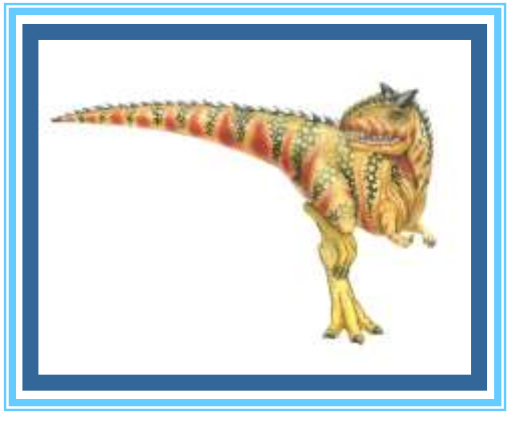
A cooperating process is one that can affect or be affected by other processes executing in the system.

So cooperating process generally share data. However, concurrent access to shared data may result in **data inconsistency.**

In this chapter, we discuss various mechanisms to ensure the **orderly execution** of cooperating processes, so that data consistency is maintained

Chapter 5: Process Synchronization



Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; //the next free position
int out = 0; // the first full position
```

- The buffer is empty when in==out
- The buffer is full when ((in + 1) % BUFFER_SIZE) = = out

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

- ❑ Can this solution solve the asynchronization issues in Pro-Con problem?

- ❑ Can only use at most Buffer_SIZE-1 elements
- ❑ How to use the full BUFFER_SIZE?

New Design - Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

- Suppose that we want to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer ***counter*** that keeps track of the number of full buffers. Initially, ***counter*** is set to 0. ***counter*** is incremented every time producer adds a new item to the buffer and is decremented every time consumer removes one item from the buffer

New Design - Consumer

```
while (true) {
       /* consume the item in next consumed */
       while (counter == 0)
               ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Has used the full BUFFER_SIZE

Can this solution solve the asynchronization issues in Pro-Con problem?

NO! Producer and consumer routines are correct separately, they may **not** function correctly when executed concurrently

Example

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "counter = 5" initially:

T_0: producer execute `register1 = counter` {register1 = 5}
T_1: producer execute `register1 = register1 + 1` {register1 = 6}
T_2: consumer execute `register2 = counter` {register2 = 5}
T_3: consumer execute `register2 = register2 – 1` {register2 = 4}
T_4: producer execute `counter = register1` {counter = 6 }
T_5: consumer execute `counter = register2` {counter = 4}

concurrent execution of counter++ and counter--

What are other potential outputs?

Incorrect !

Example

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

The only correct result is counter ==5

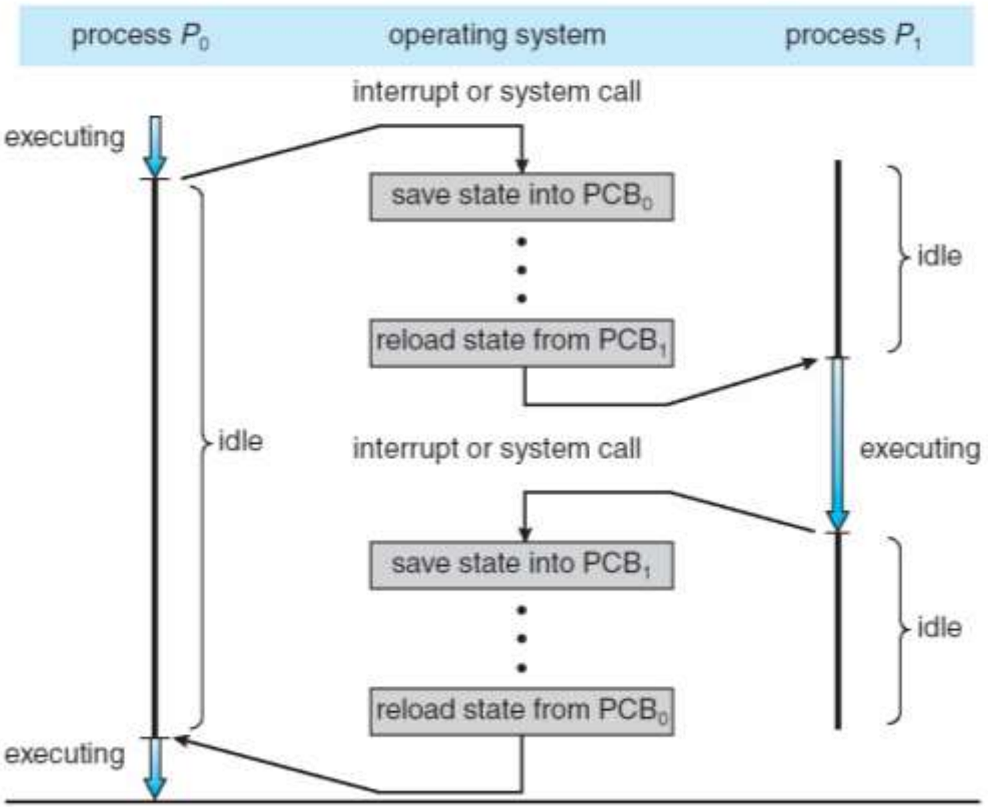
If the producer and consumer execute **separately**, finally counter == 5

But why the previous result is incorrect?

Because we allowed both processes to manipulate the variable counter **concurrently**

Background

- Processes can execute concurrently
  - May be **interrupted** at any time, partially completing execution
- Concurrent access to shared data may result in data **inconsistency**
- Maintaining data consistency requires mechanisms to ensure **the orderly execution of cooperating processes**



Race Condition

- A **race condition** is a situation where **several** processes **access** and **manipulate** the same data **concurrently** and the outcome of the execution depends on the **particular order** in which the access takes place

- Problems often occur when:
  - one process does a "**check-then-act**" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X)
  - and another process does something to the value in between the "check" and the "act".

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

Race Condition

- Problems often occur when:
  - one process does a "**check-then-act**" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X)
  - and another thread does something to the value in between the "check" and the "act".

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another process changed x in between the check and act. You have no real way of knowing.

Race Condition

- Which of the following can cause a race condition?
  - Read-read
  - Write-write
  - Read-write
    - One reader and one writer
    - One reader and multiple writers
    - Multiple readers and one writer

How to guard against the race condition?

Race Condition

To guard against the race condition above, we need to ensure that **only one process at a time can be manipulating the variable counter**. To make such a guarantee, we require that the processes be **synchronized** in some way

Put a **lock** around the **shared data** to ensure only one process can access the data at a time.

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
              // Therefore y = 10
}
// release lock for x
```

Race Condition（Cont.）

- If there is only one producer and one consumer, whether the following solution has race condition?

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Race Condition（Cont.）

- If there is only one producer and one consumer, the solution for the producer-consumer problem does **not** have race condition.
  - The program logic ensures that result will be correct.
  - How about if there are:
    - multiple R-1 W
    - multiple W-1 R
    - multiple R-Multiple W

Critical Section Problem

- **Critical Section**
  - When more than one process access a same code segment that segment is known as critical section
- Critical section contains **shared** variables or resources which are needed to be synchronized to maintain consistency of data variable.

```
P1( )
{
C = B - 1 ;
B = 2 x C ;
}
```

```
P2( )
{
D = 2 x B ;
B = D - 1 ;
}
```

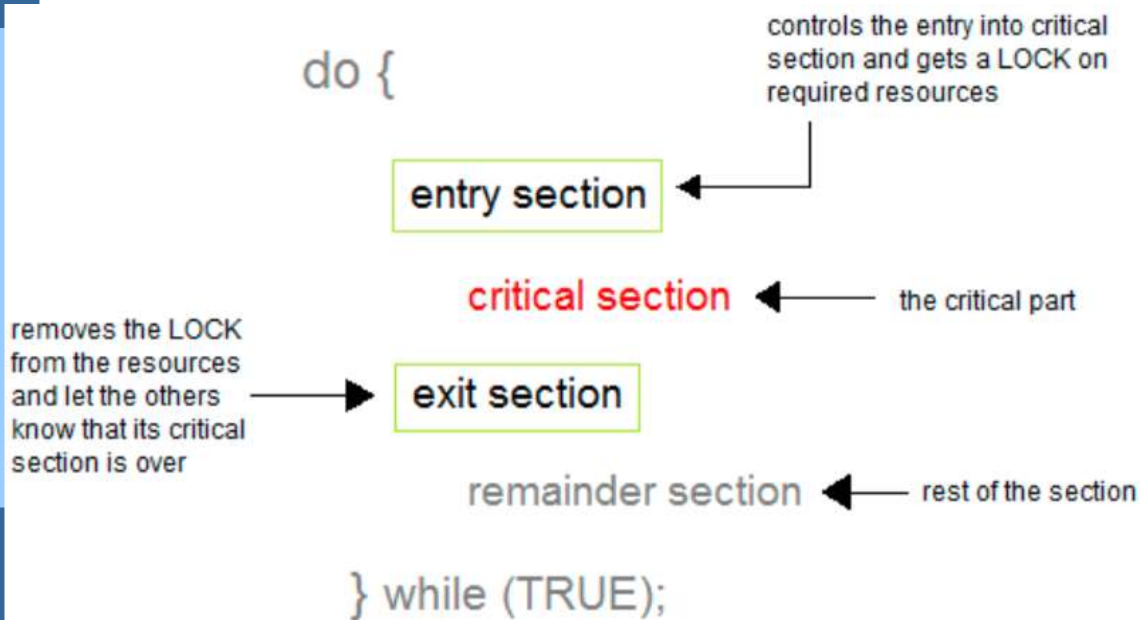
a critical section is group of instructions/statements or region of code that need to be executed **atomically**

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has a segment of code: **critical section**
  - Process may be changing common variables, updating tables, writing files, *etc*
  - When one process is executing in critical section, no other process is allowed to execute in its critical section
- No two processes are executing in their critical sections at the same time
- ***Critical section problem*** is to design a protocol to solve this problem
- Each process must request permission to enter its critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i




