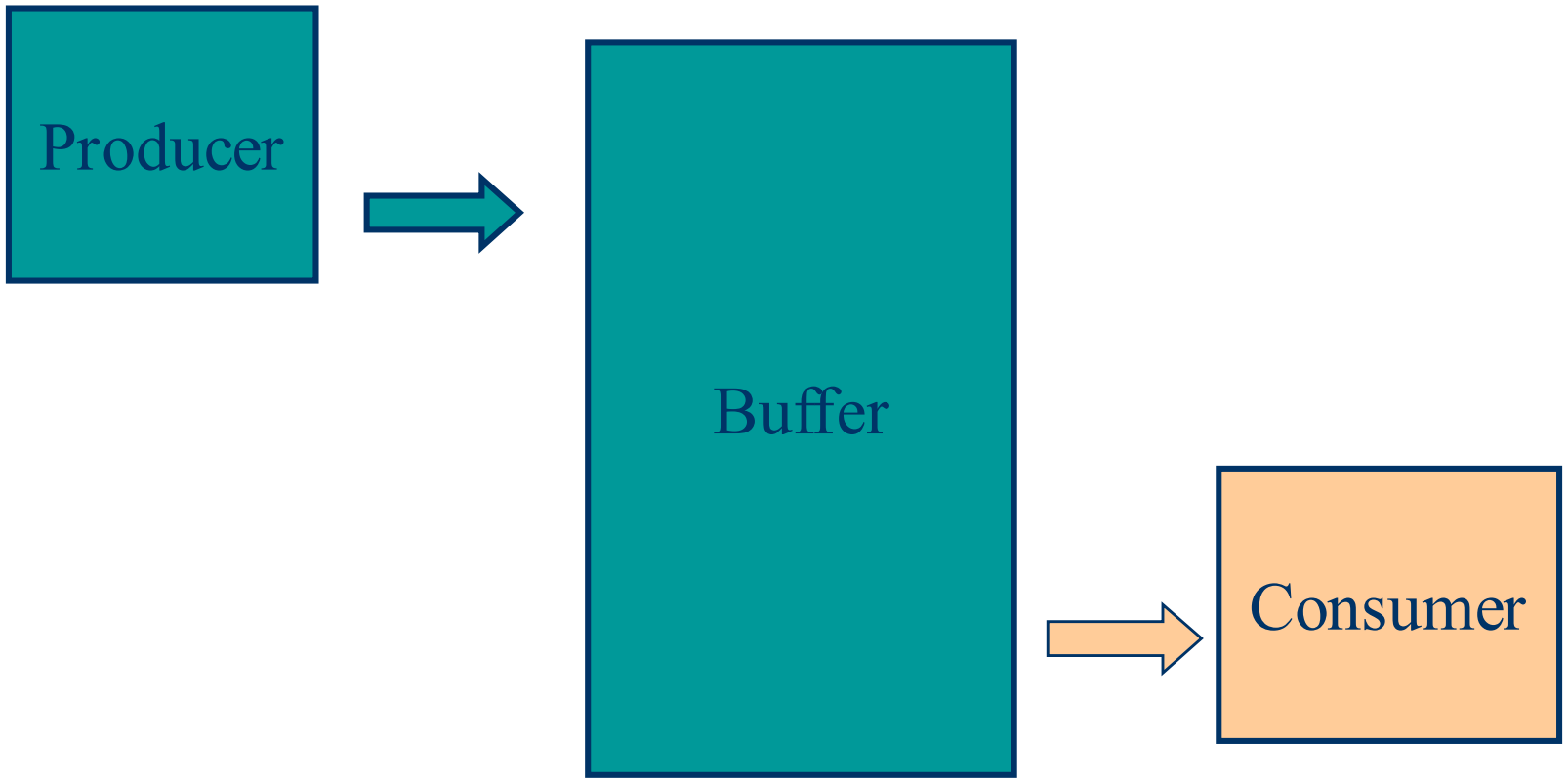
Inter-process Communications

- Passing information between processes
- Used to coordinate process activity
- May result in data inconsistency if mechanisms are not in place to ensure orderly execution of the processes.
- The problem can be identified as a race condition

Producer / Consumer Problem



Producer / Consumer Problem



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 slide:

- Does it work?
- Are all buffers utilized?

