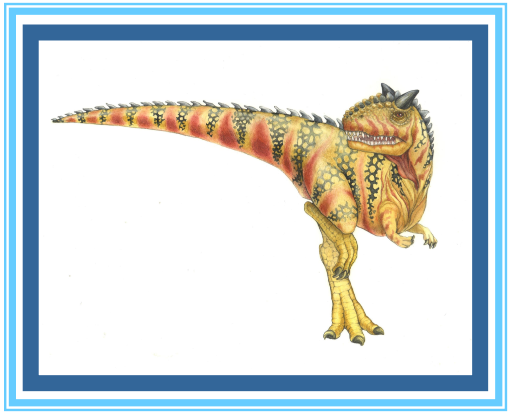
Unit 3: Process Synchronization

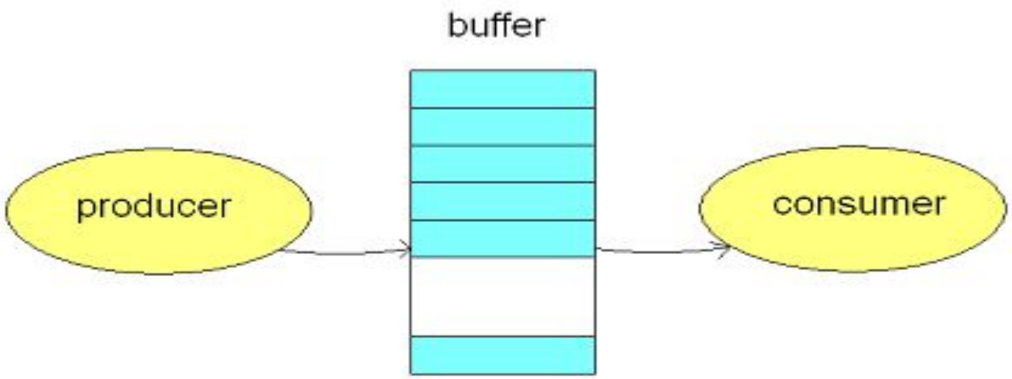


Cooperating Processes

- **Independent** process cannot affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages
- Concurrent access to shared data may result in data inconsistency

Background

- In this Unit, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

- Variables require to write producer consumer code
    - Conducer and Producer going to produce product of type item
    - Var buffer=array[0……n-1] of item
    - Pointer point to item in buffer

      int in=0……n-1

      int out=0….n-1
    - Nextproduce & nextconsume :item

Bounded-Buffer – Producer

```
Item nextproduce;
while (true) {
        /* Producer produce item in nextproduce  */
        while (( (in + 1) % BUFFER SIZE)  == out)
          ;   /* do nothing -- no free buffers */
         buffer[in] = nextproduce;
         in = (in + 1) % BUFFER SIZE;
    }
```

Bounded Buffer – Consumer

```
Item nextconsume;
while (true) {
        while (in == out)
              ; // do nothing -- nothing to consume

        // remove an item from the buffer & store in nextconsume
        nextconsume = buffer[out];
        out = (out + 1) % BUFFER SIZE;
    return item;
    }
```

Problem-

Only BUFFER SIZE-1 element can store

Producer

- One possibility is to add an integer variable counter, initialized to 0.
- Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer

```
while (true) {
      /*  produce an item and put in nextProduced


       while (count == BUFFER_SIZE)
        ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer

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

Race Condition

- count++ could be implemented as

```
  S0:-  register1 = count
  S1:- register1 = register1 + 1
  S2:- count = register1
```

- count-- could be implemented as

```
    S3:- register2 = count
    S4:- register2 = register2 - 1
    S5:-count = register2
```

- S0,S1,S3,S4,S2,S5
- S3,S4,S0,S1,S5,S2

What should be the answer?????

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = count {register1 = 5}

S1: producer execute register1 = register1 + 1 {register1 = 6}

S3: consumer execute register2 = count {register2 = 5}

S4: consumer execute register2 = register2 - 1 {register2 = 4}

S2: producer execute count = register1 {count = 6 }

S5: consumer execute count = register2 {count = 4}

Several Process access and manipulate the same data concurrently and outcome of the execution depends on the particular order(order of execution) in which access take place ,is called **Race Condition**

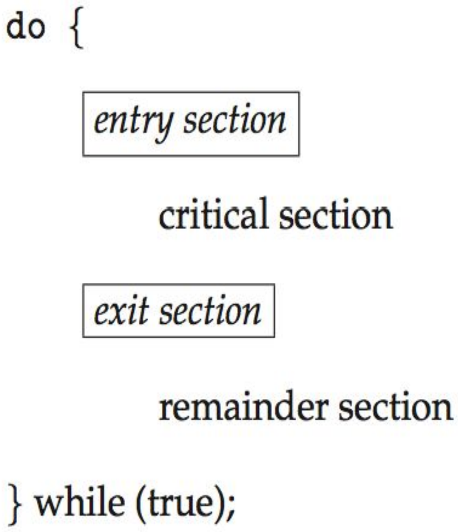
Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol that the processes can use to co-operate
- Each process must ask permission to enter **critical section**

- The section of code implementing this request is **entry section**
- The critical Section followed by **exit** and **remainder section**

Critical Section

- General structure of process P_i



```
do {

     entry section

          critical section

     exit section

          remainder section

} while (true);
```

Solution to Critical-Section Problem

A solution to critical section problem must satisfy following Requirements:

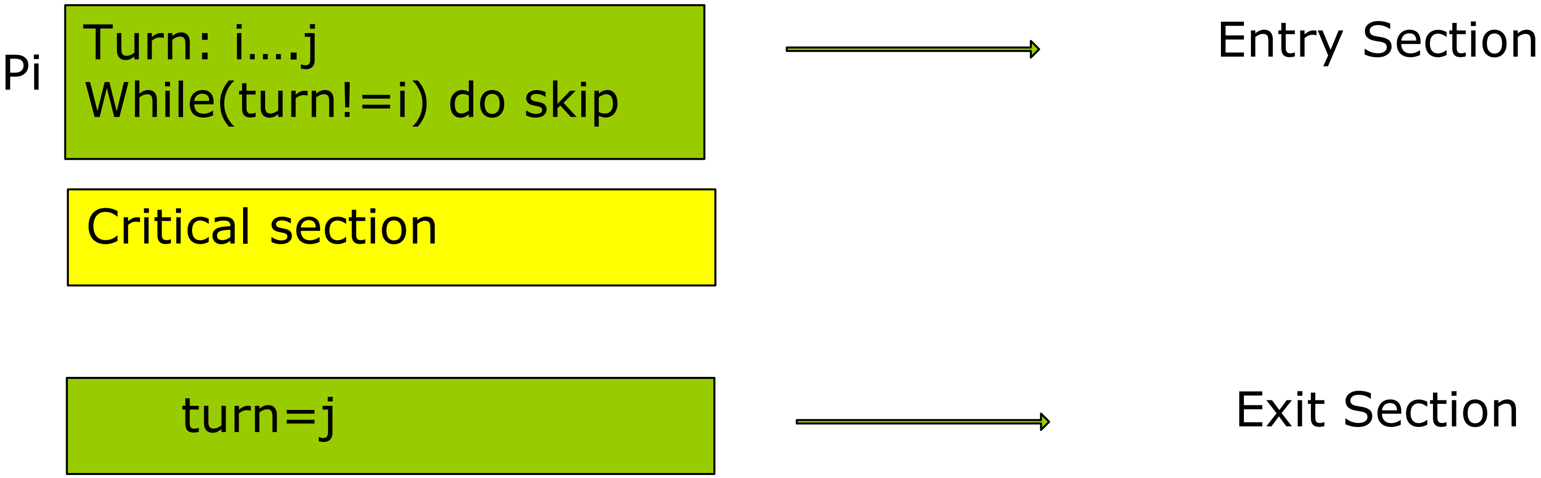
1. **Mutual Exclusion –** Only one process can execute CS at a time

2. **Progress –** only those processes that are interested in entering CS can participate in deciding which will enter its critical section next

3. **Bounded Waiting -** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections

Different approach

- 1st approach



Pi

Turn: i….j
While(turn!=i) do skip

→ Entry Section

Critical section

turn=j

→ Exit Section

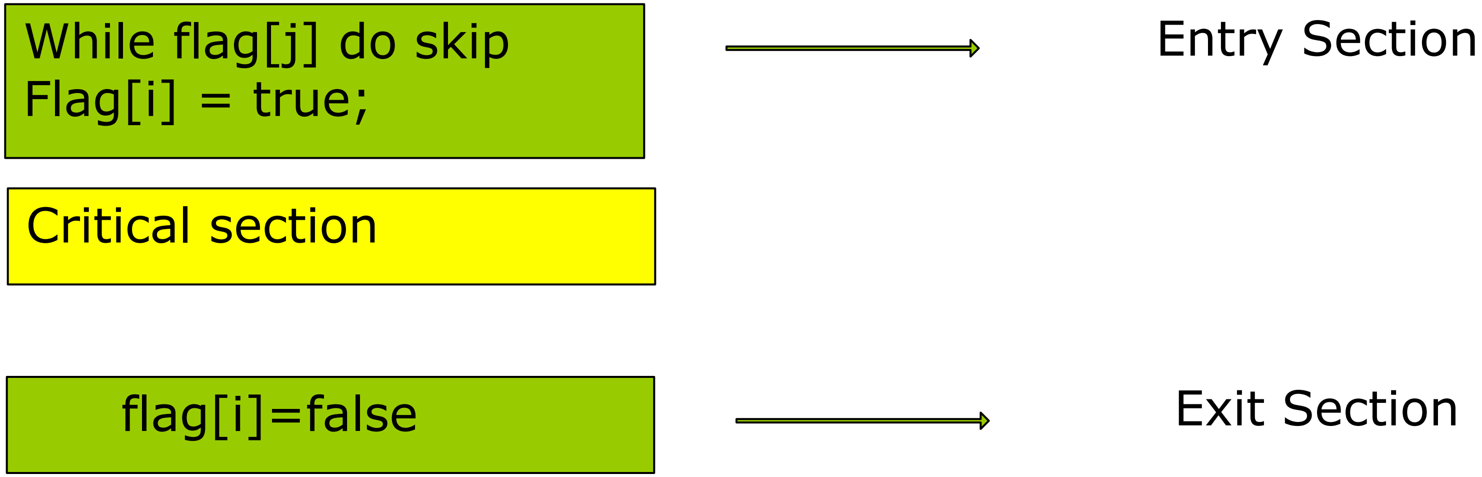
Requirement:-

✔ME

X Progress

Different approach

- **2^{nd} approach**