```
do {

     while (turn == j);

          critical section

     turn = i;

          remainder section

 } while (true);
```

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements

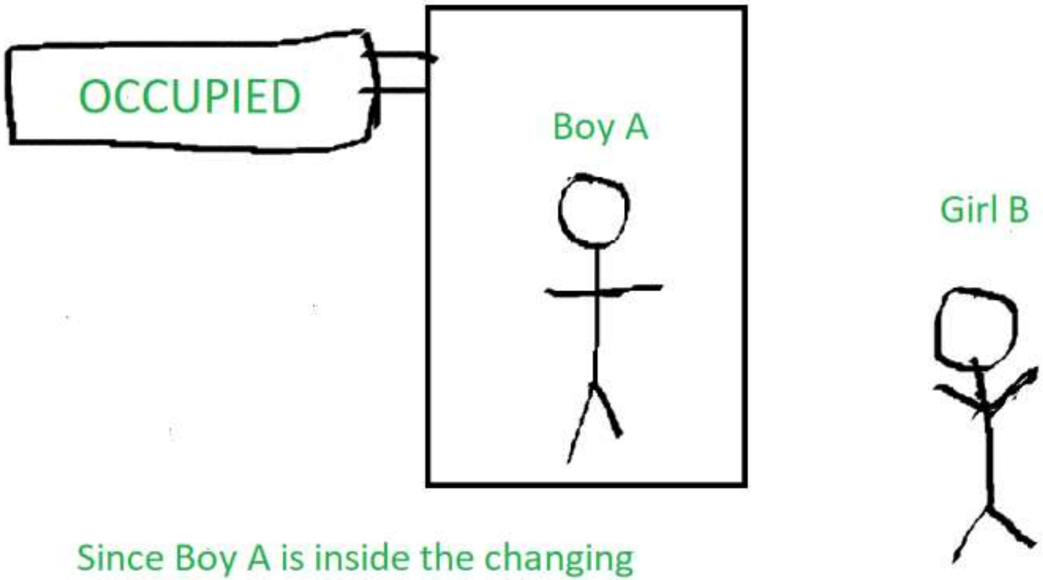
1. Mutual Exclusion

2. Progress

3. Bounded Waiting

Solution to Critical-Section Problem

1. **Mutual Exclusion** - When one process is executing in its critical section, **no** other process is allowed to execute in its critical section.



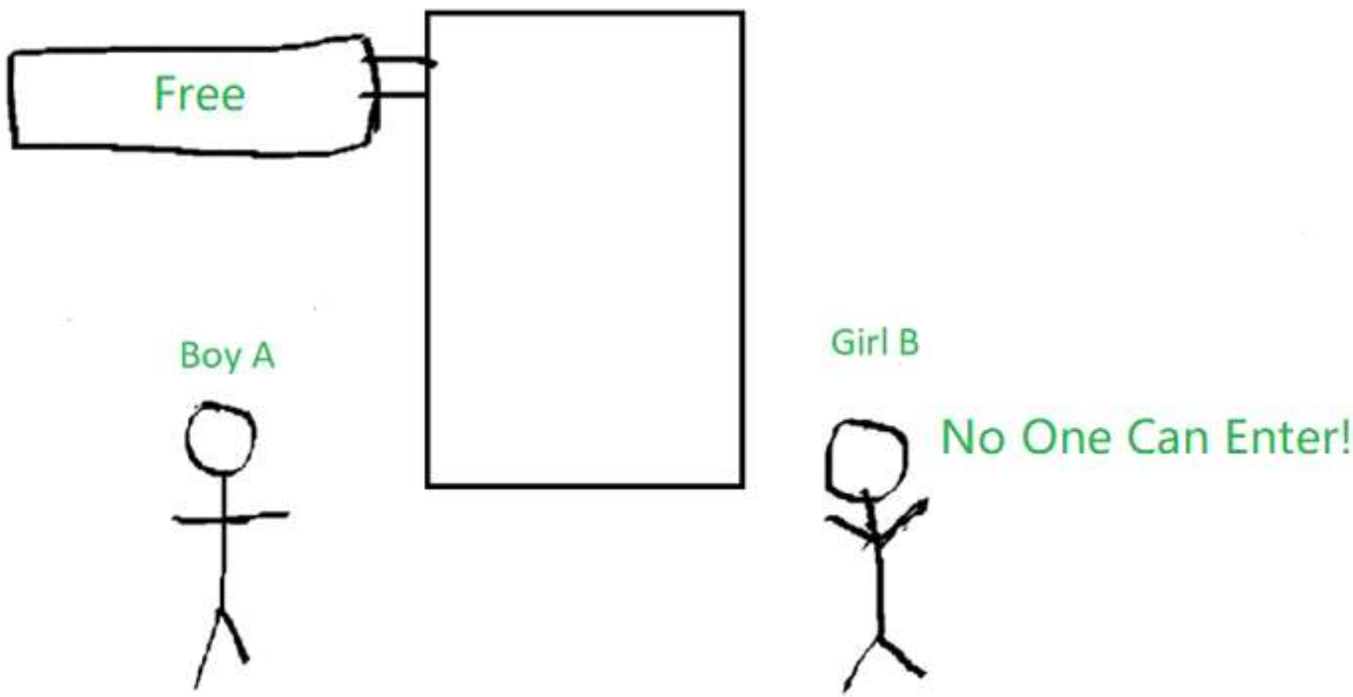
Girl B has to wait outside the changing room till Boy A comes out.

Since Boy A is inside the changing room, the sign on it prevents the others from entering the room.

Is mutual exclusion enough to solve the critical-section problem?

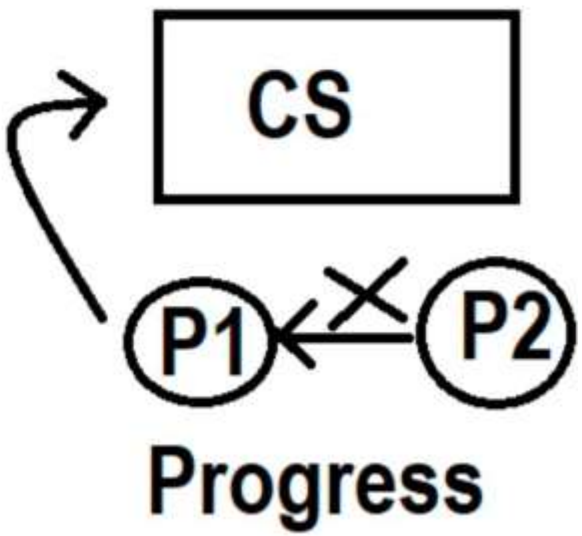
Solution to Critical-Section Problem

Is mutual exclusion enough to solve the critical-section problem?



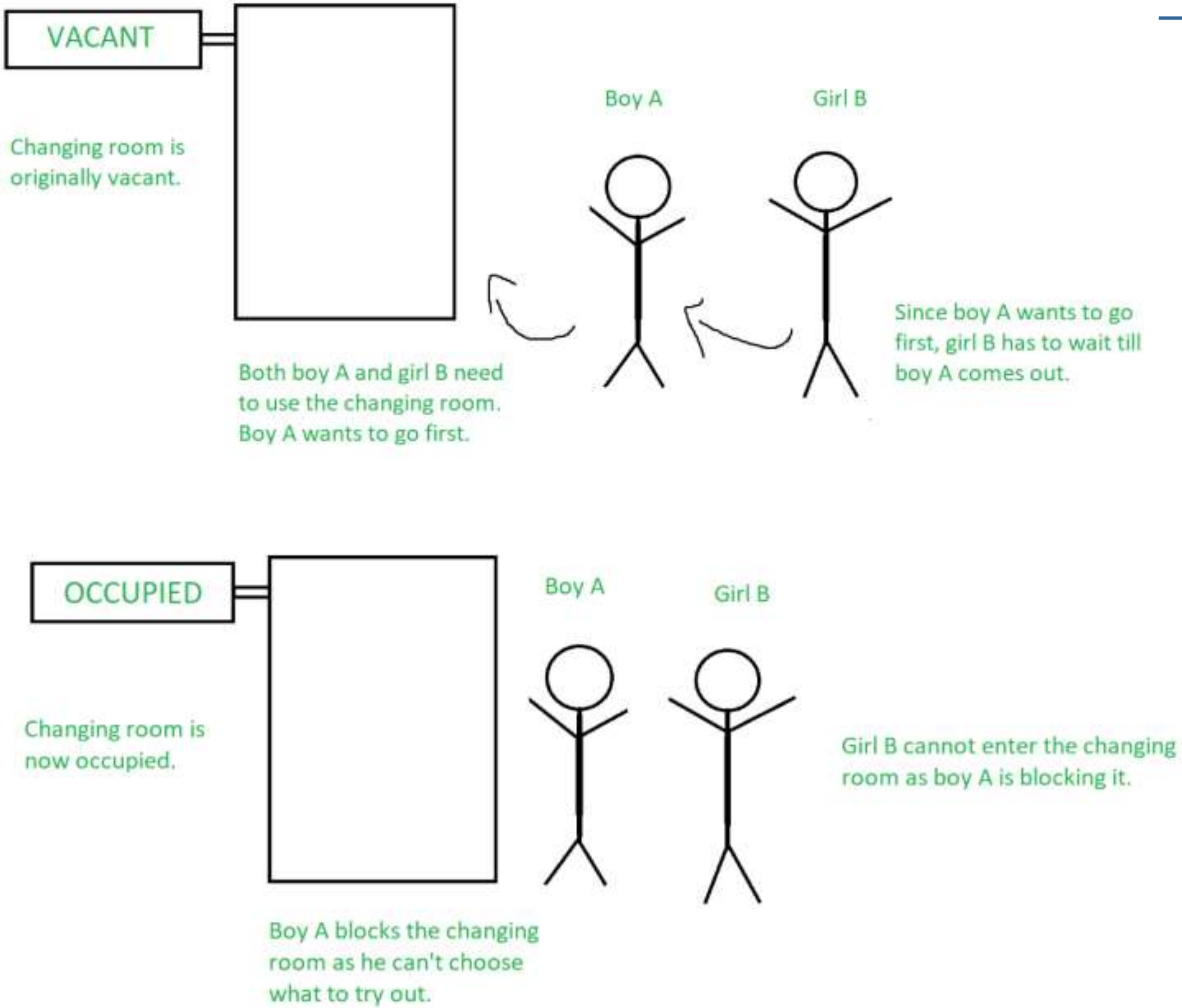
Solution to Critical-Section Problem

2. **Progress** – No process running outside the critical section should block the other interesting process from entering into a critical section when in fact the critical section is free.



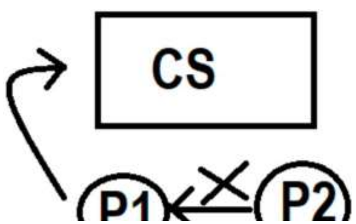
If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then **the selection of the processes that will enter the critical section next cannot be postponed indefinitely**

Solution to Critical-Section Problem

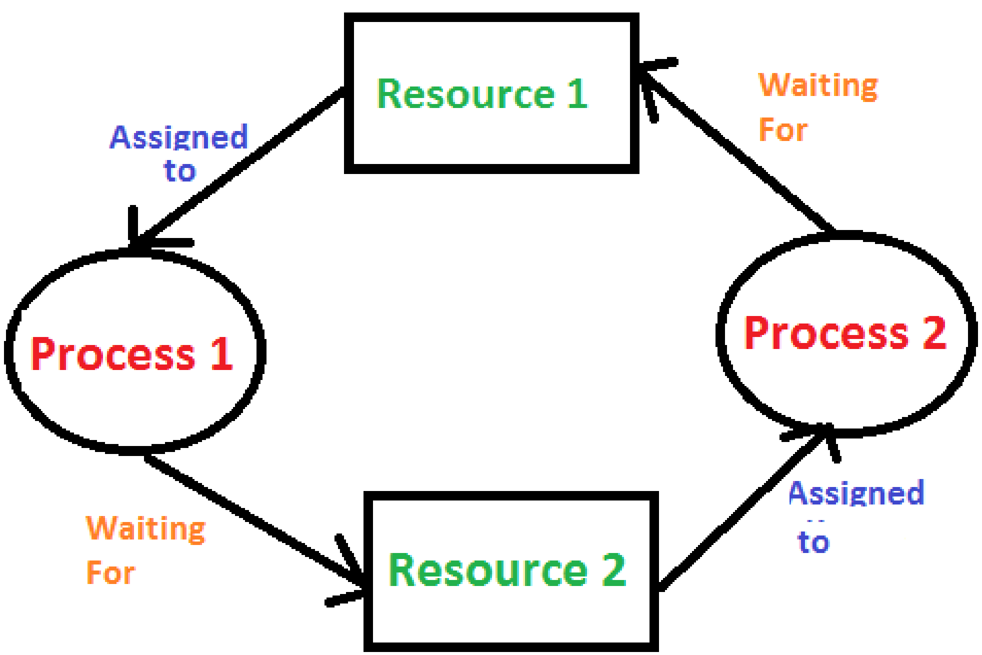


Solution to Critical-Section Problem

2. **Progress** – No process running outside the critical section should block the other interesting process from entering into a critical section when in fact the critical section is free.



If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then **the selection of the processes that will**
tical section next cannot be
ndefinitely



Deadlock!

Deadlock ==> Progress Violation

Solution to Critical-Section Problem

2. **Progress** –

The main job of progress is to ensure one process is executing in the critical section at any point in time (so that some work is always being done by the processor).

This decision cannot be “postponed indefinitely” – in other words, it should take a limited amount of time to select which process should be allowed to enter the critical section. If this decision cannot be taken in a finite time, it leads to a deadlock.

Solution to Critical-Section Problem

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

No process should have to wait forever to enter into the critical section. there should be a boundary on getting chances to enter into the critical section.

Examples of bounded waiting?

quizzes:

Which of the following is not exar
A. Read - Read Conflict
B. Read - Write Conflict
C. Write - Read Conflict
D. Write - Write Conflict

What does atomic instruction mea
A. CPU can context switch only o
B. CPU will not context switch wh
C. CPU will execute these instruct
D. CPU will execute these instruct

Considering the following algorith
satisfied? A
A. Mutual exclusion
B. Progress
C. Bounded waiting
D. None of the above

