UNIT-3

Chapter 1: Process Synchronization

CONTENTS

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

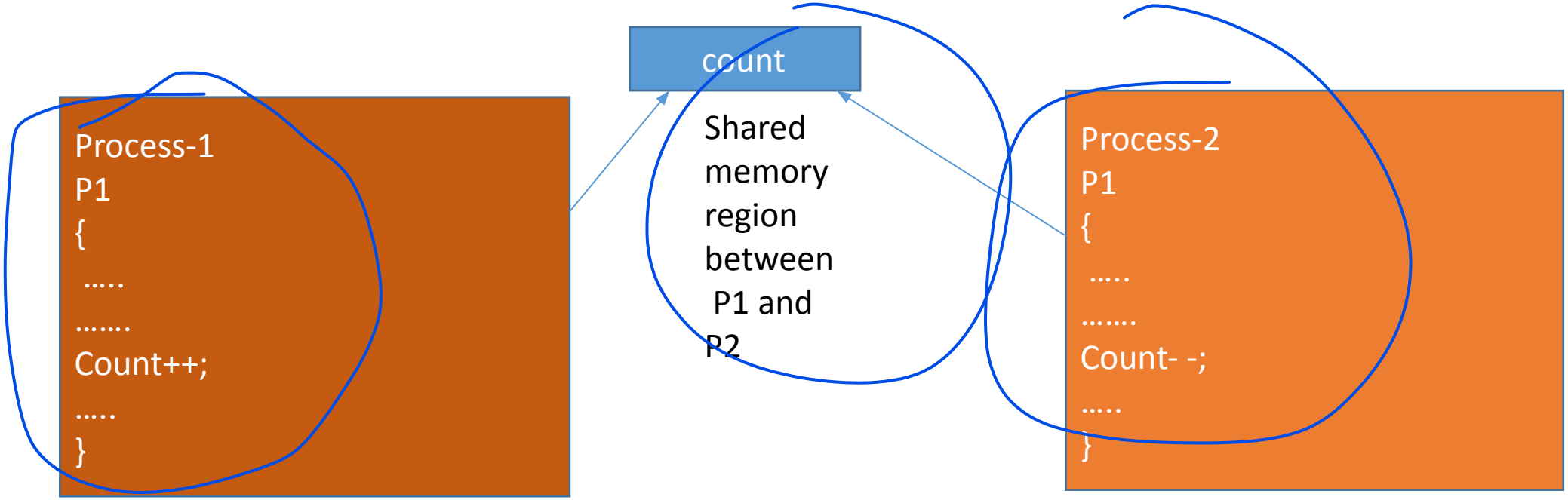
Background

- To improve CPU utilization we go for multiprogramming in which more than one process is placed in the main memory at the same time.
- As a result, multiple processes will be executing in the concurrent fashion(parallel execution).
- Such processes which are executing in parallel are broadly classified into two types:
  1. Independent Processes
  2. Cooperative processes(interdependent processes)
- Process synchronization is required in case of cooperative processes

Examples to illustrate cooperative processes

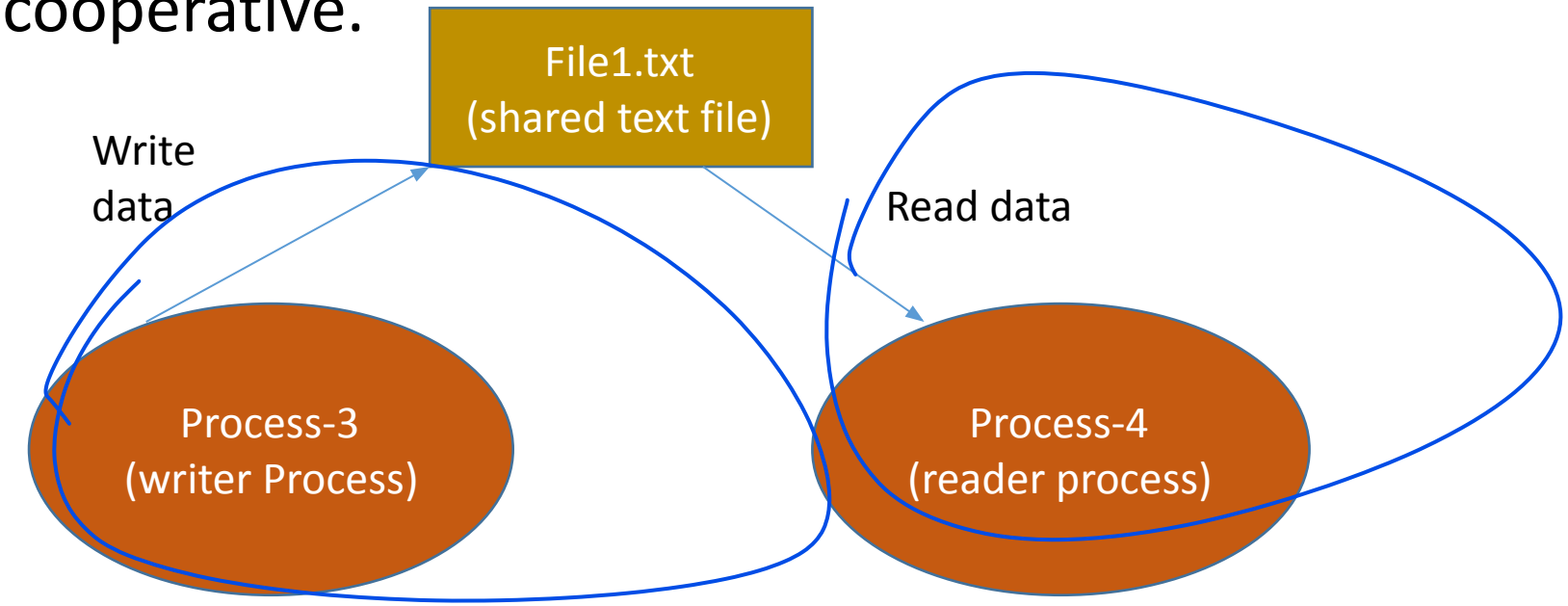
1.Producer consumer processes:

Consider two processes P1(producer) and P2 (consumer) both will share a common shared variable count. Both P1 and P2 are writing to the same shared variable count. Since,P1 and P2 are sharing common shared memory region they are cooperative processes.



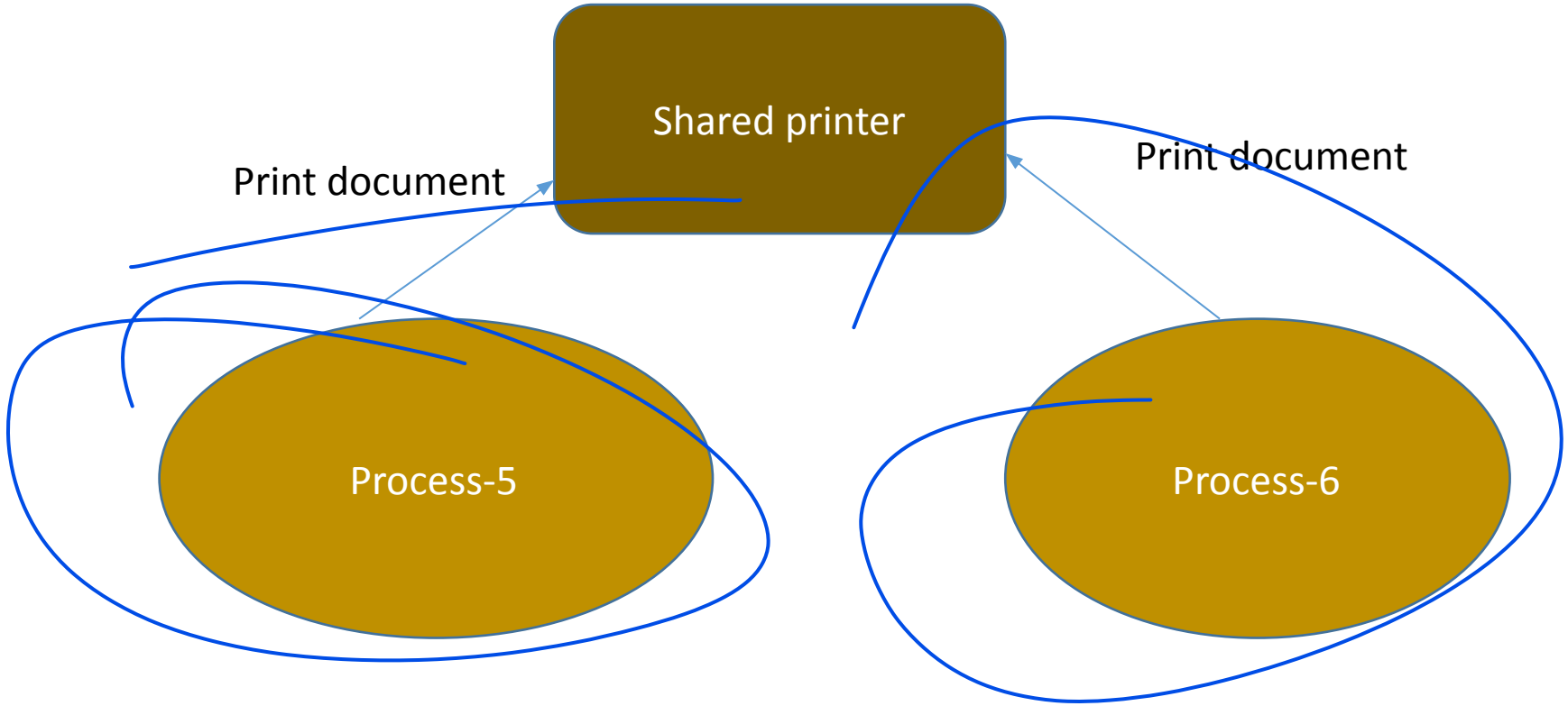
2.Reader writer Processes

Consider two processes P3(writer Process) and P4(reader process) performing write and read operation to the same file File1.txt. Hence they are cooperative.



