PROCESS SYNCHRONIZATION

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Types of Process

- Cooperating Process
- Competitive Process
- Producer Consumer Problem is a type of cooperative process.

Interprocess communication

- Processes frequently need to communicate with other processes.
- For example, in a shell pipeline, the output of the first process must be passed to second process, and so on down the line.
- Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts.
- This we refer as InterProcess Communication or IPC.

Interprocess communication

Very briefly, there are three issues here.

- **How one process can pass information to another.**
- **How two process can work in isolation at the time of shared resource**.
- **How to maintain proper sequencing when dependencies are present:**

Interprocess communication

- In some operating systems, processes that are working together may share some common storage that each one can read and write.
- Example
  - Memory Block
  - Shared file
  - Shared Variable

Example of IPC

A print spooler.

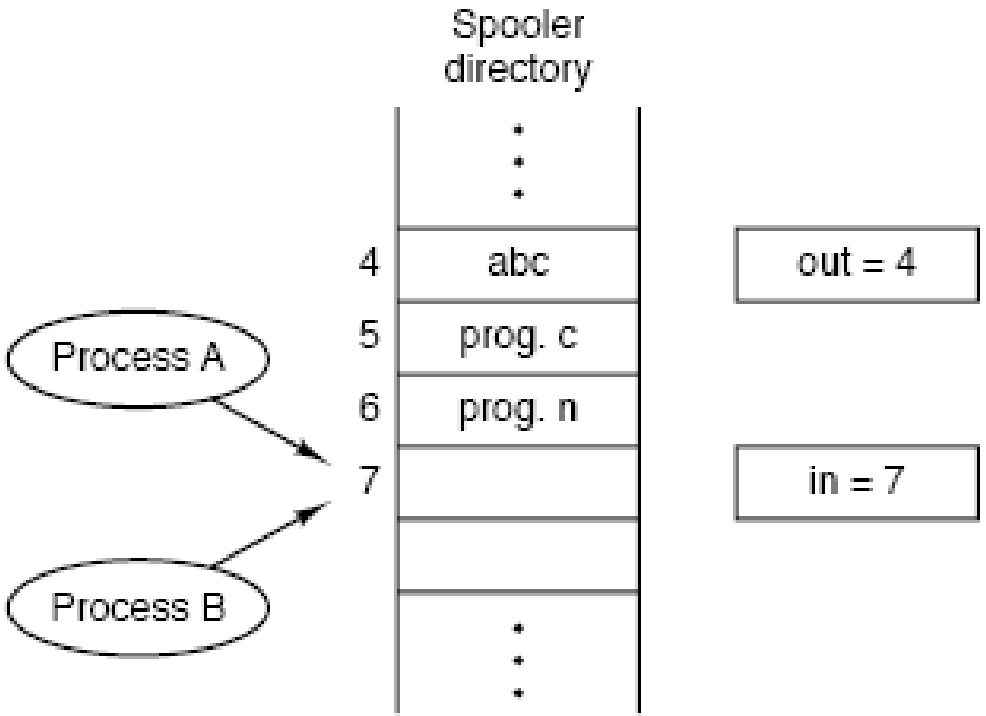
- When a process wants to print a file, it enters the file name in a special spooler directory.
- Another process, the printer daemon, periodically checks to see if so are any files to be printed, and if so removes their names from the directory and do the Job.

Example of IPC

- Imagine that our spooler directory has a large (potentially infinite) number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name.
- Also imagine that there are two shared variables:
  - **Out:** points to the next file to be printed.
  - **In:** points to the next free slot in the directory.

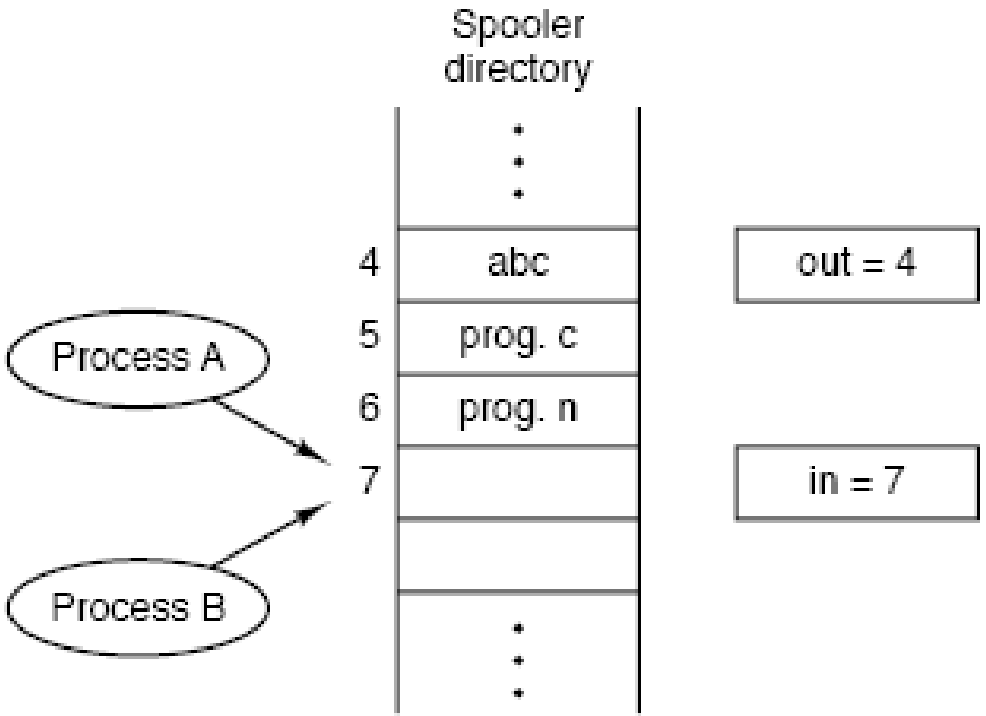
Example of IPC

- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed).