```
Int thread_num = 1;

thread1 {
 While (true) {
  while(thread_num == 2);
      /* start of critical section */
  ...
  /* end of critical section */
  thread_num == 2
      ...
  }
}

thread2 {
 While (true) {
   while(thread_num == 1);
      /* start of critical section */
  ...
  /* end of critical section */
  thread_num == 1
      ...
  }
}
```

Locks and Unlocks

program 0

```
{
  *
  *
  lock(L)
  counter++
  unlock(L)
  *
}
```

shared variable

```
int counter=5;
lock_t L;
```

program 1

```
{
  *
  *
  lock(L)
  counter--
  unlock(L)
  *
}
```

- lock(L) : acquire lock L **exclusively**
  – Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  – Permitting other processes to access the critical section

How to Implement Locking?

Using Interrupts?

Process 1

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
    other code
}
```

lock

unlock

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
    other code
}
```

- Simple
  - When interrupts are disabled, context switches won’t happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Software Solution (Attempt 1)

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2; // unlock
    other code
}
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

Can this solution satisfy mutual exclusion, progress, or bounded waiting?

- Only satisfies mutual exclusion
- Needs to alternate execution in critical section

process1 → process2 → process1 → process2

Problem with Attempt 1

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
   critical section
   turn = 2; // unlock
   other code
}
```

Process 2

```
while(1){
   while(turn == 1); // lock
   critical section
   turn = 1; // unlock
   other code
}
```

- Had a common **turn** flag that was modified by both processes
- This required processes to alternate.
- Possible Solution: Have **two flags** – one for each process

Review: Software Solution (Attempt 2)

shared

```
p2_inside = False, p1_inside = False
```

Process 1

lock

unlock

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

- Can this solution satisfy mutual exclusion, progress, or bounded waiting?

Both p1 and p2 can enter into the critical section at the same time

Problem with Attempt 2

shared

p2_inside = False, p1_inside = False

Process 1

```
while(1){
    while(p2_inside == True);     <- lock
    p1_inside = True;
    critical section
    p1_inside = False;            <- unlock
    other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

- The flag (p1_inside, p2_inside), is set after we break from the while loop.

Software Solution (Attempt 3)

globally defined

p2_wants_to_enter, p1_wants_to_enter

Process 1

```
while(1){
    p1_wants_to_enter = True
    while(p2_wants_to_enter = True);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock

unlock

Process 2

```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

What’s the drawback of this solution?

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

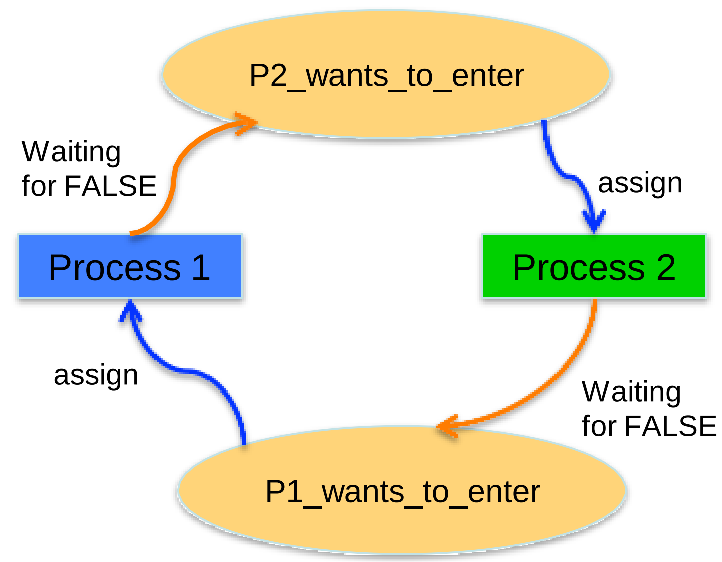
Deadlock

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved
Each process is waiting for the other

this is a deadlock



Review

A solution to the critical section problem must satisfy three conditions:

① Mutual Exclusion: Only one process can be in the critical section at a time -- otherwise what critical section?

② Progress: No process is forced to wait for an available resource -- otherwise very wasteful.

③ Bounded Waiting: No process can wait forever for a resource -- otherwise an easy solution: no one gets in.

Review: example 1

```
/* process i */

  while (true)
  {
    while (turn != i);  /* spin until it's my turn */

    <<< critical section >>>

    turn = j;

    <<< code outside critical section >>>
  }
```

```
/* process j */

  while (true)
  {
    while (turn != j);  /* spin until it's my turn */

    <<< critical section >>>

    turn = i;

    <<< code outside critical section >>>
  }
```

Mutual exclusion?
Progress?
Bounded waiting?

This solution does ensure mutual exclusion, but it is not correct.

The proposed solution violates both the progress criteria and the bounded waiting criteria:

Review: example 1

```
/* process i */

  while (true)
  {
    while (turn != i);  /* spin until it’s my turn */

    <<< critical section >>>

    turn = j;

    <<< code outside critical section >>>
  }
```

```
/* process j */

  while (true)
  {
    while (turn != j);  /* spin until it’s my turn */

    <<< critical section >>>

    turn = i;

    <<< code outside critical section >>>
  }
```

Since the processes are forced strictly into alternating turns, one process could be prevented from accessing an unused resource, simple because it was not its turn.

Consider P0, a process that must be in the critical section 70% of the time, and another process, P1, that must be in the critical section 1% of the time. P1's infrequent use of the critical section will block P0 most of the time (Progress criteria violated)

Review: example 1

```
/* process i */

  while (true)
  {
    while (turn != i);  /* spin until it's my turn */

    <<< critical section >>>

    turn = j;