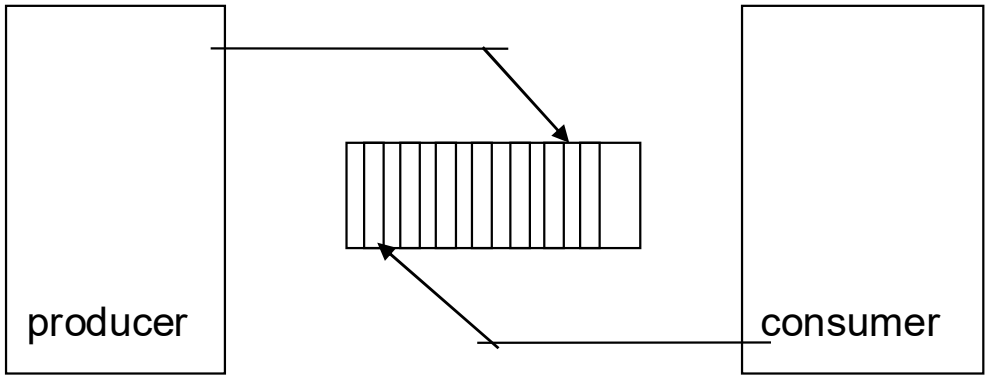
PROCESS SYNCHRONIZATION

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

PRODUCER

```
item    nextProduced;

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



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;
```