```
While flag[j] do skip
Flag[i] = true;
```

Entry Section

Critical section

```
flag[i]=false
```

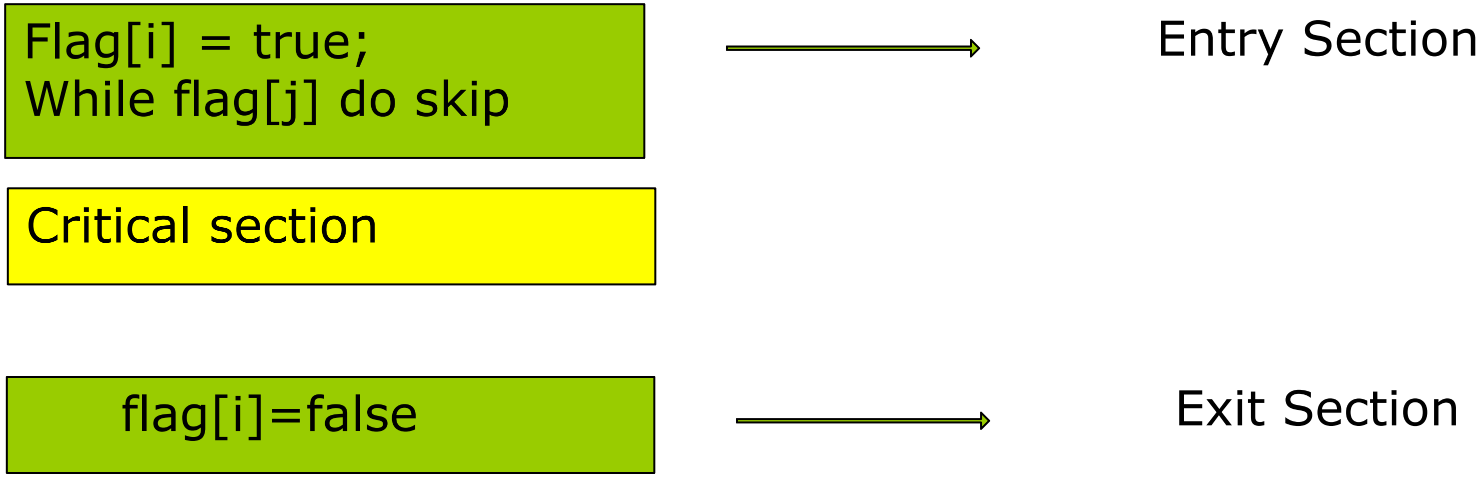
Exit Section

Requirement:-

X ME

Different approach

- 3rd approach



```
Flag[i] = true;
While flag[j] do skip
```

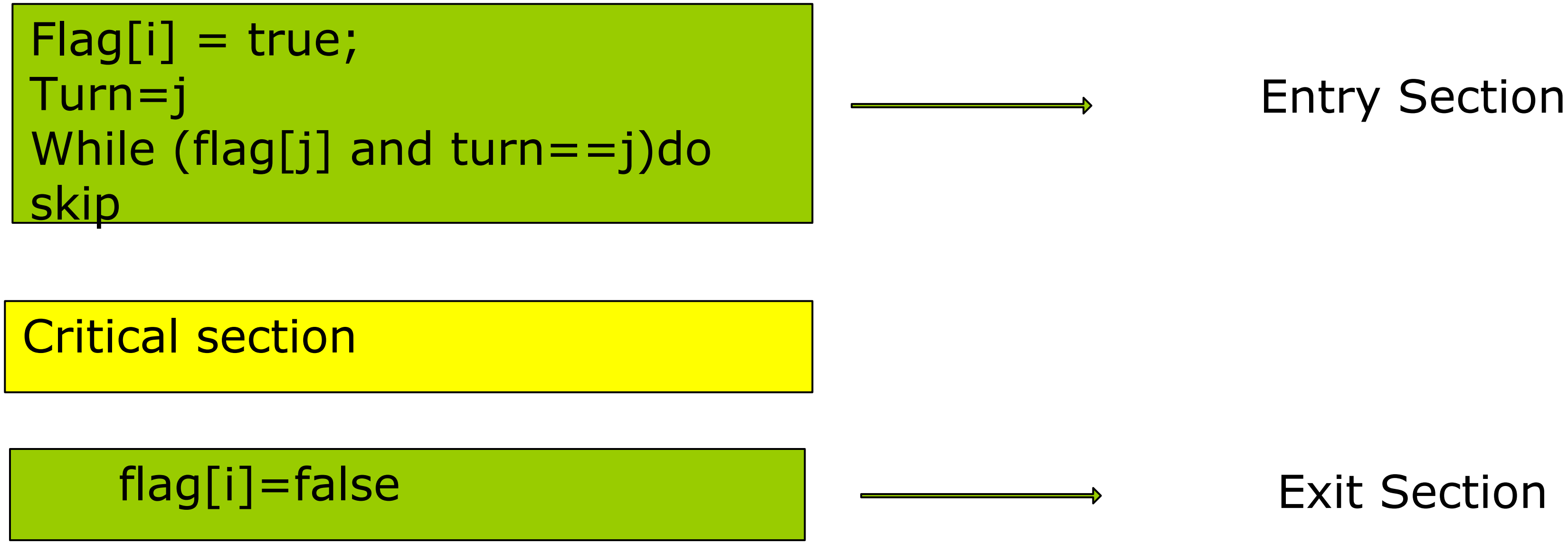
Entry Section

Critical section

```
flag[i]=false
```

Exit Section

Correct solution/Peterson’s Solution



```
Flag[i] = true;
Turn=j
While (flag[j] and turn==j)do
skip
```

Entry Section

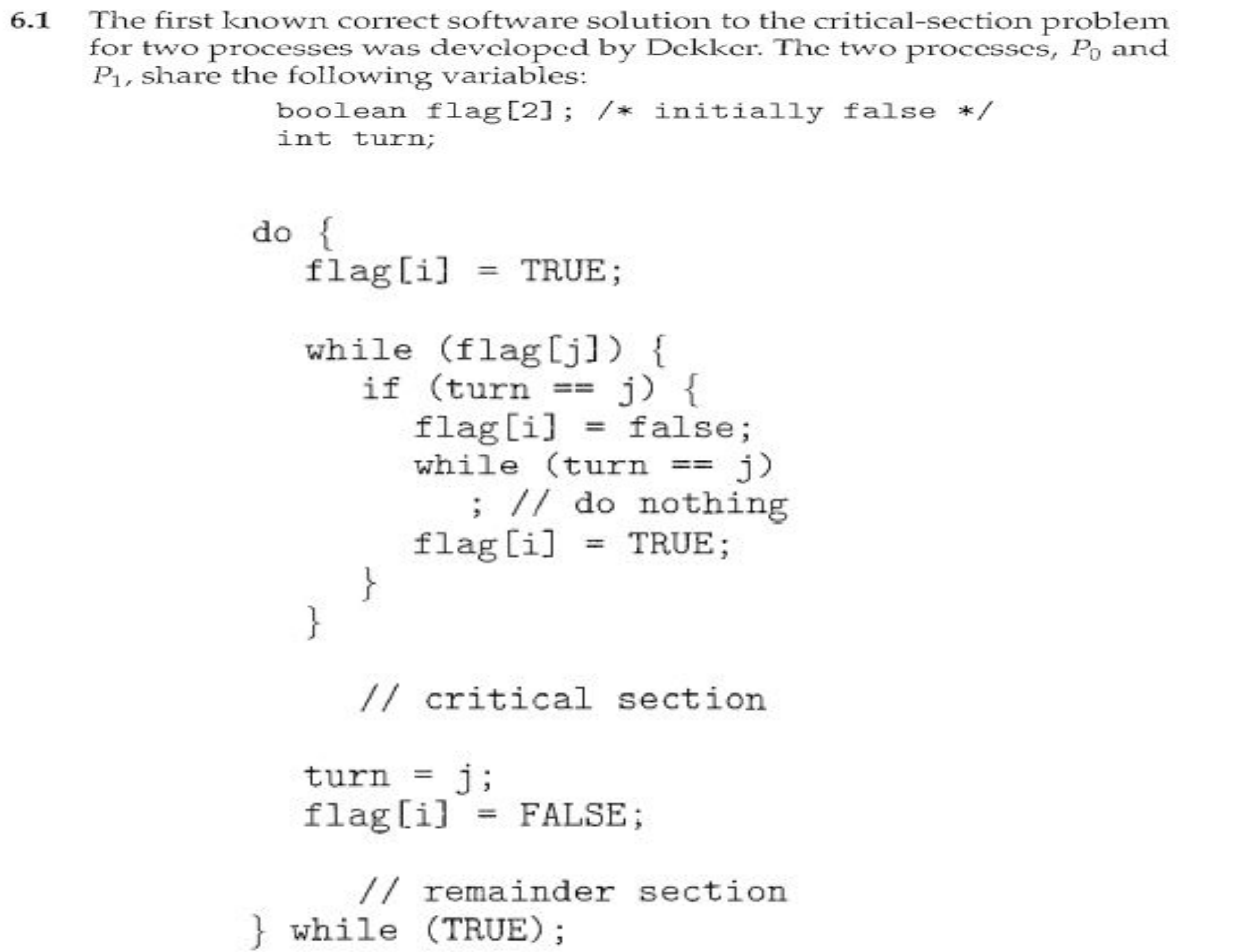
Critical section