    <<< code outside critical section >>>
  }
```

```
/* process j */

  while (true)
  {
    while (turn != j);  /* spin until it's my turn */

    <<< critical section >>>

    turn = i;

    <<< code outside critical section >>>
  }
```

Consider also the termination of either process. When a process terminates, one of two things is true: it has the "turn", or it will get the "turn" again soon.

The problem is, if it isn't running, it can't give the "turn" back. When a process ends, it takes its turn with it into an immortal existence. The other process is forever blocked.

Review: example 2

```
/* process i */
  while (true)
  {
    while (state[j]  == inside);  /* is the other one inside? */

    state[i] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[i] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

```
/* process j */
  while (true)
  {
    while (state[i]  == inside);  /* is the other one inside? */

    state[j] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[j] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

Mutual exclusion?
Progress?
Bounded waiting?

Review: example 2

```
/* process i */
  while (true)
  {
    while (state[j]  == inside);  /* is the other one inside? */

    state[i] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[i] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

Initially: state[i] = outside
state[j] = outside

Pi Pass the while loop, enter

```
/* process j */
  while (true)
  {
    while (state[i]  == inside);  /* is the other one inside? */

    state[j] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[j] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

Pj Pass the while loop, enter

Review: example 2

```
/* process i */
  while (true)
  {
    while (state[j]  == inside);  /* is the other one inside? */

    state[i] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[i] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

Initially: state[i] = outside
state[j] = outside

Pi sets state[i] = inside

And enter CS

```
/* process j */
  while (true)
  {
    while (state[i]  == inside);  /* is the other one inside? */

    state[j] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[j] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

Pj sets state[j] = inside

And enter CS

Review: example 2

```
/* process i */
  while (true)
  {
    while (state[j]  == inside);  /* is the other one inside? */

    state[i] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[i] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

```
/* process j */
  while (true)
  {
    while (state[i]  == inside);  /* is the other one inside? */

    state[j] = inside;  /* get in and flip state */

    <<< critical section >>>

    state[j] = outside;  /* revert state */

    <<< code outside critical section >>>
  }
```

How to make some changes to ensure the three properties be satisfied?

Initially: state[i] = outside
state[j] = outside

1. Pi Pass the while loop, enter.
2. And Pi sets state[i] = inside
3. And enter CS

Pj stops at the while loop, only after pi set
state[i] = outside

Review: example 3

```
/* process i */
while (true)
  {
    state[i] = interested;  /* declare interest */

    while (state[j]  == interested);  /* stay clear till safe */

    <<< critical section >>>

    state[i] = notinterested;  /* we're done */

    <<< code outside critical section >>>
  }
```

Mutual exclusion?
Progress?
Bounded waiting?

```
/* process j */
while (true)
  {
    state[j] = interested;  /* declare interest */

    while (state[i]  == interested);  /* stay clear till safe */

    <<< critical section >>>

    state[j] = notinterested;  /* we're done */

    <<< code outside critical section >>>
  }
```

deadlock

Peterson's Solution

- Good algorithmic description of solving the problem
- **Two processes**' solution
- The two processes share two variables:

```
int turn;
Boolean flag[2]
```

- The variable ***turn*** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

software-based solution

Algorithm in Peterson's Solution

If process j wants to enter. Give turn to it. (be nice !!!)

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

turn is used to break the tie when both p1 and p2 want to enter the critical section.

Algorithm in Peterson's Solution

Does Peterson's solution require two processes to alternate execute in critical section?

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

NO!

Algorithm in Peterson's Solution

Does Peterson's solution have deadlock?

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

- Deadlock broken because turn can only be i or j.
  - Therefore, tie is broken. Only one process will enter the critical section
- Solves Critical Section problem for two processes

Algorithm in Peterson's Solution

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Try to show the running procedure of Pi and Pj in a concurrent manner!

Peterson's Solution: Pi runs first, 1 instruction/each

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Flag[i] = true

turn = j

Since the while condition is not satisfied, pi will enter critical section

Flag[j] = true

turn = i (overwrite, now **turn = i**)

Since the while condition is satisfied, pj will stuck in the while loop, thus cannot enter critical section

Peterson’s Solution: Pj runs first, 1 instruction/each

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Flag[i] = true

turn = j (overwrite, now **turn = j**)

Since the while condition is satisfied, pi will stuck in the while loop, thus cannot enter critical section

Flag[j] = true

turn = i

Since the while condition is not satisfied, pj will enter critical section

Peterson's Solution: Pi runs first, 2 instructions/each

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Flag[i] = true

turn = j

Since the while condition is not satisfied, pi will enter critical section

Flag[j] = true

turn = i (overwrite, now **turn = i**)

Since the while condition is satisfied, pj will stuck in the while loop, thus cannot enter critical section

Peterson's Solution: Pj runs first, 2 instructions/each

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Flag[i] = true

turn = j (overwrite, now **turn = j**)

Since the while condition is satisfied, pi will stuck in the while loop, thus cannot enter critical section

Flag[j] = true

turn = i

Since the while condition is not satisfied, pj will enter critical section

Peterson's Solution (Cont.)

- Prove that the three critical section requirement are met:

1. Claim: Mutual exclusion is preserved

Proof: prove the above claim by **contradiction**

If mutual exclusion is not preserved, then P_i and P_j can enter their critical section **simultaneously**:

`Pi` enters critical section only if:

either **`flag[j] = false`** or **`turn = i`**

`Pj` enters critical section only if:

either **`flag[i] = false`** or **`turn = j`**

Then this two conditions must be satisfied simultaneously

`turn = i = j`

this is a contradiction!!!

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Peterson's Solution (Cont.)

- Provable that the three critical section requirement are met:

2. Claim: **Progress** is preserved

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Peterson's Solution (Cont.)

- Provable that the three critical section requirement are met:

3. Claim: **Bounded-waiting** is preserved

```
//proces i:

flag[i] = true;
turn = j;
while(flag[j] == true && turn==j);

<critical section>

flag[i] = false;

<remainder section
```

```
//proces j:

flag[j] = true;
turn = i;
 while(flag[i] == true && turn == i);

<critical section>

flag[j] = false;

<remainder section>
```

Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport

Eat when 196 displayed

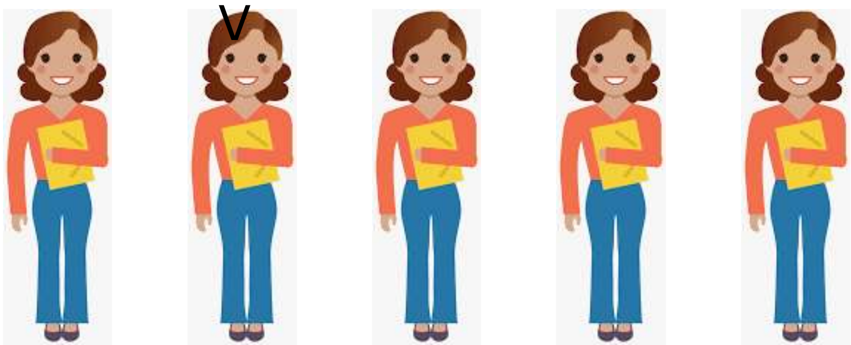
wait your turn!!

http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport

A B C D E
V



2 3 6 7 9

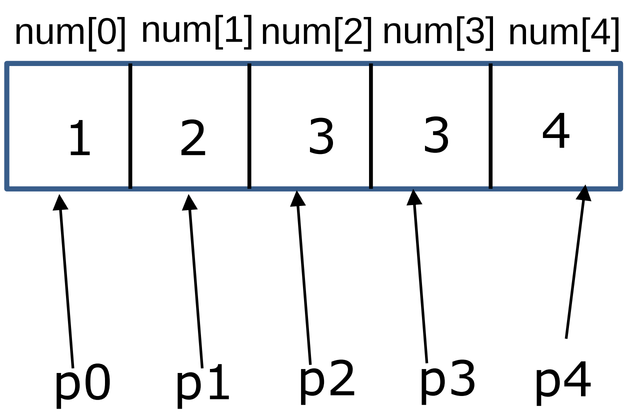
A will be served first

D will be served until A, B and C have been served

First come, first served

Simplified Bakery Algorithm

- Processes numbered 0, 1 … N-1
- Each process i has an integer variable num[i], initially 0
- num is an array N integers (initially 0).