Example of IPC

- More or less simultaneously, processes A and B decide they want to queue a file for printing.



Example of IPC

- **The following might well happen.**
- Process A reads **in** and stores the value, 7, in a local variable called next free slot.
- Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- Process B also reads **in,** and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8.
- Then it goes off and does other things.

Example of IPC

- Eventually, process A runs again, starting from the place it left off last time.
- It looks at next free slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.
- Then it computes next free slot + 1, which is 8, and sets in to 8.

RACE CONDITION

- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.
- **Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.**

Race Conditions

- A race condition occurs when two processes can interact and the outcome depends on the order in which the processes execute.
- Assume two processes both accessing x (initial value 10).
- One process is to execute x = x+1
- The other is to execute x = x-1
- When both are finished x should be 10
  - But we might get 9 or 11!

Critical Sections

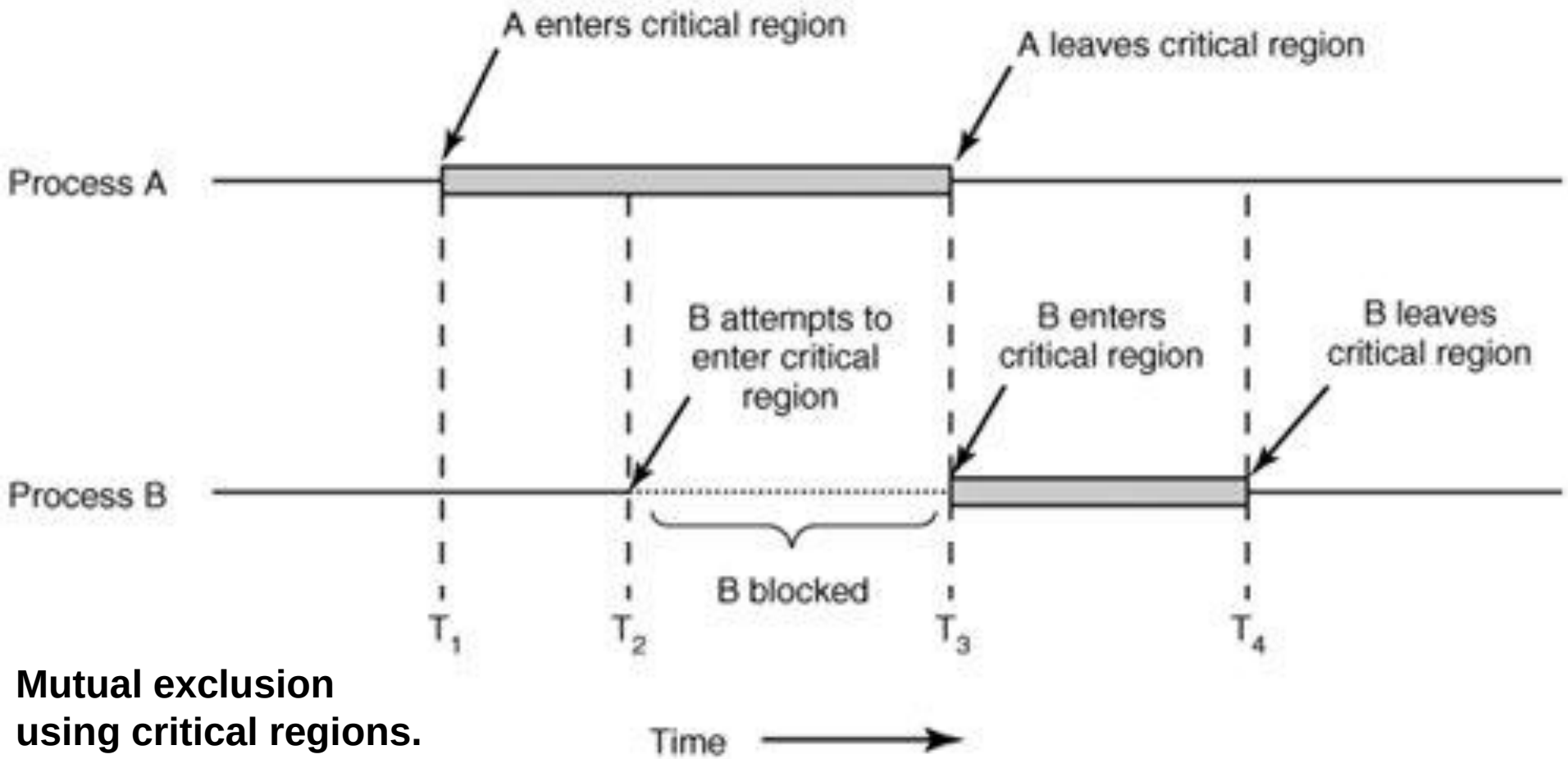
- How do we avoid race conditions?
- It is important to find some way to prohibit more than one process from reading and writing the shared data at the same time.
- **what we need is mutual exclusion**
- *The difficulty above occurred because process B started using one of the shared variables before process A was finished with it.*

Critical Sections

- *That part of the program where the shared memory is accessed is called the critical region or critical section.*
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