```
      flag[i]=false
```

Exit Section

Peterson’s Soution

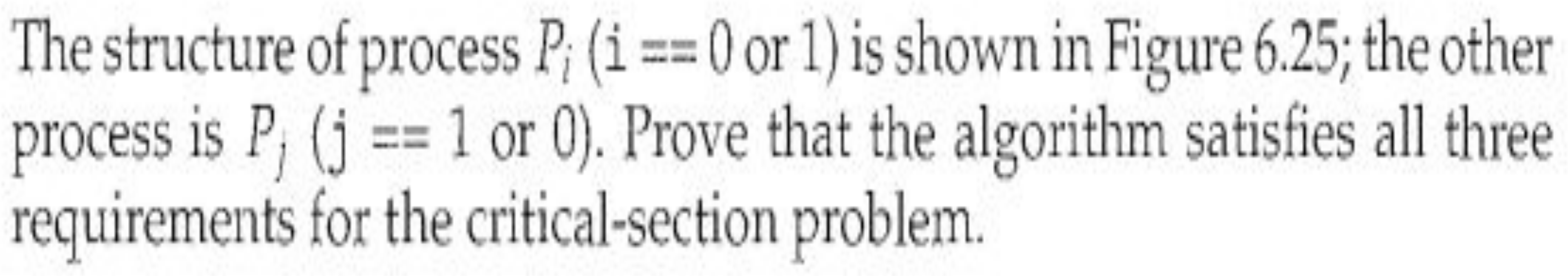
- A classic software based solution to critical section problem
- May not work correctly on modern computer architecture
- But it provide good algorithmic description to solve critical section problem and illustrate what complexity occur while addressing the requirement of mutual exclusion, Bounded waiting and progress



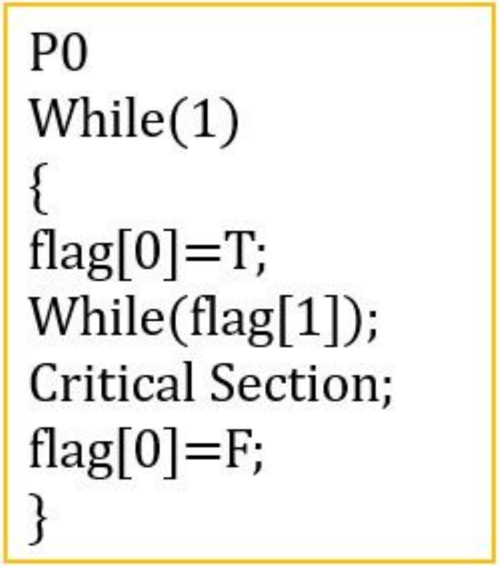
6.1 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

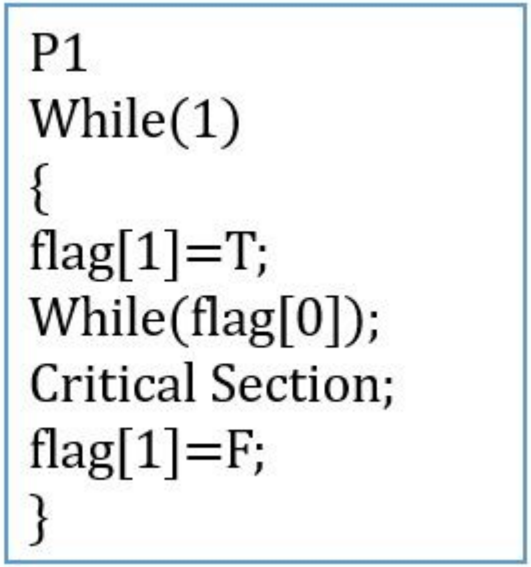
```
do {
  flag[i] = TRUE;

  while (flag[j]) {
    if (turn == j) {
      flag[i] = false;
      while (turn == j)
        ; // do nothing
      flag[i] = TRUE;
    }
  }

    // critical section

  turn = j;
  flag[i] = FALSE;

    // remainder section
} while (TRUE);
```



The structure of process P_i (i == 0 or 1) is shown in Figure 6.25; the other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.



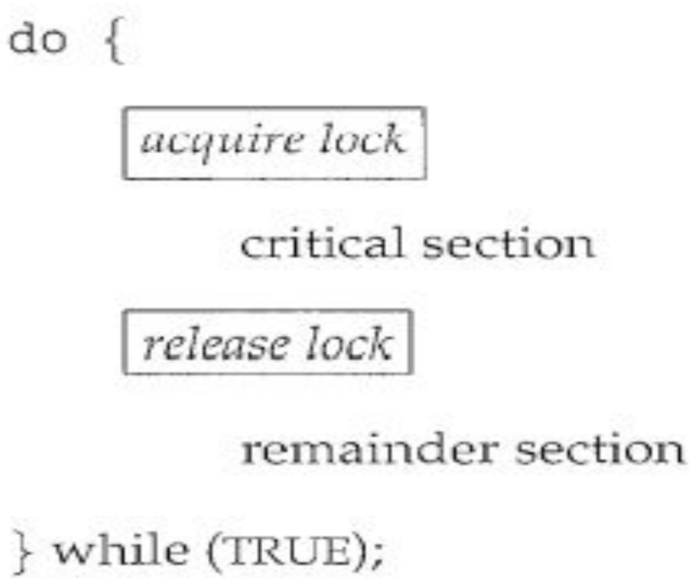
```
P0
While(1)
{
flag[0]=T;
While(flag[1]);
Critical Section;
flag[0]=F;
}
```



```
P1
While(1)
{
flag[1]=T;
While(flag[0]);
Critical Section;
flag[1]=F;
}
```

Synchronization Hardware

- Peterson's are not guaranteed to work on modern computer architectures.
- Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a lock.



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

Synchronization Hardware

- Definition:

```
boolean  TestAndSet (boolean target)
 {
     boolean  rv = target;
     target = TRUE;
     return  rv;
 }
```

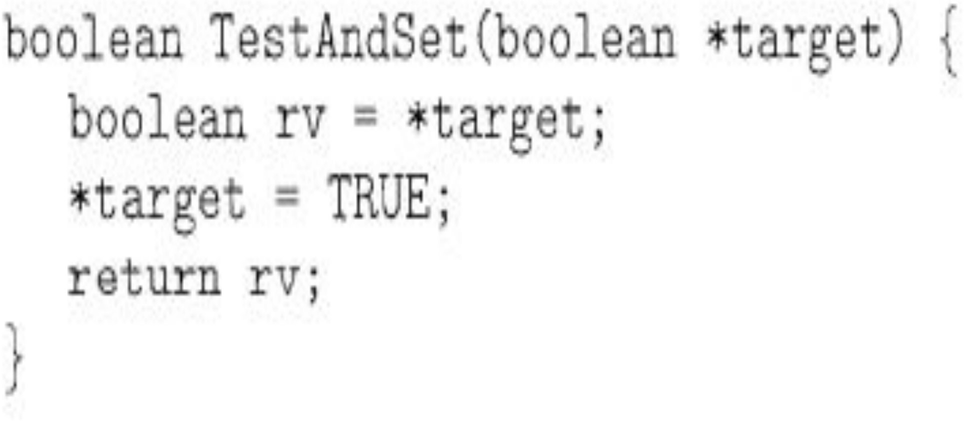
Must be executed atomically

- We discuss simple hardware instructions that are available on many systems and showing how they can be used effectively in solving the critical-section problem.
- Hardware features can make any programming task easier and improve system efficiency.
- Many modern computer systems therefore provide special **hardware instructions** that allow us either to **test** and **modify** the content atomically
- It work like one uninterruptible unit.
- These special instructions used to solve the critical-section problem

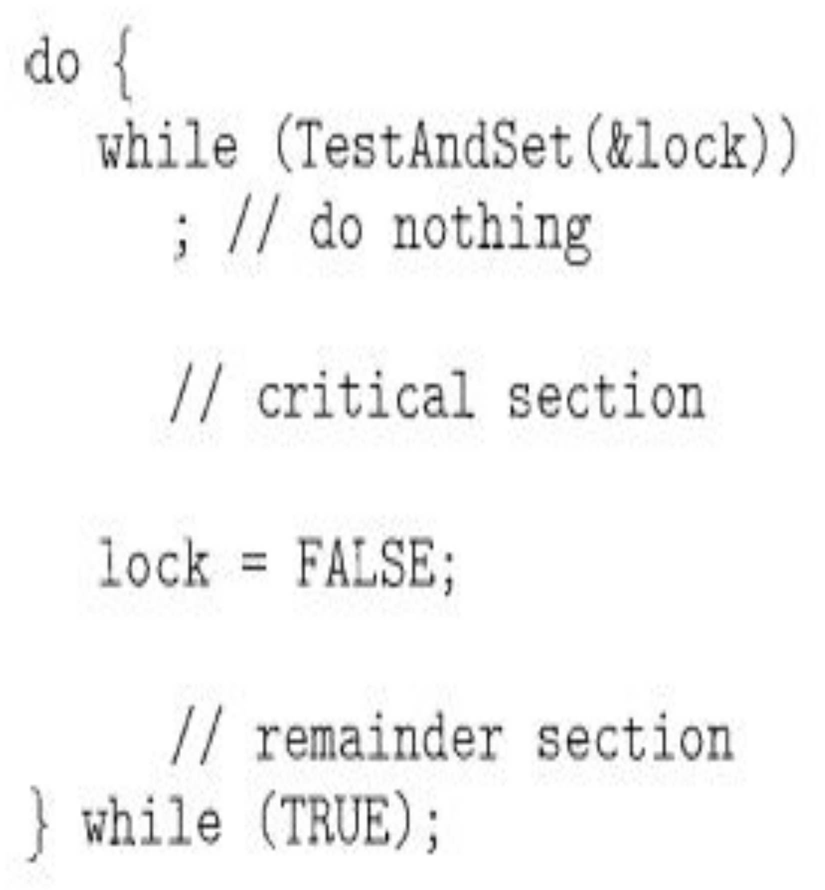
TestAndSet ()

- The important characteristic of this instruction is that it is executed atomically.
- Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P; is shown

Boolean lock=false
TestAndSet(&lock)



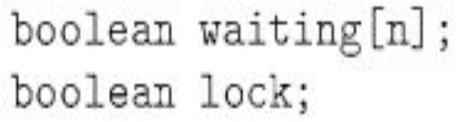
```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```



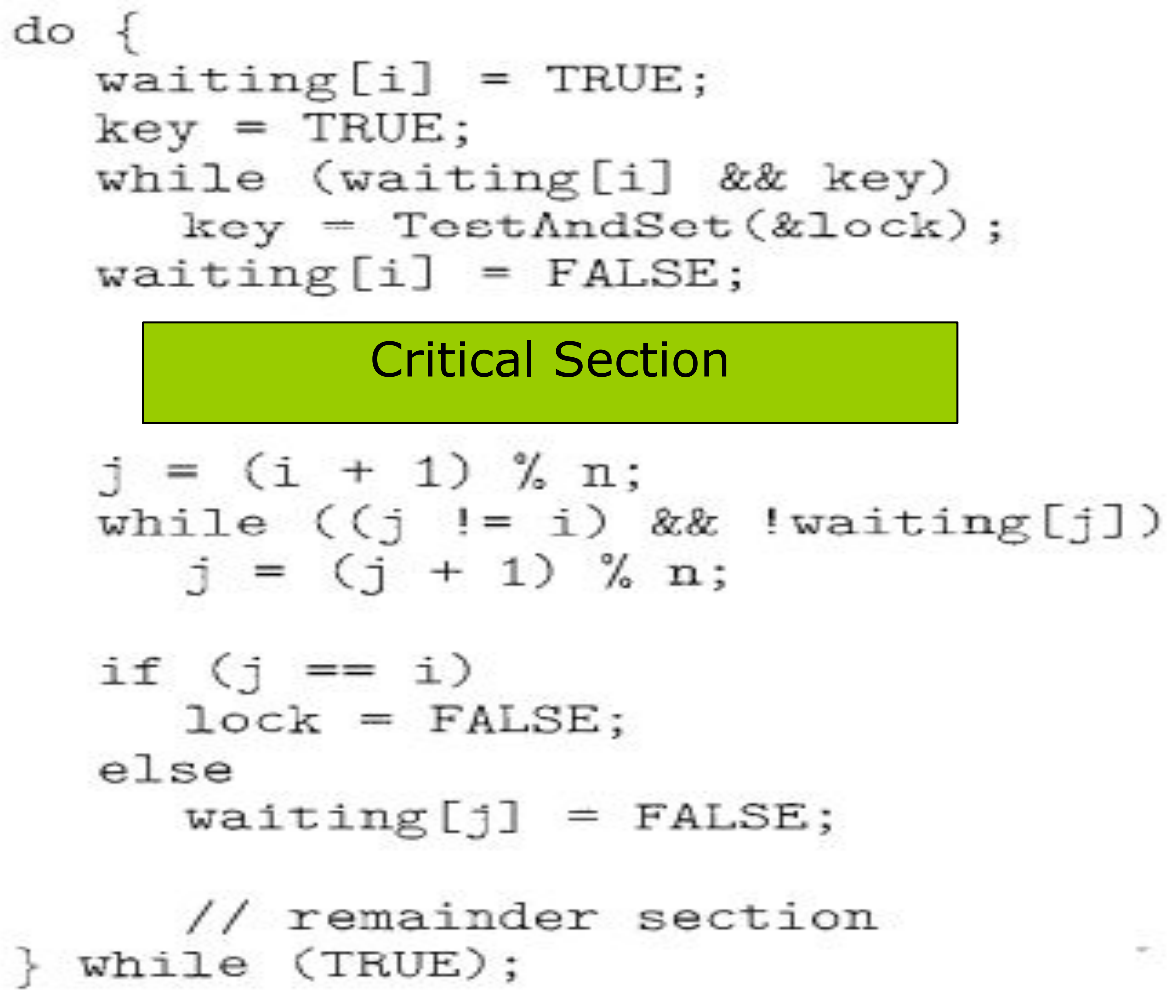
```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Not Satisfy Progress and bounded waiting

Correct solution



```
boolean waiting[n];
boolean lock;
```



```
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);
```

Bounded-waiting Mutual Exclusion & Progress with TestandSet()

```
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);
```

Entry Section

Exit Section

Semaphore

- Semaphore is proposed by Edsger Dijkstra ,is technique to manage concurrent processes by using simple integer value, which is known as semaphore
- Semaphore is simply a variable which is non-negative and shared between thread. This variable is used to solve the critical section problem and to achieve process synchronization in multiprocessing environment
- A semaphore S is a integer variable that can access through two standard atomic operation Wait() and Signal().
- **wait ->**P (from the Dutch word proberen, means "to test"); **signal->**V (from the Dutch word verhogen, means "to increment").

```
• wait (S) {
          while S <= 0
               ; // no-op
            S--;
    }

• signal (S) {
       S++;
   }
```

- All 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.

Two Type of Semaphore

- **Binary**:-The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
- **Counting:-**The value of a counting semaphore can range over an unrestricted domain ie Counting semaphores can be used to control access to a given resource consisting of a multiple instances

Exercise

- P1 with a statement S1 and P2 with a statement S2 . Suppose we require that S2 be executed only after S1 has completed Provide solution using semaphore variable.

Synchronizing Processes using Semaphores

- Two processes:
  - P_1 and P_2
- Statement S_1 in P_1 needs to be performed before statement S_2 in P_2
- Need to make P_2 wait until P_1 tells it is OK to proceed

- Define a semaphore “synch”
  - Initialize synch to 0

- Put this in P_2:
  - wait(synch);
  - S_2;

- And this in in P_1:
  - S_1;
  - signal(synch);

Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
     wait (mutex);
          // Critical Section
      signal (mutex);
        // remainder section
} while (TRUE);
```

Disadvantage of Semaphore

- The main disadvantage of the semaphore definition given here is thatit 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 entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively
- This type of semaphore is also called spinlock a because the process "spins" while waiting for the lock.

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations
- 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.

