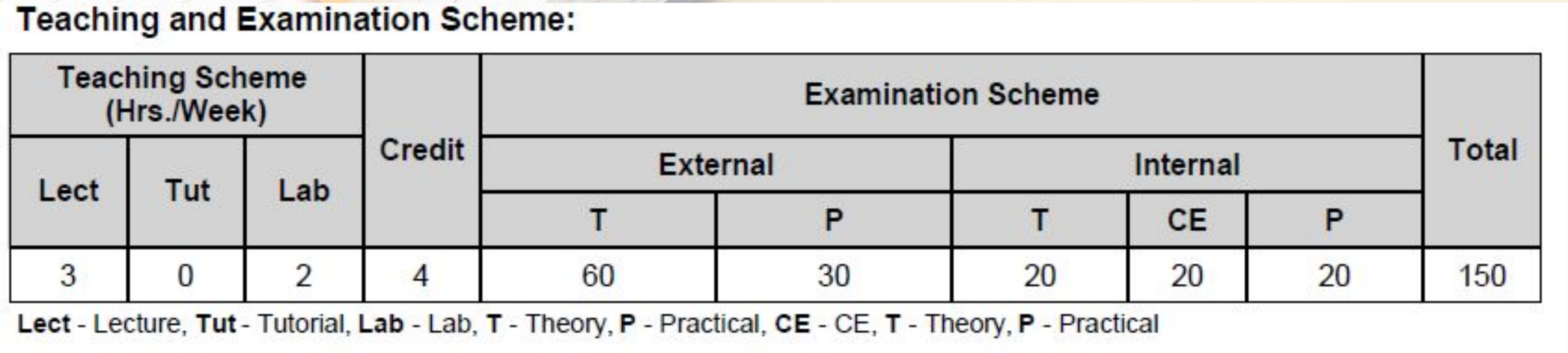
UNIT 3
INTER-PROCESS COMMUNICATION

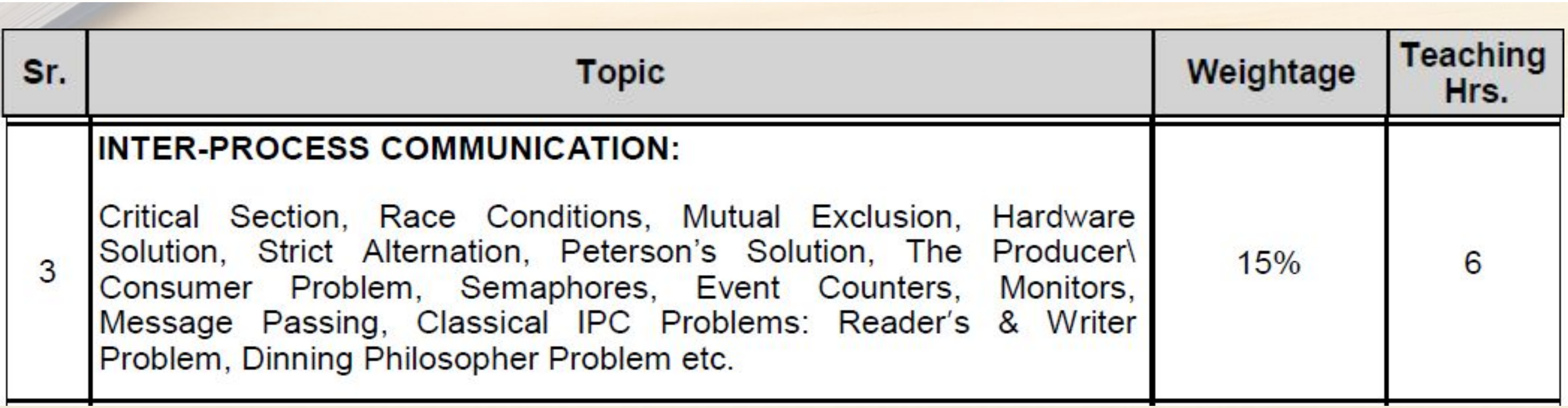
Syllabus



Teaching and Examination Scheme:

Teaching Scheme (Hrs./Week)			Credit	Examination Scheme					Total
Lect	Tut	Lab		External		Internal			
				T	P	T	CE	P	
3	0	2	4	60	30	20	20	20	150

Lect - Lecture, **Tut** - Tutorial, **Lab** - Lab, **T** - Theory, **P** - Practical, **CE** - CE, **T** - Theory, **P** - Practical



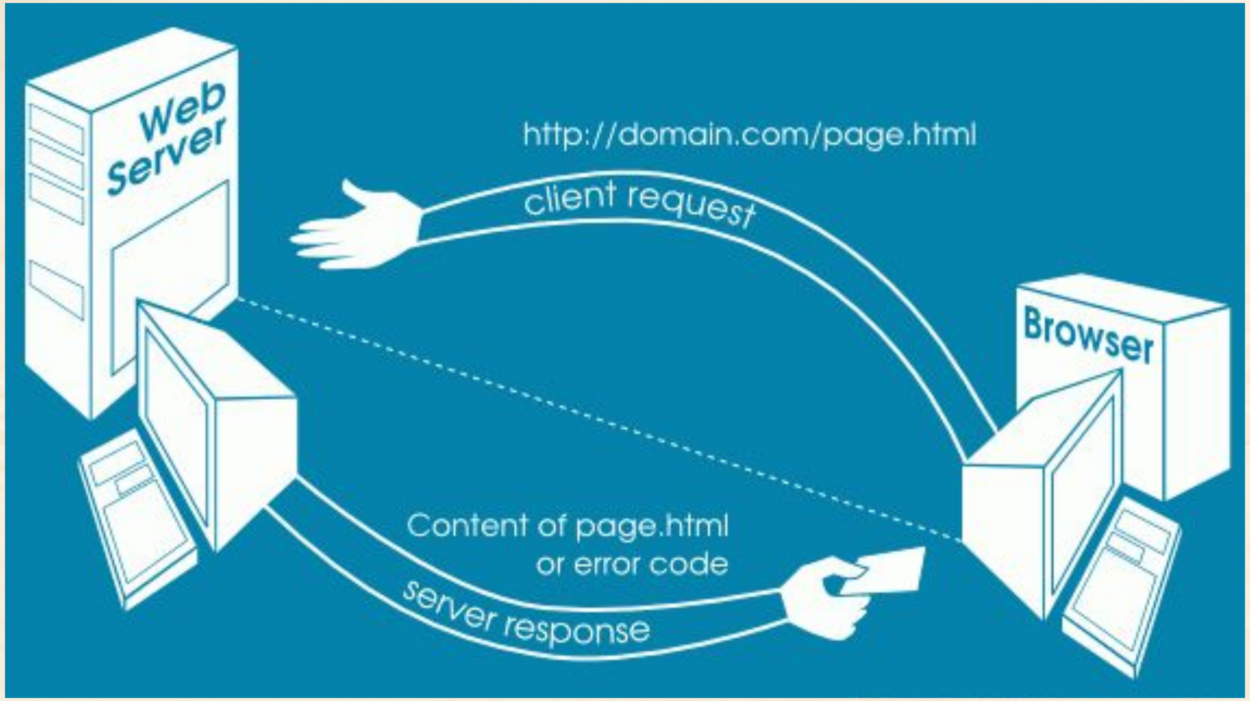
Sr.	Topic	Weightage	Teaching Hrs.
3	**INTER-PROCESS COMMUNICATION:** Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer\ Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc.	15%	6

Inter Process Communications(IPC)

- Processes in a system can be independent or cooperating.
  1. *Independent process* cannot affect or be affected by the execution of another process.
  2. *Cooperating process* can affect or be affected by the execution of another process.
- Cooperating processes need inter process communication mechanisms.

IPC

- It is a communication between two or more processes.



IPC

- Reasons of process cooperation
  1. Information sharing
  2. Computation speed-up
  3. Modularity
  4. Convenience

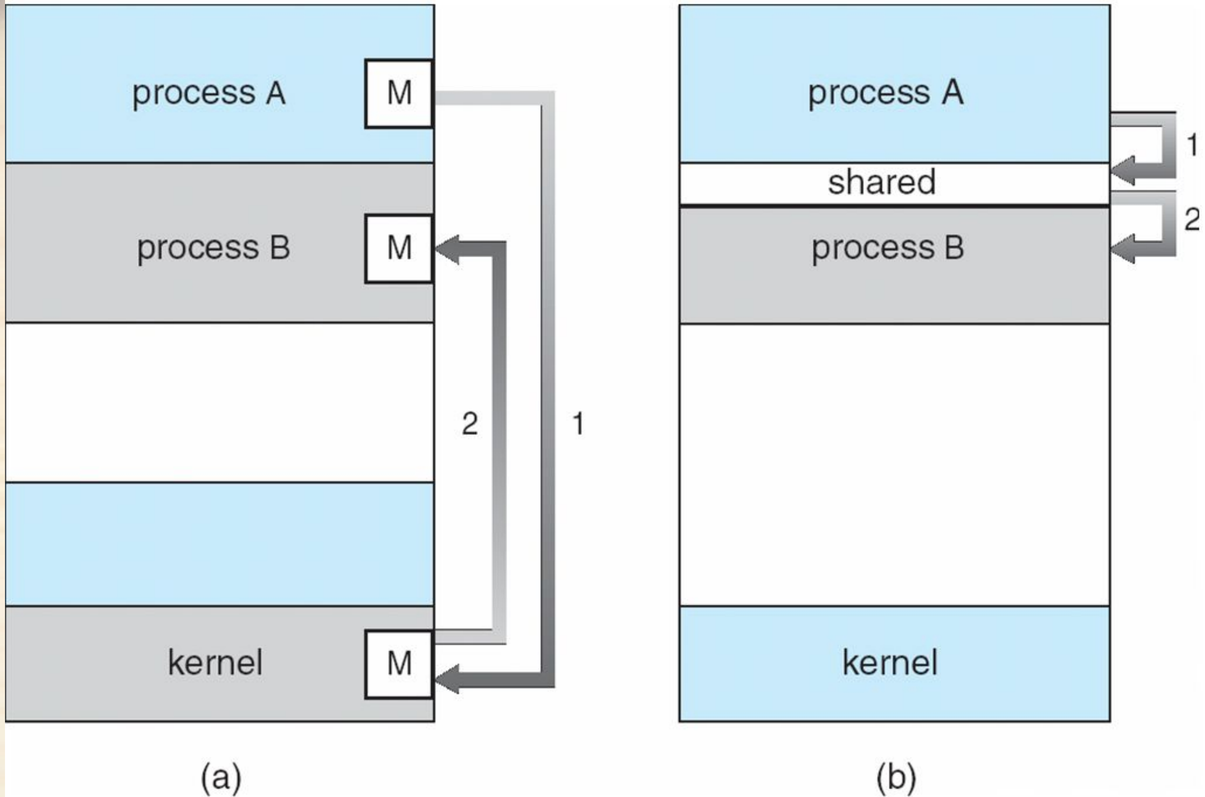
- Issues of process cooperation
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing

IPC

- There are two models for IPC

a. Message Passing

b. Shared Memory

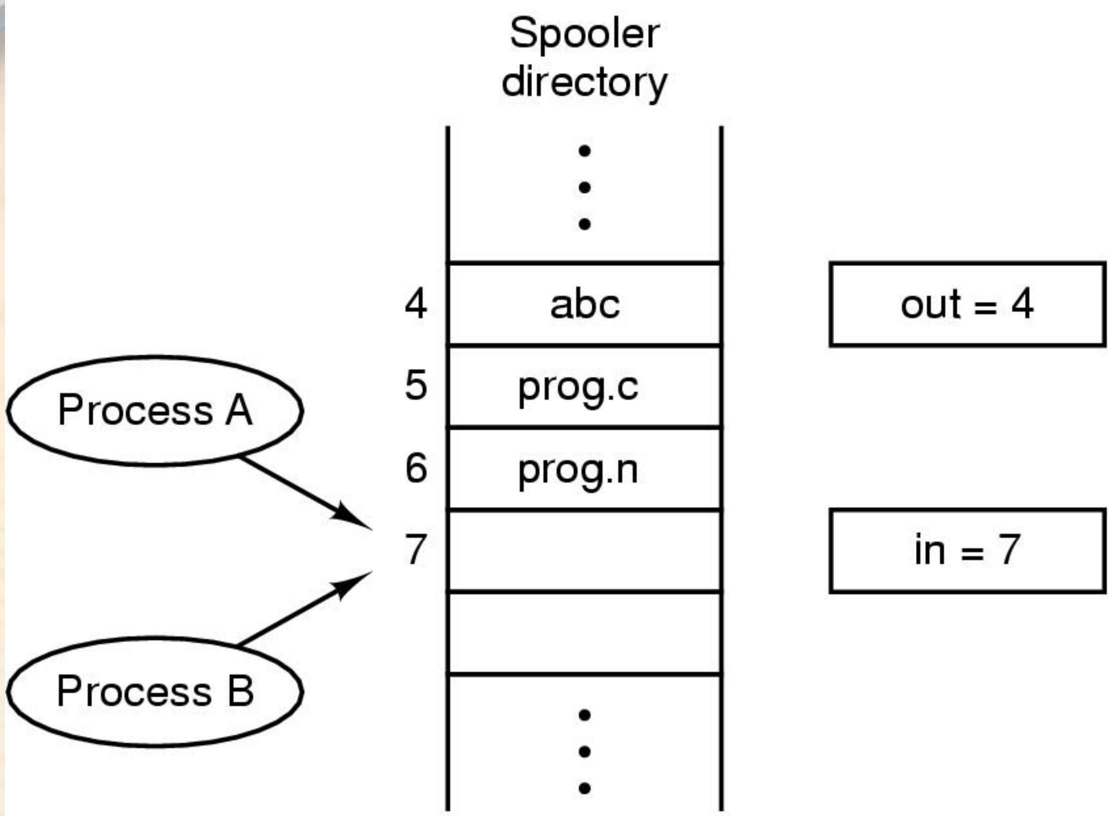


Race Condition

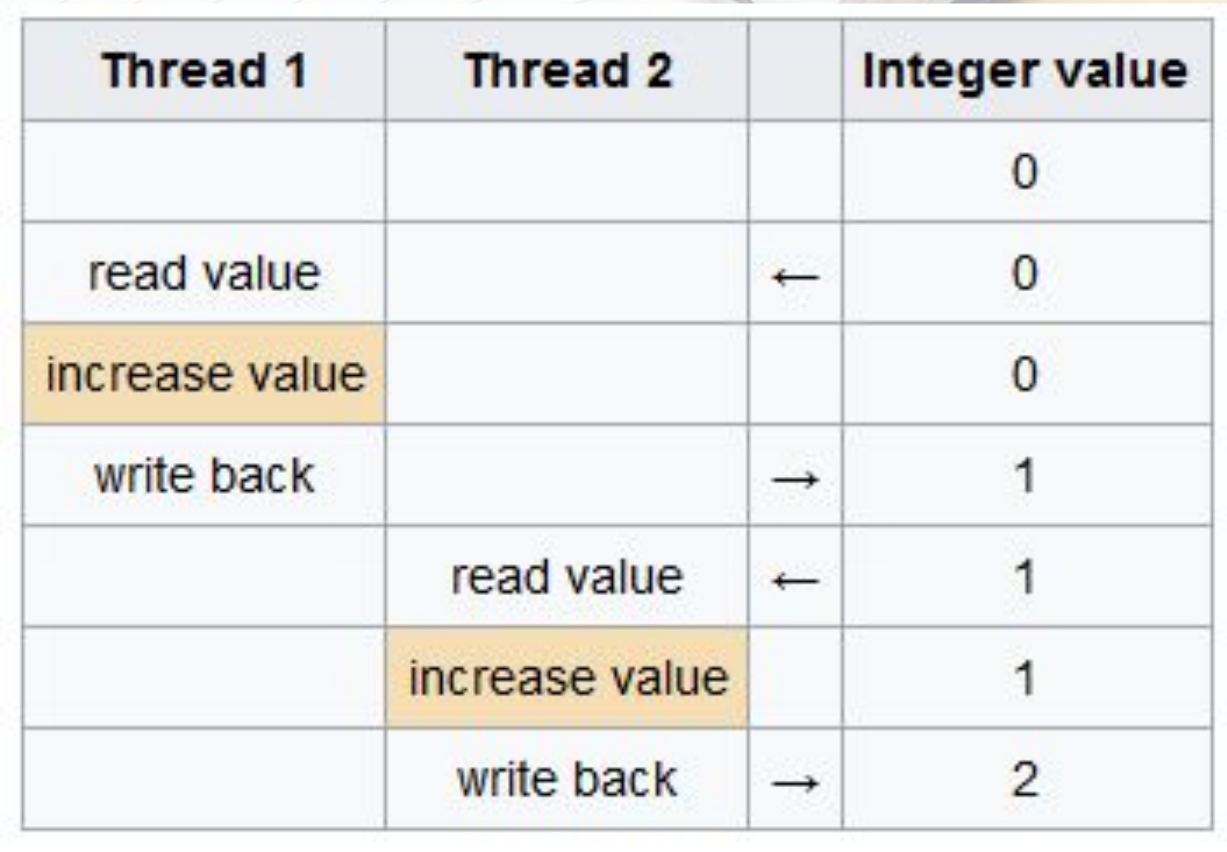
- **Race Condition**:
  - A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time.
  - But, because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.
- Reasons for Race Condition
  1. Exact instruction execution order cannot be predicted
  2. Resource (file, memory, data etc…) sharing

Example of Race Condition

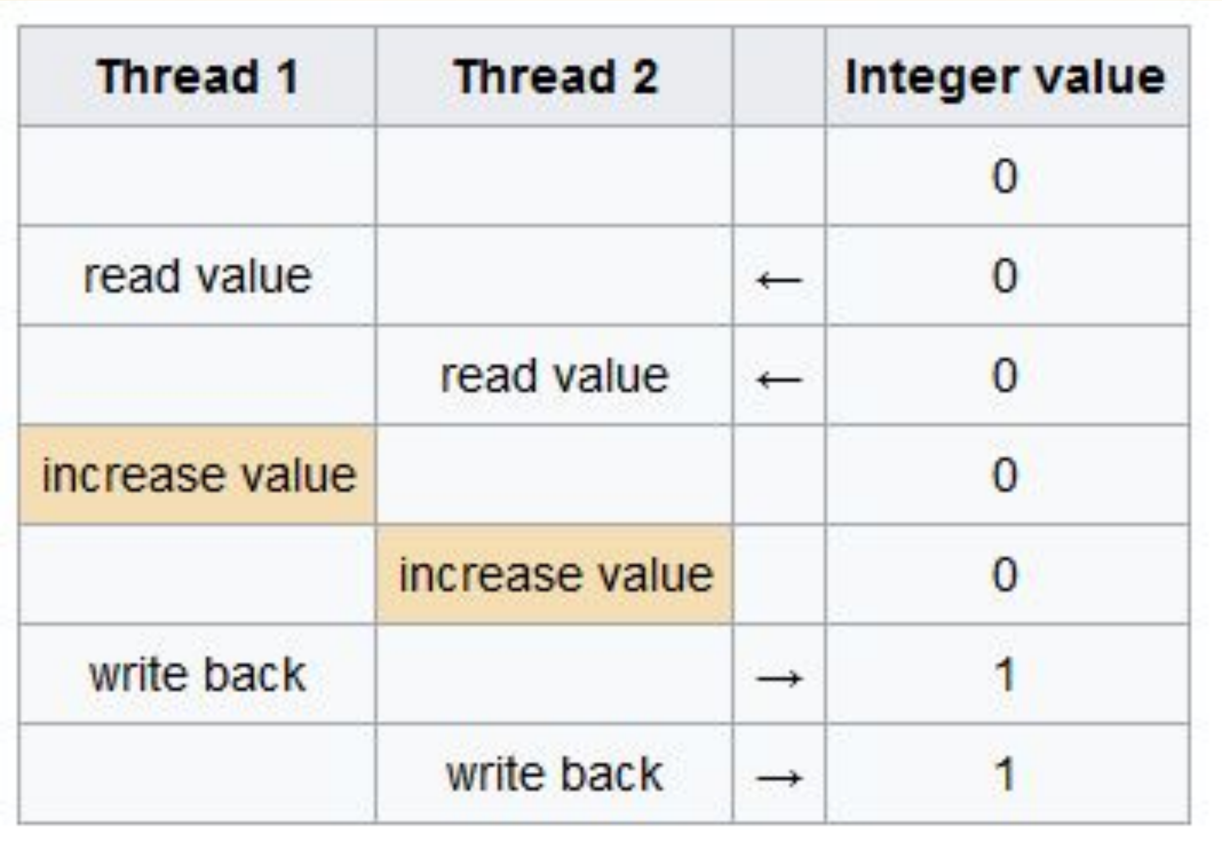
- Print spooler directory example : Two processes want to access shared memory at the same time.



Example of Race Condition



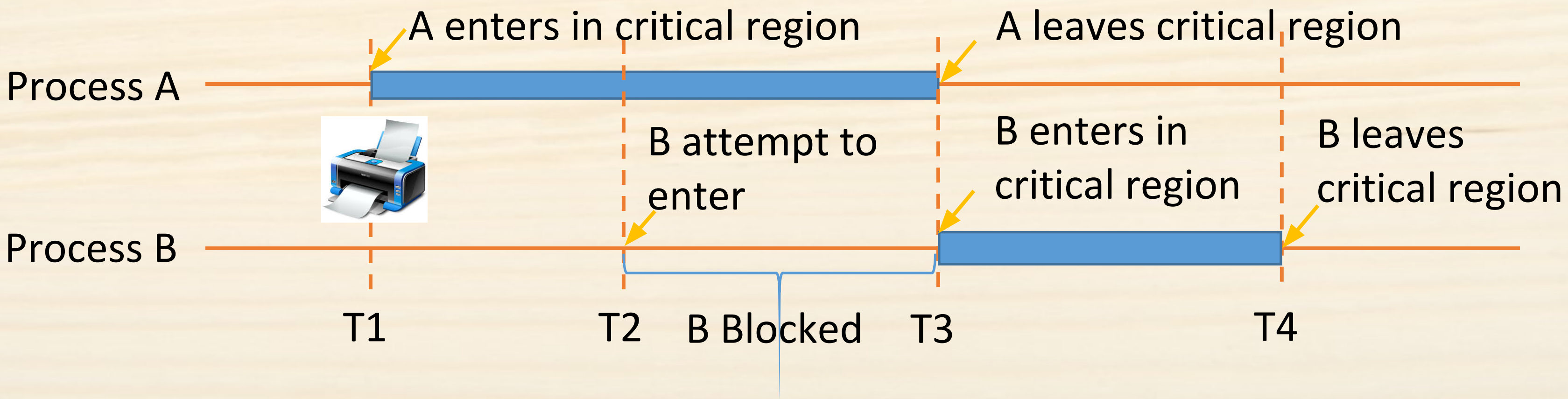
Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

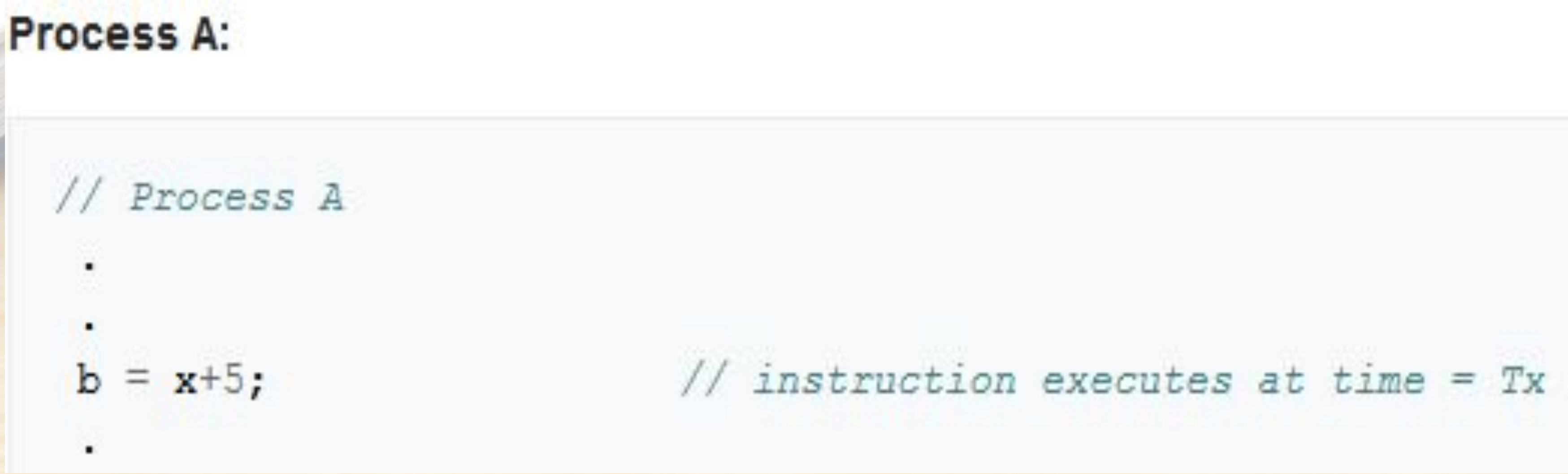
Critical Section

- ***Critical section*** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed is protected. This protected section is the critical section or critical region.



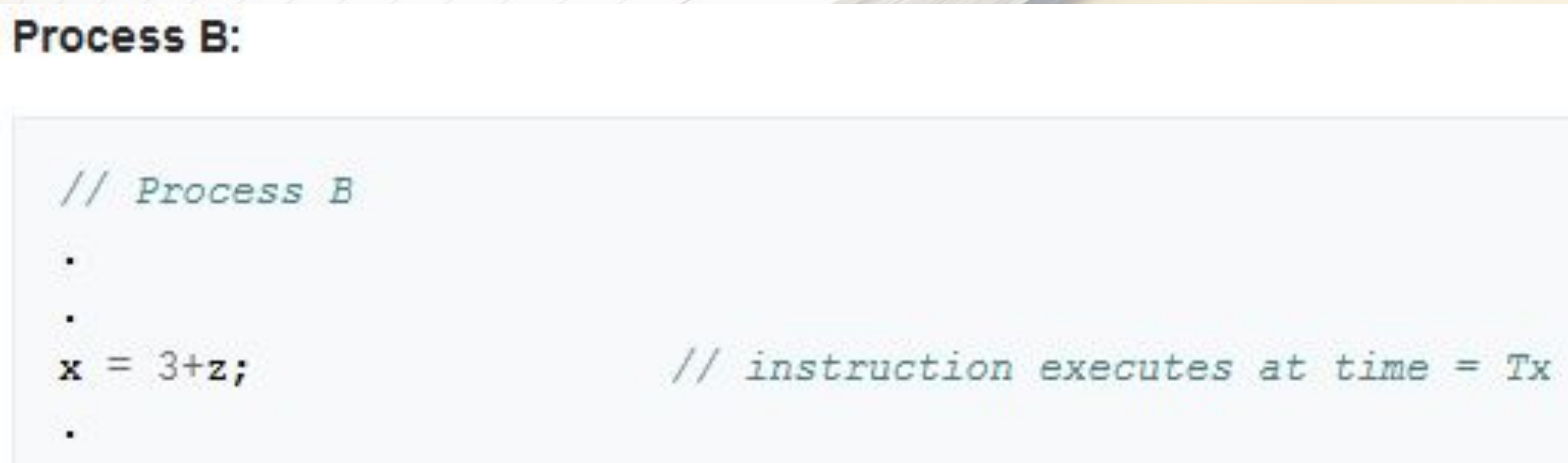
Need for Critical Sections

- For e.g., a variable ‘x’ is to be read by process A and process B has to write to the same variable ‘x’ at the same time.



Process A:

```
// Process A
 .
 .
 b = x+5;               // instruction executes at time = Tx
 .
```



Process B:

```
// Process B
.
.
x = 3+z;                    // instruction executes at time = Tx
.
```

- if A needs to read the updated value of ‘x’, executing Process A and Process B at the same time may not give required results.
- To prevent this, variable ‘x’ is protected by a critical section.
- First, B gets the access to the section. Once B finishes writing the value, A gets the access to the critical section and variable ‘x’ can be read.

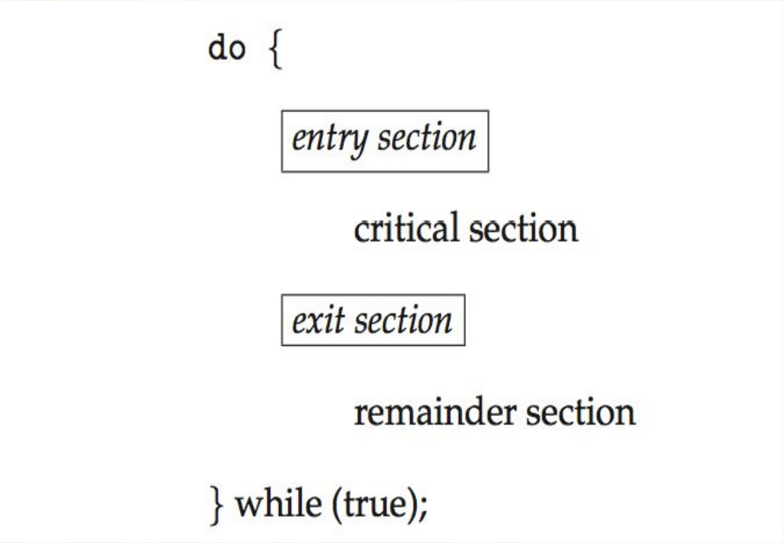
Mutual Exclusion

- A way of making sure that if one process is using a shared variable or file; the other process will be excluded (stopped) from doing the same thing.