Critical Sections

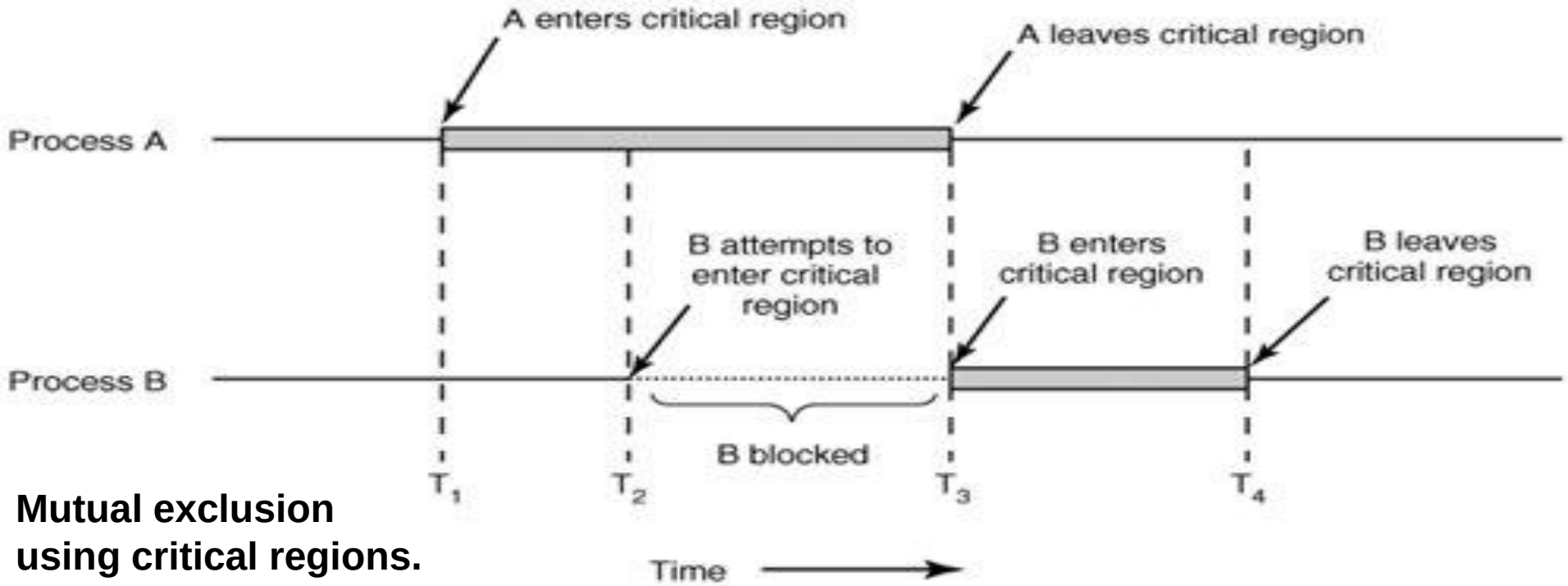
- Here process *A enters its critical region at time* T_1



Mutual exclusion using critical regions.

Critical Sections

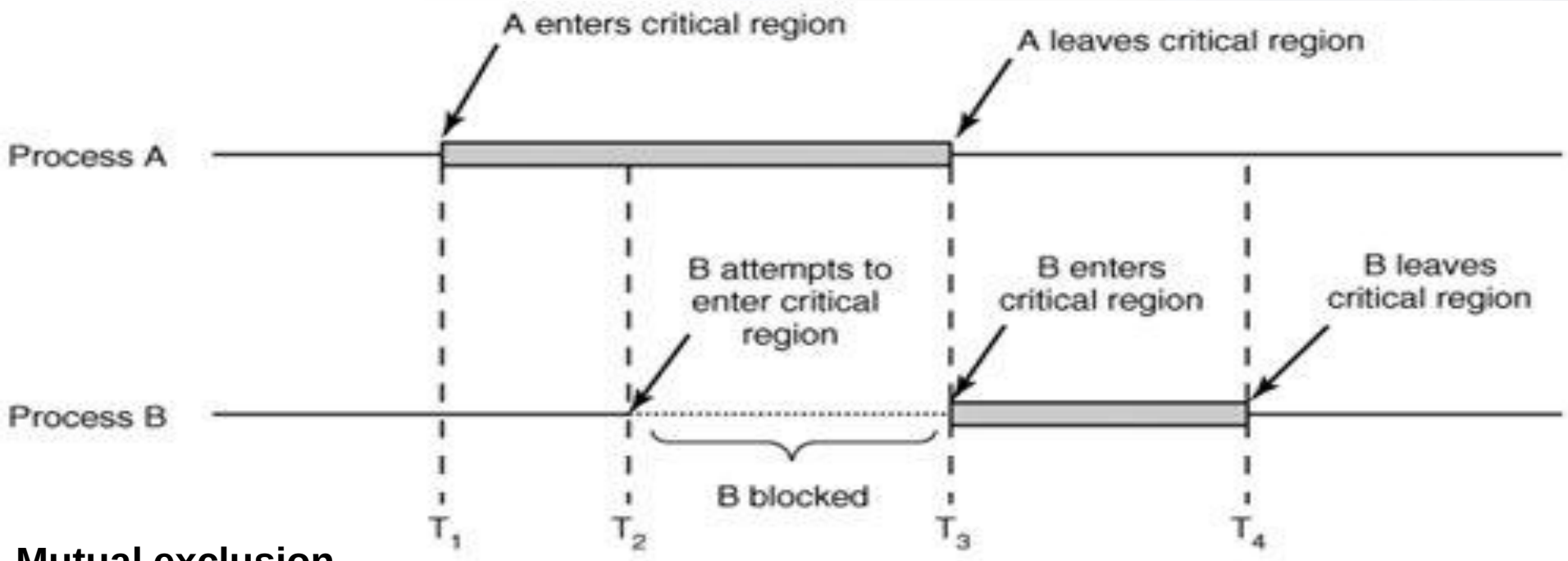
- At time T_2 *process B attempts to enter its critical region but fails because another* process is already in its critical region and we allow only one at a time.



Mutual exclusion using critical regions.