3.Printing processes

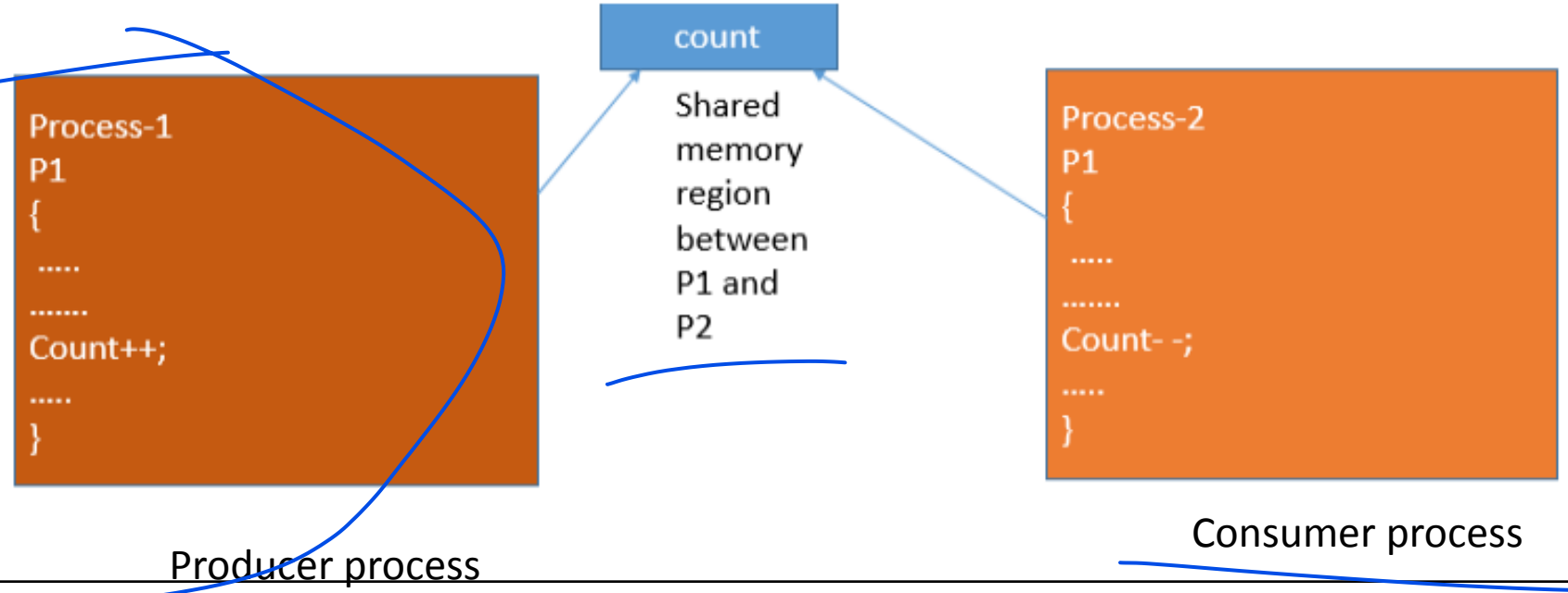
Consider two processes p5,p6 both sharing a common resource , printer to print their documents. Hence, they are cooperative.



What is the need for Processes (cooperative)to synchronize?/ Why Process Synchronization?

Example to illustrate Problems associated with concurrent execution of cooperative processes.

Let us consider the same producer consumer process example.



- P1 increments the value of the shared variable count, and P2 decrements the value of the shared variable count and since both process share a common memory location, they are cooperative.
- Suppose if both processes are executed in non parallel fashion, one after the another ,(i.e, P1 executes and finishes its execution and then P2 starts its execution).

Assume the value of count=5;

Upon execution of P1 followed by P2 ;

P1:count++ count=6

P2:count-- count=5

In this case no change in the value of the variable count.

Lets assume, P2 executes first and then P1.(one after the other)

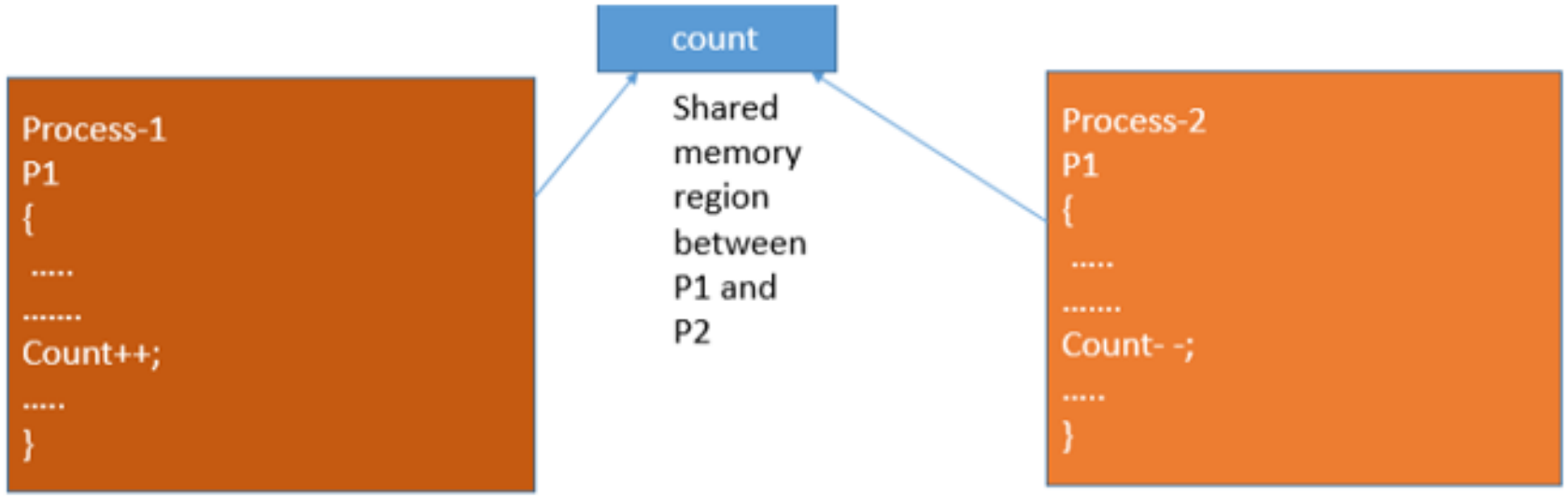
Initial value of Count=5;

P2:Count-- count=4

P1:count++ count=5

In this case also there is no change in the value of the variable count.

Suppose if both processes are executed in parallel fashion.

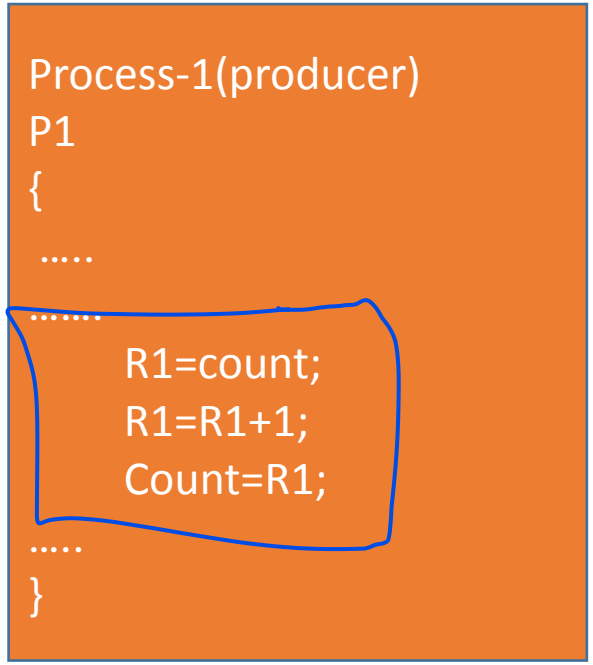


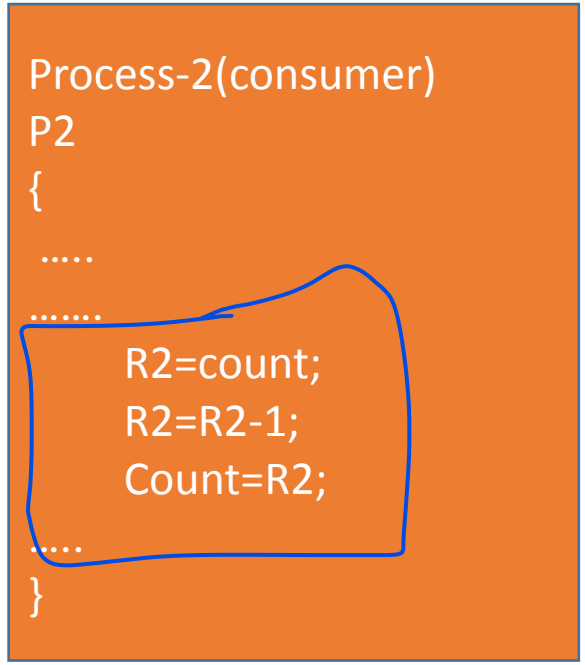
When converted to machine language, the count operation is achieved as follows:

R1=count;

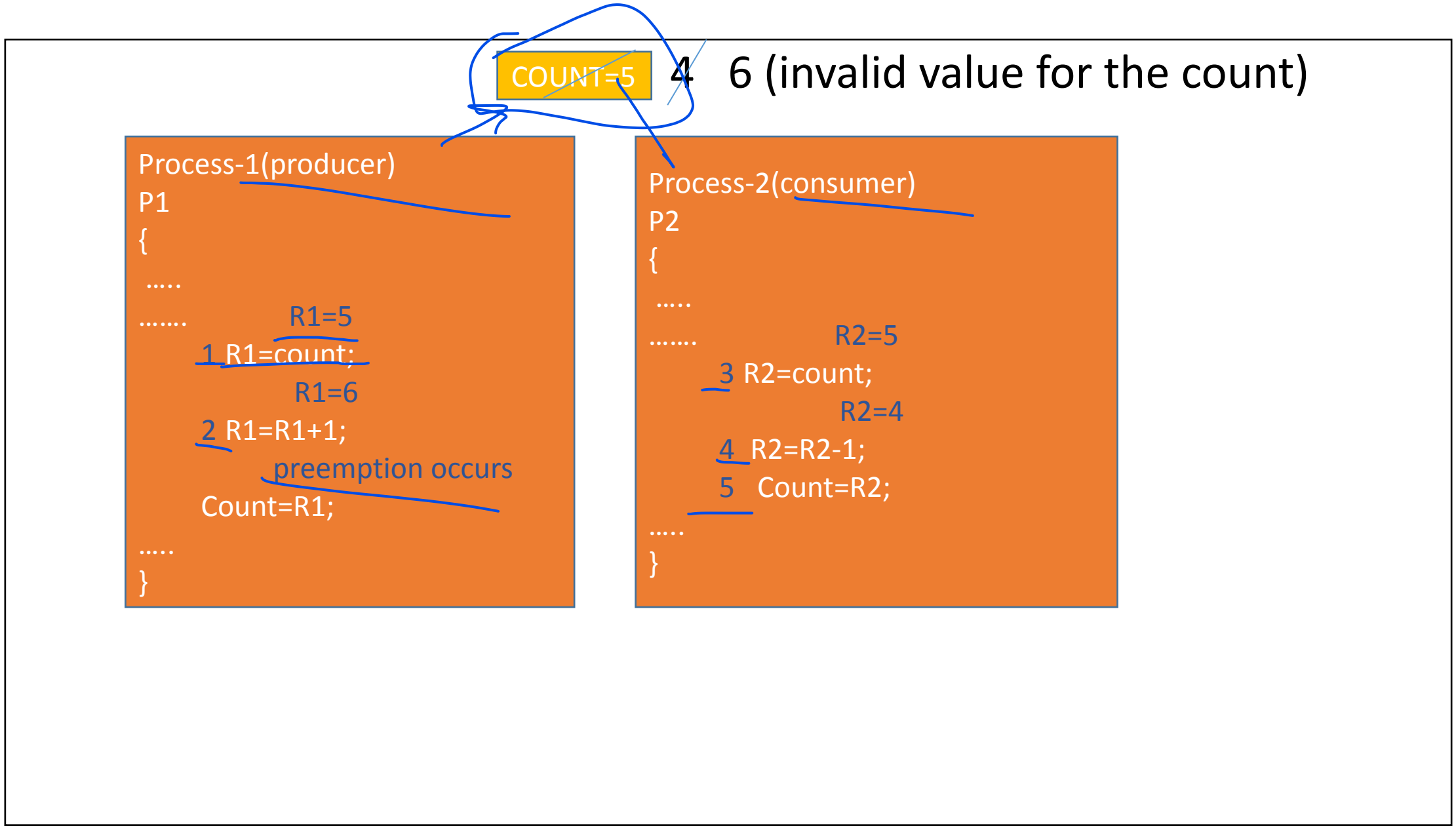
R1=R1+1;

Count=R1;





Let us assume count=5,and p1 and p2 executes in parallel. Depending on the order of execution of the processes the final value of count varies as shown in the upcoming slides.