- A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource.
- This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

Critical-Section Problem

- Consider system of ***n*** processes $\{P_0, P_1, \ldots, P_{n-1}\}$
- 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 is allowed to be in its critical section
- ***Critical section problem*** is to design protocol to solve this
  - Each process
    - Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**,
    - may follow critical section with **exit section**,
    - then **remainder section**



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

Solving Critical-Section Problem

Any good solution to the problem must satisfy following four conditions:

1. Mutual Exclusion:
    - No two processes may be simultaneously inside the same critical section.
2. Bounded Waiting:
    - No process should have to wait forever to enter a critical section.
3. Progress:
    - No process running outside its critical region may block other processes.
4. Arbitrary Speed:
    - No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed).

Mutual Exclusion with busy waiting

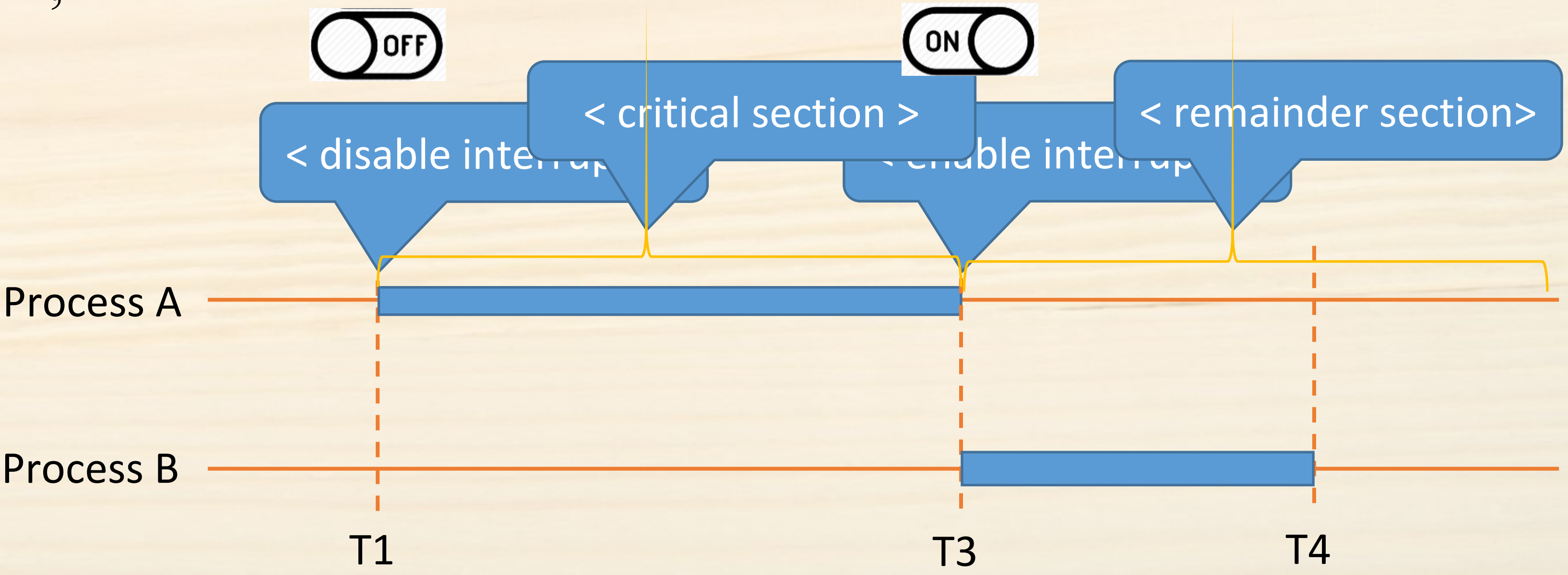
- Mechanisms for achieving mutual exclusion with busy waiting
    - Disabling interrupts
    - Shared lock variable
    - Strict alteration
    - TSL (test and set lock) instruction
    - Exchange instruction
    - Peterson’s solution

Disabling Interrupts

- Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section.
- With interrupts turned off the CPU could not be switched to other process.
- Hence, no other process will enter its critical and mutual exclusion achieved

Disabling interrupts

```
• while (true)
    {
        < disable interrupts >;
        < critical section >;
        < enable interrupts >;
        < remainder section>;

    }
```



Disabling Interrupts

- **Problems:**
  - Unattractive or unwise to give user processes the power to turn off interrupts.
  - What if one of them did it (disable interrupt) and never turned them on (enable interrupt) again? That could be the end of the system.
  - If the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

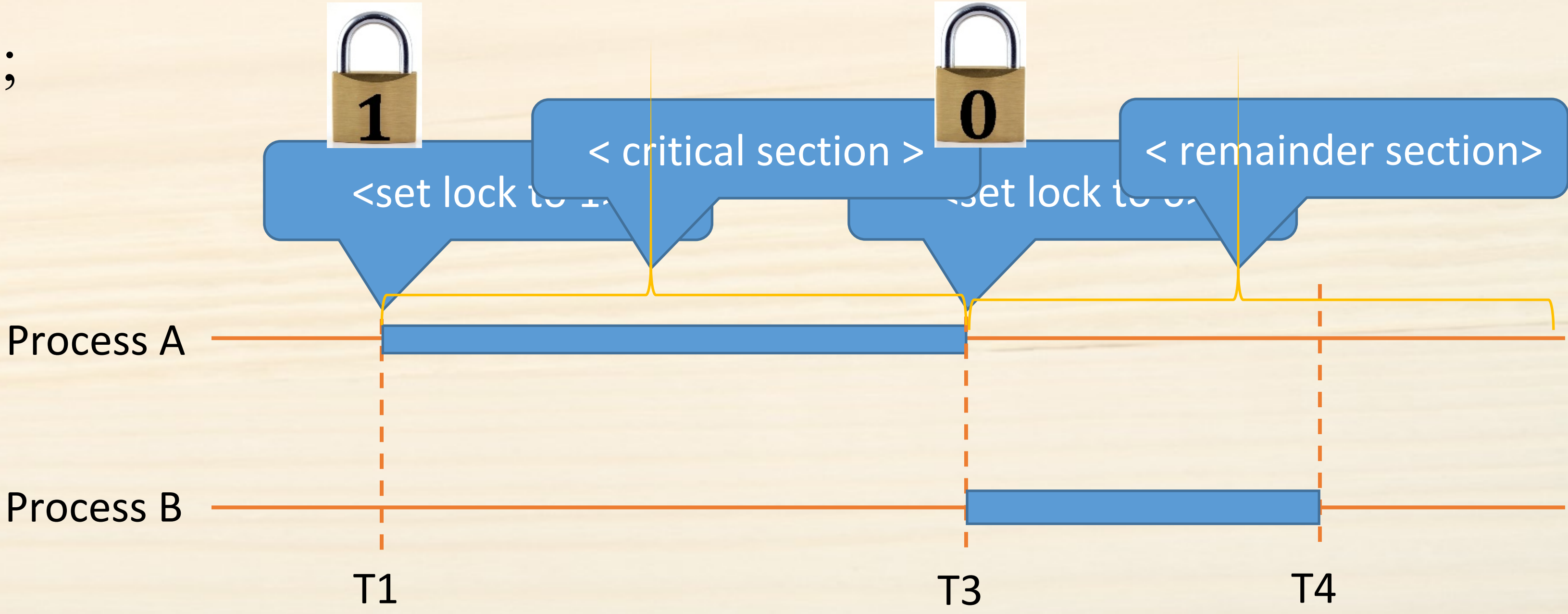
Shared lock variable

- A shared variable lock having value 0 or 1.
- Before entering into critical region a process checks a shared variable lock’s value.
    - If the value of lock is 0 then set it to 1 before entering the critical section and enters into critical section and set it to 0 immediately after leaving the critical section.
    - If the value of lock is 1 then wait until it becomes 0 by some other process which is in critical section.

Shared lock variable

 Algorithm:

```
• while (true)
      {    < set shared variable to 1>;
           < critical section >;
           < set shared variable to 0>;
           < remainder section>;
      }
```

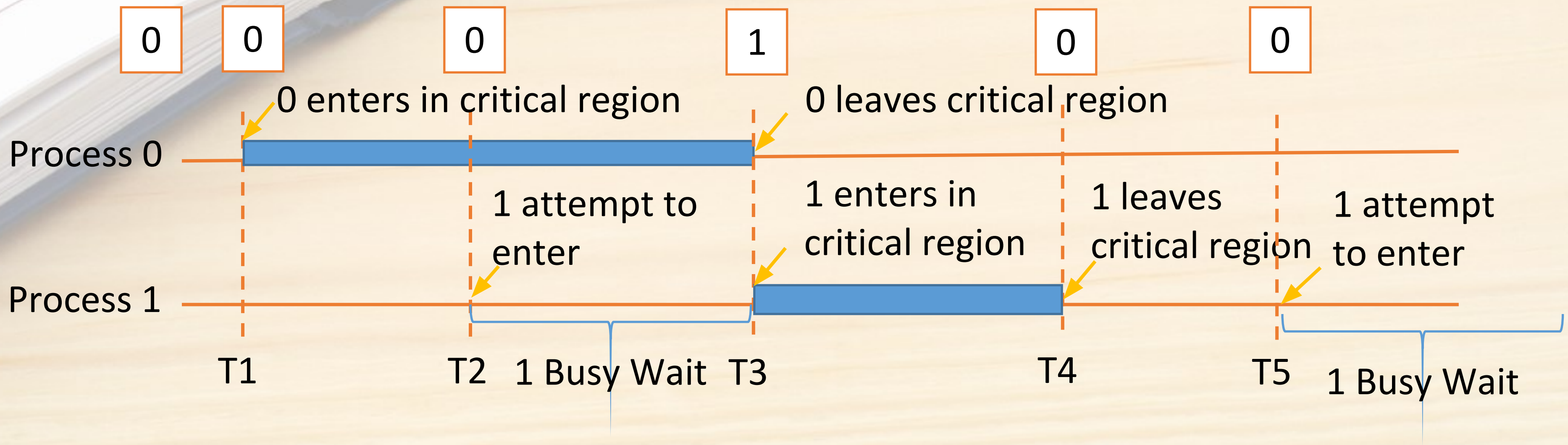


- Problem:
  - If process P0 sees the value of lock variable 0 and before it can set it to 1 context switch occurs.
  - Now process P1 runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
  - At some point of time P0 resumes, sets the value of lock variable to 1, enters critical region.
  - Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion condition.

Strict Alteration

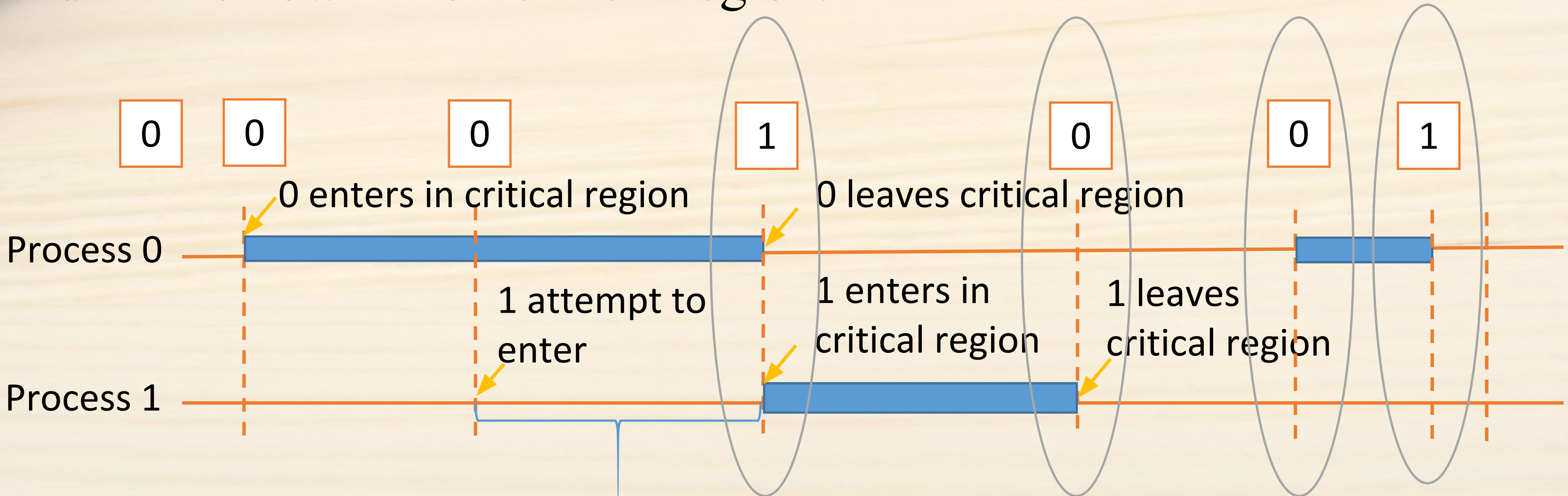
- Integer variable 'turn' keeps track of whose turn is to enter the critical section.
- Initially turn=0. Process 0 inspects turn, finds it to be 0, and enters in its critical section.
- Process 1 also finds it to be 0 and therefore sits in a loop continually testing 'turn' to see when it becomes 1.
- *Continuously testing a variable waiting for some event to appear is called the busy waiting.*
- When process 0 exits from critical region it sets turn to 1 and now process 1 can find it to be 1 and enters in to critical region.
- In this way, both the processes get alternate turn to enter in critical region.