Critical Sections

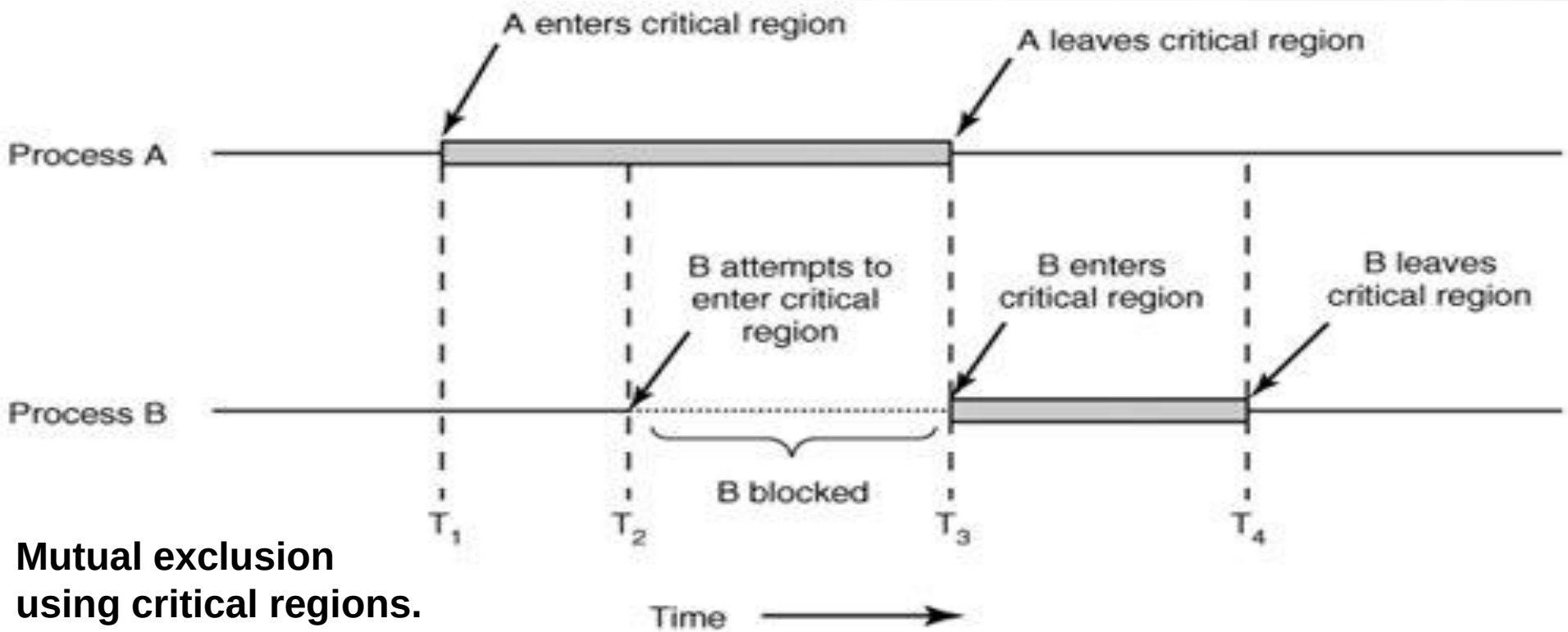
- Consequently, *B is* temporarily suspended until time *T 3 when A leaves its critical region, allowing B to enter*



Mutual exclusion using critical regions.

Critical Sections

- Eventually *B leaves (at T_4) and we are back to the original situation with no* processes in their critical regions.



Mutual exclusion using critical regions.

Solution to Critical-Section Problem

Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Solution to Critical-Section Problem

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

Solution to Critical-Section Problem

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

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

How to implement Mutual Exclusion

- Disabling interrupts
- Lock Variables
- Turn Variables
- Peterson solution

Disabling Interrupts

- The simplest solution : **each process disable all interrupts just after entering its critical region and allow them just before leaving it.**
- With interrupts disabled, no clock interrupts can occur.
- CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without intervention of another process.

Disabling Interrupts

- **This approach is generally unattractive because**
  - it is unwise to give user processes the power to turn off interrupts.
  - Suppose that one of them did, and then never turned them on again?

Lock Variables

- Consider having a single, shared, (lock) variable, initially **0**.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is **0**, the process sets it to **1** and enters the critical region.

Lock Variables

- If the lock is already **1**, the process just waits until it becomes **0**.
- Thus, a **0** means that no process is in its critical region, and a **1** means that some process is in its critical region.

Drawback with Lock Variables

- Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.
- Suppose that one process reads the lock and sees that it is **0**.
- Before it can set the lock to **1**, another process is scheduled, runs, and sets the lock to **1**.
- When the first process runs again, it will also set the lock to **1**, and two processes will be in their critical regions at the same time.

Turn Variable: Busy Waiting

- The integer variable turn , initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process 0 inspects turn , finds it to be 0, and enters its critical region.
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continuously testing a variable until some value appears is called **busy waiting .**

Turn Variable: Busy Waiting

- **It should usually be avoided, since it wastes CPU time.**
- Only when there is a reasonable expectation that the wait will be short is busy waiting used.
- A lock that uses busy waiting is called a **spin lock**

Turn Variable: Busy Waiting

```
while (TRUE){
     while(turn != 0)          /* loop* /;
     critical_region();
     turn  = 1;
     noncritical_region();
}
```

(a)

```
while (TRUE) {
     while(turn != 1)          /* loop* /;
     critical_region();
     turn  = 0;
     noncritical_region();
}
```

(b)

Peterson's solution for achieving mutual exclusion.

- Before using the shared variables (i.e., before entering its critical region), each process calls enter_region with its own process number, 0 or 1, as the parameter.
- This call will cause it to wait, if need be, until it is safe to enter.
- After it has finished with the shared variables, the process calls leave_region to indicate that it is done and to allow the other process to enter, if it so desires.

