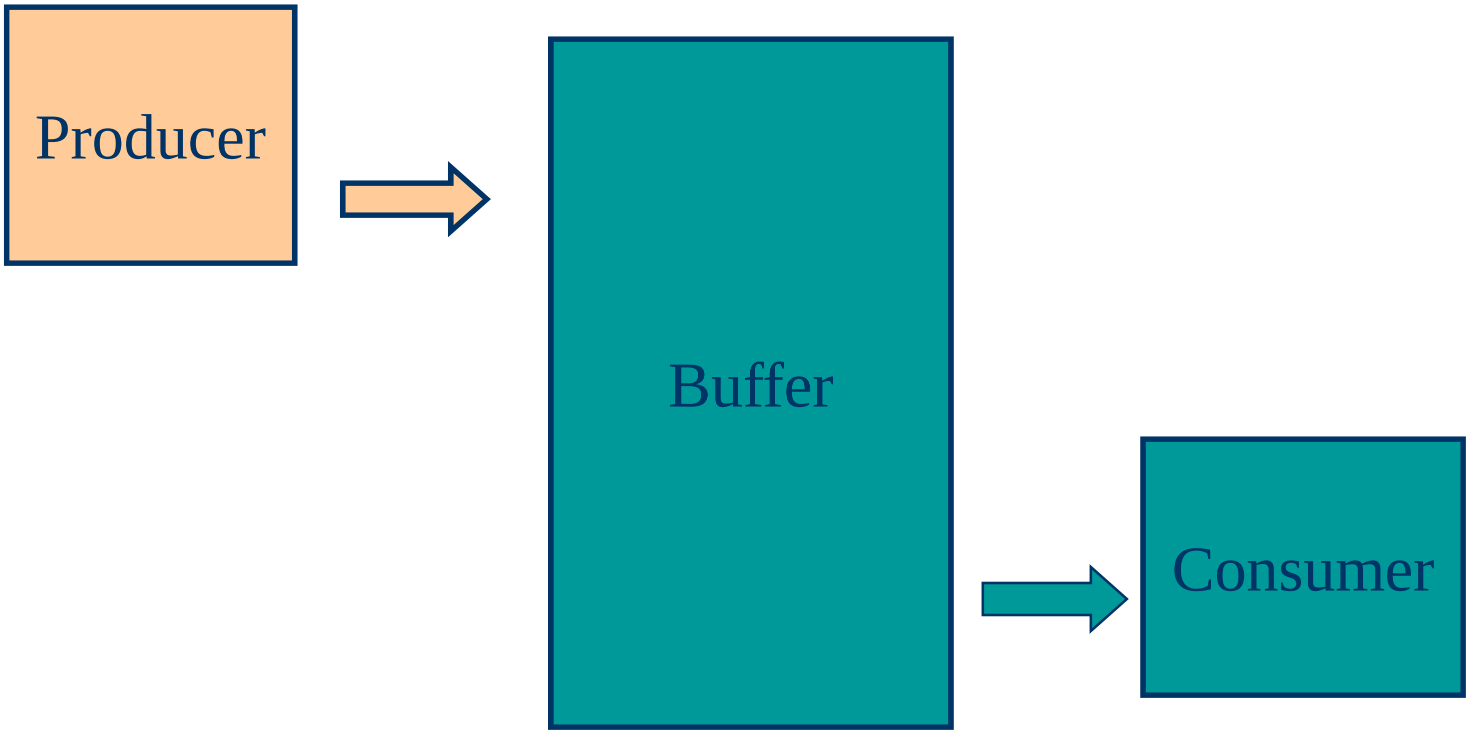
Process

Communication / Synchronization

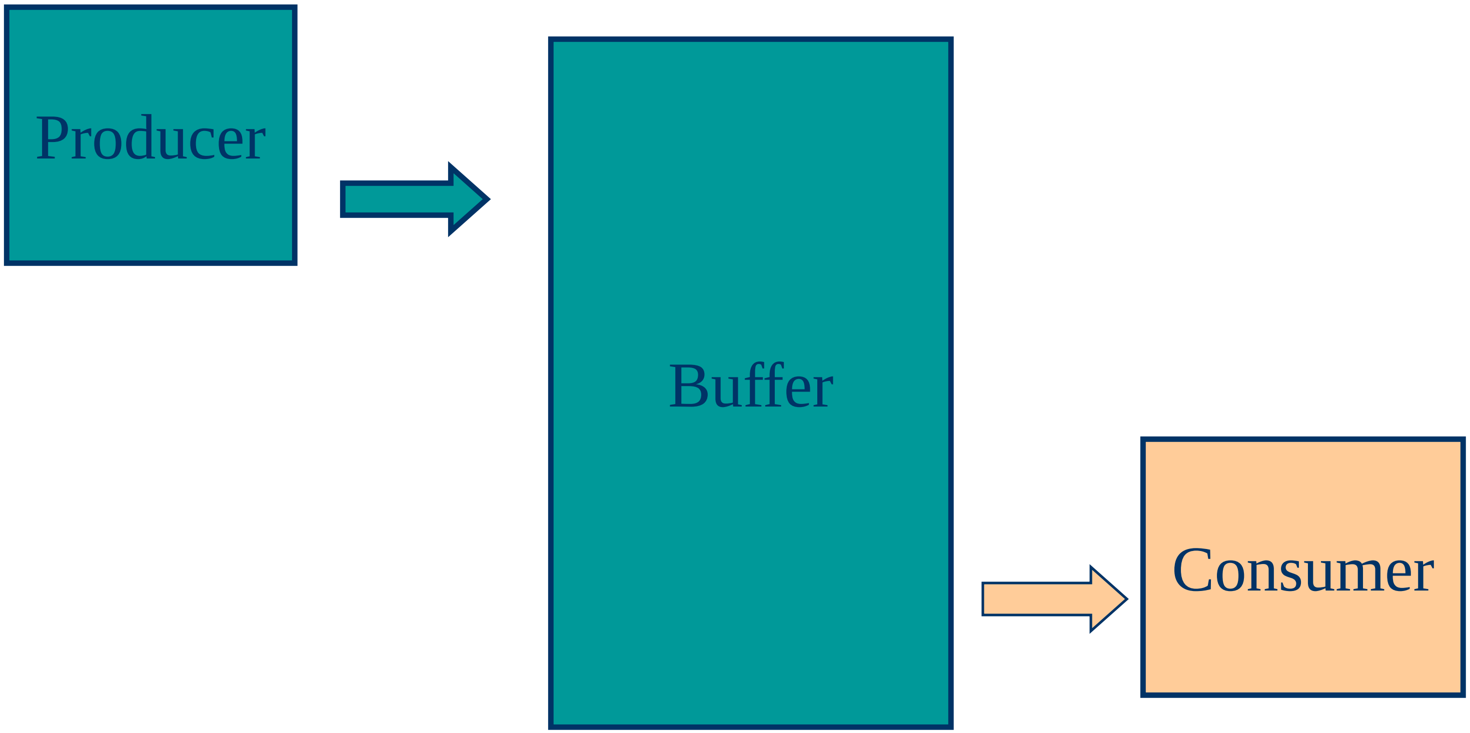
Inter-process Communications

- 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?