COUNT=5
4
6 (invalid value for the count)
Process-1(producer)
P1
{
.....
.....
R1=5
1 R1=count;
R1=6
2 R1=R1+1;
preemption occurs
Count=R1;
.....
}
Process-2(consumer)
P2
{
.....
.....
R2=5
3 R2=count;
R2=4
4 R2=R2-1;
5 Count=R2;
.....
}

COUNT=5 ~~6~~ 4 (invalid value for the count)

```
Process-1(producer)
P1
{
 …..
…….            R1=5
       3 R1=count;
                 R1=6
       4 R1=R1+1;
       5 Count=R1;
…..
}
```

```
Process-2(consumer)
P2
{
 …..
…….                R2=5
       1 R2=count;
                   R2=4
       2  R2=R2-1;
                 preemption occurs
        6  Count=R2;
…..
}
```

COUNT=5 4 6 (invalid value for the count)

```
Process-1(producer)
P1
{
 …..
…….            R1=5
       1 R1=count;
                 R1=6
       5 R1=R1+1;
       6 Count=R1;   COUNT=6
…..
}
```

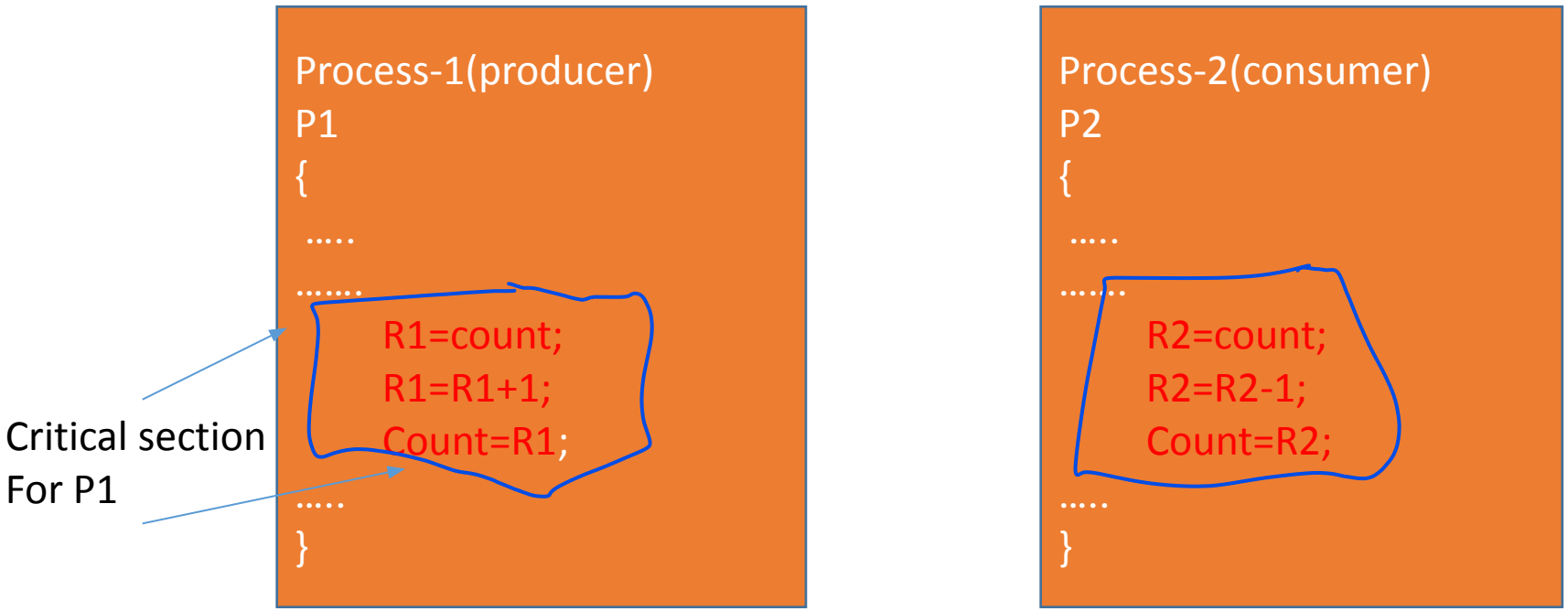
```
Process-2(consumer)
P2
{
 …..
…….                R2=5
       2 R2=count;
                   R2=4
       3  R2=R2-1;

        4  Count=R2;  COUNT=4
…..
}
```

We have arrived at this in correct state because we allowed both processes to manipulate the shared variable concurrently.

This kind of 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 is called **race condition**.

The Critical Section Problem

- The problem of race condition occurs when two or more cooperative processes try to access the shared resource or data at the same time.
- The piece of code/part of program in which the process is accessing shared variable or resource is called **critical section**.



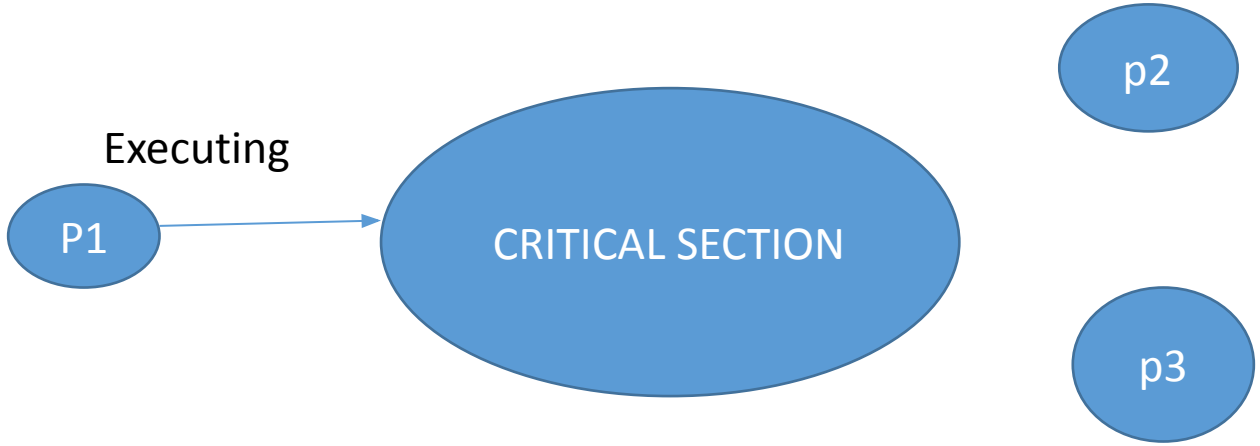
- Consider system consisting of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has a segment of code called **critical section.**
- **In critical section,** Process may be changing common variables, updating table, writing file, etc.
- The important feature is that, When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- The *critical-section problem is to design a protocol that the processes can use to cooperate.*
- Each process can have one or more critical sections.
- Each process must request permission to enter its **critical section**.
- The section of code implementing this request is the **entry section.**
- The critical section may be followed by an **exit section.**
- The remaining code is the **remainder section**.

General structure of a process Pi

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

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

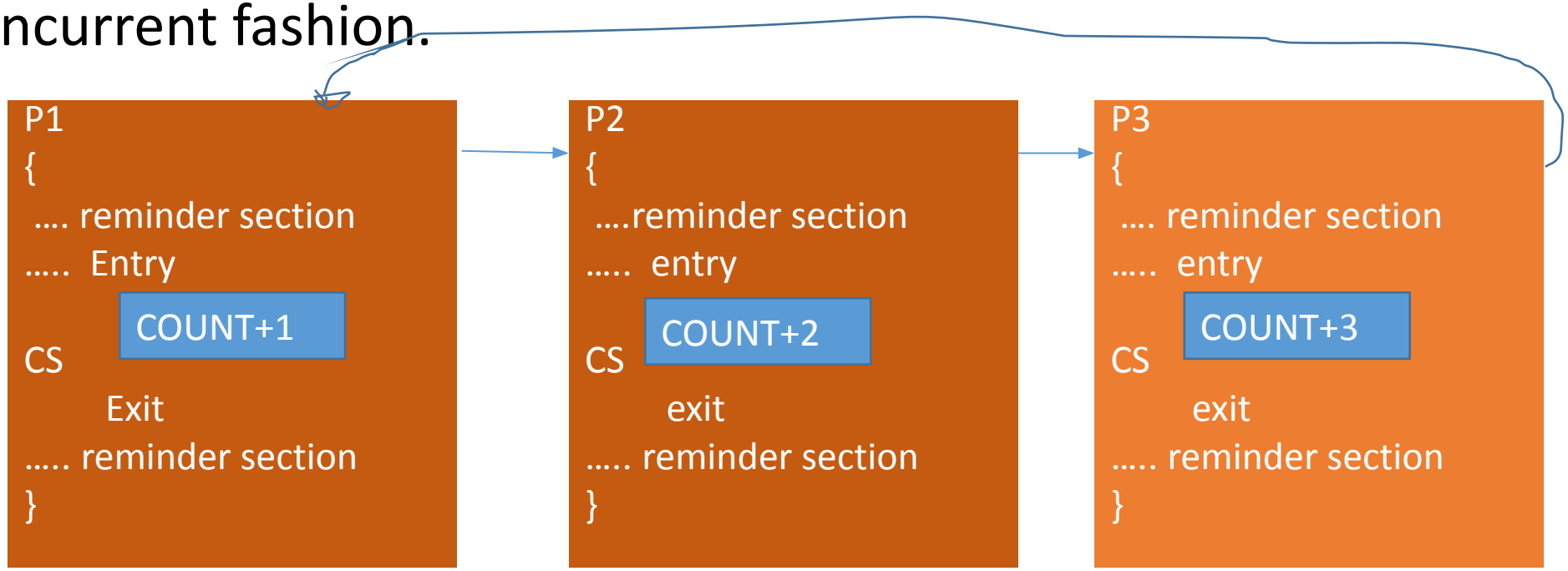
1. **Mutual exclusion.** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.



In this case, P1 is executing in its critical section ,hence p2 and P3 should not be allowed to enter their critical section.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

Consider three cooperative processes P1,P2,P3 executing in a concurrent fashion.



- Suppose a proposed solution to the critical section problem in order to ensure mutual exclusion allows the processes to enter their critical section in the order p1,p2,p3 in round robin fashion.
- Suppose if P2 is executing in its critical section and P3 is executing in its remainder section and p1 had made a request to enter its critical section. Then upon p2 exiting from the critical section the process p1 will not be allowed to enter the critical section because p3 is the next process to enter the critical section. But since p3 is still executing in its remainder section p1 has to wait.
- Thus, the progress of P1 is blocked by P3 who is in the reminder section.
- To avoid this kind of situation a solution to the critical section problem should ensure **progress.**

3.Bounded waiting. There exists a bound, or limit, 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.