Strict Alteration (Algorithm)



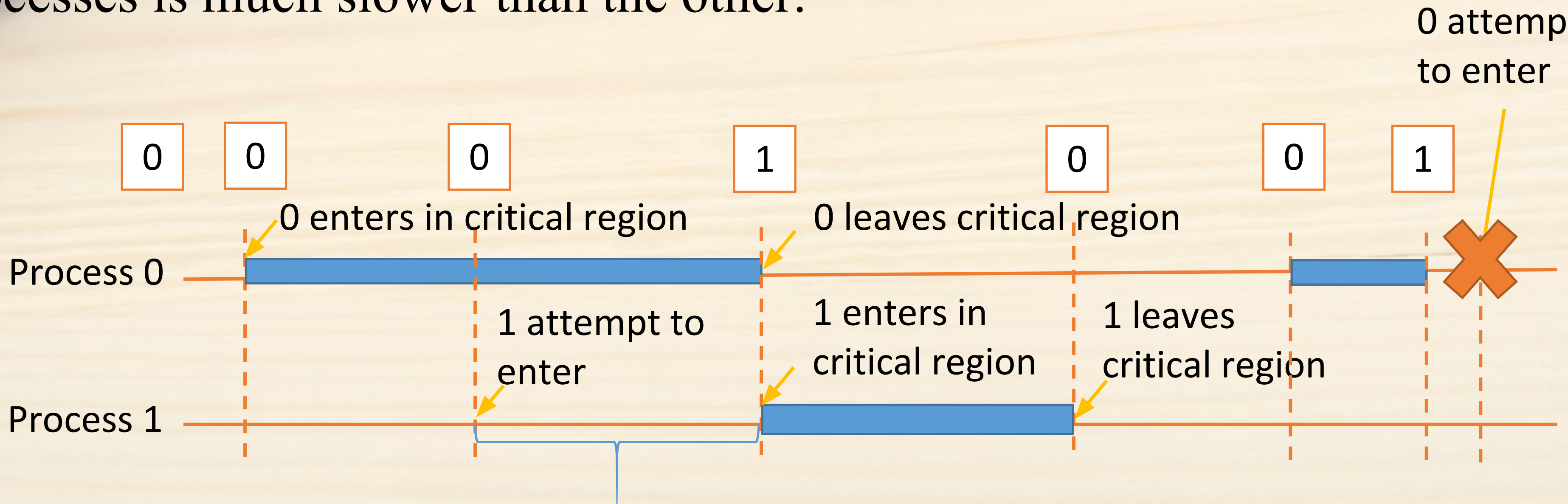
Strict Alteration (Disadvantages)

- Consider the following situation for two processes P0 and P1.
- P0 leaves its critical region, set turn to 1, enters non critical region.
- P1 enters and finishes its critical region, set turn to 0.
- Now both P0 and P1 in non-critical region.
- P0 finishes non critical region, enters critical region again, and leaves this region, set turn to 1.
- P0 and P1 are now in non-critical region.



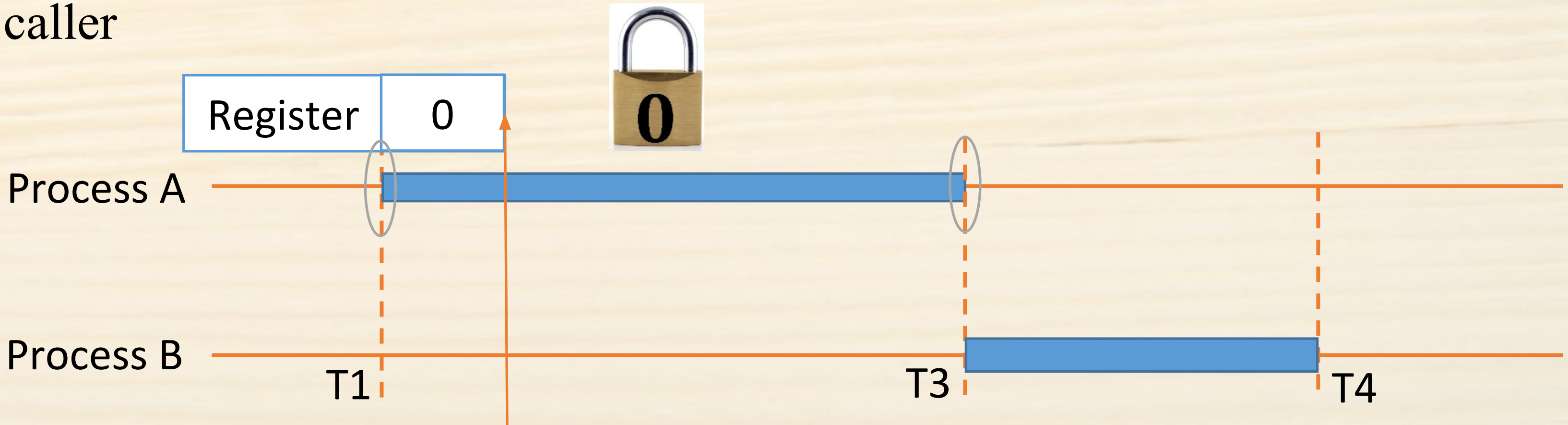
Strict Alteration (Disadvantages)

- P0 finishes non critical region but cannot enter its critical region because turn = 1 and it is turn of P1 to enter the critical section.
- Hence, P0 will be blocked by a process P1 which is not in critical region. This violates one of the conditions of mutual exclusion.
- It wastes CPU time, so we should avoid busy waiting as much as we can.
- **Another Disadvantage:** Taking turns is not a good idea when one of the processes is much slower than the other.



TSL (Test and Set Lock) Instruction

```
enter_region:        (Before entering its critical region, process calls enter_region)
 TSL REGISTER,LOCK   |copy lock variable to register set lock to 1
 CMP REGISTER,#0     |was lock variable 0?
 JNE enter_region    |if it was nonzero, lock was set, so loop
 RET                 |return to caller: critical region entered
leave_region:        (When process wants to leave critical region, it calls leave_region)
 MOVE LOCK,#0        |store 0 in lock variable
 RET                 |return to caller
```



Exchange Instruction

- Algorithm

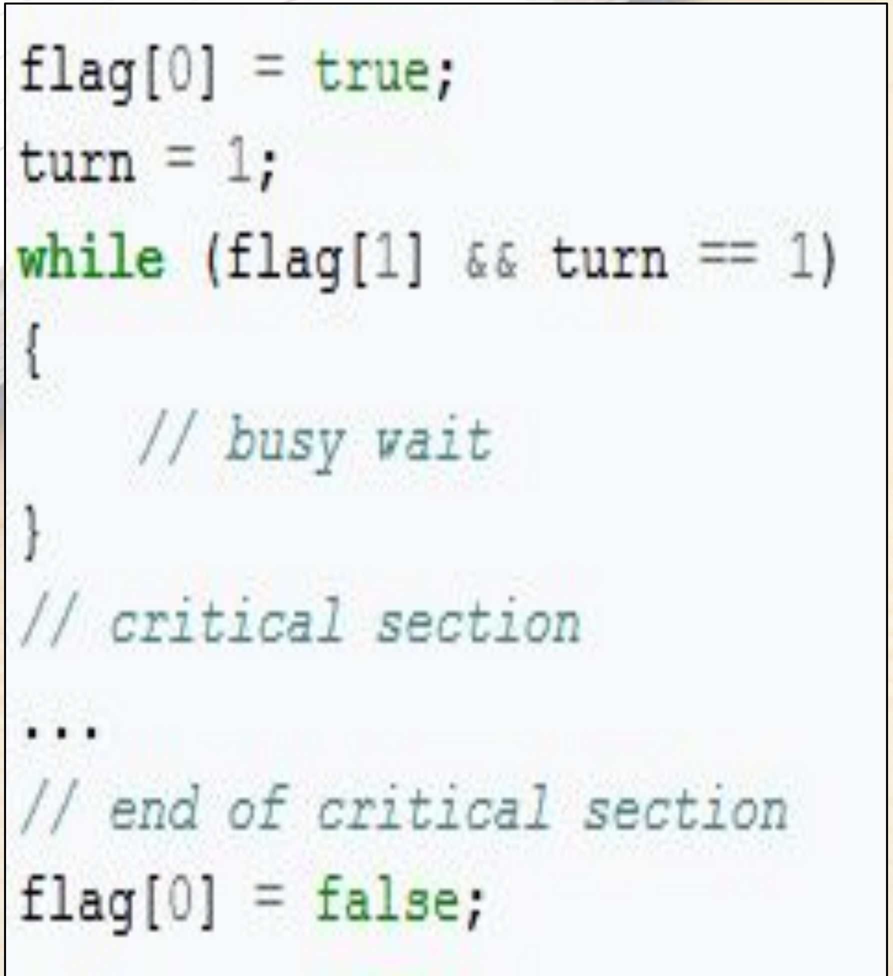
```
enter_region:        (Before entering its critical region, process calls enter_region)
MOVE REGISTER,#1      |put 1 in the register
XCHG REGISTER,LOCK    |swap content of register & lock variable
CMP REGISTER,#0       |was lock variable 0?
JNE enter_region      |if it was nonzero, lock was set, so loop
RET                   |return to caller: critical region entered
leave_region:        (When process wants to leave critical region, it calls leave_region)
MOVE LOCK,#0          |store 0 in lock variable
RET                   |return to caller
```

Peterson’s Solution

- Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.
- In the algorithm two processes share two variables:
  - □ **`int turn;`**
  - □ **`Boolean flag[2]`**
- The variable **`turn`** indicates **whose turn it is** to enter the critical section.
- The **`flag`** array is used to indicate if a process is **ready** to enter the critical section. **`flag[i] = `*`true`*** implies that process $\mathbf{P}_i$ is ready!

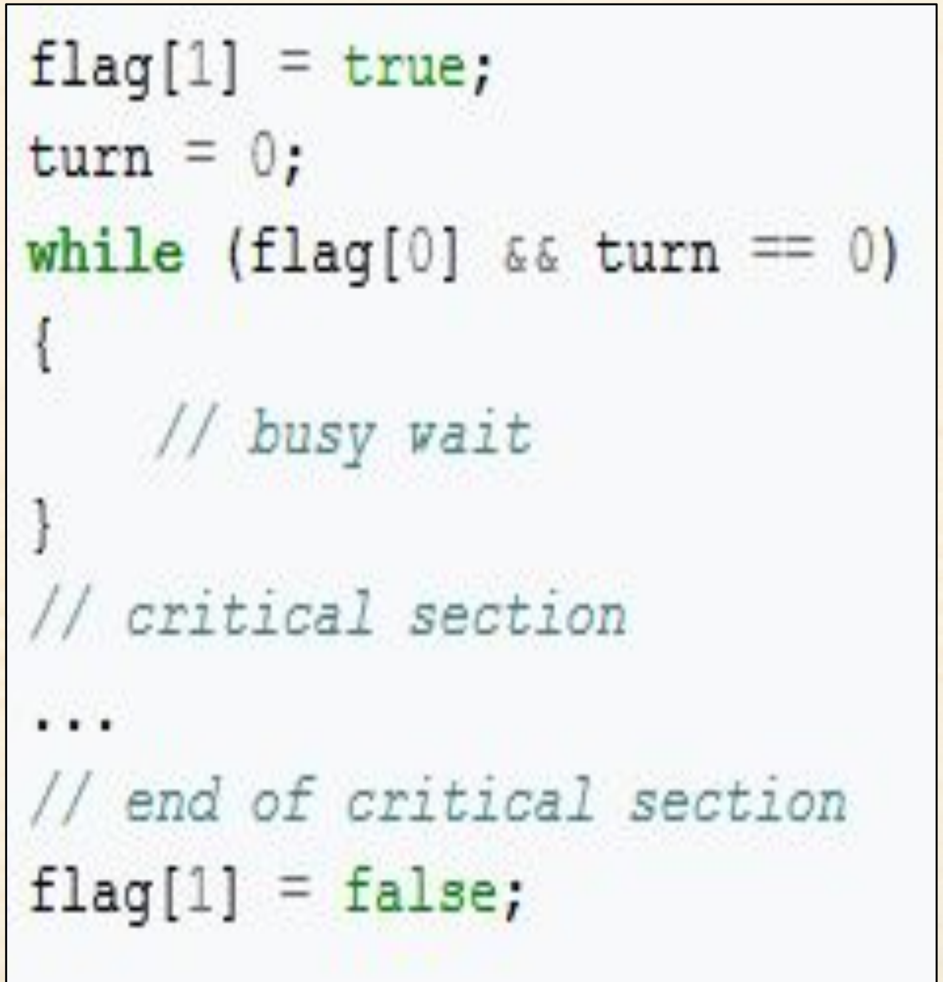
Peterson’s Solution(cont.)

Process-0



```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

Process-1



```
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Peterson’s Solution(cont.)

□Three Essential Criteria

[1] Mutual exclusion

- P0 and P1 can never be in the critical section at the same time:

- If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0.

[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 making the decision as to which process will enter its critical section next.

[3] Bounded waiting

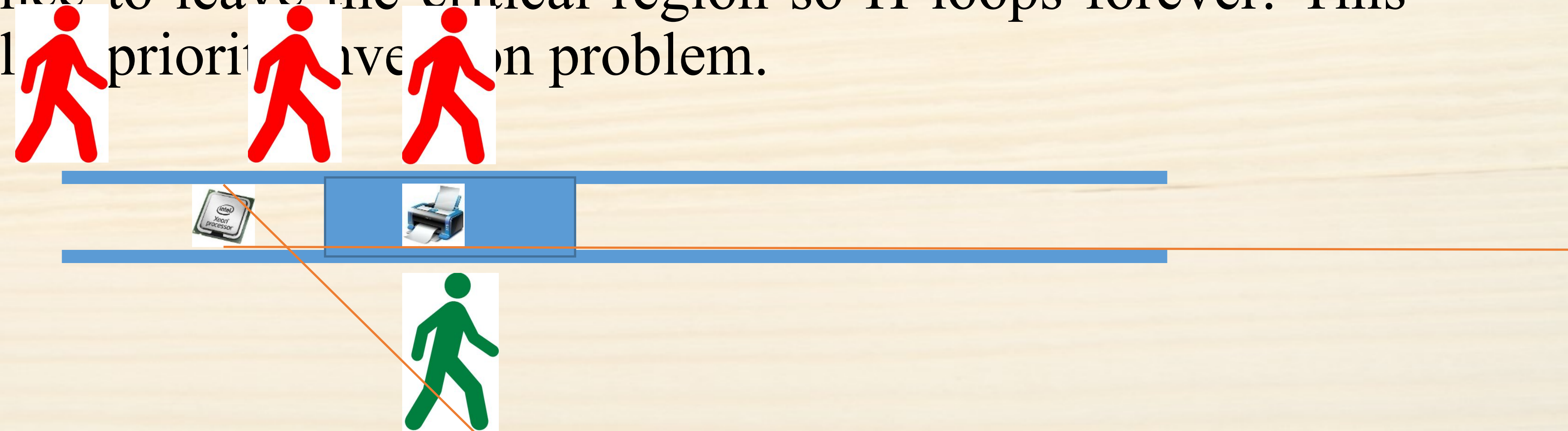
- Bounded waiting means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.