Semaphore structure

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
Typedef  struct
{   int value;
    struct process *list;
}s;
```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

- Implementation of signal:

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

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

```
        P0                      P1
      wait (S);               wait (Q);
      wait (Q);               wait (S);
 .          .
 .          .
 .          .
      signal  (S);            signal (Q);
      signal (Q);             signal (S);
```

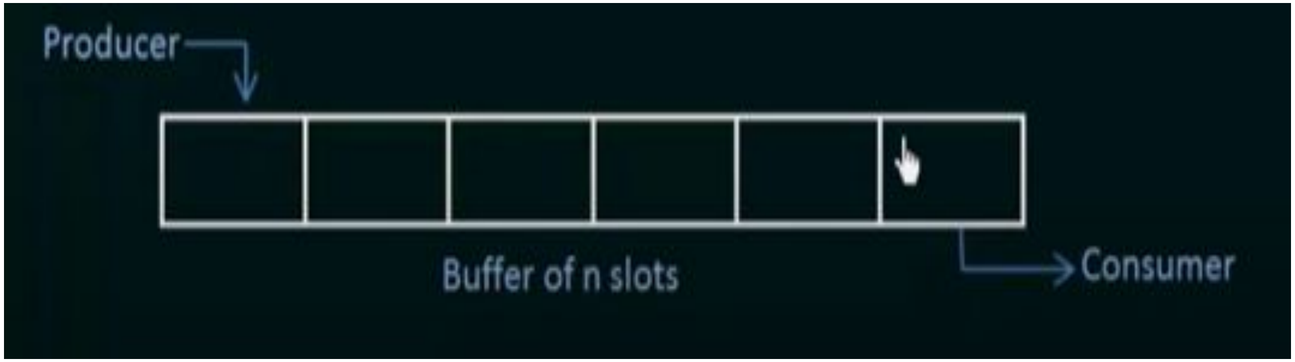
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- There is a buffer of n slot buffers and each slot is capable of holding one item.
- *There are two processes running , namely ,producer and consumer, which are operating on buffer*



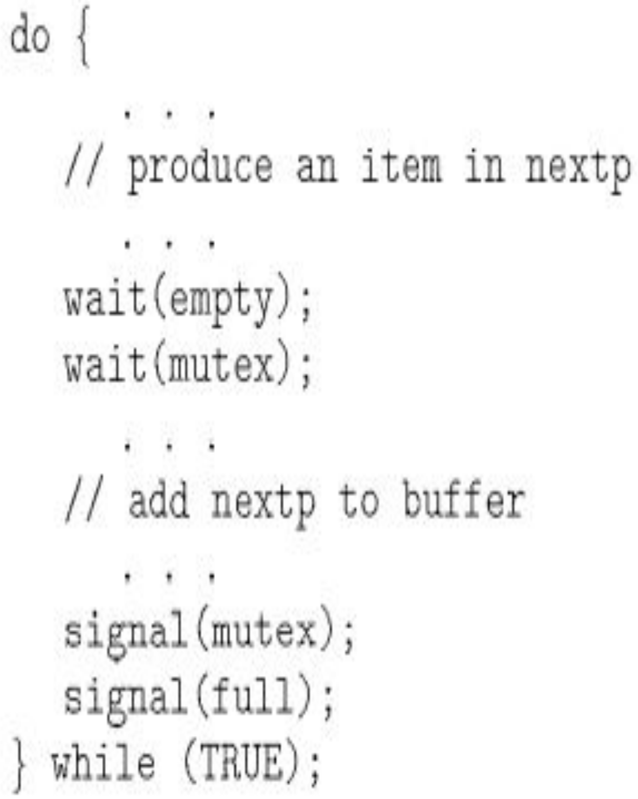
- *Producer tries to insert data into empty slot of the buffer*
- *Consumer tries to remove data from filled slot of the buffer*

Problem

- Producer must not insert data when the buffer is full
- Consumer must not remove data when buffer is empty
- Producer and consumer should not insert and remove data simultaneously

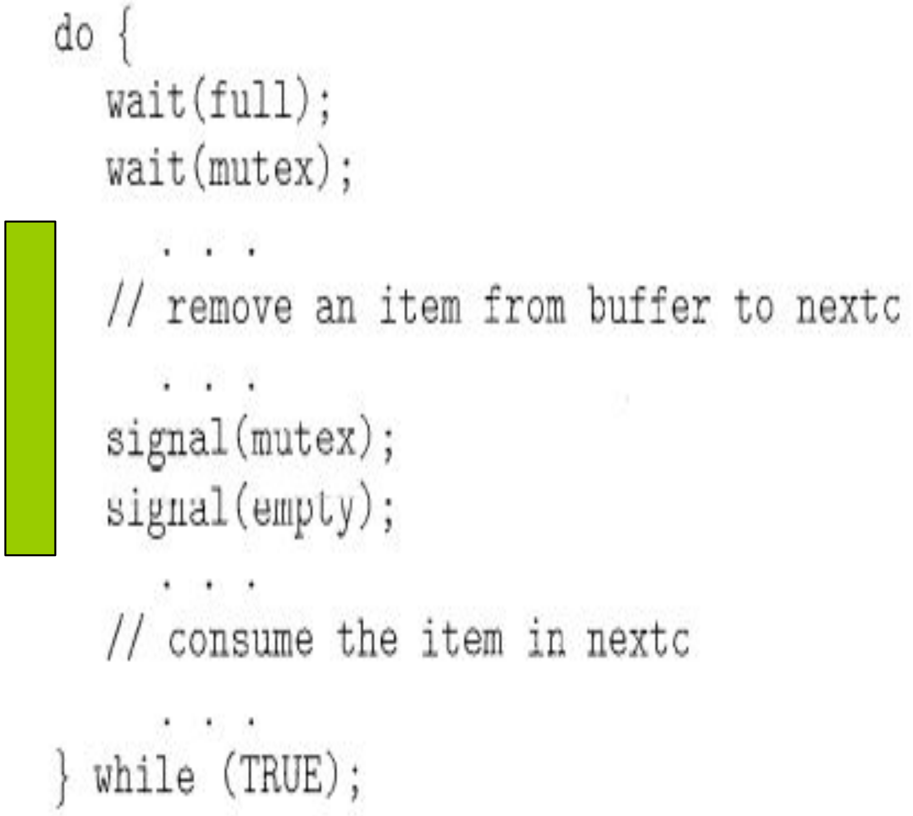
Solution to bounded buffer problem using semaphore

- **Mutex** ,binary semaphore which is uses to acquire and release lock ,initialize to 1
- **Empty**, a counting semaphore whose initial value is the number of slots in the buffer, since initially all slots are empty
- **Full**, counting semaphore whose initial value is 0



```
do {
     . . .
   // produce an item in nextp
     . . .
   wait(empty);
   wait(mutex);
     . . .
   // add nextp to buffer
     . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

Figure 6.10 The structure of the producer process.



```
do {
   wait(full);
   wait(mutex);
     . . .
   // remove an item from buffer to nextc
     . . .
   signal(mutex);
   signal(empty);
     . . .
   // consume the item in nextc
     . . .
} while (TRUE);
```

Figure 6.11 The structure of the consumer process.

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

Problem

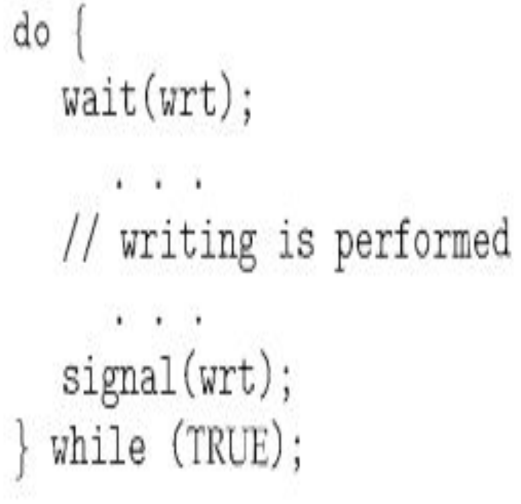
- Allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- To ensure that these difficulties do not arise, we require that writer have exclusive access to shared data

Readers-Writers Problem

Solution to Reader Writer problem using semaphore

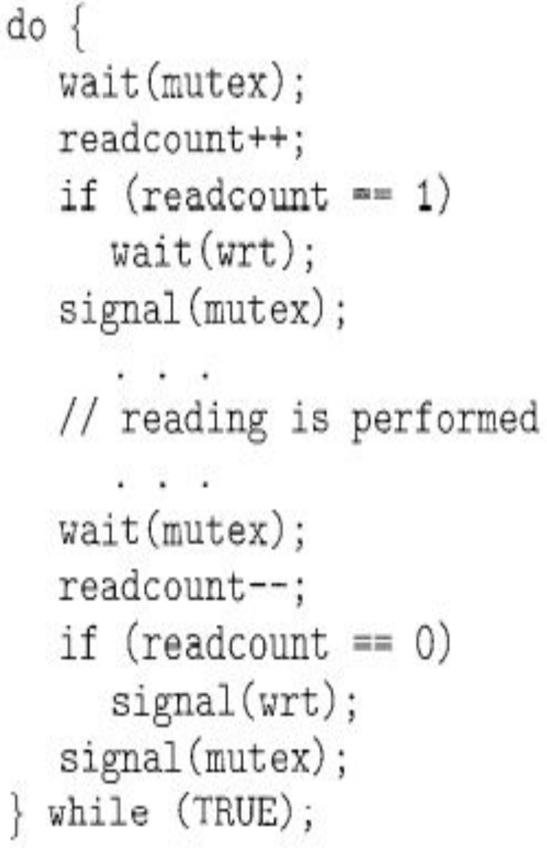
We make the use of two semaphore variable and an integer variable

1. **mutex** , a semaphore variable (initialized to 1) which is uses to ensure mutual exclusion when read count is updated ie when any reader enter and exit from critical section
2. **wrt** , a semaphore (initialize to 1) common to both reader and writer process
3. **readcount** is integer variable(initialize to 0) that keep track of how many processes are currently reading the object



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

Figure 6.12 The structure of a writer proce



```
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);
```

Figure 6.13 The structure of a reader process.