CONSUMER

```
item  nextConsumed;

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

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

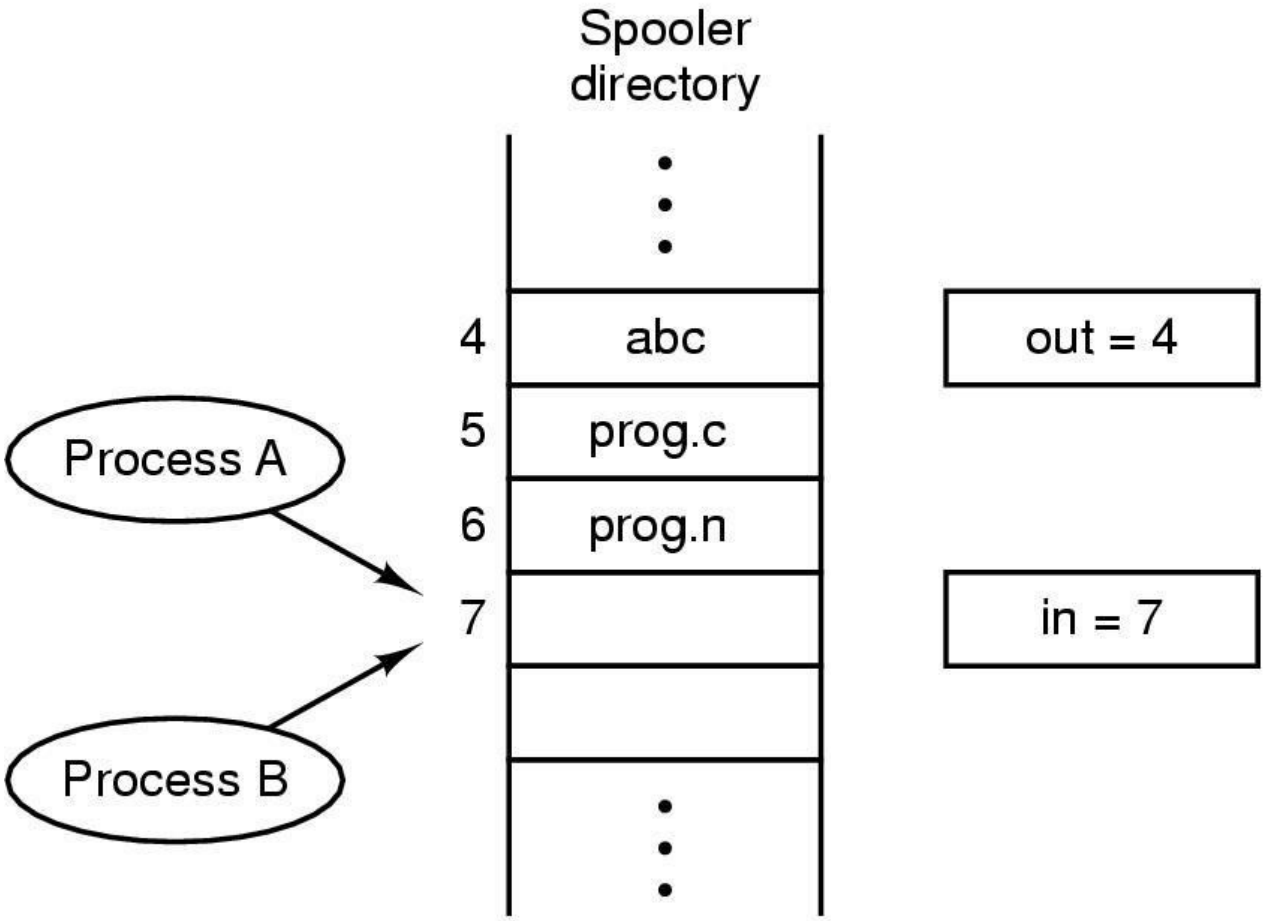
At a micro level, the following scenario could occur using this code:

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**

The Problem:

atomic execution of counter changes

Race Conditions



Two processes want to access shared memory at the same time.

The Critical Section / Region Problem

- Occurs in systems where multiple processes all compete for the use of shared data.
- Each process includes a section of code (the **critical section**) where it accesses this shared data.
- The problem is to ensure that **only one process at a time is allowed** to be operating in its critical section.

PROCESS SYNCHRONIZATION

Critical Section / Region Problem

Critical section must ENFORCE ALL 3 of the following rules:

Mutual Exclusion: No more than one process can execute in its critical section at one time.