N = 5

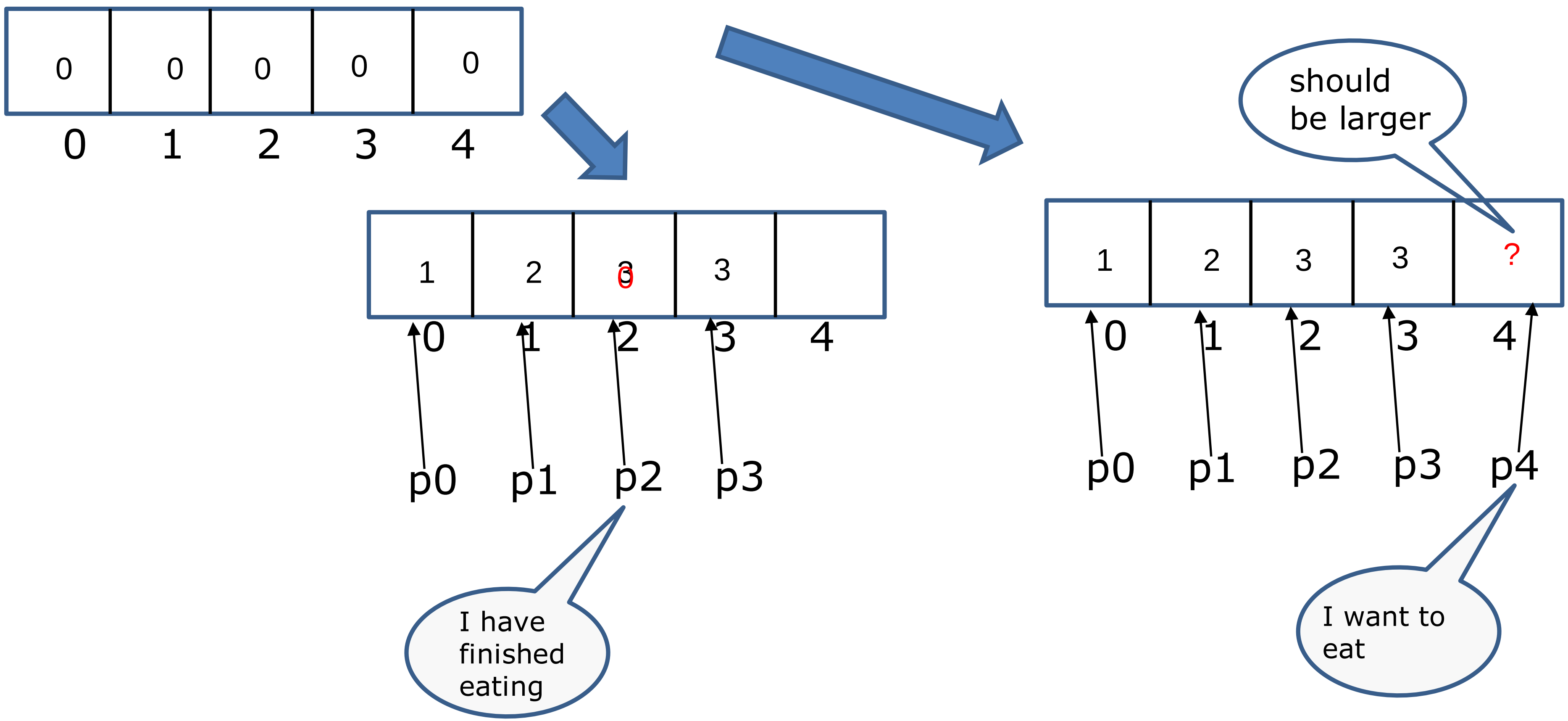
The numbering scheme always generates numbers in **increasing order** of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5, …

Holder of the smallest number enters the critical section first

If processes Pi and Pj receive the same number

if i < j
    Pi is served first;
else
    Pj is served first.

Simplified Bakery Algorithm



Intuitively, when process p4 want to enter its critical section, it should set num[4] to a value **higher than** the num of every other process, i.e., **num[4] = max(num[0], num[1], num[2], num[3])+1**;

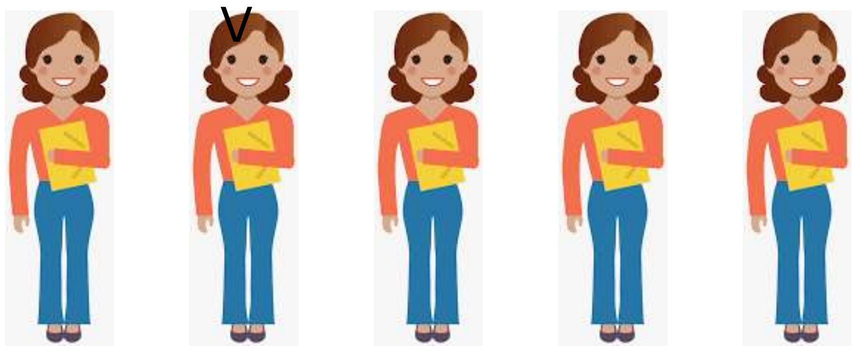
Simplified Bakery Algorithm

How to let p4 eat a bread:

1)p4 must wait until p0, p1, p2 and p3:
a)have finished eating, i.e., **num[0]=0; num[1]=0; num[2]=0; num[3]=0**
b)OR p0, p1, p2 and p3's num is smaller the num[p4]

A B C D E

V



A	B	C	D	E
2 0	3 0	6 0	7	9
has finished eating	has finished eating	has finished eating	Can Eat Now	

Simplified Bakery Algorithm

How to let p4 eat a bread:

1)p4 must wait until p0, p1, p2 and p3:
a)have finished eating, i.e., **num[0]=0; num[1]=0; num[2]=0; num[3]=0**
b)OR p0, p1, p2 and p3's num is smaller the num[p4]

```
num[4] = max(num[0], num[1], num[2], num[3])+1;

for(k = 0; k < N; k++)
{
        while (num[k] != 0 && num[k] < num[p4]);
}
```

Simplified Bakery Algorithm

How to let p4 eat a bread:

1)p4 must wait until p0, p1, p2 and p3:
a)have finished eating, i.e., **num[0]=0; num[1]=0; num[2]=0; num[3]=0 ,**
b)OR p0, p1, p2 and p3's num is smaller the num[p4]

```
lock(i){
    num[i] = MAX(num[0], num[1], ...., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

What's the main drawback of simplified bakery algorithm?

Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False

    for(p = 0; p < N; ++p)
    {
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

critical section

```
unlock(i){
    num[i] = 0;
}
```

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False

    for(p = 0; p < N; ++p)
    {
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

It is actually a small critical section in itself !

The very purpose of the first three lines is that if a process is modifying its TICKET value, then at that time some other process should not be allowed to check its old ticket value which is now obsolete.

This is why inside the for loop before checking ticket value we first make sure that all other processes have the "choosing" variable as FALSE.

Review 2

Use the notation k.tmpX to designate local storage of the thread k

```
initially: number[1] = number[2] = 0

thread 1                                   thread 2
//choosing[1]=true
1.tmp1 = number[1]
1.tmp2 = number[2]
                                           //choosing[2]=true
                                           2.tmp1 = number[1]
                                           2.tmp2 = number[2]
                                           number[2] = max(2.tmp1,2.tmp2)+1 = 1
                                           //choosing[2]=false
                                           //while (choosing[1]) {}
                                           while (number[1]≠0 &&
                                             (number[1],1)<(number[2],2)) {}
                                           critical section ...
number[1] = max(1.tmp1,1.tmp2)+1 = 1
//choosing[1]=false
//while (choosing[2]) {}
while (number[2]≠0 &&
       (number[2],2)<(number[1],1)) {}
critical section ...
                                           critical section ...
```

number[1] here is an obsolete value!!!.

The choosing-involved instructions have been removed, any problem?

Review 2

Use the notation k.tmpX to designate local storage of the thread k

```
initially: number[1] = number[2] = 0

thread 1                                      thread 2
choosing[1]=true
1.tmp1 = number[1]
1.tmp2 = number[2]
                                              choosing[2]=true
                                              2.tmp1 = number[1]
                                              2.tmp2 = number[2]
                                              number[2] = max(2.tmp1,2.tmp2)+1 = 1
                                              choosing[2]=false
                                              while (choosing[1]) {}
                                              while (number[1]≠0 &&
                                                     (number[1],1)<(number[2],2)) {}
                                              critical section ...
number[1] = max(1.tmp1,1.tmp2)+1 = 1
choosing[1]=false
while (choosing[2]) {}
while (number[2]≠0 &&
       (number[2],2)<(number[1],1)) {}
critical section ...
                                              critical section ...
```

Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False

    for(p = 0; p < N; ++p)
    {
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

When pi compares it num with all current N processes, several new processes get new tokens, but pi only compare its num with these N processes, is it enough?

critical section

```
unlock(i){
    num[i] = 0;
}
```

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

Bakery Algorithm

- Prove that the three critical section requirements are met by Bakery Algorithm:

1. **Mutual exclusion** is satisfied. why?
2. **Progress** requirement is satisfied. why?
3. **Bounded-waiting** requirement is met. why?

Bakery Algorithm

Claim: **Mutual exclusion** is preserved

Proof: prove the above claim by **induction**

1. Assume there are only **two** processes P_i and P_j and they can enter their critical sections **simultaneously**, **i < j**

   1) **if num[P_i]=0, num[P_j]≠0, then Pj enter**

   2) **if num[P_i]≠0, num[P_j]=0, then Pi enter**

   3) **if num[P_i]=0, num[P_j]=0, then neither Pi nor Pj enter**

   4) **if num[P_i]≠0, num[P_j]≠0, num[P_i]<num[P_j] then Pi enter**

   5) **if num[P_i]≠0, num[P_j]≠0, num[P_j]<num[P_i] then Pj enter**

   6) **if num[P_i]≠0, num[P_j]≠0, num[P_i]=num[P_j] then Pi enter**

therefore, for only two processes, the above claim holds

Bakery Algorithm

Claim: **Mutual exclusion** is preserved

Proof: prove the above claim by **induction**

2. Assume there are M processes and **at least** two processes P_i and P_j can enter their critical sections **simultaneously**, **i < j**

```
1)if Pi can enter its critical section now:
  a)mum[Pi]≠0;
  b)For any other process Pk, num[Pk]≠0, num[Pi]<num[Pk];
  c)Or for any other process Pk, num[Pk]=0.