Priority inversion problem

- Priority inversion means the execution of a high priority process/thread is blocked by a lower priority process/thread.
- Consider a computer with two processes, H having high priority and L having low priority.
- The scheduling rules are such that H runs first then L will run.

Priority inversion problem

- At a certain moment, L is in critical region and H becomes ready to run (e.g. I/O operation complete).
- H now begins busy waiting and waits until L will exit from critical region. But H has highest priority than L so CPU is switched from L to H.
- Now L will never be scheduled (get CPU) until H is running so L will never get chance to leave the critical region so H loops forever. This situation is cal priorit ve on problem.



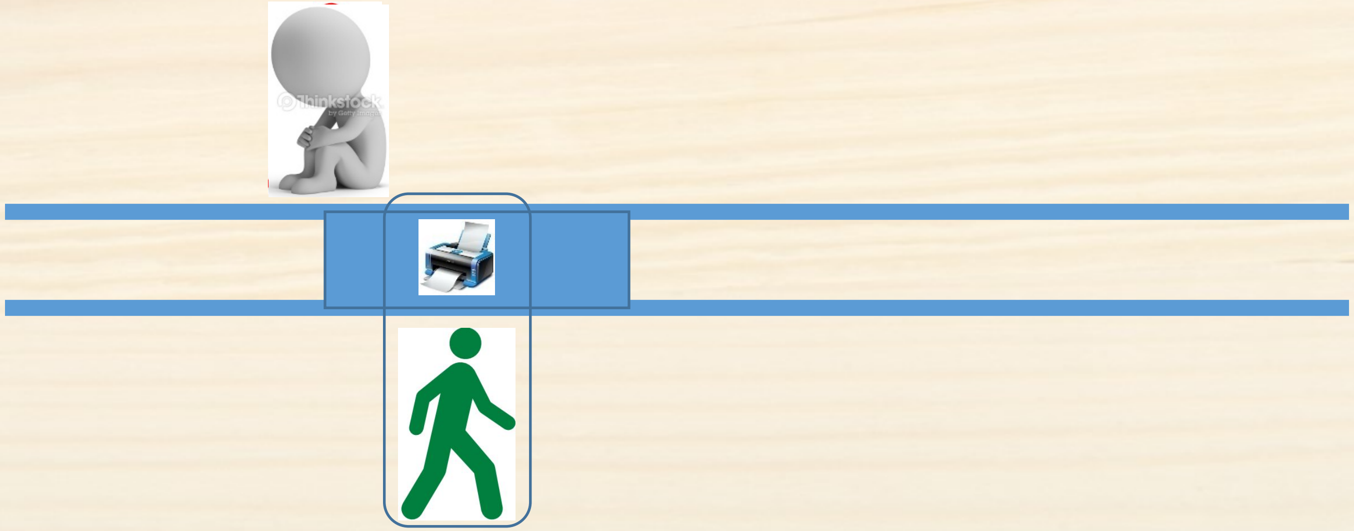
Mutual Exclusion with Busy Waiting

1. Disabling Interrupts
   - is not appropriate as a general mutual exclusion mechanism for user processes
2. Lock Variables
   - contains exactly the same fatal flaw that we saw in the spooler directory
3. Strict Alternation
   - solution violates condition 3 □ process running outside its critical region blocks other processes.
4. Peterson's Solution
5. The TSL/XCHG instruction
   - Both Peterson’s solution and the solutions using TSL or XCHG are correct.

- Limitations
  1. Busy Waiting : this approach waste CPU time
  2. Priority Inversion Problem : a low-priority process blocks a higher-priority one

Sleep and Wakeup

- Peterson’s solution and solution using TSL and XCHG have the limitation of requiring busy waiting.
    - when a processes wants to enter in its critical section, it checks to see if the entry is allowed.
    - If it is not allowed, the process goes into a loop and waits (i.e., start busy waiting) until it is allowed to enter.
    - This approach waste CPU-time.

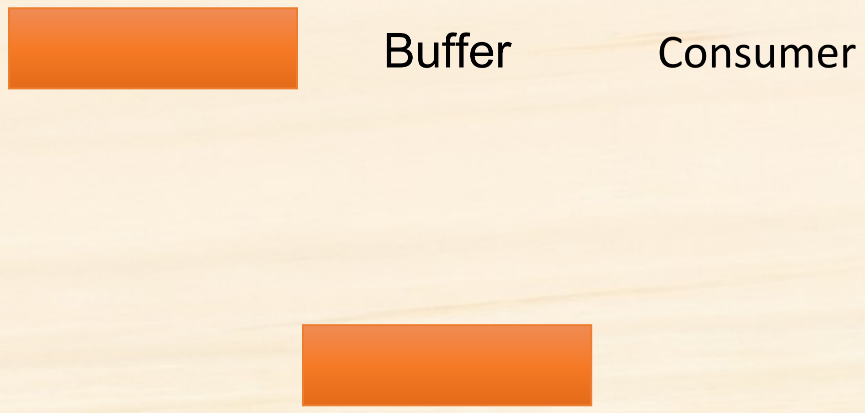


Sleep and Wakeup

- But we have interprocess communication primitives (the pair of sleep & wakeup).
  - **Sleep**: It is a system call that causes the caller to be blocked (suspended) until some other process wakes it up.
  - **Wakeup**: It is a system call that wakes up the process.
- Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups' .

Producer Consumer problem

- It is multi-process synchronization problem.
- It is also known as bounded buffer problem.
- This problem describes two processes producer and consumer, who share common, fixed size buffer.
- Producer process
  - Produce some information and put it into buffer
- Consumer process
  - Consume this information (remove it from the buffer)

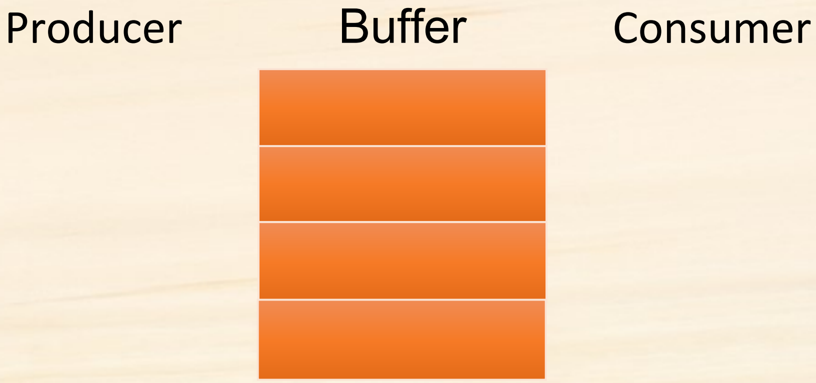


What Producer Consumer problem is?

- The problem is to make sure that the producer won’t try to add data (information) into the buffer if it is full and consumer won’t try to remove data (information) from the an empty buffer.
- Solution for producer:
  - Producer either go to sleep or discard data if the buffer is full.
  - Once the consumer removes an item from the buffer, it notifies (wakeups) the producer to put the data into buffer.
- Solution for consumer:
  - Consumer can go to sleep if the buffer is empty.
  - Once the producer puts data into buffer, it notifies (wakeups) the consumer to remove (use) data from buffer.

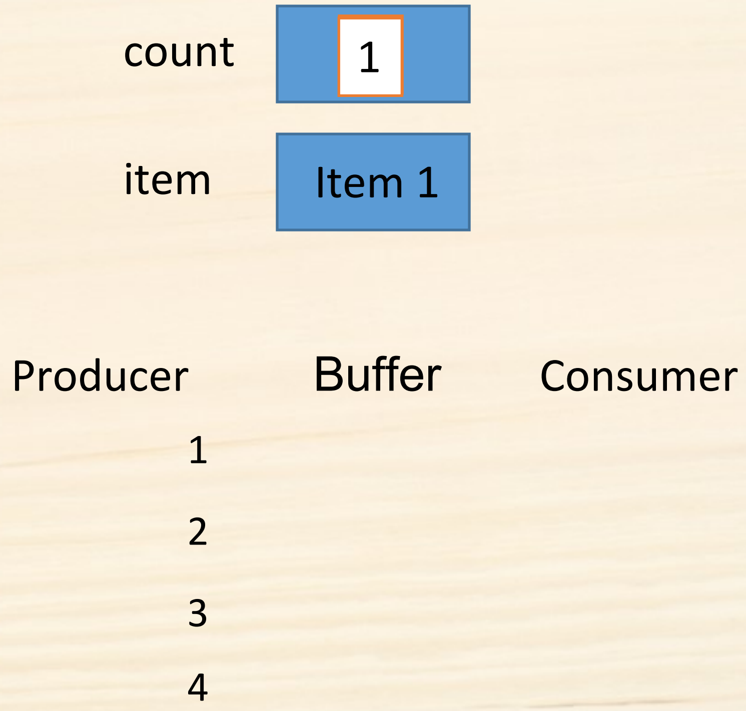
What Producer Consumer problem is?

- Buffer is empty
  - Producer want to produce √
  - Consumer want to consume X
- Buffer is full
  - Producer want to produce X
  - Consumer want to consume √
- Buffer is partial filled
  - Producer want to produce √
  - Consumer want to consume √



Producer Consumer problem using Sleep & Wakeup

```
#define N 4
int count=0;
void producer (void)
{    int item;
     while (true) {
     item=produce_item();
     if (count==N) sleep();
     insert_item(item);
     count=count+1;
     if(count==1)
wakeup(consumer);
     }
}
```



count 1

item Item 1

Producer Buffer Consumer

1

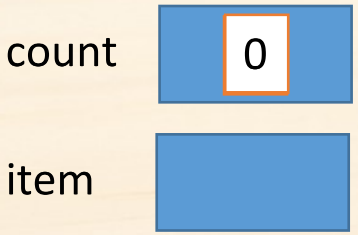
2

3

4

Producer Consumer problem using Sleep & Wakeup

```
void consumer (void)
{   int item;
    while (true)
    {
    if (count==0) sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)
        wakeup(producer);
    consume_item(item);
    }
}
```



Producer	Buffer	Consumer
1	Item 1	
2		
3		
4		

Problem

Problem with this solution is that it contains a race condition that can lead to a deadlock. **(How???)**

- The consumer has just read the variable ***count***, noticed it's zero and is just about to move inside the if block.
- Just before calling sleep, the consumer is suspended and the producer is resumed.
- The producer creates an item, puts it into the buffer, and increases ***count***.
- Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.

```
void consumer (void)
{    int item;
     while (true)
                     Context Switching
     {
     if (count==0)  sleep();
     item=remove_item();
     count=count-1;
     if(count==N-1)
          wakeup(producer);
     consume_item(item);
     }
}
```

Problem in Sleep & Wakeup

- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost.
- When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when ***count*** is equal to 1.
- The producer will loop until the buffer is full, after which it will also go to sleep.

- ***Finally, both the processes will sleep forever.*** This solution therefore is unsatisfactory.

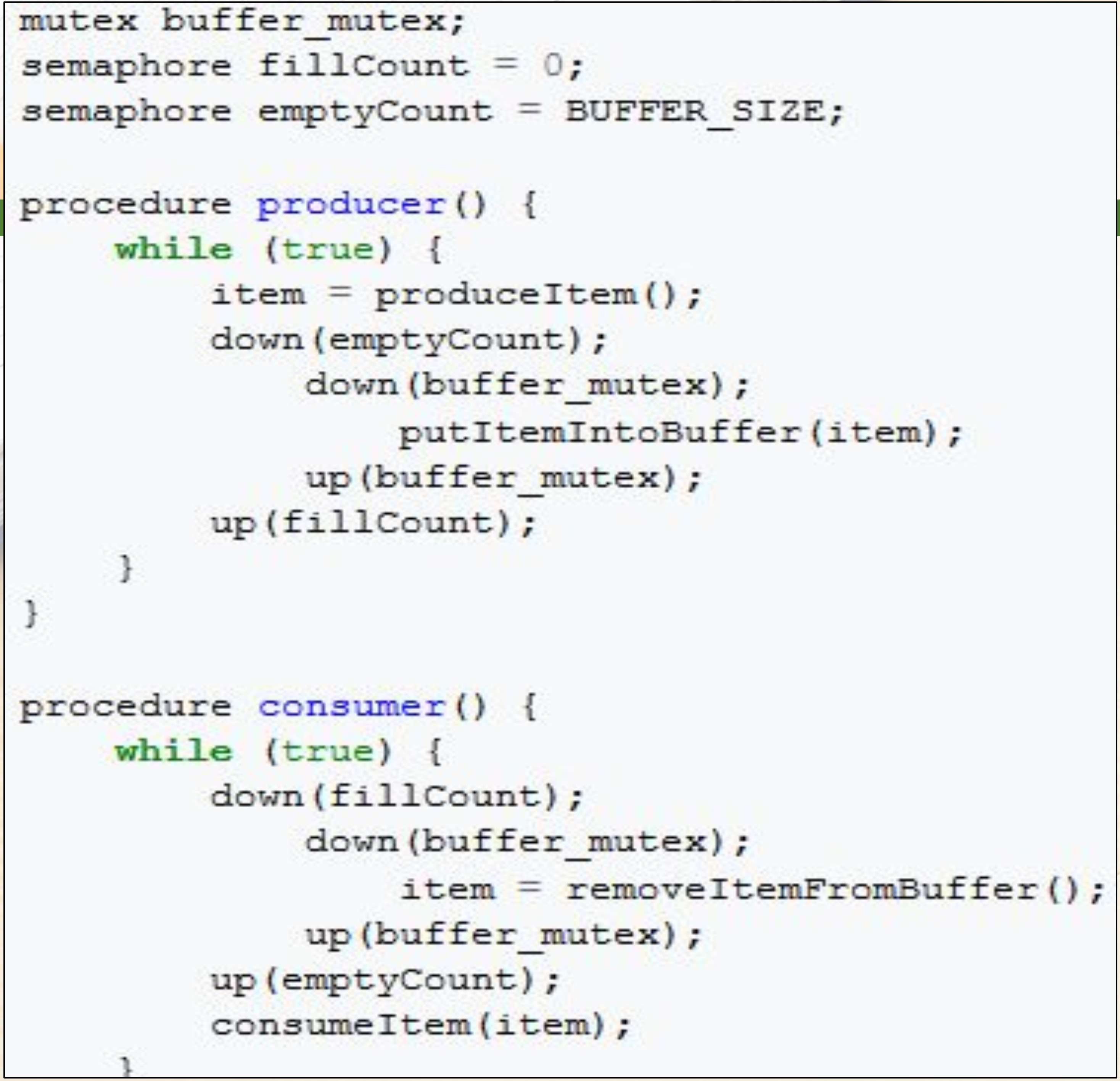
```
void consumer (void)
{     int item;
      while (true)
      {
      if (count==0) sleep();
      item=remove_item();
      count=count-1;
      if(count==N-1)
            wakeup(producer);
      consume_item(item);
      }
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by:
  - first **acquire()** a lock
  - then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
     while (!available)
        ; /* busy wait */
     available = false;
  }
 release() {
     available = true;
  }
 while (true) {
  acquire lock
     critical section
  release lock
    remainder section
  }
```



```
mutex buffer_mutex;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            down(buffer_mutex);
                putItemIntoBuffer(item);
            up(buffer_mutex);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
            down(buffer_mutex);
                item = removeItemFromBuffer();
            up(buffer_mutex);
        up(emptyCount);
        consumeItem(item);
    }
```

Semaphores

- A semaphore is a variable that provides an abstraction for controlling the access of a shared resource by multiple processes in a parallel programming environment.
- Semaphores solve the problem of lost wakeup calls.
- There are 2 types of semaphores:

1. **Binary semaphores** :-
   - Binary semaphores can take only 2 values (0/1).
   - Binary semaphores have 2 methods associated with it (up, down / lock, unlock/signal, wait).
   - They are used to acquire locks.
2. **Counting semaphores** :-
   - Counting semaphore can have possible values more than two.