Progress: If no process is in the critical section and one of them wants to go in, then the interested processes should be able to decide in a finite time who should go in.
Processes in their remainder section must not block another process interested to enter a critical section.
Selection of that process cannot be delayed indefinitely.

Bounded Wait: All requesters (processes) must eventually be let into the critical section.
There is a bound on the number of times that a waiting process can be superceded.

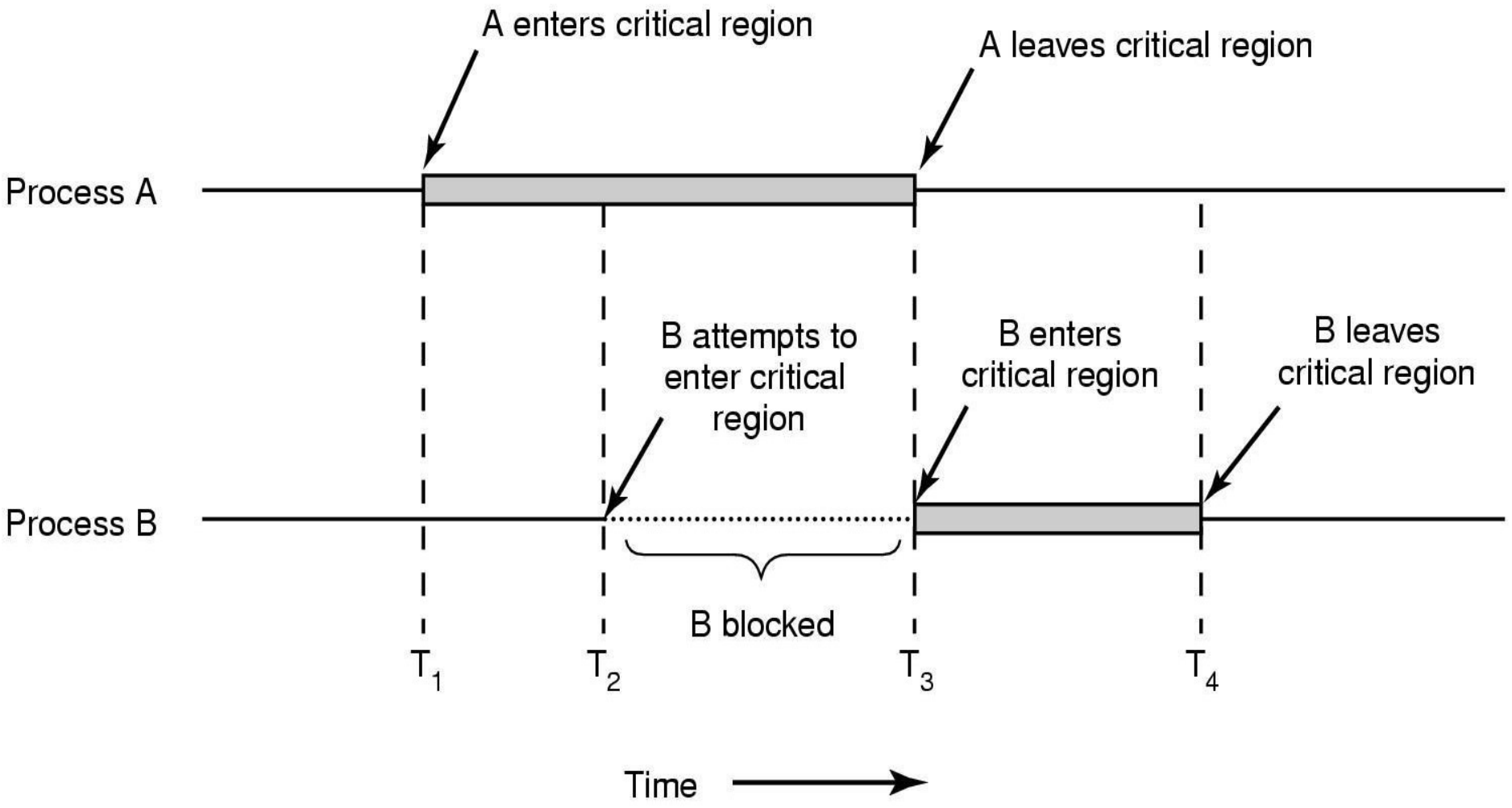
Can **Race Conditions** be avoided using above conditions?

Conditions to avoid Race Conditions

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Genesis: Three conditions on the previous slide

How Critical Regions Operate?



Mutual Exclusion Preserved!

Are there are other conditions we should worry about?

Guess?

Are there are other conditions we should worry about?

- Starvation
- Deadlock

Try this Exercise! Lets see your approach

Following solution is alleged to be a solution to the critical section problem. Argue for its correctness or show a case in which it fails.