- At any given point of time many kernel mode processes may be active in the operating system. As a result, the code implementing an operating system is subject to several possible race conditions.
- **Example:** Kernel data structure that maintains a list of all open files, structures for maintaining memory allocation, process lists, interrupt handling etc.
- Two general approaches are used to handle critical sections in operating systems:
  (1) **preemptive kernels :** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
  (2) **Nonpreemptive kernels :** A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Peterson’s Solution

- Classic software-based solution to the critical-section problem.
- Peterson’s solution is not guaranteed to work on modern computer architectures. However it provides a good algorithmic description of solving the critical section problem and illustrates the complexities involved in designing software that addresses the requirements of mutual exclusion, progress and bounded waiting requirements.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0 and P1.

- Peterson's solution requires two data items to be shared between the two processes:

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

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

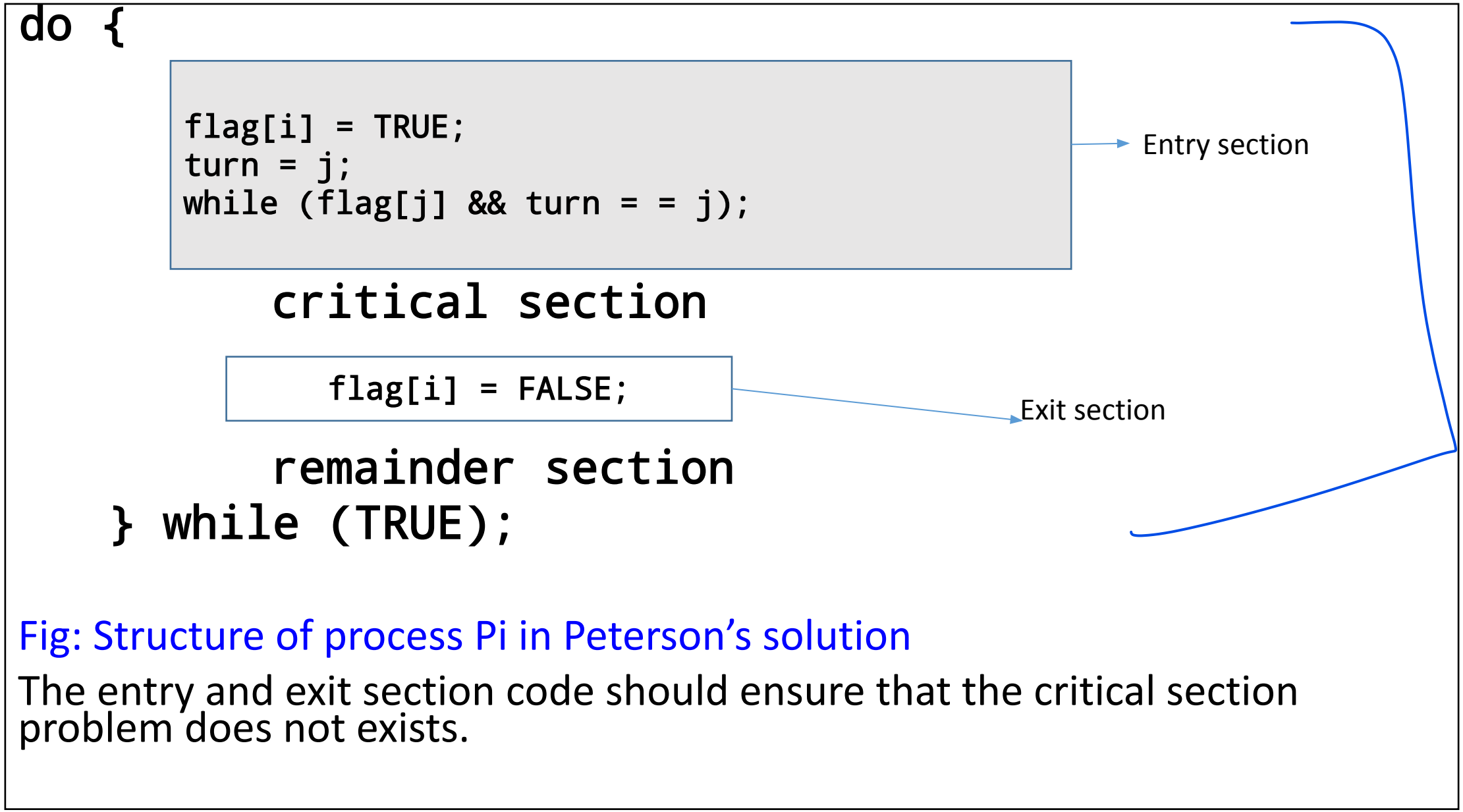


Fig: Structure of process Pi in Peterson's solution

The entry and exit section code should ensure that the critical section problem does not exists.

```
Process-0
do
{
    flag[0] = TRUE;
    turn = 1;
while (flag[1] && turn = = 1);

      //critical section

flag[0]=FALSE;

 remainder section
}
```

```
Process-1
do
{
    flag[1] = TRUE;
    turn = 0;
while (flag[0] && turn = = 0);

      //critical section

flag[1]=FALSE;

 remainder section
}
```

- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.
- In order to prove the solution is correct we need to show that
    1. Mutual exclusion is preserved.
    2. The progress requirement is satisfied.
    3. The bounded-waiting requirement is met.

- P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Mutual exclusion is preserved.
- If Pj is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn== i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.
- However, once Pi exits its critical section, it will reset flag [j] to false, allowing Pi to enter its critical section. If Pj resets flag [j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Synchronization Hardware

- Any solution to the critical-section problem requires a simple tool- a lock.
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated as shown in the figure below:

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

- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this manner, 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.
- Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming.
- Let us explore hardware instructions that are available on many systems that can be used effectively in solving the critical section problem.

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
}
```

Analyze this

- Does this scheme provide mutual exclusion?

Process 1

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

lock=0

Process 2

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

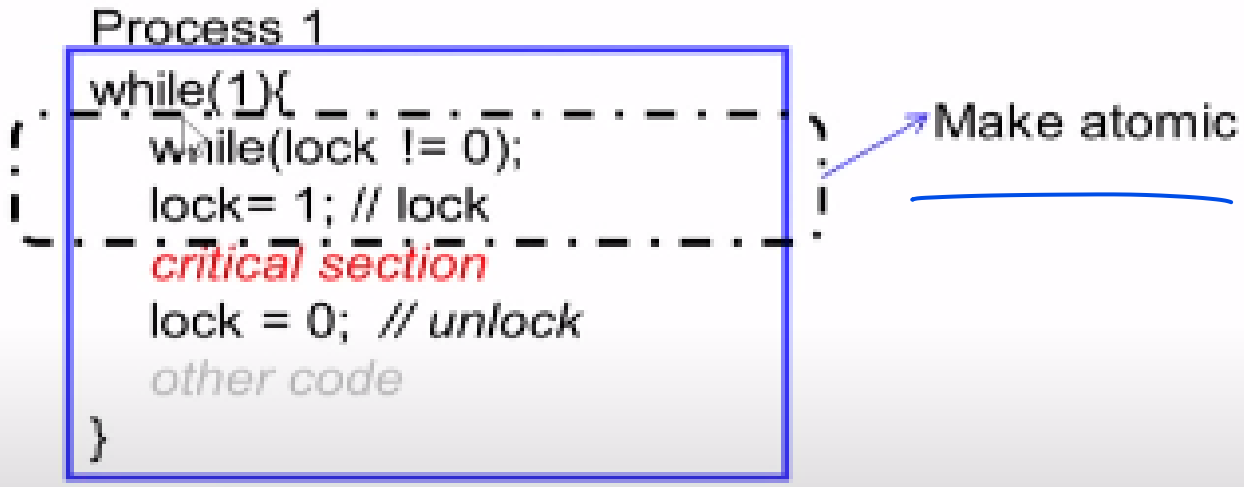
No

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

context switch

If only…

- We could make this operation atomic



- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically.(that is as one uninterruptible unit).
- This special instructions can be used to solve critical section problem.
- The following two hardware instructions will be discussed.

  1)TestAndSet ( )

  2)Swap( )

The TestAndSet () Instruction can be defined as follows:

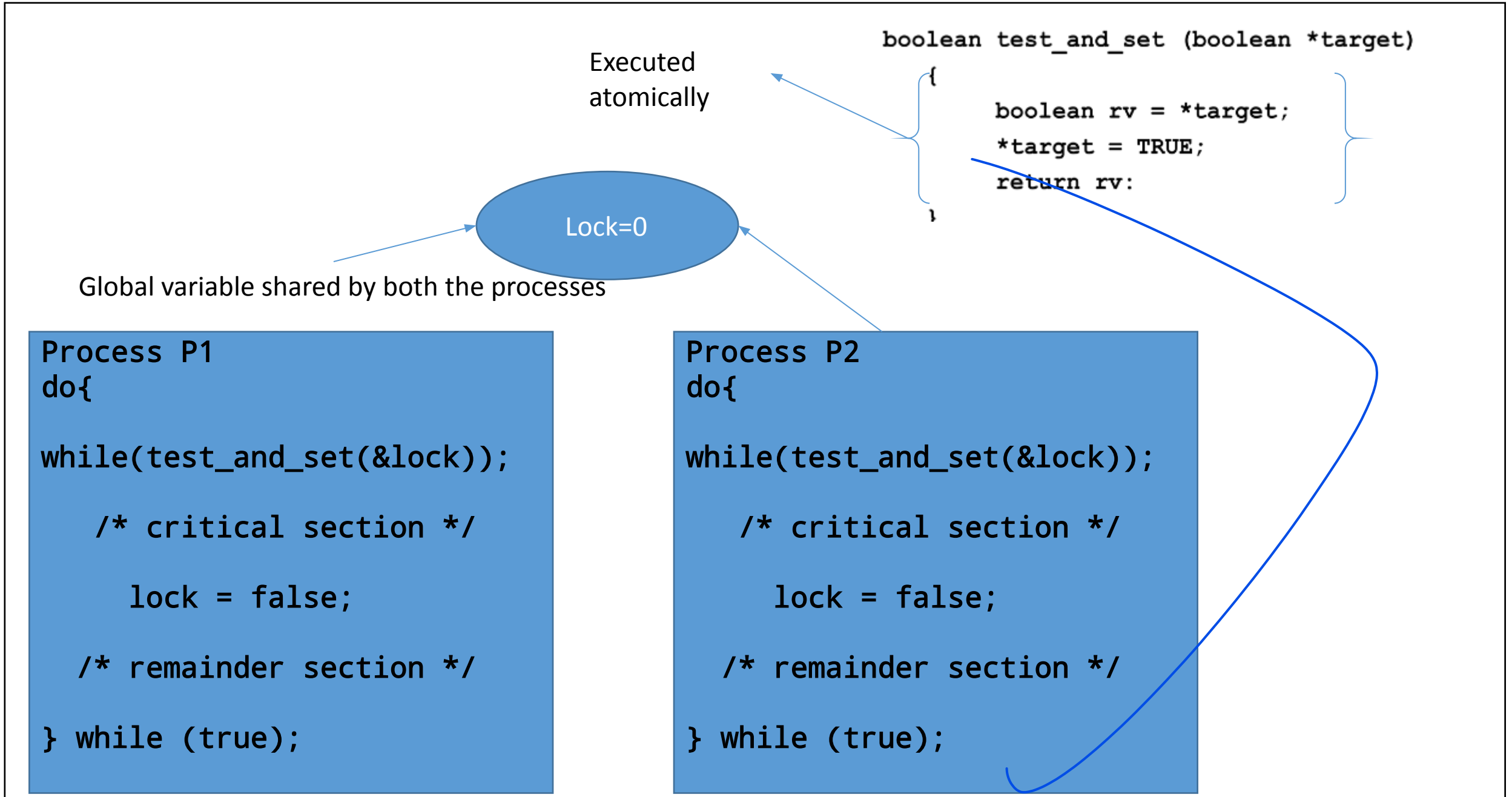
```
boolean test_and_set (boolean *target)
          {
               boolean rv = *target;
               *target = TRUE;
               return rv:
          }
```

The important characteristic of this instruction is:

1. It is Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Mutual exclusion Implementation with TestAndSet()

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


boolean test_and_set (boolean *target)
{
boolean rv = *target;
*target = TRUE;
return rv:
}
Executed atomically
Lock=0
Global variable shared by both the processes
Process P1
do{
while(test_and_set(&lock));
/* critical section */
lock = false;
/* remainder section */
} while (true);
Process P2
do{
while(test_and_set(&lock));
/* critical section */
lock = false;
/* remainder section */
} while (true);

The Swap () instruction, operates on the contents of two words; it is defined as shown in Figure below:

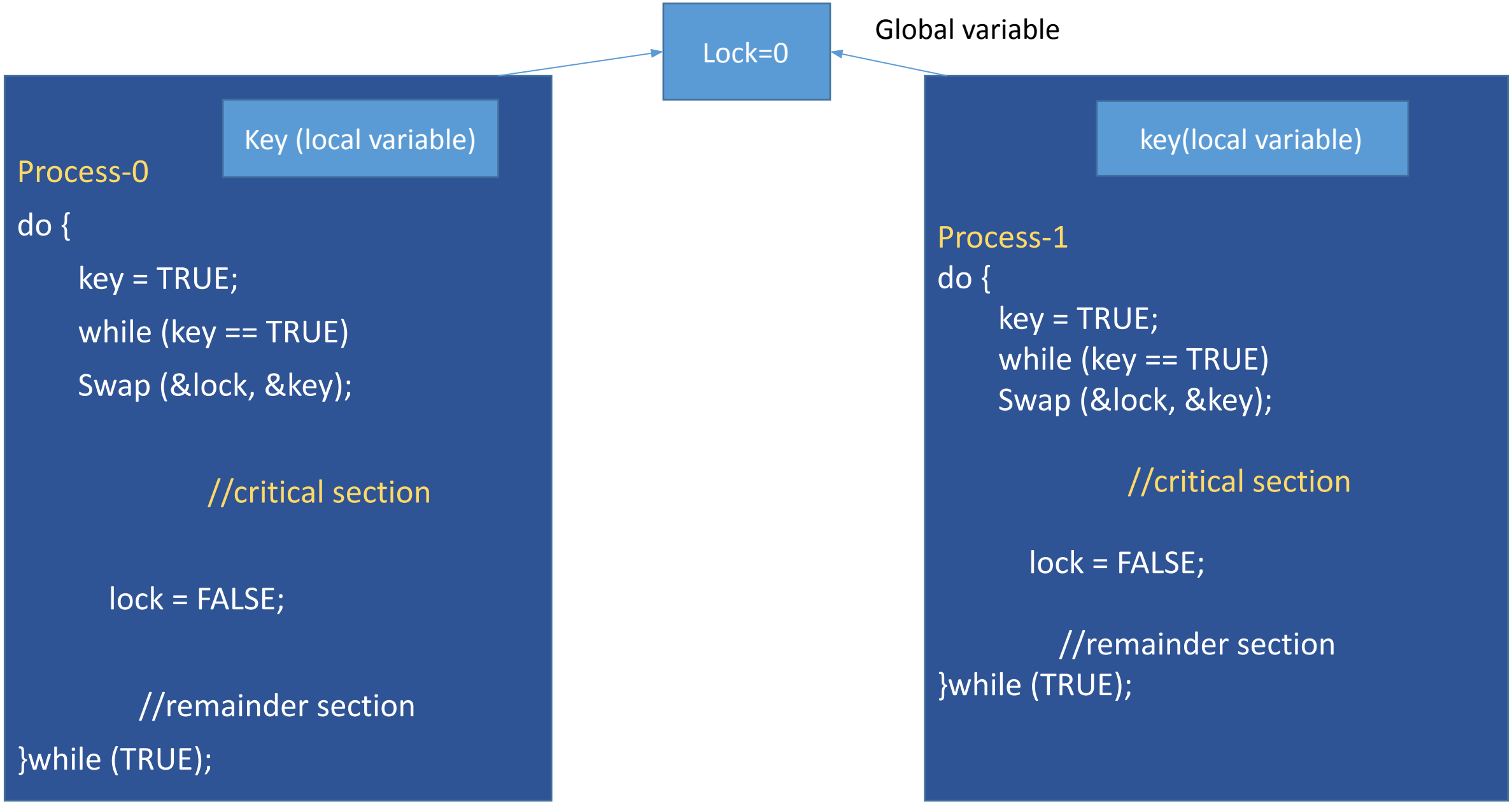
```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a= *b;
    *b = temp;
}
```

Like the TestAndSet () instruction, it is executed atomically.

- If the machine supports the Swap () instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable lock is declared and is initialized to false.
- In addition, each process has a local Boolean variable key.
- The structure of process Pi is shown in Figure below:

```
do {
        key = TRUE;
        while (key == TRUE)
        Swap (&lock, &key);

                        //critical section

            lock = FALSE;
                //remainder section
}while (TRUE);
```

Fig: Mutual-exclusion implementation with the Swap () instruction


Lock=0
Global variable
Key (local variable)
Process-0
do {
key = TRUE;
while (key == TRUE)
Swap (&lock, &key);
//critical section
lock = FALSE;
//remainder section
}while (TRUE);
key(local variable)
Process-1
do {
key = TRUE;
while (key == TRUE)
Swap (&lock, &key);
//critical section
lock = FALSE;
//remainder section
}while (TRUE);

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.
- Below figure present another algorithm using the TestAndSet( ) instruction that satisfies all the critical-section requirements.

```
do {
        waiting[i] = TRUE;
        key= TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```

P1,p2,p3,……pn

0 1 2 3 4 5

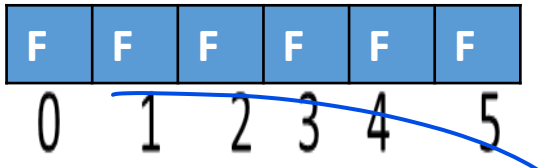
waiting

Shared variable

Lock=0

P1,P2,P3,……Pn

Waiting



Key

```
PROCESS-0
do {
        waiting[i] = TRUE;
        key= TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```

key

```
Process-1
do {
        waiting[i] = TRUE;
        key= TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```

- The common data structures are:
  - boolean waiting[n] ;
  - boolean lock;
- These data structures are initialized to false.
- To prove that the mutual exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting [i]== false or key==false.
- The value of key can become false only if the TestAndSet () is executed. The first process to execute the TestAndSet( ) will find key== false; all others must wait.
- The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

- To prove that the progress requirement is met, we note that a process exiting the critical section either sets lock to false or sets waiting[j] to false.
- Both allow a process that is waiting to enter its critical section to proceed.
- To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n - 1,0, ..., i-I). It designates the first process in this ordering that is in the entry section (waiting [j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

Semaphores

- The various hardware-based solutions to the critical-section problem (using the TestAndSet( ) and Swap( ) instructions) are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore.**
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait( ) and signal( ).
- The wait () operation was originally termed P (from the Dutch *proberen*, "to test"); signal( ) was originally called V(from *Verhogen,* "to increment").

The definition of wait () is as follows:

```
Wait(S) {
        While S<=0
            ;//no operation
        S--;
}
```

The definition of Signal( ) is as follows:

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

- All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In case of wait (S), the testing of the integer value of S (S ≤ 0), and its possible modification (S--), must also be executed without interruption.
- Two types of semaphores:
  - counting
  - binary semaphores(also called mutex locks as they are locks that provide mutual exclusion)
- The value of a counting semaphore can range over an unrestricted domain.

- We can use binary semaphores to deal with the critical-section problem for multiple processes.
- The n processes share a semaphore, mutex, initialized to 1. Each process Pi is organized as shown in Figure below:

```
do {
        waiting (mutex) ;
                // critical section
        signal (mutex) ;
                //remainder section
 }while (TRUE);
```

Figure: Mutual exclusion implementation with semaphore

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait ( ) operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal () operation (incrementing the count). 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.

Mutual Exclusion Implementation with Semaphores

```
int S;

wait(S)
{
  while(S<=0);

  S--;

}
signal(S)
{

   S++;

}
```

initialize S=1

Pi

do{

wait(S); — entry code

cs

signal(S); — exit code

rs

}while(1);

- We can also use semaphores to solve various synchronization problems.

EXAMPLE: consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements in process P1 and P2

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

```
Process 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.

Implementation

- The main disadvantage of the semaphore is that it requires busy waiting.
- A process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.
- Spinlocks do have an advantage in that 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;

- To overcome the need for busy waiting, the definition of the wait () and signal () semaphore operations can be modified.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. 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.
- Then control is transferred to the CPU scheduler, which selects another process to execute.
- 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.
- To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct
{
int value;
 struct process *list;
 } semaphore;
```

- 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.

- The wait () semaphore operation can now be defined as

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

- The signal () semaphore operation can now be defined as

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

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores S and Q, set to the value 1:

```
Process P0
Wait(S);
Wait(Q);
      .
      .
      .
Signal(S);
Signal(Q);
```

```
Process P1
Wait(Q);
Wait(S);
      .
      .
      .
Signal(Q);
Signal(S);
```

- suppose that P0 executes wait (S) and then P1 executes wait (Q). When P0 executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until P0 executes signal (S). Since these signal () operations cannot be executed, P0 and P1 are deadlocked.