Readers-Writers Problem (Cont.)

- **The structure of a writer process**

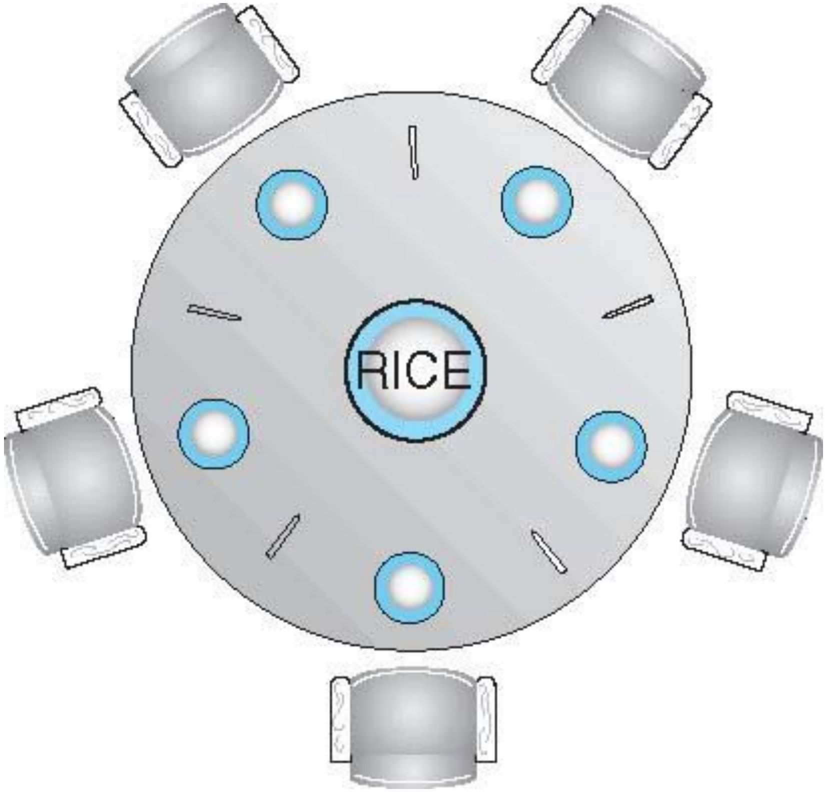
```
        do {
//write request fo CS
                wait (wrt) ;
                    //    writing is performed
// leave the critical section
                signal (wrt) ;
        } while (TRUE);
```

Readers-Writers Problem (Cont.)

- **The structure of a reader process**

```
do {
            wait (mutex) ;// the number of readers has now increase by one
            readcount ++ ;
            if (readcount == 1)
        wait (wrt) ;// this ensure that no writer can enter if there is even one
reader
            signal (mutex)// other reader can enter if current reader is inside CS
                 // reading is performed
            wait (mutex) ;
            readcount  - - ;// reader want to leave
            if (readcount  == 0)  // no reader left n CS
             signal (wrt) ; // Writer can enter
            signal (mutex) ; // reader leaves
    } while (TRUE);
```

Dining-Philosophers Problem



- **Philosopher have two work to do**
- **1. Think**:-He/she sit ideal
- **2. Eat:**-He/she tries to pickup the two chopstick which are closest to him/her.
- Philosopher will pick one chopstick at a time

Represent chopstick with **semaphore**

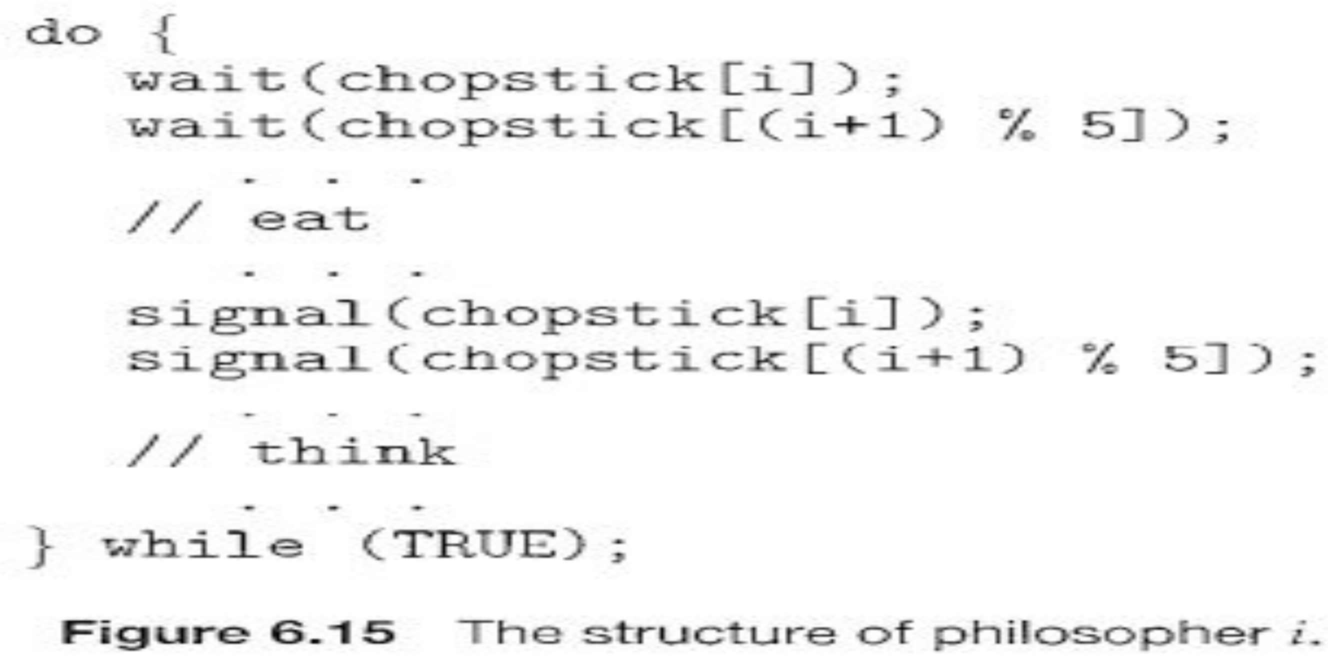
- Philosopher tries to grab chopstick by using wait() operation on semaphore variable
- Philosopher tries to release chopstick by using signal() operation on semaphore variable

Dining-Philosophers Problem (Cont.)

- 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);
```

What is the problem with the above?



```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

- 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 next.

- 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

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A abstract data type- or ADT- encapsulates private data with public methods to operate on that data. A monitor type is an ADT
- A monitor which 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.
- Only one process may be active within the monitor at a time