```
shared int turn;
shared boolean flag[2];


proc (int i) {
   while (TRUE) {
      compute;

try:   flag[i] = TRUE;

      while (flag[(i+1) mod 2]) {
         if (turn == i)
            continue;
         flag[i] = FALSE;
         while (turn != i);
         goto try;
      }

      <critical section>;


turn = (i+1) mod 2;
flag[i] = FALSE;
    }
 }

 turn = 0;
 flag[0] = flag[1] = FALSE;

 Run proc with arg of 0
 Run proc with arg of 1
```

Can you extract the structure of the above program?

Extracted Structure of the Program Code

```
shared int turn;
shared boolean flag[2];

proc (int i) {
   while (TRUE) {
      compute;

try:   flag[i] = TRUE;

      while (flag[(i+1) mod 2]) {
         if (turn == i)
            continue;
         flag[i] = FALSE;
         while (turn != i);
         goto try;
      }
```

```
      <critical section>;

turn = (i+1) mod 2;
flag[i] = FALSE;
   }
}

turn = 0;
flag[0] = flag[1] = FALSE;

Run proc with arg of 0
Run proc with arg of 1
```

Try the same Exercise (now COMMENTED)!

Following solution is alleged to be a solution to the critical section problem. Argue for its correctness or show a case in which it fails.

```
shared int turn;       // Keeps track of whose turn it is
shared boolean flag[2]; // If TRUE, indicates that
                        //   a process would like to enter its c.s.

proc (int i) {
   while (TRUE) {
      compute;
     // Attempt to enter the critical section
   try: flag[i] = TRUE;          // An atomic operation
     // While the other process's flag is TRUE
     while (flag[(i+1) mod 2]){//atomic operation
        if (turn == i)
           continue;
        flag[i] = FALSE; // Reset to let other
// process go
        while (turn != i); // Wait till it's my
// turn
        goto try;
      }
// Okay to enter the critical section
     <critical section>;

     // Leaving the critical section
  turn = (i+1) mod 2;// Set turn to other process
 flag[i] = FALSE; // Indicate no desire to enter
//  my cs
    }
 }

  turn = 0;          // Process 0 wins tie for 1st turn
  flag[0] = flag[1] = FALSE;  // Initialize
// flags before starting