Semaphore (cont…)

- We want functions insert_item and remove_item such that the following hold:
  - Mutually exclusive access to buffer: At any time only one process should be executing (either insert_item or remove_item).
  - No buffer overflow: A process executes insert_item only when the buffer is not full (i.e., the process is blocked if the buffer is full).
  - No buffer underflow: A process executes remove_item only when the buffer is not empty (i.e., the process is blocked if the buffer is empty).

Semaphore (cont…)

- We want functions insert _item and remove_item such that the following hold:
    - No busy waiting.
    - No producer starvation: A process does not wait forever at insert_item() provided the buffer repeatedly becomes full.
    - No consumer starvation: A process does not wait forever at remove_item() provided the buffer repeatedly becomes empty.

Semaphores

Two operations on semaphores are defined.

1. Down Operation
   - The down operation on a semaphore checks to see if the value is greater than 0.
   - If so, it decrements the value and just continues.
   - If the value is 0, the process is put to sleep without completing the down for the moment.
   - *Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action.*
   - It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

Semaphores

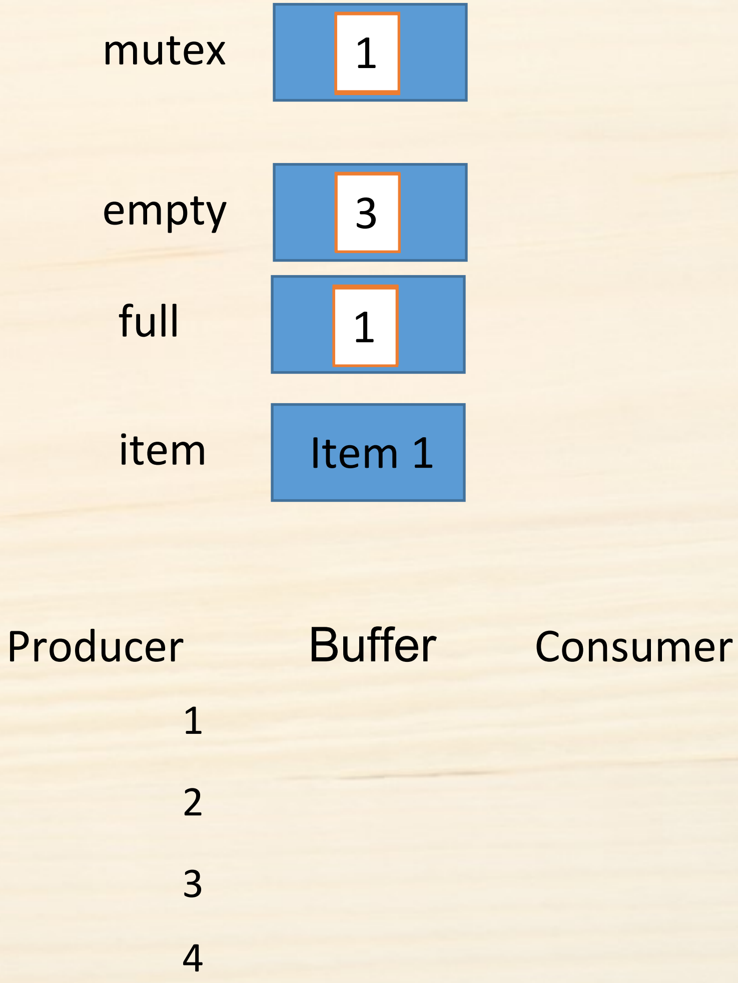
Two operations on semaphores are defined.

2. Up Operation

- The up operation increments the value of the semaphore addressed.
- If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.
- The operation of incrementing the semaphore and waking up one process is also indivisible.
- *No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.*

Producer Consumer problem using Semaphore

```
#define N 4
typedef int semaphore;
semaphore mutex=1;
semaphore empty=N;
semaphore full=0;
void producer (void)
{       int item;
        while (true)
        {
         item=produce_item();
         down(&empty);
         down(&mutex);
         insert_item(item);
         up(&mutex);
         up(&full);
        }
}
```



mutex 1

empty 3

full 1

item Item 1

Producer Buffer Consumer

1

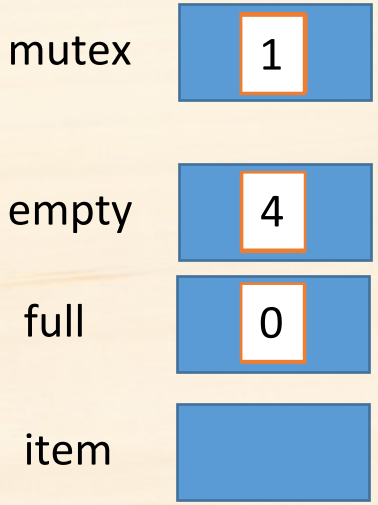
2

3

4

Producer Consumer problem using Semaphore

```
void consumer (void)
{    int item;
     while (true)
     {
      down(&full);
      down(&mutex);
      item=remove_item(item);
      up(&mutex);
      up(&empty);
      consume_item(item);
     }
}
```



mutex 1

empty 4

full 0

item

Producer	Buffer	Consumer
1	Item 1	
2		
3		
4		

Monitor

- A higher-level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, ***but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.***

Monitor

- Monitors have an important property for achieving mutual exclusion: only one process can be active in a monitor at any instant.

- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor.

- If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

Producer consumer problem using monitor

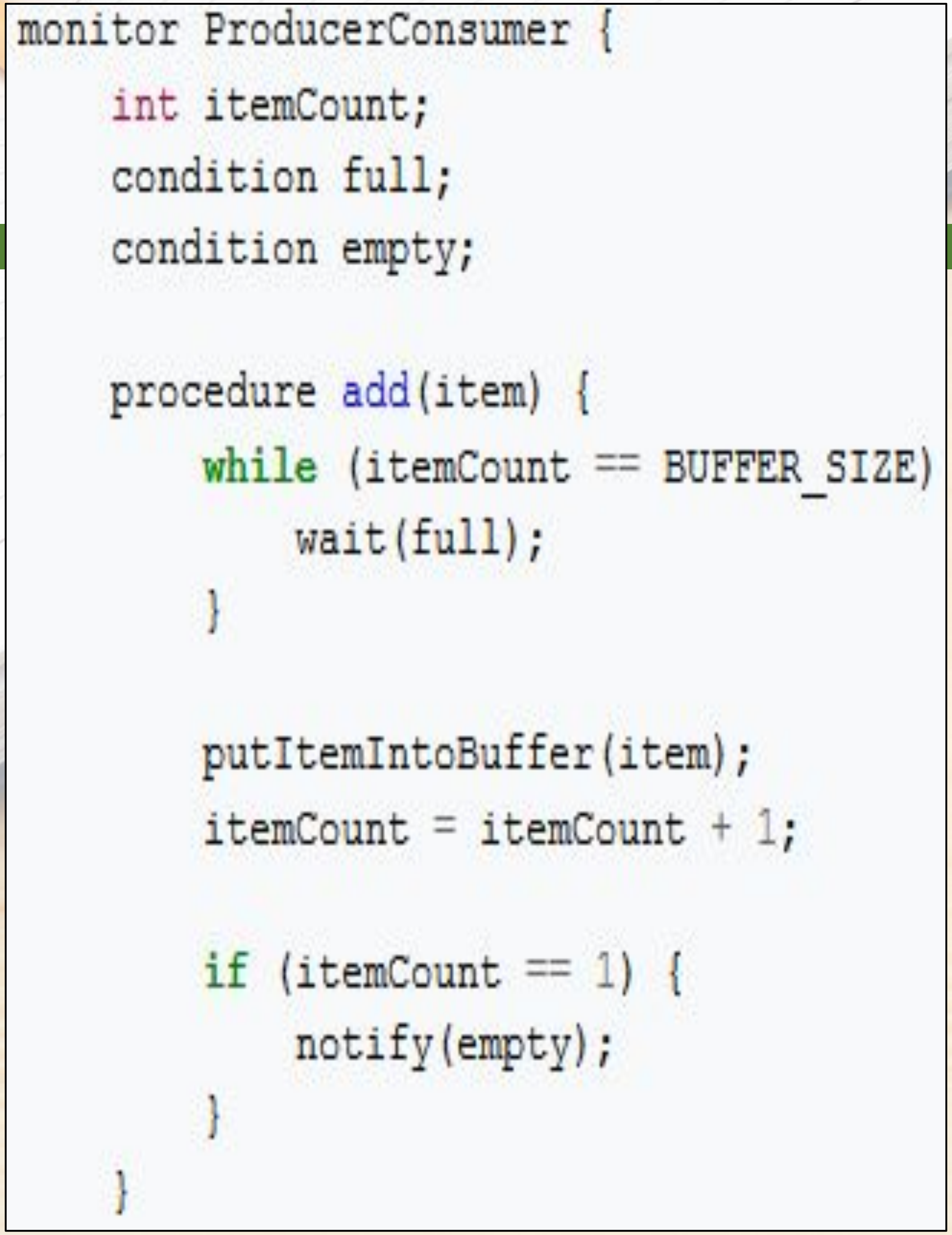
- The solution proposes *condition variables*, along with two operations on them, ***wait and signal***.

- When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, ***full***.

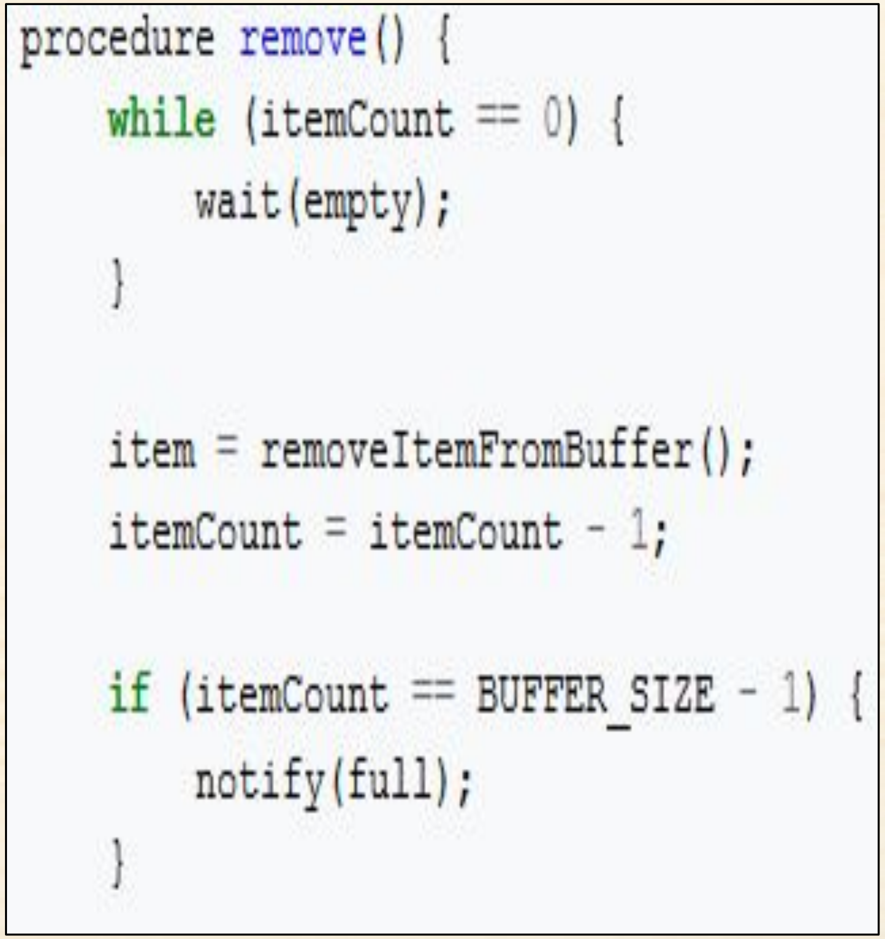
- This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