2) if Pj can enter its critical section now:
  a)mum[Pj]≠0;
  b)For any other process Pk, num[Pk]≠0, num[Pj]<num[Pk];
  c)Or for any other process Pk, num[Pk]=0.
```

Contradiction!!!

Thus, for any M processes, mutual exclusion is preserved

Bakery Algorithm

Claim: **Mutual exclusion** is preserved

Proof: prove the above claim by **induction**

1. Assume there are only **two** processes P_i and P_j and they can enter their critical sections **simultaneously**, **$i < j$**
2. Assume there are M processes and **at least** two processes P_i and P_j can enter their critical sections **simultaneously**, **$i < j$**
3. **For any n processes, mutual exclusion is preserved**

Bakery Algorithm

Claim: **Progress** is preserved

Proof: prove the above claim by **induction**:

1. Assume there are only **two** processes P_i and P_j, **$i < j$**

   1) When Pi gets stuck in the while loop, then

   num[Pj]!=0 and num[Pj]<num[Pi]

   2) When Pj gets stuck in the while loop, then

   num[Pi]!=0 and num[Pi]<num[Pj]

Contradiction!!!

Thus, for only 2 processes, progress is preserved

Do same analysis for N processes

Bakery Algorithm

Claim: **Progress** is preserved

Proof: prove the above claim by **induction**

2. Assume there are M processes, and **all M** processes get stuck in the while loop, then for process P0

1)When P0 gets stuck in the while loop, then there must exit at least 1 process Pk:

```
num[Pk]!=0 and num[Pk]<num[P0]
```

Contradiction!!!

Thus, for any M processes, progress is preserved

Thus, for any n processes, progress is preserved

Review for Progress and Bounded Waiting

A more in-depth discussion

Progress

- **If *no* process is executing in its critical section and some processes want to enter their corresponding critical sections, then**
  1. **Only those processes that are waiting to enter can participate in the competition (to enter their critical sections) and no other processes can influence this decision.**
  2. **This decision cannot be postponed indefinitely (i.e., finite decision time). Thus, one of the waiting processes can enter its critical section.**

Bounded Waiting

- **After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a *bound* on the number of turns that other processes are allowed to enter.**
- ***Finite is not the same as bounded*. The former means any value you can write down (e.g., billion, trillion, etc) while the latter means this value has to be no larger than a particular one (i.e., the bound).**

Progress vs. Bounded Waiting

- ***Progress*** **does not imply** ***Bounded Waiting***: ***Progress*** **says a process can enter with a finite decision time. It does not say which process can enter, and there is no guarantee for bounded waiting.**
- ***Bounded Waiting*** **does not imply** ***Progress***: **Even through we have a bound, all processes may be locked up in the enter section (i.e., failure of** ***Progress*****).**
- **Therefore,** ***Progress*** **and** ***Bounded Waiting*** **are independent of each other.**

A Few Related Terms:

- ***Deadlock-Freedom*: If two or more processes are trying to enter their critical sections, one of them will eventually enter. This is *Progress* without the “outsiders having no influence” condition.**
- **Since the enter section is able to select a process to enter, the decision time is certainly finite.**

A Few Related Terms:

- ***r-Bounded Waiting*****: There exists a fixed value *r* such that after a process made a request to enter its critical section and before it is granted the permission to enter, no more than *r* other processes are allowed to enter.**
- **Therefore, bounded waiting means there is a *r* such that the waiting is *r*-bounded.**

A Few Related Terms:

- ***Starvation-Freedom***: **If a process is trying to enter its critical section, it will eventually enter.**
- ***Questions***:
  1. **Does starvation-freedom imply deadlock-freedom?**
  2. **Does starvation-freedom imply bounded-waiting?**
  3. **Does bounded-waiting imply starvation-freedom?**
  4. **Does bounded-waiting *AND* deadlock-freedom imply starvation-freedom?**

A Few Related Terms:

- ***Question (1)***: **Does starvation-freedom imply deadlock-freedom?**
- ***Yes!*** **If every process can eventually enter its critical section, although waiting time may vary, it means the decision time of selecting a process is finite. Otherwise, all processes would wait in the enter section.**

A Few Related Terms:

- ***Question (2)***: **Does starvation-freedom imply bounded-waiting?**
- ***No!*** **This is because the waiting time may not be bounded even though each process can enter its critical section.**

A Few Related Terms:

- ***Question (3)***: **Does bounded-waiting imply starvation-freedom?**
- ***No.*** **Bounded-Waiting does not say if a process can actually enter. It only says there is a bound. For example, all processes are locked up in the enter section (i.e., failure of *Progress*).**
- **We need *Progress* + *Bounded-Waiting* to imply *Starvation-Freedom* (*Question (4)*). In fact, *Progress* + *Bounded-Waiting* is stronger than *Starvation-Freedom*.**

Why?

Dekker's solution

```
int turn;
Boolean flag[2];
```

```
/* process i */

flag[i] = true;
while (flag[j] = true)
{

    if (turn == j)
    {
        flag[i] = false;
        while (turn  == j);
        flag[i] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = j;
flag[i] = false;
```

```
/* process j */

flag[j] = true;
while (flag[i] = true)
{

    if (turn == i)
    {
        flag[j] = false;
        while (turn  == i);
        flag[j] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = i;
flag[j] = false;
```

Dekker's solution

int turn;
Boolean flag[2];

How about **mutual exclusion**?
if process j is in its C.S.

```
/* process i */

flag[i] = true;
while (flag[j] = true)
{

      if (turn == j)
      {
            flag[i] = false;
            while (turn  == j);
            flag[i] = true;
      }
}

/* enter Critical Section */
/* exit Critical Section */

turn = j;
flag[i] = false;
```

now:

flag[j] = true

process i stuck in the while loop.

```
/* process j */

flag[j] = true;
while (flag[i] = true)
{

      if (turn == i)
      {
            flag[j] = false;
            while (turn  == i);
            flag[j] = true;
      }
}

/* enter Critical Section */
/* exit Critical Section */

turn = i;
flag[j] = false;
```

Dekker's solution

int turn;
Boolean flag[2];

How about **mutual exclusion**?

```
/* process i */

flag[i] = true;
while (flag[j] = true)
{

    if (turn == j)
    {
        flag[i] = false;
        while (turn  == j);
        flag[i] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = j;
flag[i] = false;
```

flag[i] = true

flag[j] = true

turn can only be i or j!

Suppose turn ==j

flag[i] = false

Jump out from the while loop

```
/* process j */

flag[j] = true;
while (flag[i] = true)
{

    if (turn == i)
    {
        flag[j] = false;
        while (turn  == i);
        flag[j] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = i;
flag[j] = false;
```

Dekker's solution

int turn;
Boolean flag[2];

How about **progress**?

flag[i] = true

flag[j] = true

```
/* process i */

flag[i] = true;
while (flag[j] = true)
{

    if (turn == j)
    {
        flag[i] = false;
        while (turn  == j);
        flag[i] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = j;
flag[i] = false;
```

turn can only be i or j!

so i or j must enter the C.S.

```
/* process j */

flag[j] = true;
while (flag[i] = true)
{

    if (turn == i)
    {
        flag[j] = false;
        while (turn  == i);
        flag[j] = true;
    }
}

/* enter Critical Section */
/* exit Critical Section */

turn = i;
flag[j] = false;
```

How to Implement Locking

(Hardware Solutions and Usage)

Analyze this

- Does this scheme provide mutual exclusion?

lock=0

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No

lock = 0
P1: while(lock != 0);
P2: while(lock != 0); P2: lock = 1;
P1: lock = 1; …. Both processes in critical section

context switch

How to make some changes such that it provides mutual exclusion?

If only…

- We could make this operation atomic

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Make atomic

Hardware to the rescue….

Synchronization Hardware

1. The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variables was being modified

2. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by non-preemptive kernels

3. Disabling interrupts on a multiprocessor can be time consuming

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks

```
do {
        acquire lock
              critical section
        release lock
              remainder section
} while (TRUE);
```

- Single-processor – could disable interrupts
  - Currently running code would execute without preemption
  - Generally, too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Synchronization Hardware

- Modern machines provide special **atomic hardware instructions**
  - **Atomic** = as one non-interruptible unit

- **test_and_set()**
- **compare_and_swap()**

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

return the given value

sets the given memory address to 1

These two steps are executed **atomically:**

1) **Returns the original value of passed parameter**

2) **Set the new value of passed parameter to "TRUE".**

If two test_and_set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially some arbitrary order.

Solution using test_and_set()

- Shared Boolean variable **lock**, initialized to **FALSE**
- Solution:

"false" represents resource is free;
"true" represents resource is occupied by another process

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
} while (true);
```

acquire lock

release lock

While a process finishes its critical section and sets the lock to be false, another process may call the test_and_set() and sets the lock to be true. Why don't have a race condition here?

- The test_and_set() is atomic!

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;

    return temp;
}
```

1. Executed atomically
2. **Returns the original value of passed parameter "value"**
3. The operand ***value*** is set to ***new_value*** but only if "*value == expected*". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

"0" represents "false", the resource is free; "1" represents "true", the resource is "not free"

acquire lock

release lock

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
       /* critical section */
   lock = 0;
       /* remainder section */
} while (true);
```

1. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0.
2. Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0.

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    while (waiting[i] && test_and_set(&lock)) ;
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the **bounded-waiting** requirement.

When a process leaves a critical section, it **scans the array** waiting in the cyclic ordering.

It **designates the first process in this ordering as the next one** to enter the critical section

Mutex Locks

- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - **Boolean variable** indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions

Mutex Locks

Process 1

acquire(&locked)
critical section
release(&locked)

Process 2

acquire(&locked)
critical section
release(&locked)

```
do {
      acquire lock
            critical section
      release lock
            remainder section
} while (true);
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

Mutex Locks

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

1. A mutex lock has a boolean variable ***available*** whose value indicates if the lock is available or not

2. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.

3. A process that attempts to acquire an unavailable lock is blocked until the lock is released

4. Calls to either acquire() or release() must be performed atomically

acquire() and release()

- ```
  acquire() {
       while (!available)
           ; /* busy wait */
       available = false;
  }
  ```

- ```
  release() {
       available = true;
  }
  ```

- ```
  do {
      acquire lock
           critical section
      release lock
           remainder section
  } while (true);
  ```

available: whose value indicates if the lock is available or not
  ```

acquire() and release()

```
int N;                   // SHARED variable
pthread_mutex_t N_mutex;  // Mutex controlling access to N

void *worker(void *arg)
{
  int i;

  for (i = 0; i < 10000; i = i + 1)
  {
     pthread_mutex_lock(&N_mutex);
     N = N + 1;
     pthread_mutex_unlock(&N_mutex);
  }
}
```

Although many thread are executing simultaneously, the statement ***pthread_mutex_lock(&N_mutex);*** ensures that ***exactly ONE thread*** is successful in locking the mutex variable ***N_mutex***.