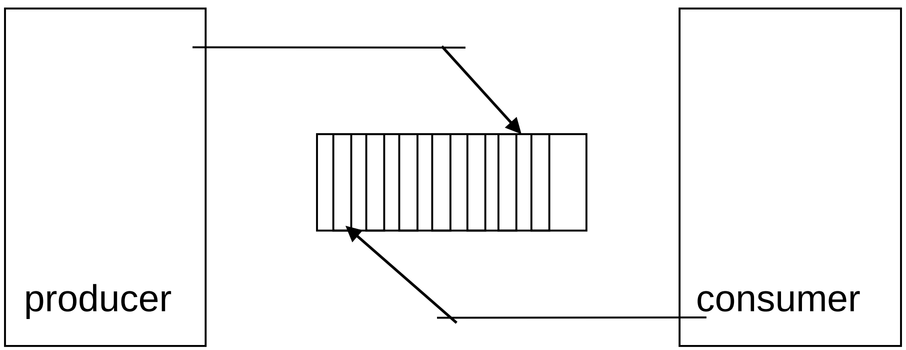
PROCESS SYNCHRONIZATION

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

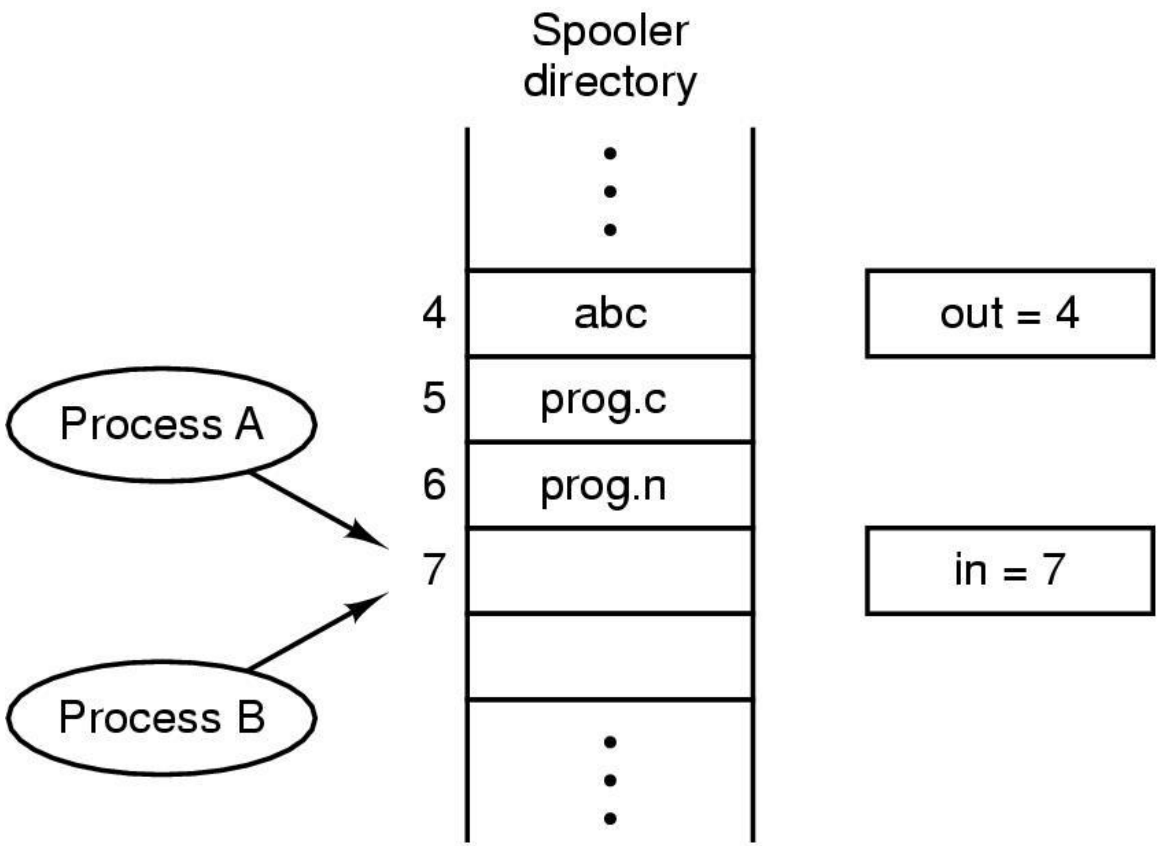
At a micro level, the following scenario could occur using this code:

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.