  Run proc with arg of 0 //process 0
  Run proc with arg of 1 //process 1
```

Visualize how this program will run?

Suppose process 1 is in its "compute;" section for an extremely long period of time, such that process 0 has time to execute its critical section several times. Will process 0 have to wait for process 1? Show why or why not. Can you give a schedule for two processes?

Solution to Exercise

The answer is that process 0 will not have to wait for process 1. It can execute its critical section several times while process 1 is in its "compute;" section.

Solution to Exercise

The answer is that process 0 will not have to wait for process 1. It can execute its critical section several times while process 1 is in its "compute;" section.

Assume turn is 0 to begin with. If process 1 is in its compute section, we know that flag[1] is FALSE. Process 0 attempts to enter its critical section, so it sets flag[0] to TRUE. Since flag[1] is FALSE, process 0 doesn't enter the while loop. It executes its critical section, and then sets turn to 1 and flag[0] to FALSE.

Process 0 then returns to the top of the while(TRUE) loop and computes. Then it attempts to enter its critical section (Process 1 is still computing). Process 0 sets flag[0] to TRUE, and again it does not enter the while loop, but moves directly to its critical section code.

Although the turn variable was set to 1 (which indicates that it is the other process's turn, it did not enter the while loop, so it didn't have to test turn==i (which would have failed).

PROCESS SYNCHRONIZATION

Critical Regions

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.

Few Approaches to Achieve Mutual Exclusion

Two categories of solutions –

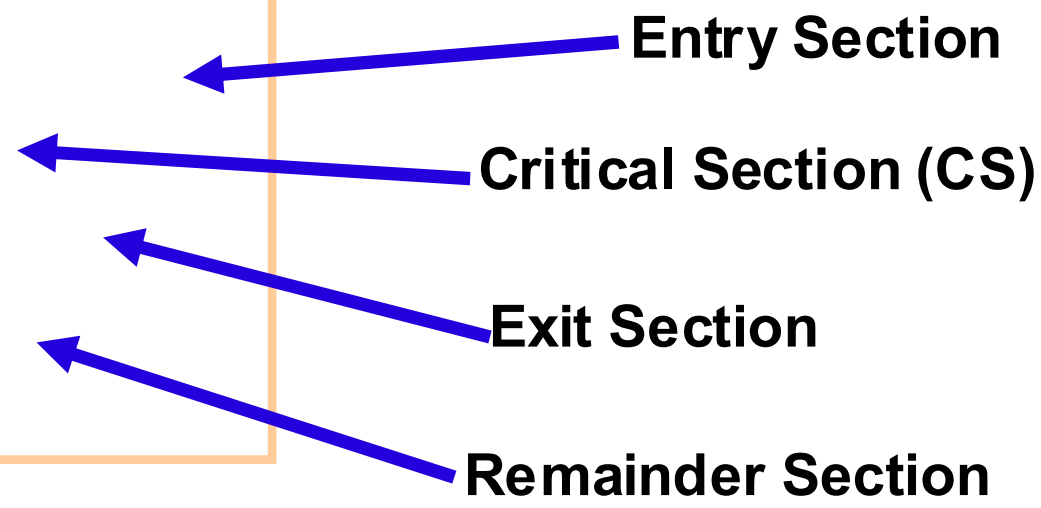
- ***Hardware solution*** - Disabling (DI / EI) interrupts, special instructions like TSL / CMP / SWAP
- ***Software solution*** - Strict alternation, Lock variables, Peterson's solution and few more.

PROCESS SYNCHRONIZATION

Two Process Software

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

```
do {
/* check conditions for CS */
/* critical section  */
/* set conditions for CS */
/* remainder section */
 } while(TRUE);
```



We will now try a succession of increasingly complicated solutions to the problem of creating valid entry / exit sections.

Approach 1:
Using one variable for process turns

Strict Alternation

NOTE: In all examples, **i** is the current process, **j** the "other" process. In these examples, envision the same code running for two processes 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?

Also called Strict Alternation

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Approach 2:
Using one variable indicating interest to enter CR

Lock Variables

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 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]) do no-op;                    /*
critical section */
flag [i] = false;
/* remainder section */
} while (1);
```

Are the three Critical Section Requirements Met?

Principle: Set the flag (LOCK) to enter CR, and Reset the flag after exit

Approach 3:
Using turns and locks together

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

How this is different then previous solutions?

Peterson's Solution to achieve Mutual Exclusion

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Producer Consumer Problem

- Using strict alternation
- Using lock variables
- Using Peterson's solution

Write pseudocode and verify for its correctness.