Classic Problems of Synchronization

- This section, presents a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In the solutions to the problems, we use semaphores for synchronization.
- Three classical problems of synchronization are discussed:
  - The bounded buffer problem
  - The Readers-Writers Problem
  - The Dining-Philosophers Problem

The Readers-Writers Problem

- A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.
- If two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the readers-writers problem.
- The readers writers problem has several variations, all involving priorities.
- The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

- In this section, we present a solution to the first readers-writers problem
- In the solution to the first readers-writers problem, the reader processes share the following data structures:

  semaphore mutex, wrt;

  int readcount;
- The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0.
- The semaphore wrt is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as 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 readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure:

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

The code for a reader process is shown in Figure:

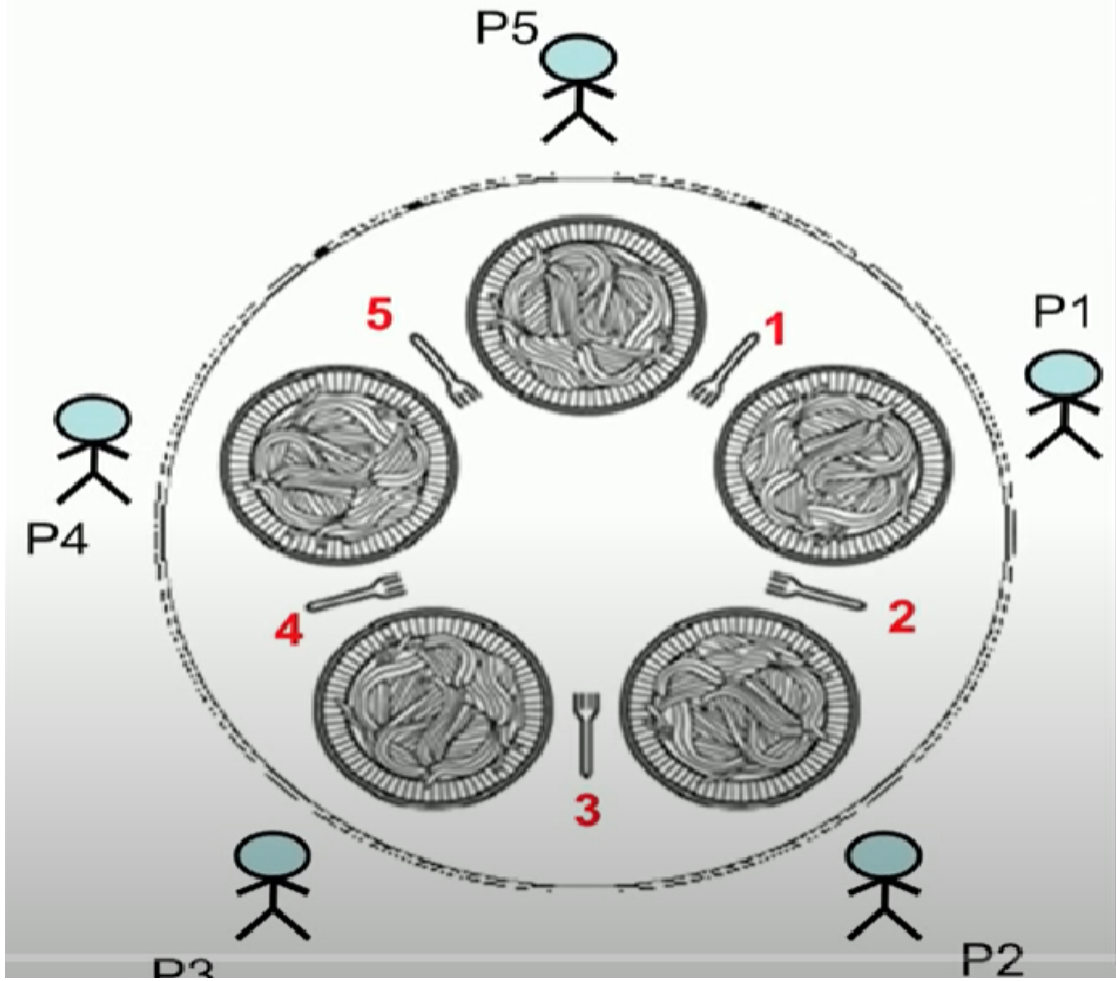
```
do {
        wait (mutex) ;
        readcount++;
         if (readcount ==1)
        wait (wrt) ;
        signal (mutex) ;

                // reading is performed

         wait (mutex) ;
         readcount--;
        if (readcount ==0)
        signal (wrt) ;
        signal (mutex) ;
}while (TRUE) ;
```

- If a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex.
- When a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer.
- Acquiring a reader-writer lock requires specifying the mode of the lock: either read or 'write access.
- When a process only wishes to read shared data, it requests the reader-writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode.
- Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode; only one process may acquire the lock for writing as exclusive access is required for writers.

Reader-writer locks are most useful in the following situations:

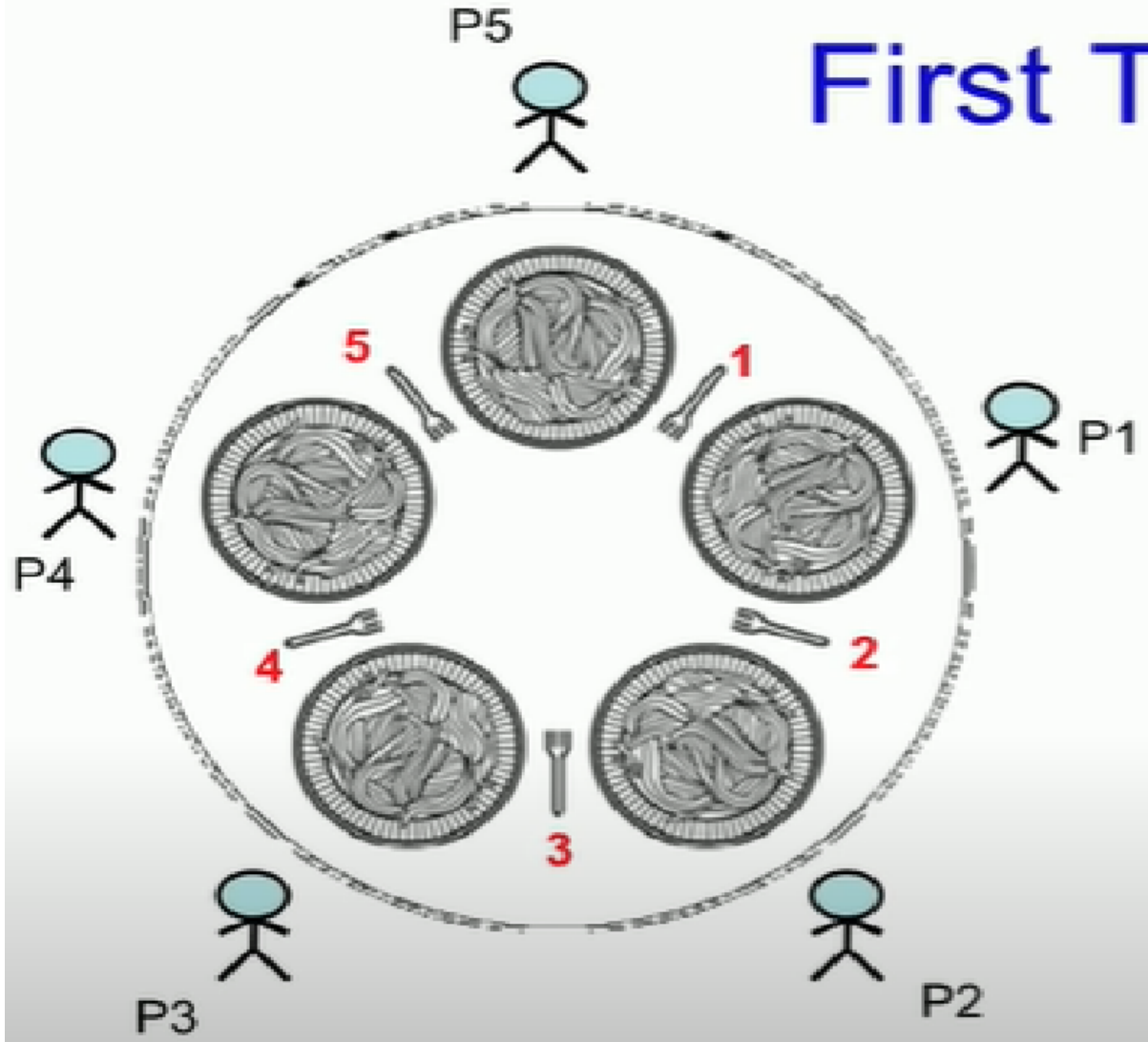
- In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
- In applications that have more readers than writers.

Dining Philosophers Problem



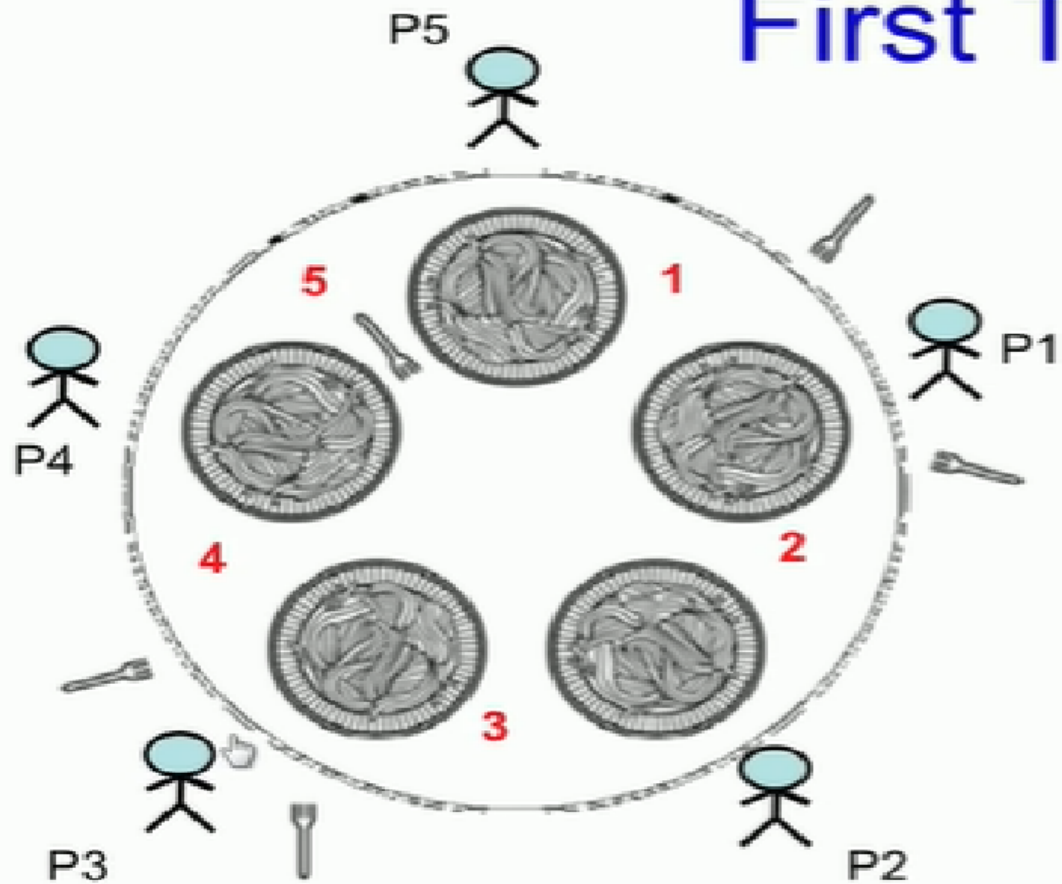
- **Philosophers either think or eat**
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- Problem Statement : Develop an algorithm where no philosopher starves.