Producer consumer problem using monitor

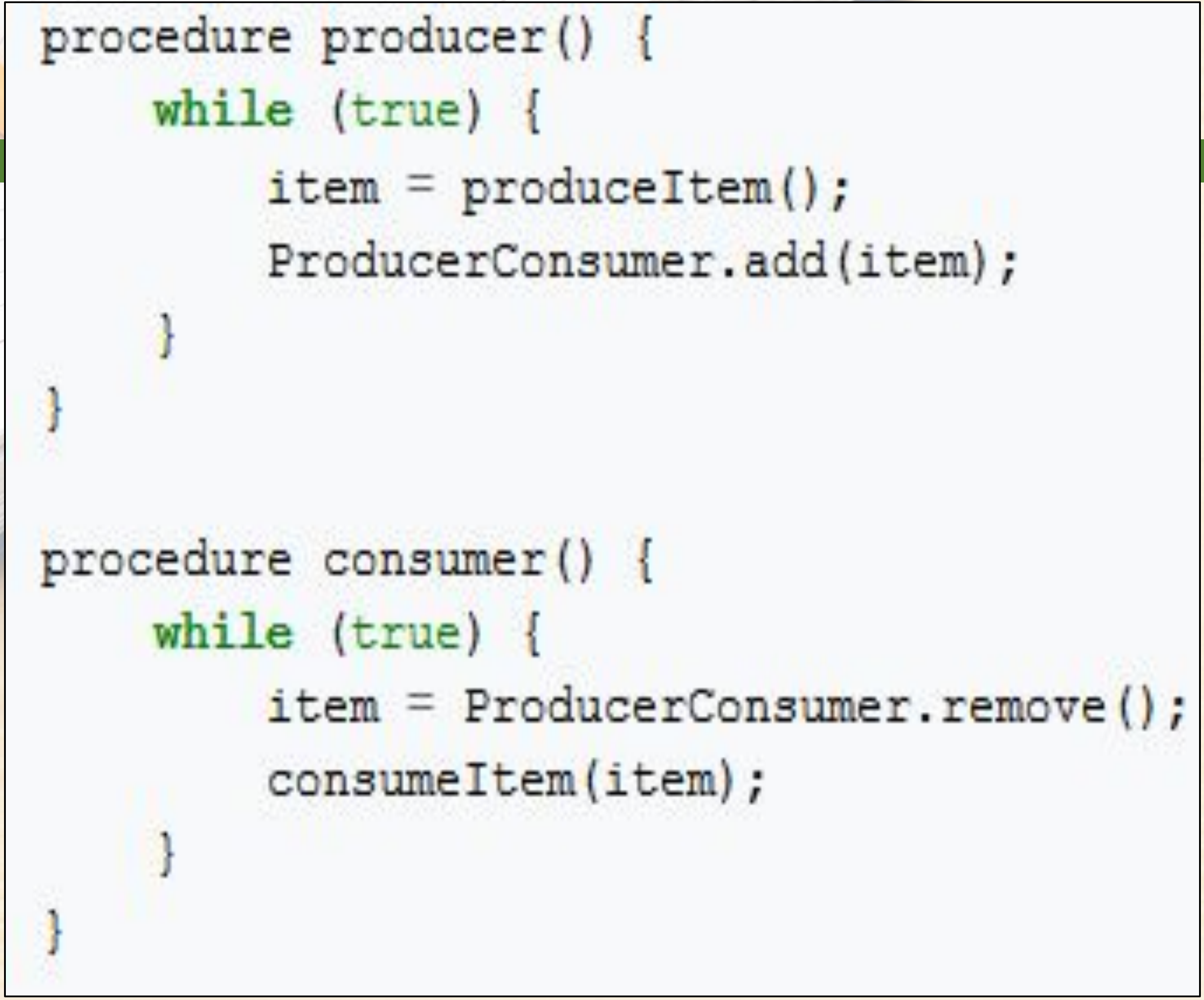
- This other process the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on.
- To avoid having two active processes in the monitor at the same time a signal statement may appear only as the final statement in a monitor procedure.
- If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.



```
monitor ProducerConsumer {
    int itemCount;
    condition full;
    condition empty;

    procedure add(item) {
        while (itemCount == BUFFER_SIZE)
            wait(full);
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            notify(empty);
        }
    }
```



```
procedure remove() {
    while (itemCount == 0) {
        wait(empty);
    }

    item = removeItemFromBuffer();
    itemCount = itemCount - 1;

    if (itemCount == BUFFER_SIZE - 1) {
        notify(full);
    }
```



```
procedure producer() {
    while (true) {
        item = produceItem();
        ProducerConsumer.add(item);
    }
}

procedure consumer() {
    while (true) {
        item = ProducerConsumer.remove();
        consumeItem(item);
    }
}
```

Readers Writer Problem

- Readers writer problem is another example of a classic synchronization problem.

□Problem Statement

- There is a shared resource which should be accessed by multiple processes.
- There are two types of processes in this context. They are **reader** and **writer**.

Problem Statement

- Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource.
- When a **writer** is writing data to the resource, no other process can access the resource.
- A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Solution

- From the above problem statement, it is evident that readers have higher priority than writer.

- Here, we use one mutex (m) and a semaphore (w). An integer variable read_count is used to maintain the number of readers currently accessing the resource.

- The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Cont.

- Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

writer process

```
while(TRUE) {
  wait(w);   // waits on the w semaphore until it gets a chance to write to the resource
  /*perform the
write operation */
     signal(w); //it increments w so that the next writer can access the resource.
}
```

Reader process

```
while(TRUE) {
    wait(m);     //acquire lock
    read_count++;
    if(read_count == 1)
          wait(w);
    signal(m);    //release lock
       /* perform the
        reading operation */
    wait(m);     // acquire lock
    read_count--;
    if(read_count == 0)
            signal(w);
    signal(m);    // release lock
}
```

OR

```
void Reader (void)
{     while (true){
      down(&mutex);      //gain access to reader count
      reader_count++;    //increment reader counter
      if(reader_count==1)   //if this is first process to read DB
            down(&db)            //prevent writer process to access DB
      up(&mutex)                 //allow other process to access reader_count
      read_database();
      down(&mutex);                   //gain access to reader count
      reader_count--;  //decrement reader counter
      if(reader_count==0)             //if this is last process to read DB
            up(&db)            //leave the control of DB, allow writer process
      up(&mutex)                 //allow other process to access reader_count
      use_read_data();}          //use data read from DB (non-critical)
}
```

Cont.

- On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process.
- When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements the read_count value.
- The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section.

Cont.

- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

Message Passing

- One of the two techniques for communicating between parallel processes (the other being shared memory).

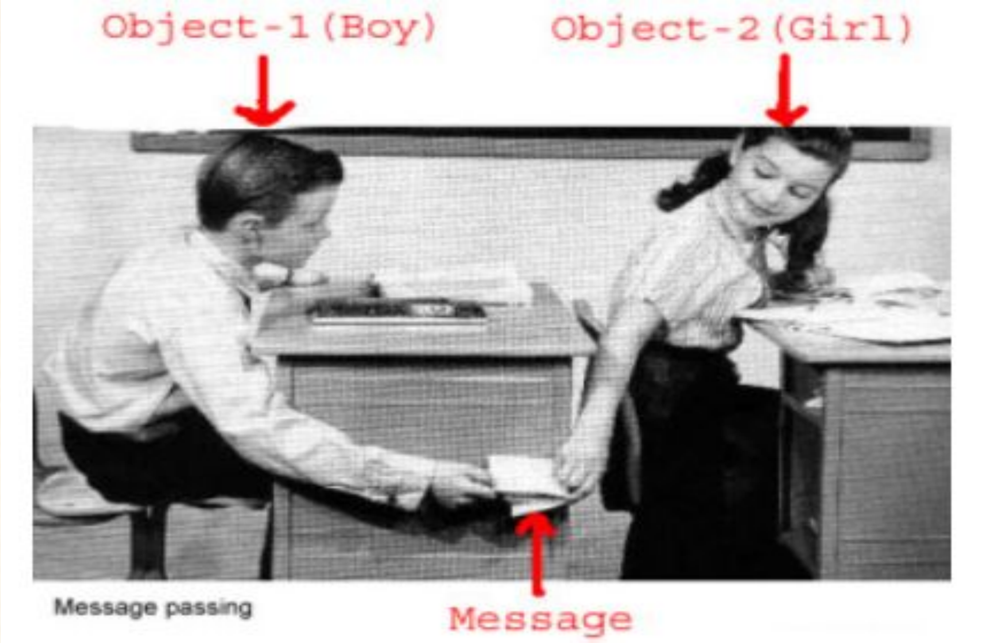
- A common use of message passing is for communication in a parallel computer.

- A process running on one processor may send a message to a process running on the same processor or another.

Cont.

▯ Message passing definition

- Message passing is a form of communication between objects, processes or other resources used in object-oriented programming, inter-process communication and parallel computing.



Cont.

- Message passing can be synchronous or asynchronous.
- Synchronous message passing systems require the sender and receiver to wait for each other while transferring the message.
- In asynchronous communication the sender and receiver do not wait for each other while transfer of messages is being done.

Message Passing

- This method will use two primitives

1. Send: It is used to send message.
   - Send (destination, &message)
   - In above syntax destination is the process to which sender want to send message and message is what the sender wants to send.
2. Receive: It is used to receive message.
   - Receive (source, &message)
   - In above syntax source is the process that has send message and message is what the sender has sent.

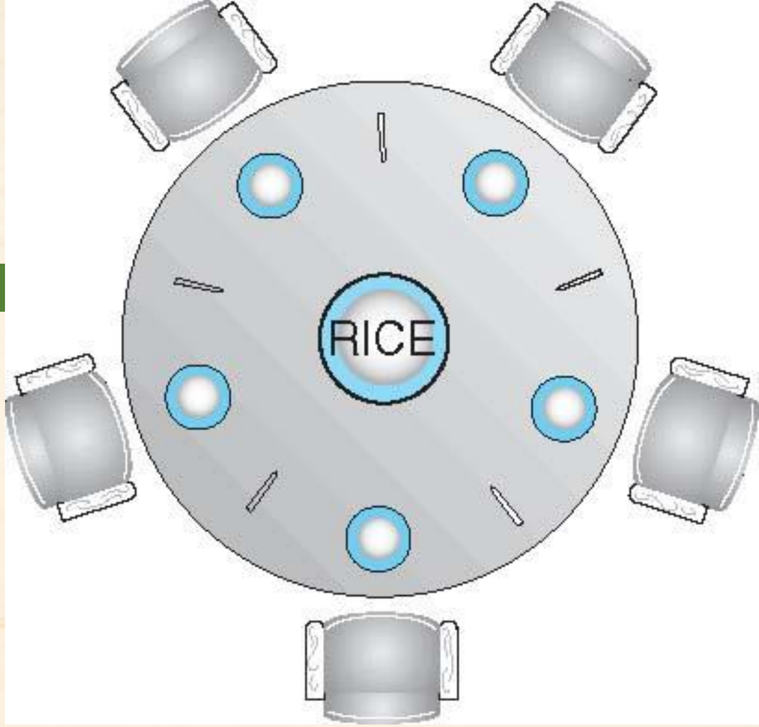
Producer Consumer problem using message passing

```
#define N 100                  //number of slots in buffer
void producer (void)
{
    int item;
    message m;                 //message buffer
    while (true)
    {
    item=produce_item();       //generate something to put in buffer
    receive(consumer, &m);     //wait for an empty to arrive
    build_message(&m, item);   //construct a message to send
    send(consumer, &m);        //send item to consumer
    }}
```

Producer Consumer problem using message passing

```
void consumer (void)
{
    int item, i;
    message m;
    for (i=0; i<N; i++) send (producer, &m);//send N empties
    while (true)
    {
    receive (producer, &m);     //get message containing item
    item=extract_item(&m);      //extract item from message
    send (producer, &m);        //send back empty reply
    consume_item (item);        //do something with the item
    }}
```

Dining Philosopher’s Problem



PROBLEM STATEMENT

- Consider there are five philosophers sitting around a circular dining table.
- The dining table has five chopsticks and a bowl of rice in the middle as shown in the figure.
- At any instant, a philosopher is either eating or thinking.
- When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.

- The case of 5 philosophers (and 5 chopsticks only)
    - Shared data
        - Bowl of rice (data set)
        - Semaphore chopstick [5] initialized to 1 (free)

- **Observation:** Occasionally try to pick up 2 chopsticks (left and right) to eat from bowl
    - One chopstick at a time
    - Need both chopsticks to eat, then release both when done
    - **Problem:** not enough chopsticks for all
        - N philosophes and N chopsticks (not 2N)

- **The structure of Philosopher *i*:**

```
while (true) {
 wait (chopstick[i]);   // wait to get the left stick
 wait (chopstick[(i + 1) % 5]); // get the right

 //  eat

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

   //  think  ///////////////////
};
```

- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick.
- Then he waits for the right chopstick to be available, and then picks it too.
- After eating, he puts both the chopsticks down.

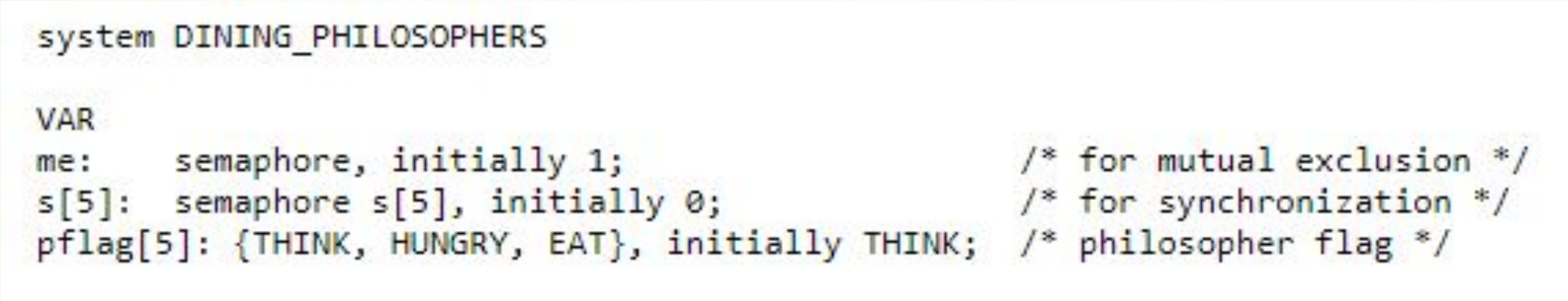
Problem: If all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

□ ***The possible solutions for this are:***

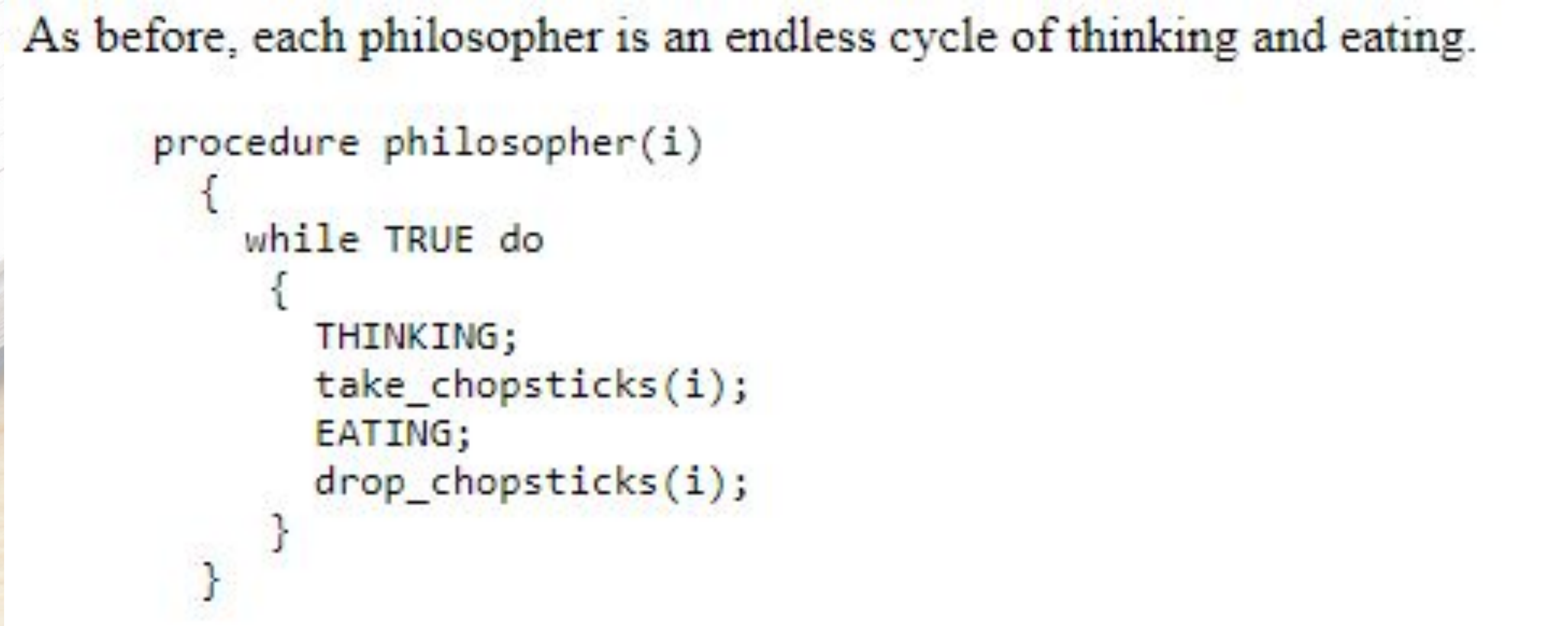
- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers starts eating. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Solution

- This solution uses only Boolean semaphors.
- There is one global semaphore to provide mutual exclusion for exectution of critical protocols.
- There is one semaphore for each chopstick.
- In addition, a local two-phase prioritization scheme is used, under which philosophers defer to their neighbors who have declared themselves "hungry."