This particular thread will then be the only thread that will update the variable ***N***, thus ensuring that ***N*** is ***updated sequential (one thread after another)***

acquire() and release()

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long count;

void increment_count()
{
        pthread_mutex_lock(&count_mutex);
        count = count + 1;
        pthread_mutex_unlock(&count_mutex);
}

long get_count()
{
        long c;
        pthread_mutex_lock(&count_mutex);
        c = count;
        pthread_mutex_unlock(&count_mutex);
        return (c);
}
```

Mutex Locks (cont.)

- ```
  acquire() {
       while (!available)
          ; /* busy wait */
       available = false;
  }
  ```

- ```
  release() {
       available = true;
  }
  ```

- ```
  do {
       acquire lock
          critical section
       release lock
          remainder section
  } while (true);
  ```

What's the advantage and disadvantage of mutex locks?
  ```

Mutex Locks (cont.)

- This solution requires **busy waiting**
  - While a process is in its critical section, any other process that tries to enter its critical section **must loop continuously in the call to acquire()**
  - This lock therefore called a **spinlock**
- Spinlock wastes CPU cycles that some other processes might be able to use productively
- Spinlock has one advantage
  - No context switch is required when a process must wait on a lock, and a context switch may take considerable time
  - Thus, when locks are expected to be held for short times, spinlocks are useful

Spinlocks (when should it be used?)

- Characteristic: busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** is an integer variable, represents the **usage sate** of one type of resources
  - *S* > 0, **represents the number of usable resource**
  - *S* < 0, **represents the number of waiting processes for such resources**
- Can only be accessed via two atomic operations
  - wait() and signal()
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore

- Definition of the **`wait() operation`**

```
wait(S) {
     while (S <= 0)
         ; // busy wait
     S--;
}
```

Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).

- Definition of the **`signal() operation`**

```
signal(S) {
     S++;
}
```

When a process releases a resource, it performs a signal() operation () (incrementing the count)

In addition, in the case of wait(S), the testing of the integer value of S (S<=0), as well as its possible modification (S--), must be executed without interruption

When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems:
- Consider two concurrently running processes P_1 with statement S_1 and P_2 with statement S_2. Now require S_1 to happen before S_2

  Create a shared semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed

Producer-Consumer with Semaphores

In this problem, **buffer is the critical section.**

2 semaphores:

full: keeps track of number of items in the buffer

empty: keeps track of number of unoccupied slots

Initialization of semaphores:

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Producer-Consumer with Semaphores

Solution for Producer:

```
do{

   //produce an item

   wait(empty);
   wait(mutex);

   //place in buffer

   signal(mutex);
   signal(full);

}while(true)
```

Solution for Consumer:

```
do{

     wait(full);
     wait(mutex);

     // remove item from buffer

     signal(mutex);
     signal(empty);

     // consumes item

}while(true)
```

what problem will be incurred if we reverse wait(empty) and wait(mutex)

Implementation with no Busy waiting (Cont.)

- **Busy waiting** problem: like the implementation of mutex locks, wait() and signal() semaphore presents the same problem
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait

How to solve the busy waiting problem?

- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state
- The control is transferred to the CPU scheduler, which selects another process to execute

Implementation with no Busy waiting (Cont.)

- With each semaphore there is an associated **waiting queue**
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
  typedef struct{
          int value;
          struct process *list;
   }semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a **wakeup()** operation, which changes the process from the waiting state to the ready state
- The process is then placed in the ready queue
- When a process must wait on a semaphore, it is added to the list of processes. **A signal()** operation removes one process from the list of waiting processes and awakens that process

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P

In this implementation, semaphore values may be negative. If it is negative, its magnitude is the number of processes waiting on that semaphore t.)

```
var occupied;         // 1 if critical section is occupied, 0 if not.
var blocked;          // counts the number of blocked processes

Enter_Region:         // process enters its critical section
{
  IF (occupied){                    // if critical section occupied
    THEN blocked = blocked + 1;     // increment blocked counter
    sleep();                        // go to "sleep", or block
    }
  ELSE occupied = 1;                // if can enter critical section,
                                    // increment counter

}
    ...
    ...                             // critical section
    ...

Exit_Region:           // process exits its critical section
{
   occupied = 0;
   IF (blocked){
     THEN wakeup(process);          // if another process is sleeping,
                                    // wake the process up.
     blocked = blocked - 1;         // decrement blocked counter
     }
}
```

Is there any problem in this design?

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Suppose that P_0 executes wait(S) and P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal() operation cannot be executed, P_0 and P_1 are deadlocked

Deadlock and Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
  - Indefinite blocking may occur if we remove processes from the list associate with a semaphore in LIFO (last in, first out) order

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- Also known as the producer-consumer problem
- In this problem, the producer and consumer processes share the following data structures
  - ***n*** buffers, each can hold one item
  - Semaphore **mutex** initialized to the value 1
    - Provides mutual exclusion for accesses to the buffer
  - Semaphore **full** initialized to the value 0
    - Counter the number of full buffers
  - Semaphore **empty** initialized to the value *n*
    - Counter the number of empty buffers

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
       /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
       ...
      /* add next produced to the buffer */
       ...
   signal(mutex);
   signal(full);
 } while (true);
```

We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

Discussions

- Why Semaphore **mutex**? Shouldn't the other two semaphores guarantee that *n* buffers will be written and read correctly?
  - Semaphore **mutex** can be removed when there is only one producer and one consumer
  - With multiple producers sharing the same buffer, or multiple consumers sharing the buffer, removing **mutex** could result in two or more processes reading or writing into the same slot at the same time

Because such queue will usually be implemented as a circular queue. Producer will be writing to the tail of the queue, while consumer reads from the head. They never access the same memory at the same time.

The idea here is that both consumer and producer can track the position of the tail/head independently.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- How is this different from the producer-consumer problem?
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

Readers-Writers Problem

- A solution for the Reader-Writer problem
- Shared Data
  - Dataset
  - Semaphore **rw_mutex** initialized to 1
    - A mutual exclusion semaphore for the writers
    - It is also used by the first or last reader that enters or exits the critical section.
    - It is not used by reader who enter or exit while other readers are in their critical section
  - Integer **read_count** initialized to 0
  - Semaphore **mutex** initialized to 1
    - It is used to ensure mutual exclusion when the variable **read_count** is updated

Readers-Writers Problem (Cont.)

- The structure of a **writer** process

```
do {
      wait(rw_mutex);
         ...
      /* writing is performed */
         ...
       signal(rw_mutex);
  } while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);
        signal(mutex);
         ...
        /* reading is performed */
         ...
        wait(mutex);
        read_count--;
        if (read_count == 0)
            signal(rw_mutex);
        signal(mutex);
} while (true);
```

why not place signal(mutex) before if(read_count==1)

Dining-Philosophers Problem

It is a simple representation of the need to allocate several resources among several processes in a **deadlock-free** and **starvation-free** manner

- Five philosophers spend their lives alternating thinking and eating
- The philosophers share a circular table surrounded by five chairs, each belong to one philosopher
- In the center of the table is a bowl of rice, and the table is laid with **five single chopsticks**
- From time to time, a philosopher gets hungry and tries to pick up the **two chopsticks that are closest to her** (the chopsticks that are between her and her left and right neighbors)
- A philosopher may pick up **only one chopstick at a time**
- Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher **has both her chopsticks** at the same time, she eats **without releasing** the chopsticks.
- When she is finished eating, she puts down **both chopsticks** and starts thinking again.

Dining-Philosophers Problem (cont.)

- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

            //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

starting to eat until gets two chopsticks

deadlock

Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.

- What is the problem with this algorithm?
- How to solve the problem?

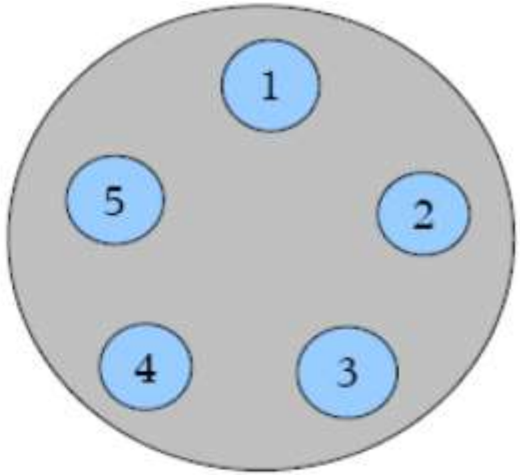
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - **Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section.**
  - Use an asymmetric solution--- an odd-numbered philosopher picks up first her left chopstick and then her right chopstick. Even-numbered philosopher picks up first her right chopstick and then her left chopstick.

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - **Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section.**

Dining-Philosophers Problem Algorithm (Cont.)



To simplify the analysis, let's assume that all the philosopher does very minimal thinking. That is, they
eat for a while, give up the chopstick, then immediately come back to wait for eating again. Suppose,
all philosophers indicated their wish to eat around the same time. Consider the following sequence:

- 1 and 3 start eating, 2, 4 and 5 are blocking
- 3 stops eating. 2 and 5 remain blocked since 1 is eating. So 4 gets the chopstick and start eating.
- 4 stops eating. 2 and 5 remain blocked since 1 is eating. So 3 gets the chopstick and start eating.
- 1 stops eating. 2 and 4 remain blocked since 3 is eating. So 5 gets the chopstick and start eating.
- 5 stops eating. 2 and 4 remain blocked since 3 is eating. So 1 gets the chopstick and start

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
    - Several processes may be executing in their critical sections simultaneously
  - wait (mutex) … wait (mutex)
    - Deadlock will occur
  - Omitting of wait (mutex) or signal (mutex) (or both)
    - Either mutual execution is violated, or a deadlock will occur
- Deadlock and starvation are possible.

Exercise

Three processes are involved in printing a file (pictured below). Process A reads the file data from the disk to Buffer 1, Process B copies the data from Buffer 1 to Buffer 2, finally Process C takes the data from Buffer 2 and print it.