Peterson's solution for achieving mutual exclusion.

- Let us see how this solution works.
- Initially, neither process is in its critical region.
- Now process 0 calls enter_region .
- It indicates its interest by setting its array element and sets turn to 0.
- Since process 1 is not interested, enter_region returns immediately.

Peterson's solution for achieving mutual exclusion.

- If process 1 now calls enter_region, it will hang there until interested [0] goes to FALSE , an event that only happens when process 0 calls leave_region to exit the critical region.

Peterson's solution for achieving mutual exclusion.

- Now consider the case that both processes call enter_region almost simultaneously.
- Both will store their process number in turn . Whichever store is done last is the one that counts; the first one is lost.
- Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

Peterson's solution for achieving mutual exclusion.

```
#define FALSE 0
#define TRUE 1
#define N      2                    /* number of processes */
int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE)*/
void enter_region(int process)      /* process is 0 or 1 */
{
int other;                          /* number of the other process */
other = 1 - process;                /* the opposite of process */
interested[process] = TRUE; /* show that you are interested */
turn = process;                             /* set flag */
```

Peterson's solution for achieving mutual exclusion.

```
while (turn == process && interested[other] == TRUE)
                                        /* null statement */;
}
void leave_region(int process)
                                    /* process: who is leaving */
{
interested[process] = FALSE;
                        /* indicate departure from critical region */

}
```

The TSL Instruction

- Now let us look at a proposal that requires a little help from the hardware.
- Many computers, especially those designed with multiple processors in mind, have an instruction
- **TSL RX, LOCK**
- **Test Set Lock**
- **RX (register)**
- **Lock (memory)**

The TSL Instruction

- It reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK .

- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

The TSL Instruction

- To use the TSL instruction, we will use a shared variable, LOCK , to coordinate access to shared memory.
- When LOCK is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory.
- When it is done, the process sets LOCK back to 0 using an ordinary move instruction.

Sleep and Wakeup

- Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting.
- Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The wakeup call has one parameter, the process to be awakened.

Producer Consumer Problem

- Producer process produce information that is consumed by consumer process.
- Complier may produce an assembly code which is consumed by an assembler.
- An assembler may produce an object code which is consumed by the loader.
- Producer Consumer problem is very similar to Client Server Environment.

Background

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
- We can do so by having an integer count that keeps track of the number of full buffers.
- Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer Consumer Problem

- Suppose Server can work as a producer process and client can work as a consumer process.
- For example: Web server can produces or provide HTML pages, contents and images which are consumed or read by Web browsers as clients.

Producer Consumer Problem

- **Solution for the Producer Consumer Problem is shared memory.**
- To allow producer consumer process to run concurrently we have available a buffer of items that can be filled by the producer and emptied by the consumer.
- The buffer may reside in a portion of memory which is shared by both producer and the consumer.

Producer Consumer Problem

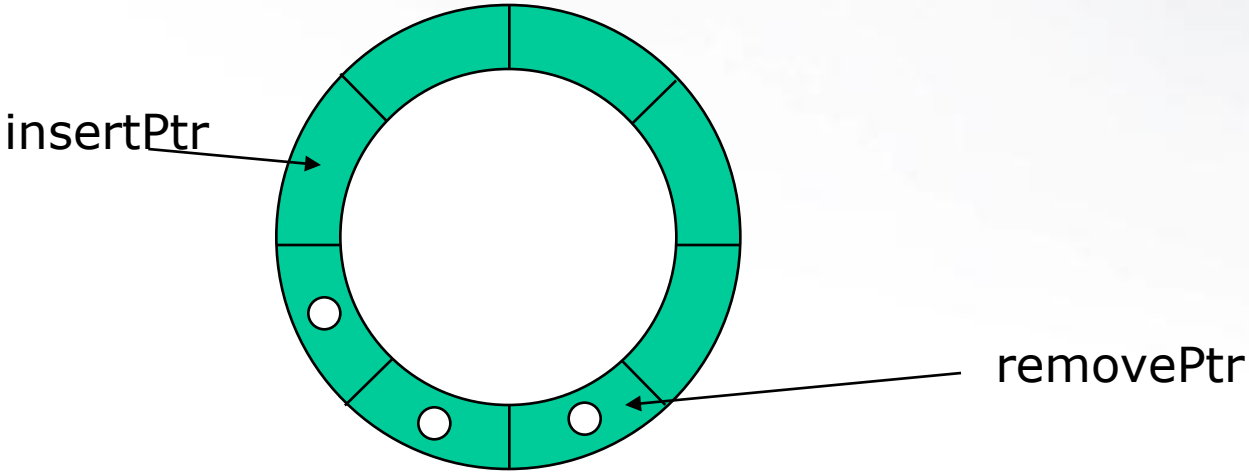
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- If the producer encounters a full buffer, or if the consumer encounters an empty buffer, the process blocks.

Producer Consumer Problem

- Two types of Buffer has been used
- **Unbounded Buffer:** No practical limit on the size of the Buffer.
- **Bounded Buffer:** Fixed Buffer Size.