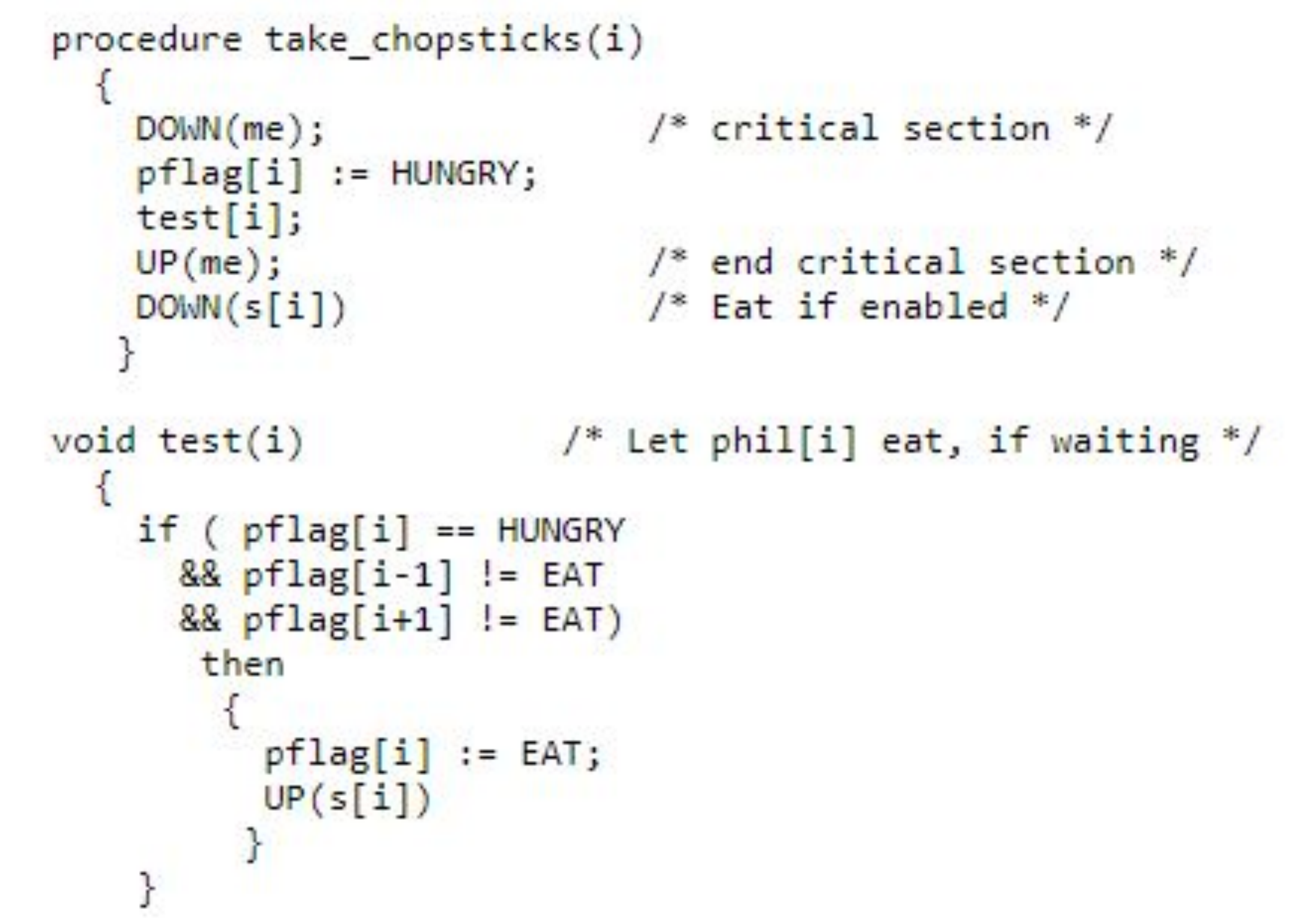
```
system DINING_PHILOSOPHERS

VAR
me:    semaphore, initially 1;                     /* for mutual exclusion */
s[5]:  semaphore s[5], initially 0;                /* for synchronization */
pflag[5]: {THINK, HUNGRY, EAT}, initially THINK;  /* philosopher flag */
```



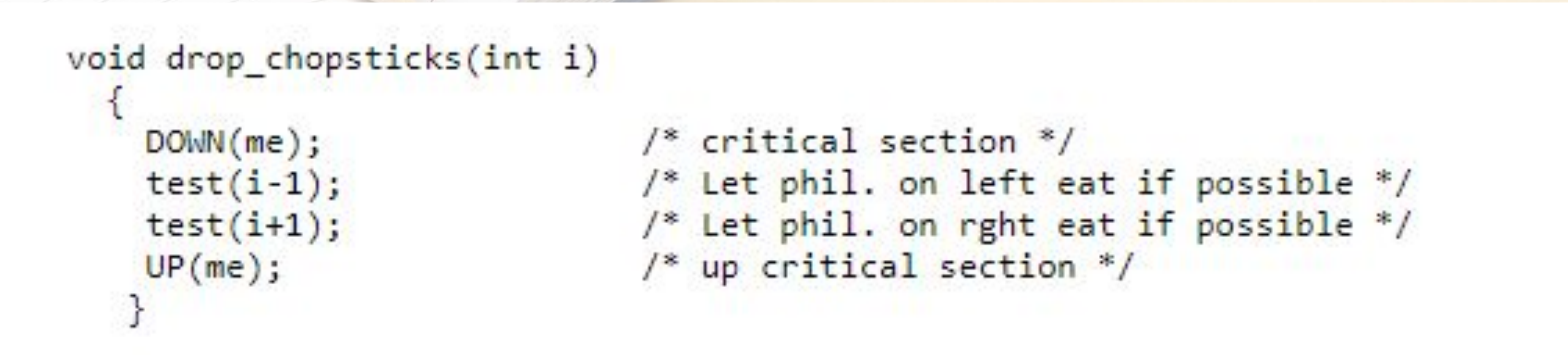
As before, each philosopher is an endless cycle of thinking and eating.

```
procedure philosopher(i)
  {
    while TRUE do
     {
       THINKING;
       take_chopsticks(i);
       EATING;
       drop_chopsticks(i);
     }
  }
```

- The take_chopsticks procedure involves checking the status of neighboring philosophers and then declaring one's own intention to eat. This is a two-phase protocol; first declaring the status HUNGRY, then going on to EAT.



```
procedure take_chopsticks(i)
  {
    DOWN(me);                  /* critical section */
    pflag[i] := HUNGRY;
    test[i];
    UP(me);                    /* end critical section */
    DOWN(s[i])                 /* Eat if enabled */
   }

void test(i)               /* Let phil[i] eat, if waiting */
  {
    if ( pflag[i] == HUNGRY
      && pflag[i-1] != EAT
      && pflag[i+1] != EAT)
       then
        {
          pflag[i] := EAT;
          UP(s[i])
         }
    }
```



```
void drop_chopsticks(int i)
  {
    DOWN(me);                /* critical section */
    test(i-1);               /* Let phil. on left eat if possible */
    test(i+1);               /* Let phil. on rght eat if possible */
    UP(me);                  /* up critical section */
   }
```

Event counters

- An event counter is a special data type that contains an integer value that can only be incremented. Three operations are defined for event counters:
- read(E): return the current value of event counter E
- advance(E): increment E
- await(E,v): wait until E has a value greater than or equal to v

Cont.

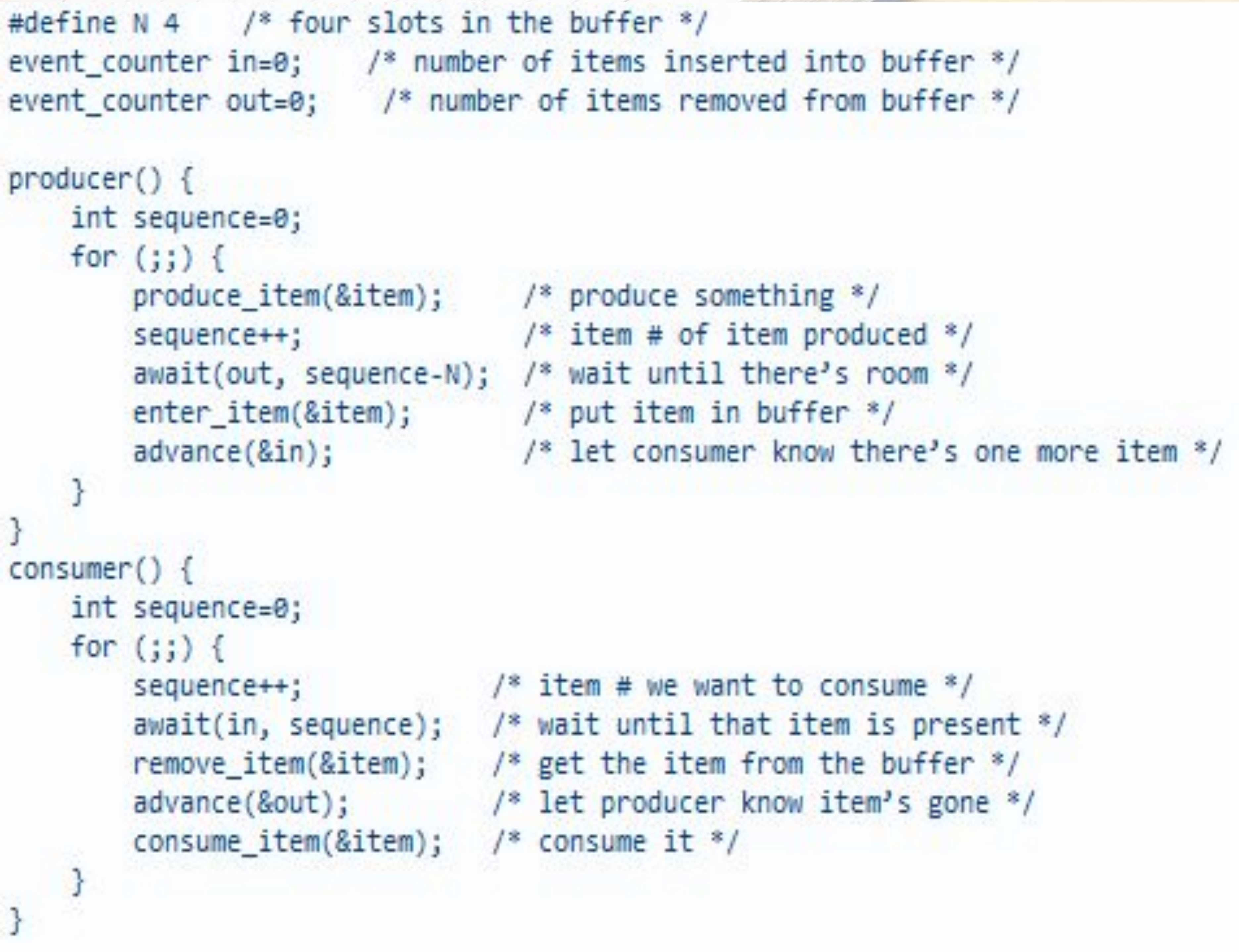
- Here is an example of the producer-consumer problem implemented with event counters.
- Both the consumer and producer maintain a sequence number locally.
- Think of the sequence number as the serial number of each item that the producer produces.
- From the consumer's point, think of the sequence number as the serial number of the next item that the consumer will consume.

Cont.

- *in* is the number of the latest item that was added to the buffer. The event counter *out* is the serial number of the latest item that has been removed from the buffer.

- The producer needs to ensure that there’s a free slot in the buffer and will wait (sleep) until the difference between the sequence number and *out* (the last item consumed) is less than the buffer size.

- The consumer needs to wait (sleep) until there is at least one item in the buffer; that is, *in* is greater than or equal to the next sequence number that it needs to consume.



```
#define N 4    /* four slots in the buffer */
event_counter in=0;    /* number of items inserted into buffer */
event_counter out=0;    /* number of items removed from buffer */

producer() {
    int sequence=0;
    for (;;) {
        produce_item(&item);    /* produce something */
        sequence++;             /* item # of item produced */
        await(out, sequence-N); /* wait until there’s room */
        enter_item(&item);      /* put item in buffer */
        advance(&in);           /* let consumer know there’s one more item */
    }
}
consumer() {
    int sequence=0;
    for (;;) {
        sequence++;           /* item # we want to consume */
        await(in, sequence);  /* wait until that item is present */
        remove_item(&item);   /* get the item from the buffer */
        advance(&out);        /* let producer know item’s gone */
        consume_item(&item);  /* consume it */
    }
}
```

ABHIJITSINH PARMAR