First Try




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

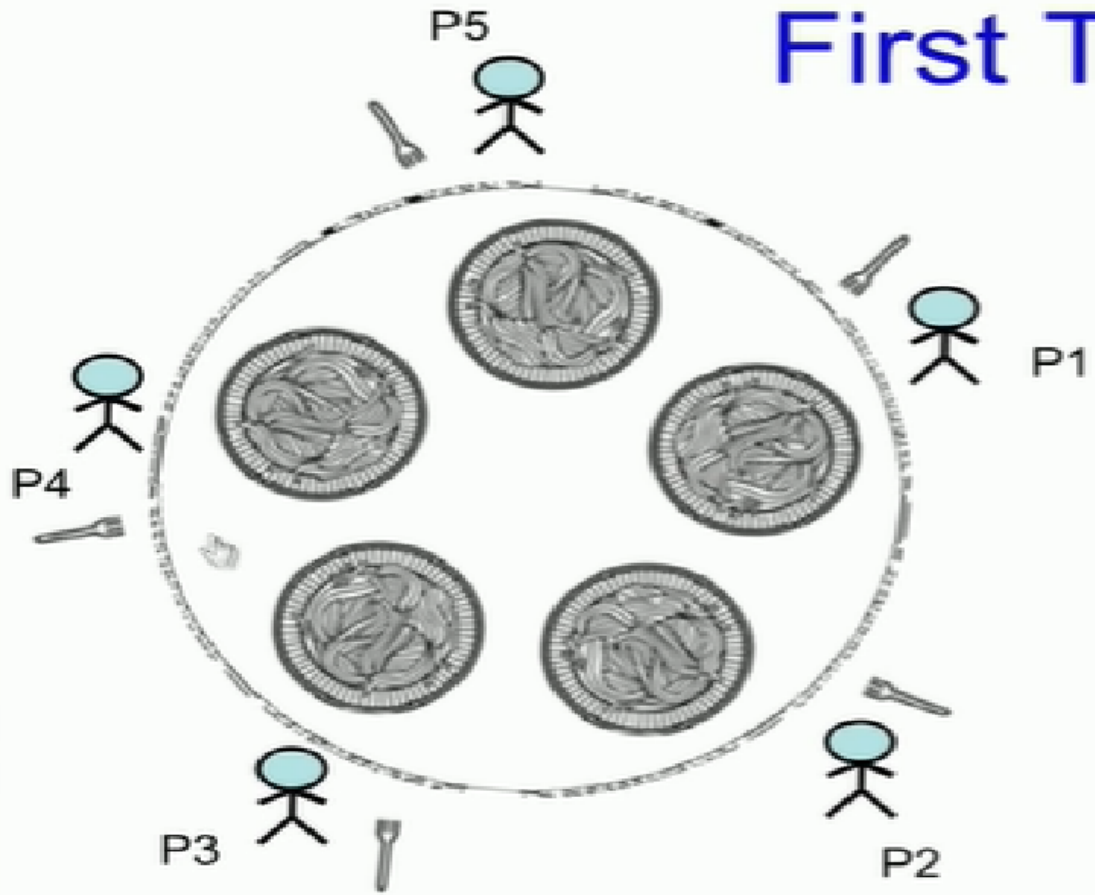
First Try




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

What happens if only philosophers P1 and P3 are always given the priority?
P4, P5, and P2 starves… so scheme needs to be fair

First Try

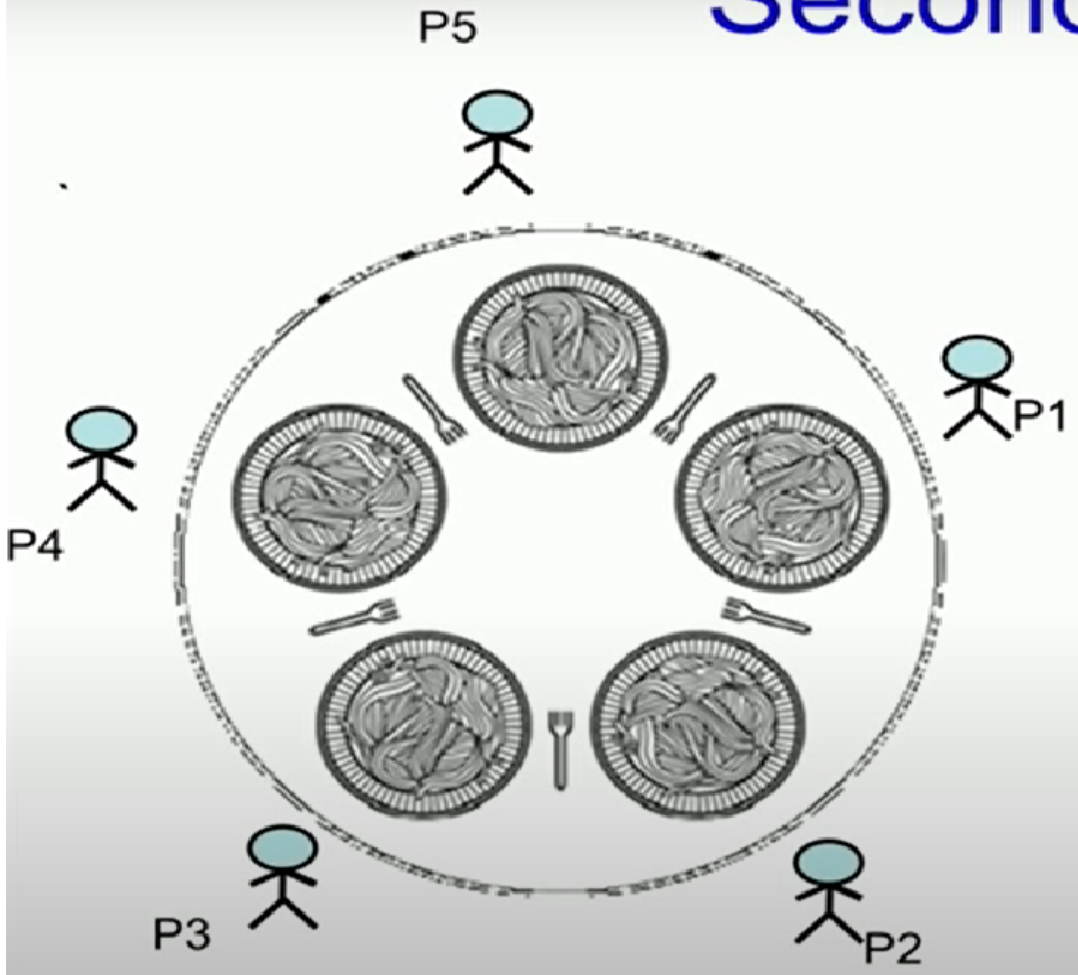



```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

What happens if all philosophers decide to pick up their right forks at the same time?

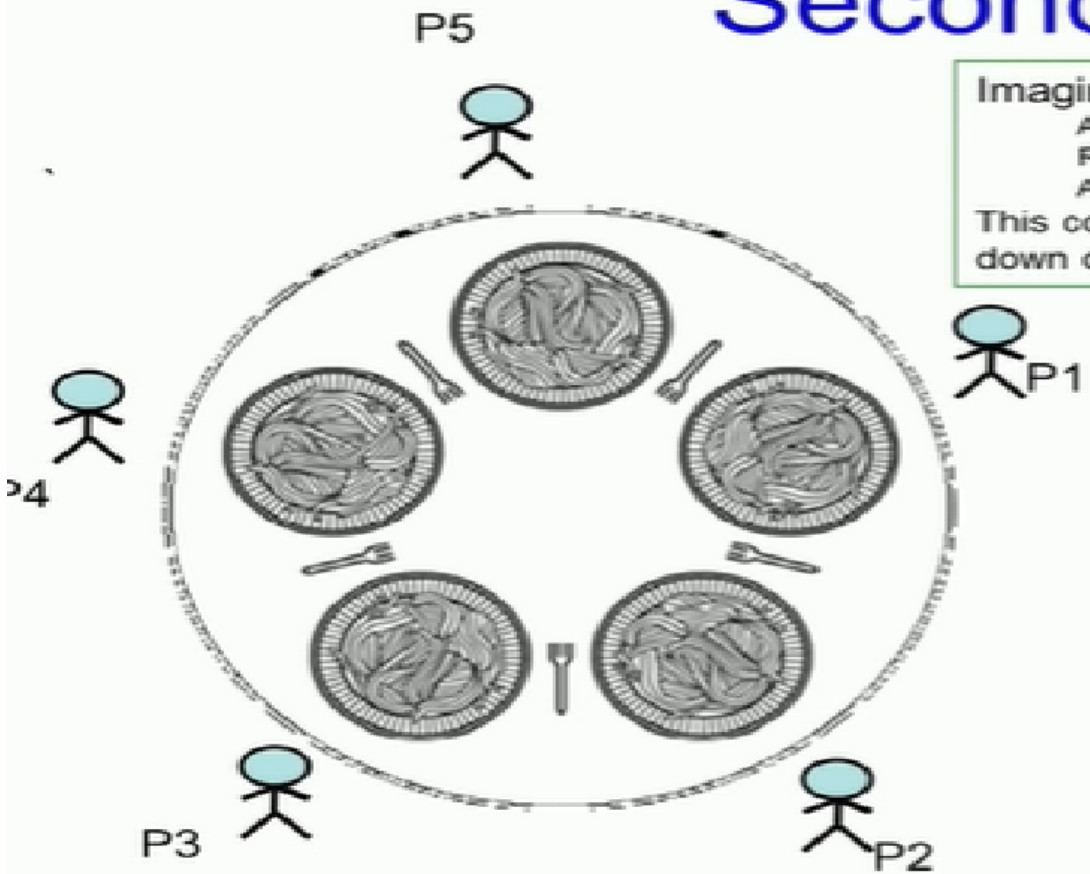
Possible starvation due to deadlock

Second try




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think();
      take_fork(Ri);
      if (available(Li){
        take_fork(Li);
        eat();
        put_fork(Ri);
        put_fork(Li);
      }else{
         put_fork(Ri);
         sleep(T)
      }
```