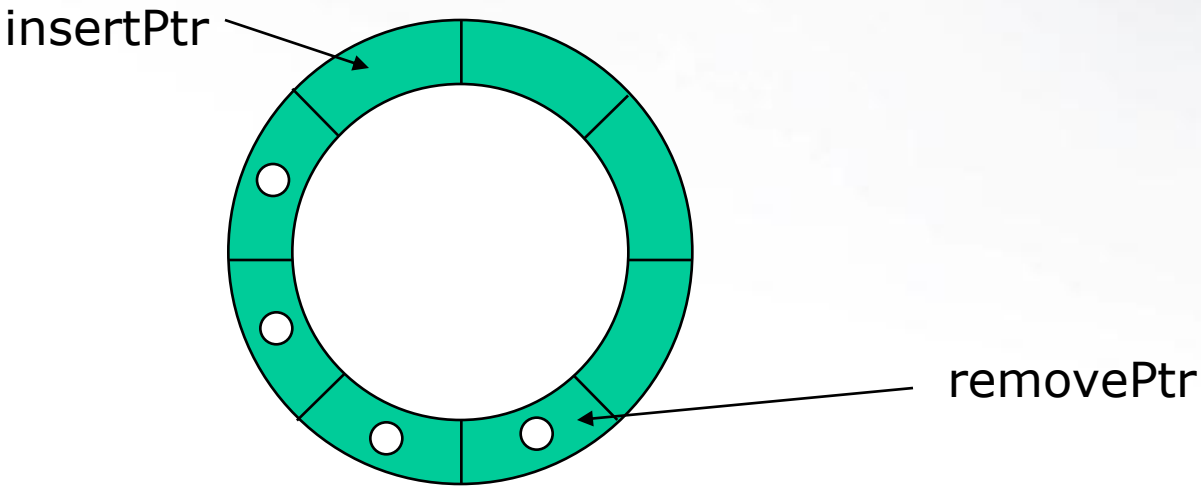
Producer-Consumer

Chef = Producer
Customer = Consumer



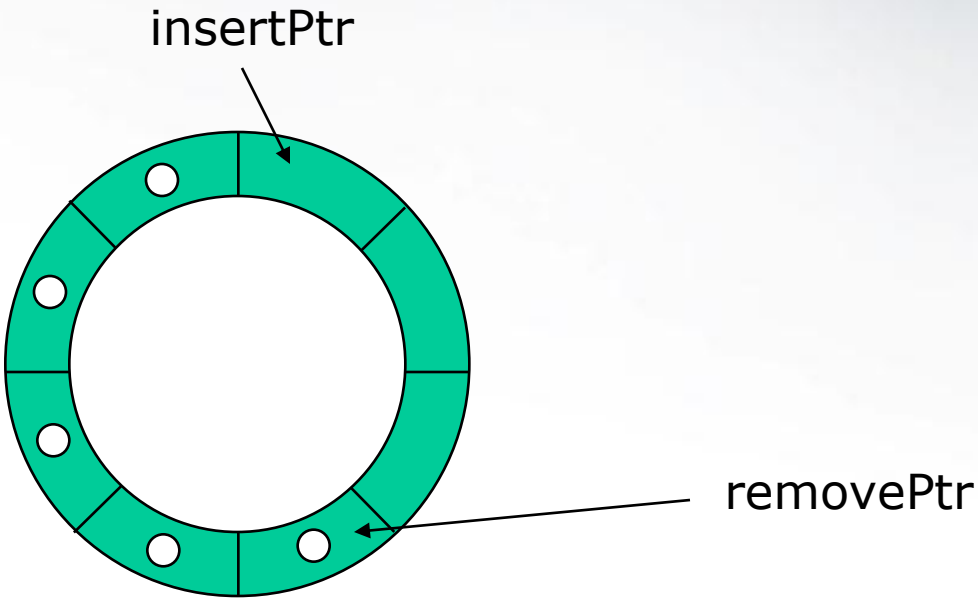
Producer-Consumer

Chef = Producer
Customer = Consumer



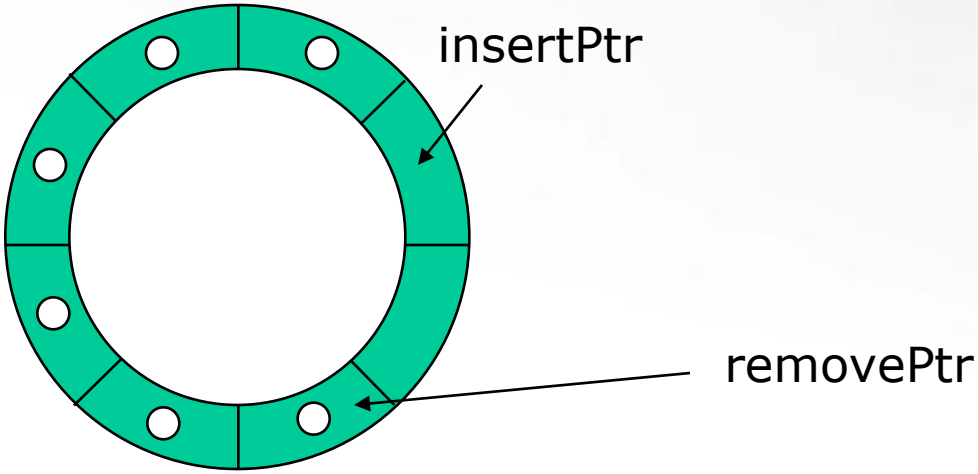
Producer-Consumer

Chef = Producer
Customer = Consumer