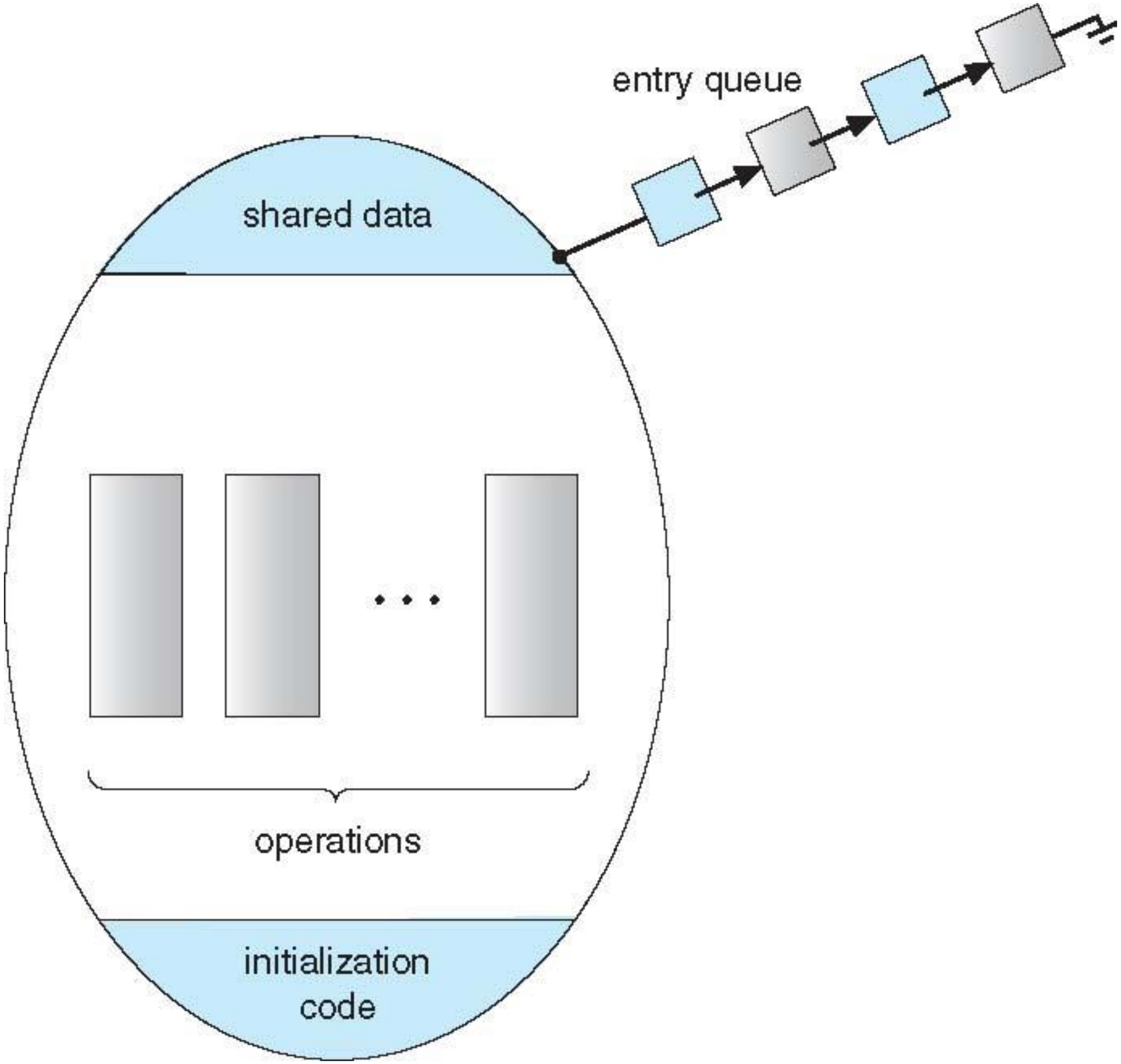
Monitors

Syntax of Monitor

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
        …
   procedure Pn (…) {……}
    Initialization code ( ….) { … }
        …
   }
}
```

- a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time is active within the monitor.

Schematic view of a Monitor



example

```
Monitor account
{
   double balance;
   Withdraw(amount)
  {
     balance=balance-amount
     return balance
  }
}
```

T1….T2…T3 Thread

Condition Variables

- However the monitor construct, as defined so far is not sufficiently powerful for modeling some synchronization schemes
- Conditional variable provide synchronization inside monitor
- If process want to sleep inside monitor or it allows awaiting process to continue, in that case conditional variable is used inside monitor
- Two operation can be perform on conditional variable(wait and signal)
- Wait operation:-If resources are not currently available then current process put to sleep .It release the lock for monitor
- Signal Operation:-Signal operation wakeup sleeping process

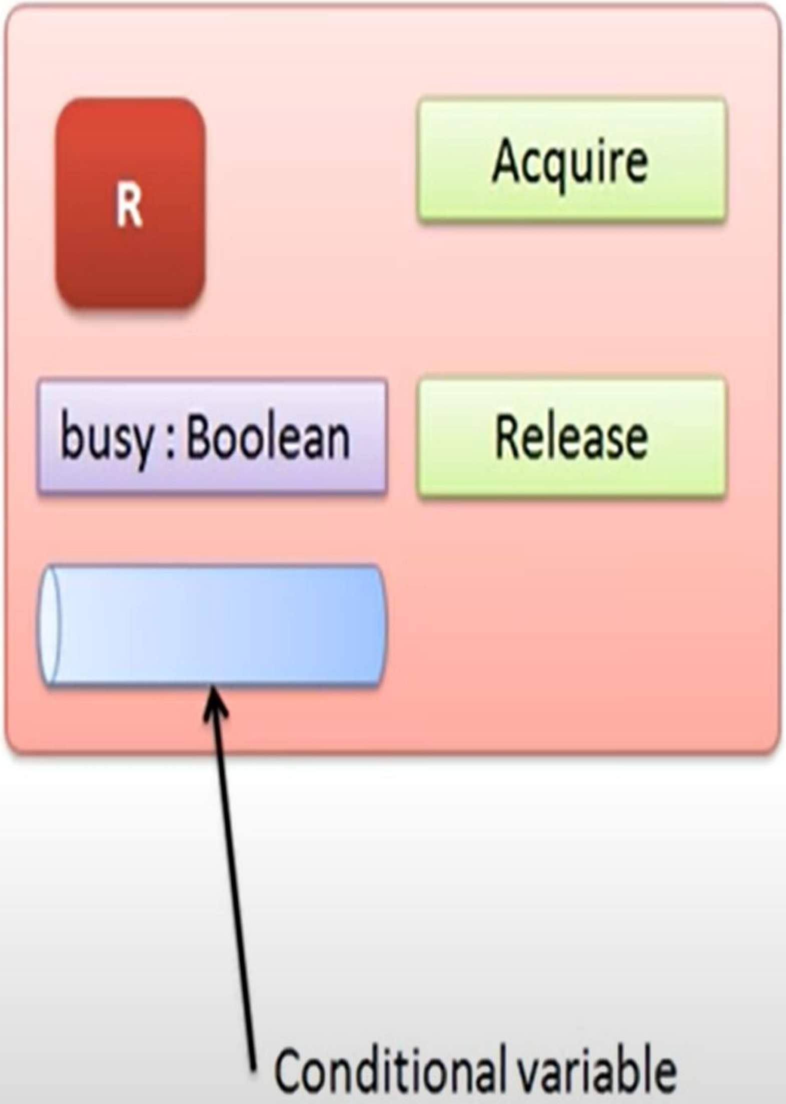
For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.

condition x, y;

Two operations on a condition variable:

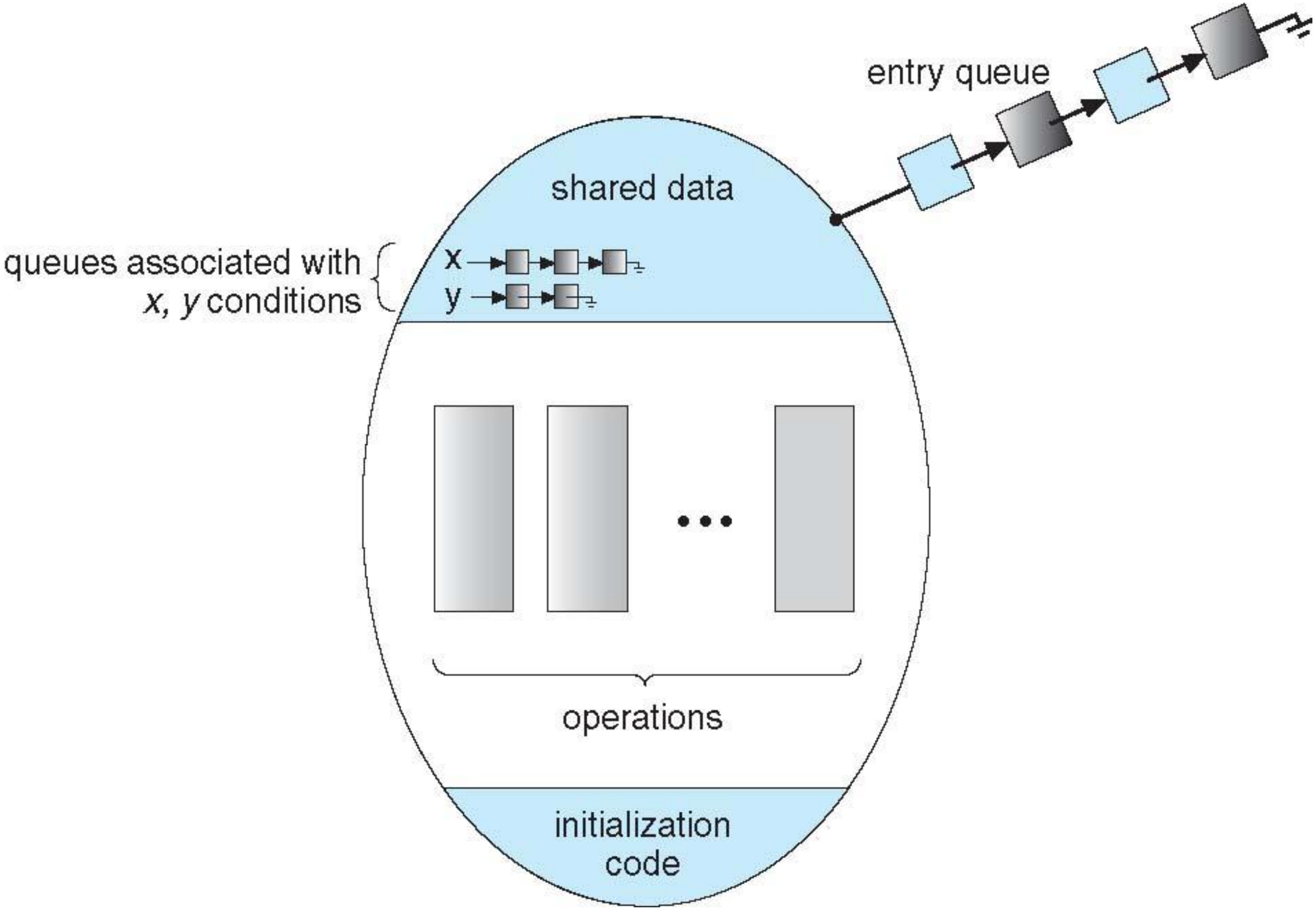
x.wait () –means that the process that invokes this operation is suspended until another process invoke x.signal() operation.

x.signal () – resumes one of processes (if any) that invoked x.wait ()



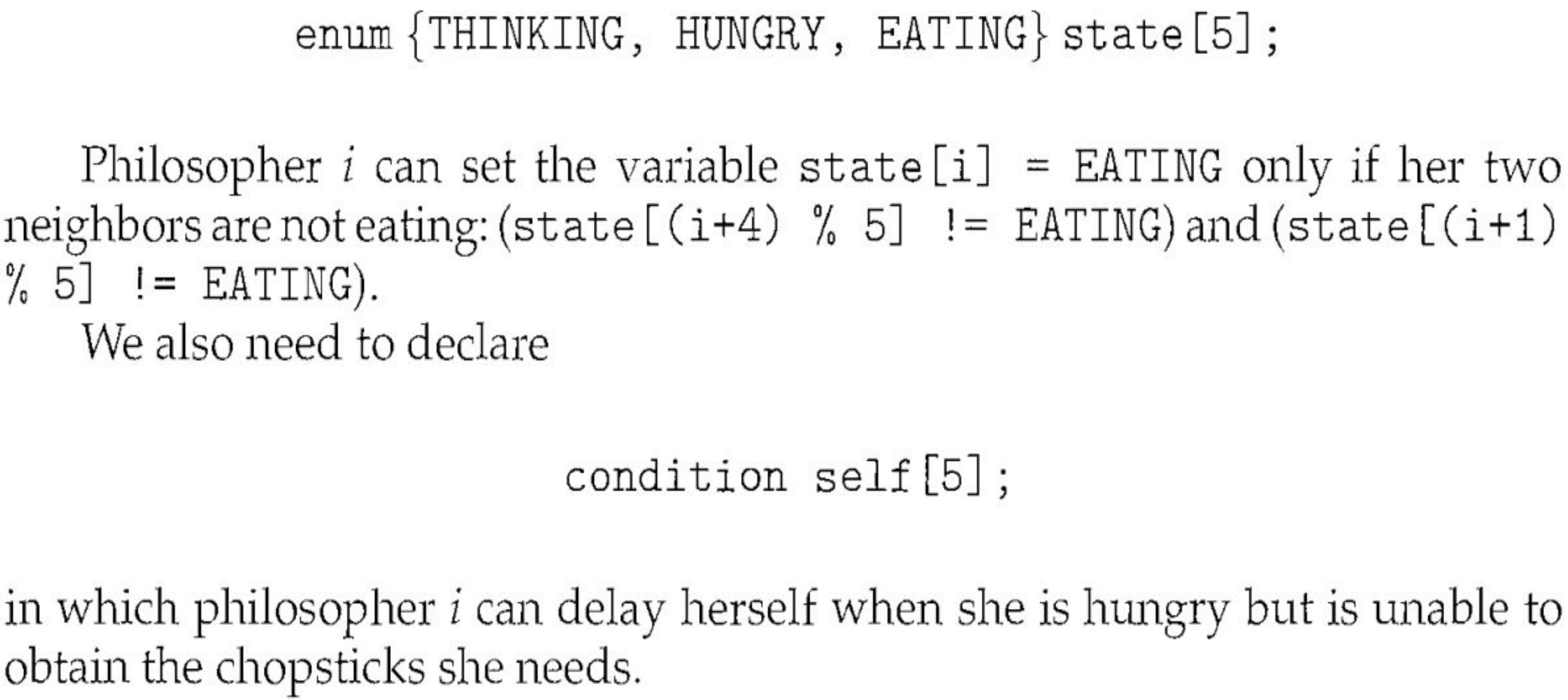
```
monitor single_resource
{
  boolean busy;
  condition nonbusy;