Second try

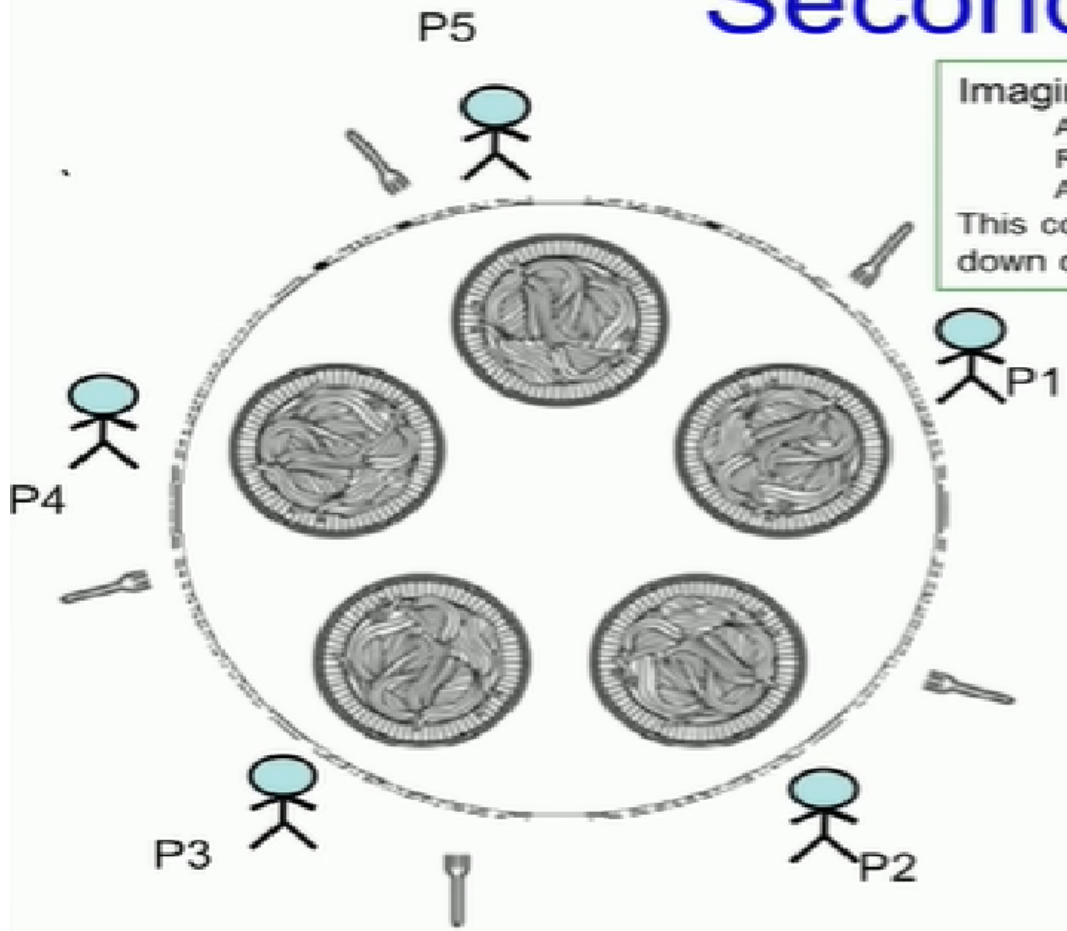


Imagine,
- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
    }else{
       put_fork(Ri);
       sleep(T)
    }
}
```

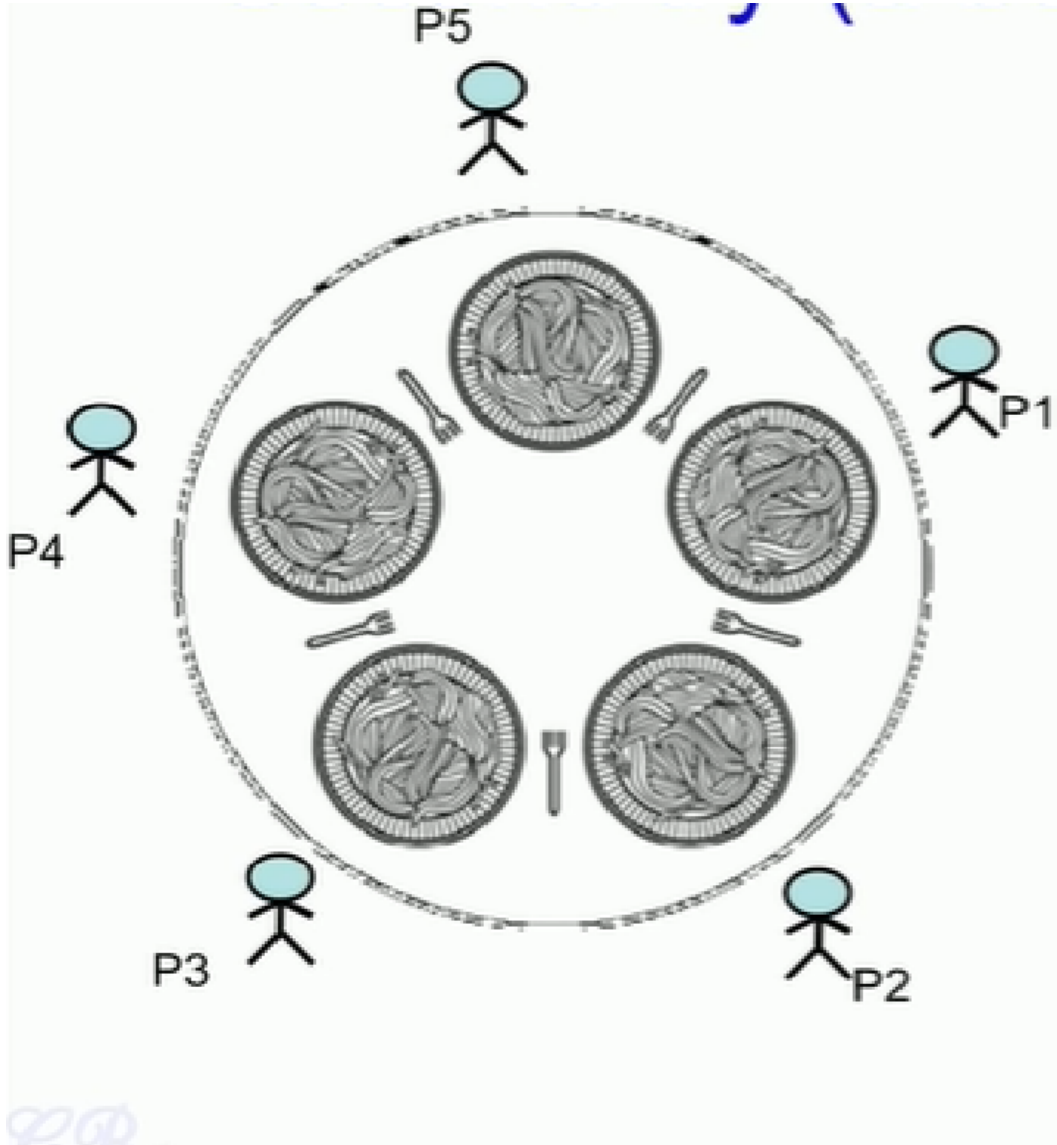
Second try



Imagine,
- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
        take_fork(Li);
        eat();
        put_fork(Ri);
        put_fork(Li);
    }else{
        put_fork(Ri);
        sleep(T)
    }
}
```




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think();
      take_fork(Ri);
      if (available(Li){
        take_fork(Li);
        eat();
        put_fork(Li);
        put_fork(Ri);
      }else{
          put_fork(Ri);
          sleep(random_time);
      }
}
```

- The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal () operation on the appropriate semaphores.
- Thus, the shared data are

  semaphore chopstick [5] ;

where all the elements of chopstick are initialized to i.

- The structure of philosopher i is shown in Figure below:

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

            // eat

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

            //think

}while (TRUE);
```

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Monitors

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect.

Examples

- Let us review the semaphore solution to the critical section problem.
- All processes share a semaphore variable mutex, which is initialized to 1.
- Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

- Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
..
critical section
..
wait(mutex);
```

- In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
critical section
wait(mutex);
```

- In this case, a deadlock will occur.
- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Usage

- Monitor is A type, or abstract data type, encapsulates private data with public methods to operate on that data.
- A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

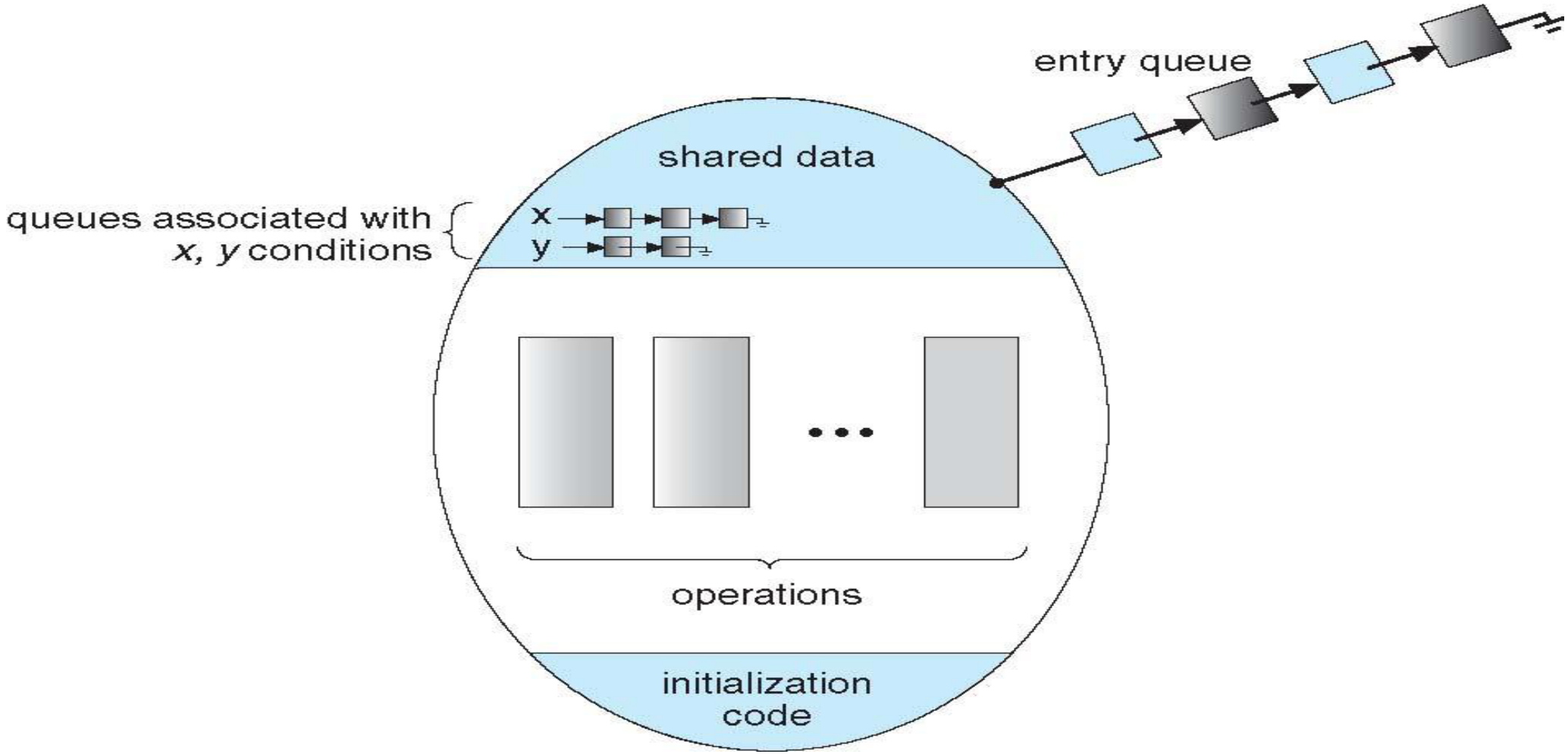
- The syntax of a monitor is shown in Figure

```
monitor monitor name
{
 //shared variable declarations

procedure P1( . . . ) {
….
}
procedure P2 ( . . . ) {

}
.
.
.
procedure Pn ( . . . ) {
….
}

initialization code (. . . ) {
…
….
}
}
```

- The monitor construct ensures that only one process at a time can be active within the monitor.
- However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.
- programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

  example: condition x, y;
- The only operations that can be invoked on a condition variable are wait( ) and signal( ).
- The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x. signal( ) ;
- The x. signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed.

Monitor with Condition Variables



- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide

Monitor Solution to Dining Philosophers problem

```
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING)
            self[i].wait;
}

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

Solution to Dining Philosophers (Cont.)

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

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

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0/ on which the signaling processes may suspend themselves.
- An integer variable next count is also provided to count the number of processes suspended on next.

- Thus, each external procedure F is replaced by

```
wait(mutex);
     body of  F
if (next_count > 0)
     signal(next);
else
     signal(mutex);
```

Mutual exclusion within a monitor is ensured.

- We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x_sem and an integer variable x_count; both initialized to 0. The operation x .wait () can now be implemented as

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex );
wait (x_sem) ;
x_count--;
```

Monitor Implementation (Cont.)

- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
  access the resurce;
    ...

R.release;
```

- Where R is an instance of type `ResourceAllocator`

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

Thank you