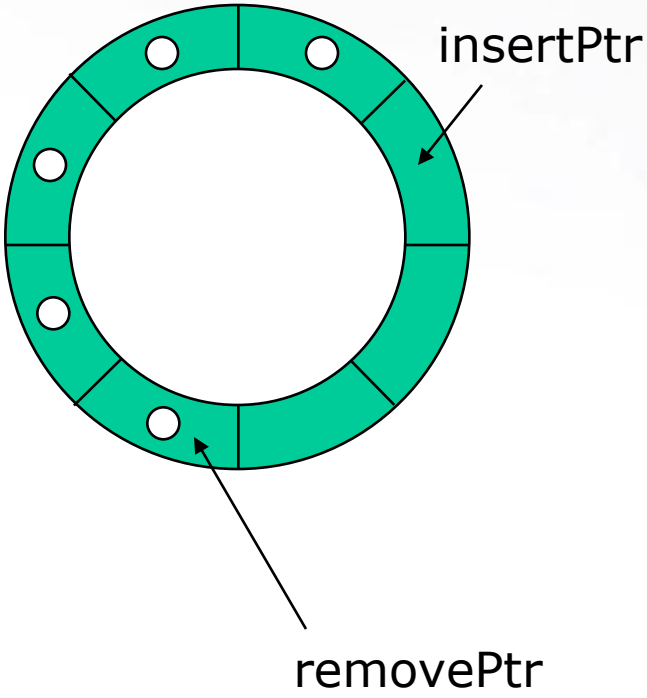
Producer-Consumer

Chef = Producer
Customer = Consumer



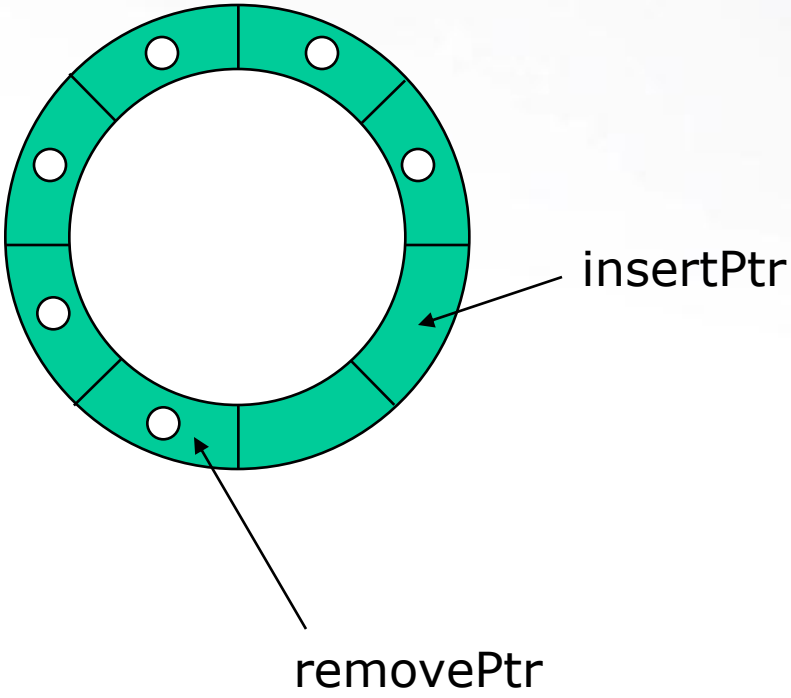
Producer-Consumer

Chef = Producer
Customer = Consumer



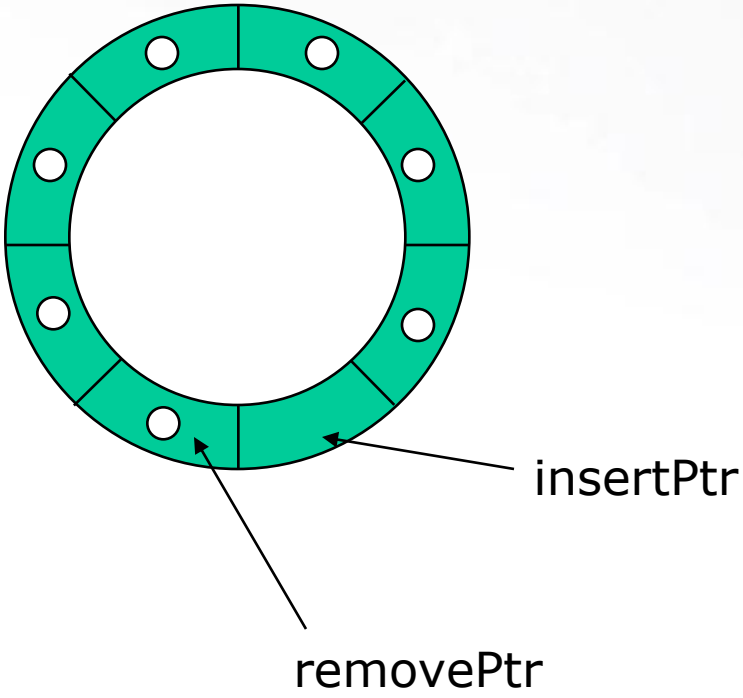
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