```
  Process A           -------------        Process B          ------------      Process C
--------------------->|  Buffer 1 |------------------------------>|  Buffer 2 |------------------------------>
Read from File        -------------          copy             -------------        print
```

Assume all three processes operate on one (file) record at a time, both buffers' capacity are one record.

Write a program to coordinate the three processes using semaphores.

Exercise 2

```
semaphore empty1 = 1;
semaphore empty2 = 1;
semaphore full11 = 0;
semaphore full12 = 0;

Process_A () {
    while(1) {
        wait(empty1);
        read(next_file(), Buffer_1);
        signal(full1);

    }
}

Process_B () {
    while(1) {
        wait(full1);
        wait(empty2);
        copy(Buffer_2, Buffer_1);
        signal(empty1);
        signal(full2);
    }
}

Process_C () {
    while(1) {
        wait(full2);
        print(Buffer_2);
        signal(empty2);
    }
}
```

Monitors in Process Synchronization

Monitors

- What are the monitors and how can we use them?
- Monitor Semantics/Structure
- Solving synchronization problems with monitors
- Comparison with semaphores

Revisiting semaphores!

- Semaphores: Common programming errors

Process i	**Process j**	**Process k**	**Process m**
`P(S)`	`V(S)`	`P(S)`	`P(S)`
`CS`	`CS`	`CS`	`if(something or other)`
`P(S)`	`V(S)`		`return;`
			`CS  V(S)`

Revisiting semaphores!

- **Semaphores** are very “low-level” primitives
  - Users could easily make small **errors**
  - Similar to programming in assembly language
  - Small error brings system to grinding halt
  - Very **difficult to debug**

- Simplification: Provide concurrency support in compiler
  - **Monitors**

Monitors

- The monitor is one of the ways to achieve Process synchronization.
- The monitor is supported by programming languages to achieve mutual exclusion between processes.
- For example, Java Synchronized methods. Java provides wait() and notify() constructs.

Monitors

- It is the collection of **condition variables** and **procedures** combined together in a special kind of module or a package.
- The processes running outside the monitor **can't access** the internal variable of the monitor **but can call procedures** of the monitor.
- **Only one process** at a time can execute code inside monitors.

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Monitors

- **Condition Variables**: **two** different **operations** are performed on the condition variables of the monitor.
  - wait
  - signal
- Let say we have 2 condition variables
  - condition x, y
- **Wait** operation
  - **x.wait()**: process performing **wait** operation on any **condition variable** are **suspended**. The suspended processes are placed in **block queue** of that condition variable.
  - Each condition variable has its **unique block queue**.

Monitors

- **Signal** operation
  - **x.signal()**: When a process performs **signal** operation on **condition variable**, one of the **blocked processes** is given chance.

If (x block queue empty)

    // Ignore signal

else

    // Resume a process from block

queue.

Monitors

- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute a monitor procedure at any time.
    - “in the monitor”
  - If second thread invokes a monitor procedure at that time
    - It will block and wait for entry to the monitor
    - Need for a wait queue

Structure of a Monitor

```
Monitor monitor_name
{
      // shared variable declarations

      procedure P1(. . ..) {
            . . . .
            }


      procedure P2(. . ..) {
            . . . .
      .}
      .
      procedure PN(. . . .) {
           . . . .
      }

      initialization_code(. . . .) {
           . . . .
      }
}
```

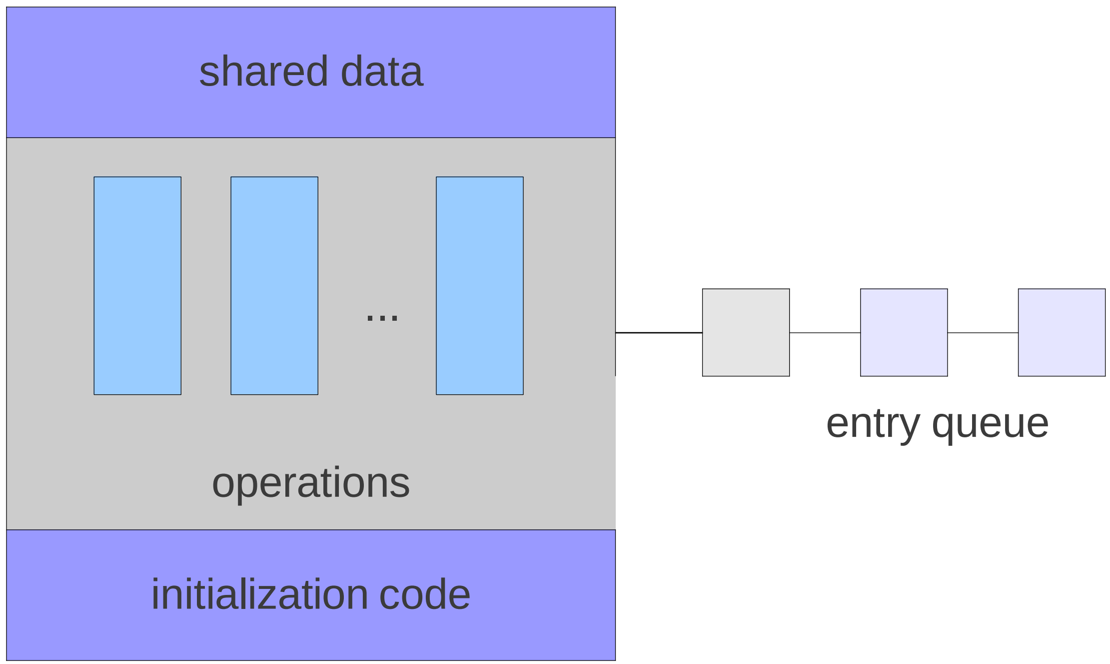
For example:

```
Monitor stack
{
int top;
       void push(any_t *)
       {
       . . . .
          }

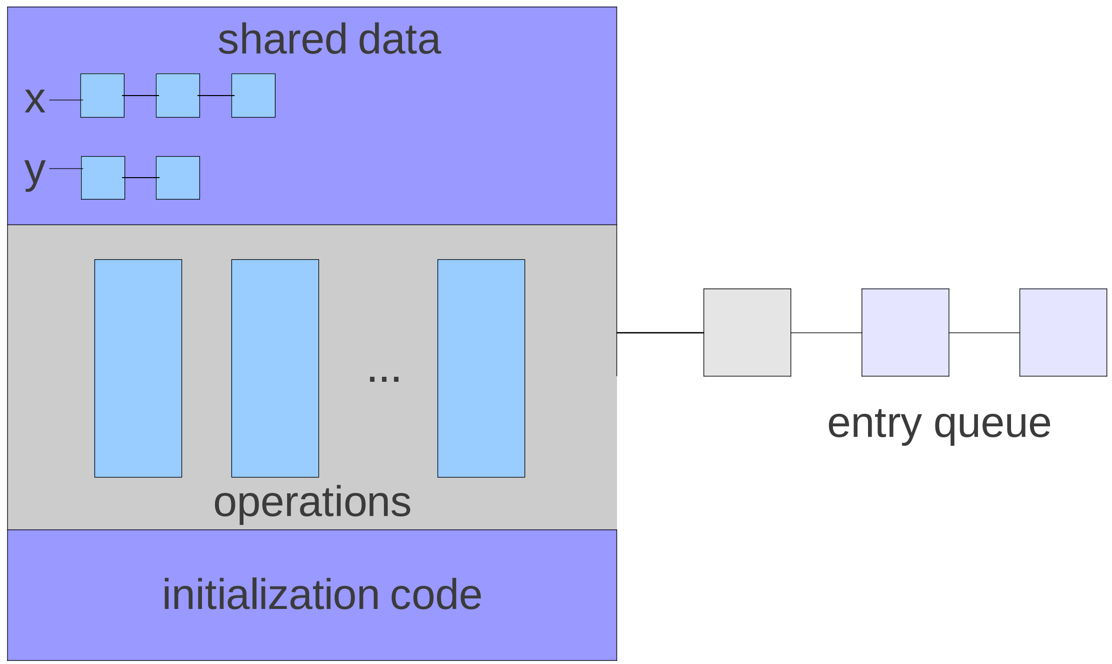
       any_t * pop() {
      }     . . . .

      initialization_code() {
           . . . .
      }
}
```

Structure of a Monitor



Synchronization Using Monitors



Reader-Writers solution using Monitors

- Considering a shared Database our objectives are:
  - Readers can access database only when there are no writers.
  - Writers can access database only when there are no readers or writers.
  - Only one thread can manipulate the state variables at a time.
- Basic structure of a solution

```
Reader()
   Wait until no writers
   Access database
   Check out – wake up a waiting writer
Writer()
   Wait until no active readers or writers
   Access database
   Check out – wake up waiting readers or writer
```

```
monitor ReadersWriters
  condition OKtoWrite, OKtoRead;
  int ReaderCount = 0;
  Boolean busy = false;


 procedure StartRead()
 {
  if (busy)                          // if database is not free, block
    OKtoRead.wait;
  ReaderCount++;                     // increment reader ReaderCount
  OKtoRead.signal();

 }

 procedure EndRead()
 {
   ReaderCount-- ;                   // decrement reader ReaderCount
   if ( ReaderCount == 0 )
      OKtoWrite.signal();
 }
```

```
procedure StartWrite()
{
 if ( busy || ReaderCount != 0 )
    OKtoWrite.wait();
 busy = true;
}

procedure EndWrite()
{
 busy = false;
 If (OKtoRead.Queue)
   OKtoRead.signal();
 else
   OKtoWrite.signal();
 }
```

```
Reader()
 {
   while (TRUE)              // loop forever
   {
     ReadersWriters.StartRead();
          readDatabase();              // call readDatabase function in monitor
     ReadersWriters.EndRead();
   }
 }

 Writer()
 {
   while (TRUE)              // loop forever
   {
          make_data(&info);                          // create data to write
     ReaderWriters.StartWrite();
          writeDatabase();             // call writeDatabase function in monitor
     ReadersWriters.EndWrite();
   }
 }
```

End of Chapter 5