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 where 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)
```

Producer-Consumer Problem

Assume there are three semaphores mutex, empty, and full.

Write a pseudocode for the producer-consumer problem using semaphore operations.

Analyze your pseudocode.

Analyze semaphore solution to PC problem (HW)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Sync with Semaphore

semaphore sync=0;

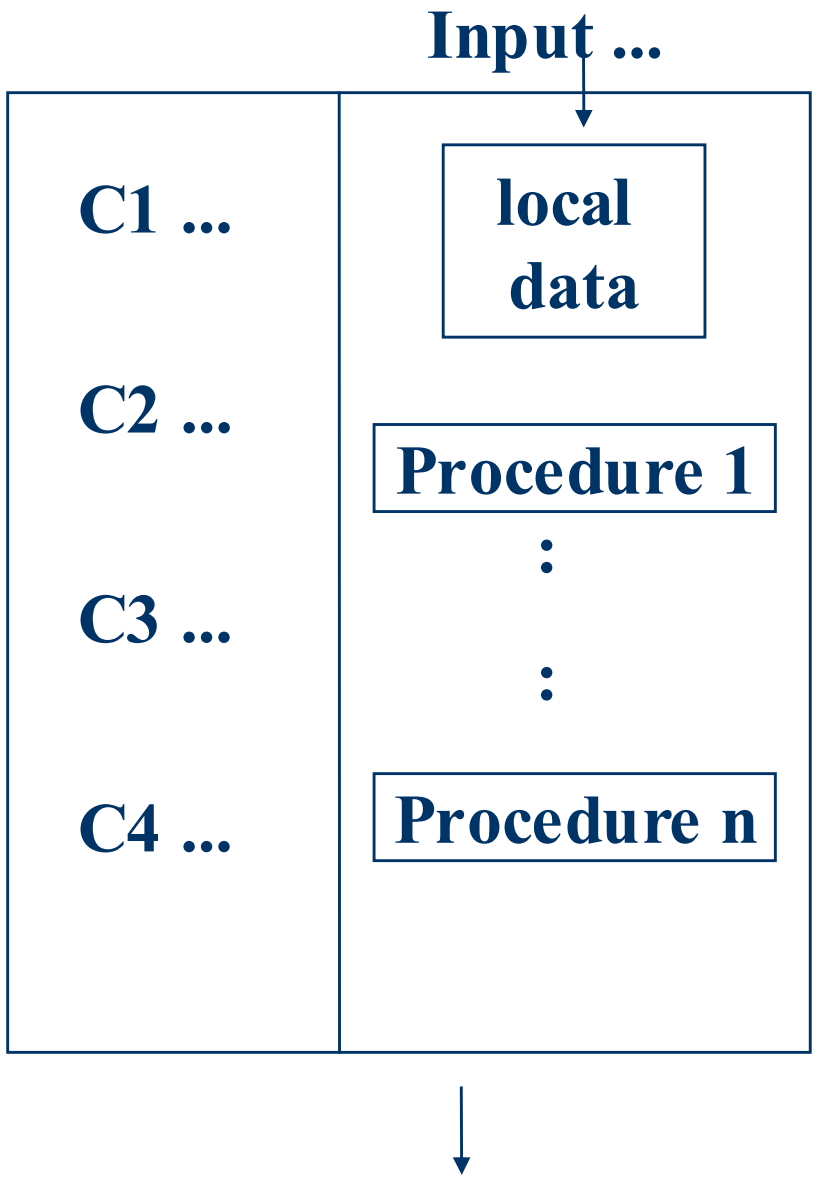
Process 1	Process 2
:	wait (sync);
A	B
signal (sync);	:
:	:

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.

For every wait call, a corresponding signal call exists.
Ordering of wait and signal calls is very important.
A wrong sequence, or a missing/additional wait/signal call could lead to errorenous solution and even to deadlocks.

Monitors

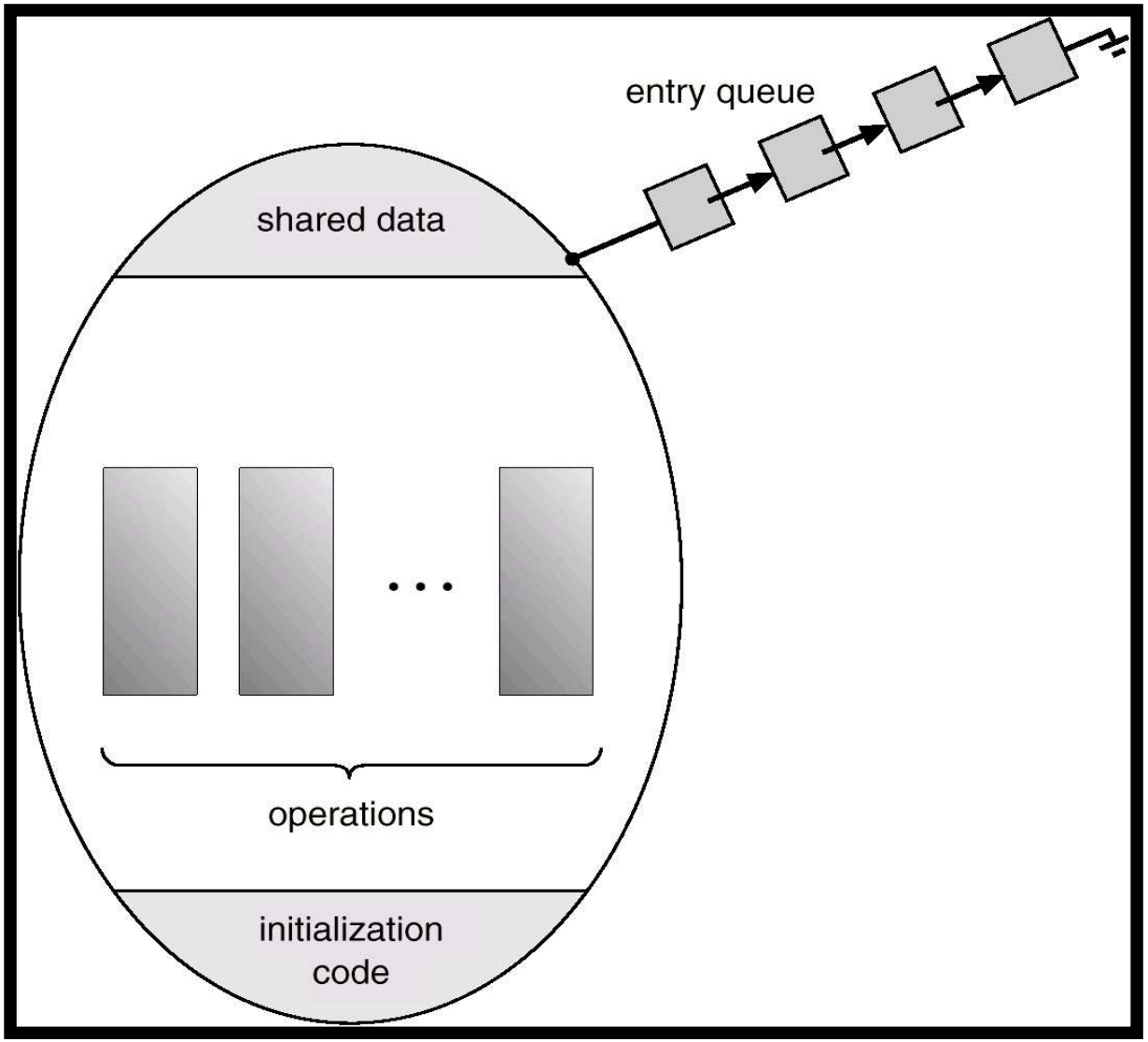
- Another high level sync construct
- Programmer defined operations – which in turn use earlier discussed operations (wait/signal).
- Local data accessible only through monitor procedures.
- Only 1 process can be executing in the monitor at a time.



PROCESS SYNCHRONIZATION

Monitors

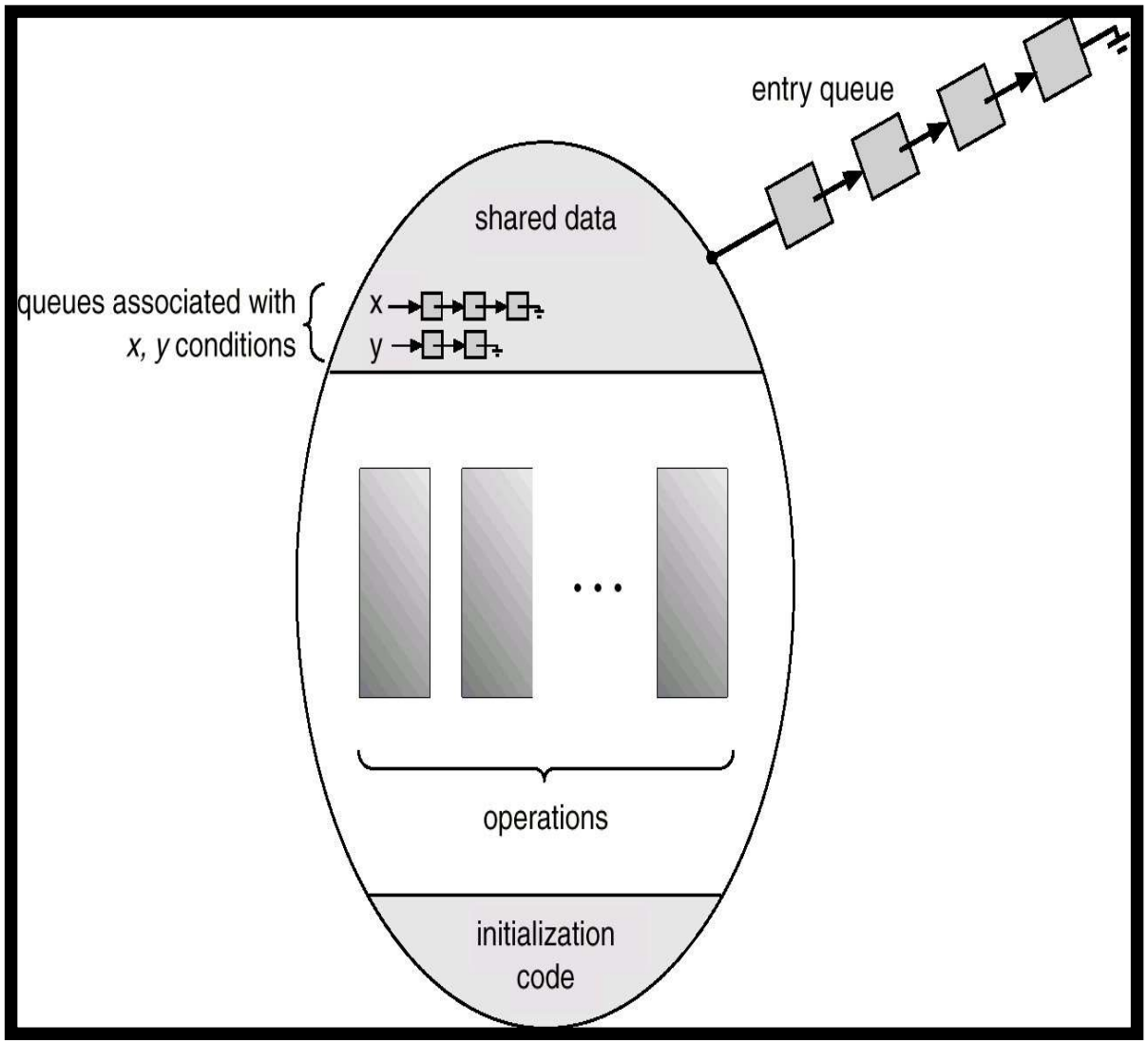
Schematic View of a Monitor



PROCESS SYNCHRONIZATION

Monitors

Monitor With Condition Variables



PROCESS SYNCHRONIZATION

Monitors

High-level synchronization construct that allows the safe sharing of data 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() or wait(x);** means that the process invoking this operation is blocked until another process invokes **x.signal() or signal(x);**
    - The **x.signal()** or **signal(x)** operation resumes exactly one blocked process.
  - If no process is blocked, then the **signal** operation has no effect.

Producer-Consumer Problem

Assume there are two condition variables empty and full to maintain the buffer status.

Write a pseudocode for a monitor program ProducerConsumer with two operations insert() and remove(). A non-condition variable count is used to maintain the item count in the buffer.

Monitors

An outline of the producer-consumer problem with monitors.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Semaphore Deadlock? Yes/No - How?

P

```
wait(B);
wait(A);
:
:
signal(B);
signal(A);
```

Q

```
wait(A);
wait(B);
:
:
signal(A);
signal(B);
```

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

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:

```
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;
```

Analyze and comment on the code

The Readers and Writers Problem ...(1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}
```

. . .

A solution to the readers and writers problem.

The Readers and Writers Problem ...(2)

. . .

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

A solution to the readers and writers problem …cont'd