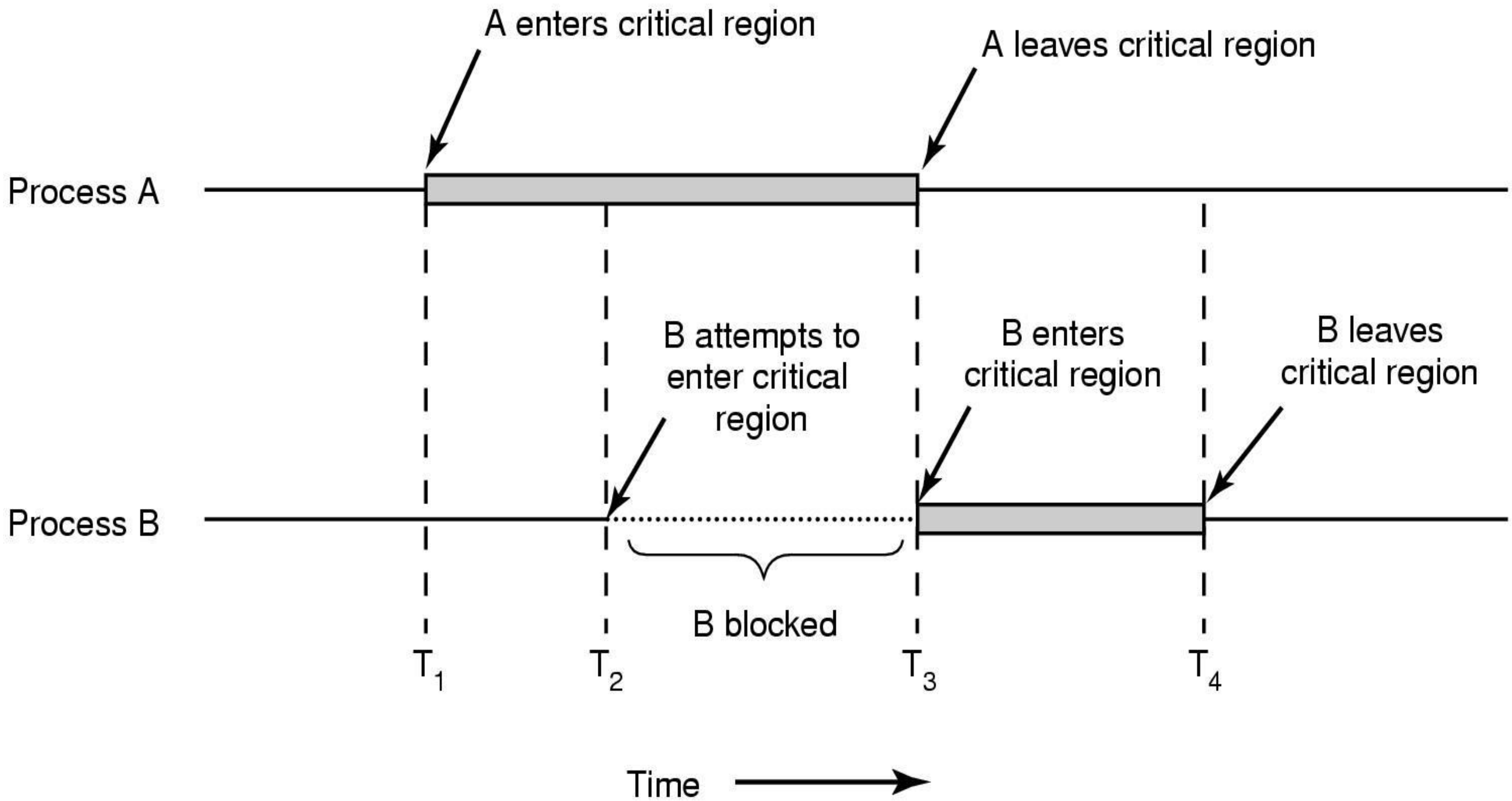
Can **Race Conditions** be avoided using above conditions?

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);
```

Entry Section

Critical Section (CS)

Exit Section

Remainder Section

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** ⇒ 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** ⇒ 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 */
}
```