  Acquire()
  {
      if busy then nonbusy.wait
      else  busy=true
   }
 Release()
  {
     busy=false
     nonbusy.signal
  }
}
```

Monitor with Condition Variables



Dining-Philosophers Solution Using Monitors

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.
- For this purpose, we introduce the following data structure:



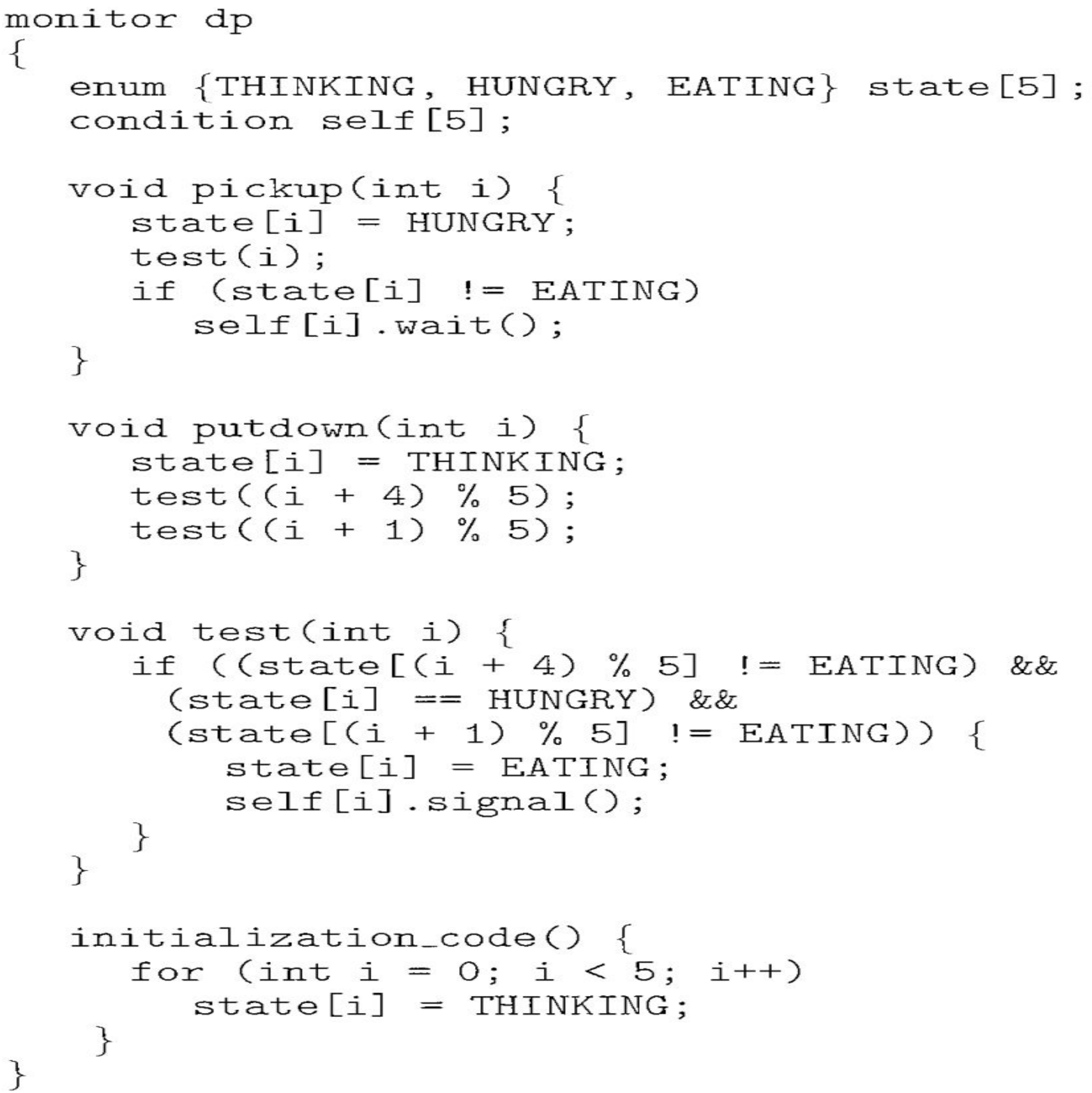
```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable `state[i] = EATING` only if her two neighbors are not eating: (`state[(i+4) % 5] != EATING`) and (`state[(i+1) % 5] != EATING`).

We also need to declare

```
condition self[5];
```

in which philosopher *i* can delay herself when she is hungry but is unable to obtain the chopsticks she needs.



```
monitor dp
{
   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((i + 4) % 5);
      test((i + 1) % 5);
   }

   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;
    }
}
```