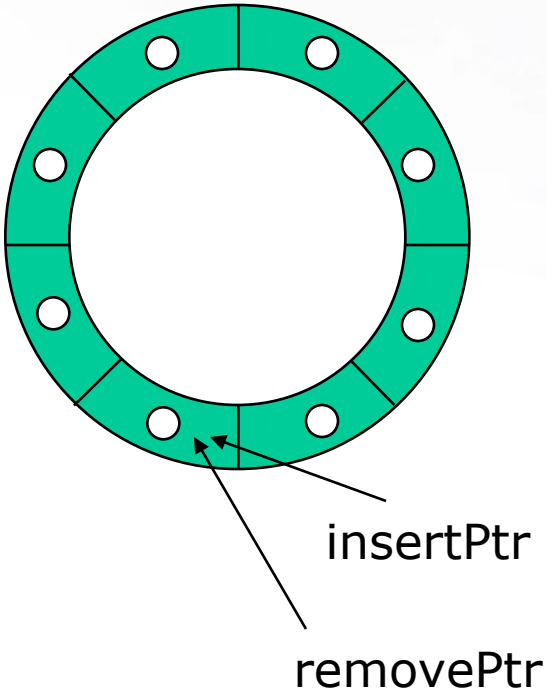
Chef = Producer
Customer = Consumer



Producer-Consumer

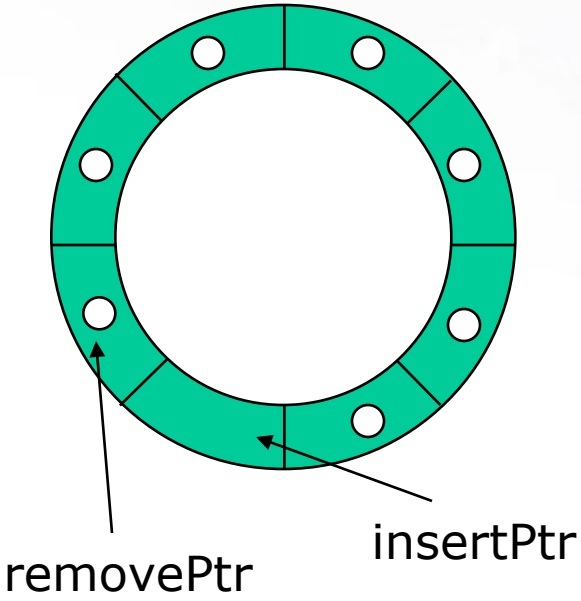
Chef = Producer
Customer = Consumer

BUFFER FULL: Producer must be blocked!



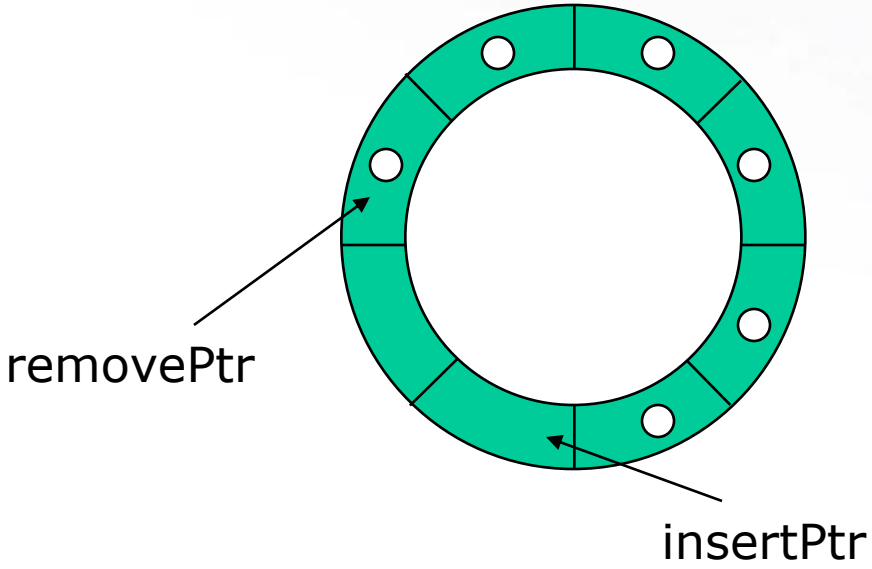
Producer-Consumer

Chef = Producer
Customer = Consumer



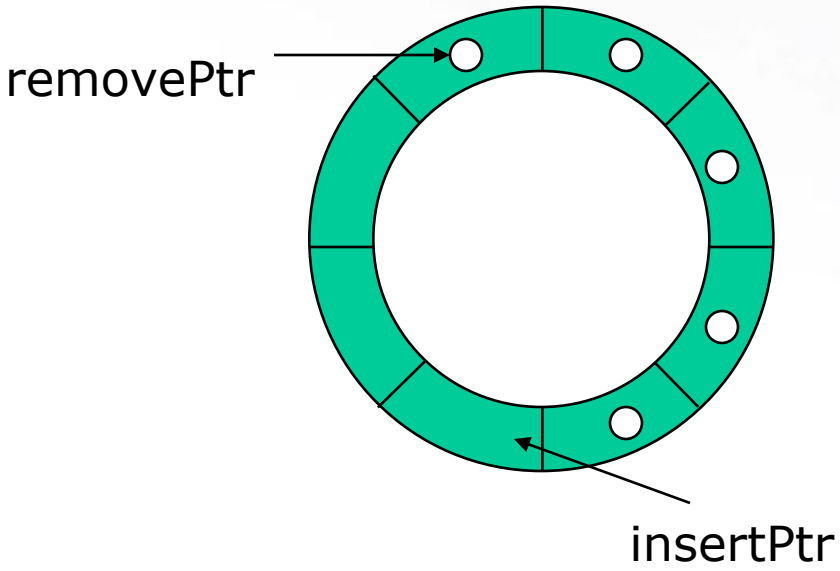
Producer-Consumer

Chef = Producer
Customer = Consumer



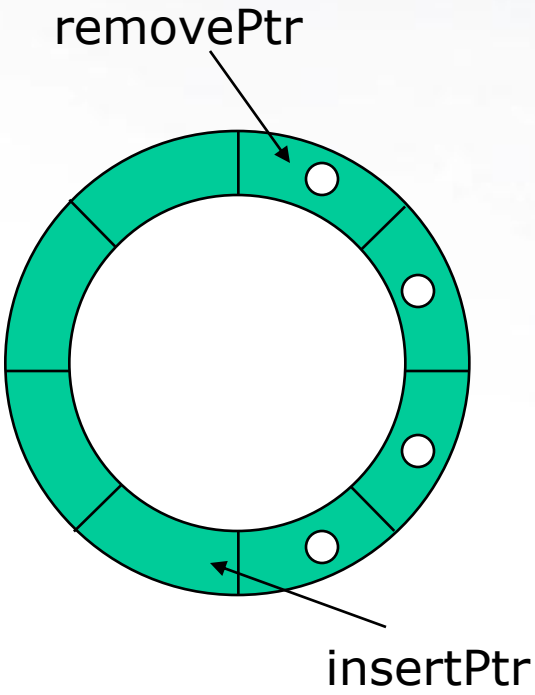
Producer-Consumer

Chef = Producer
Customer = Consumer



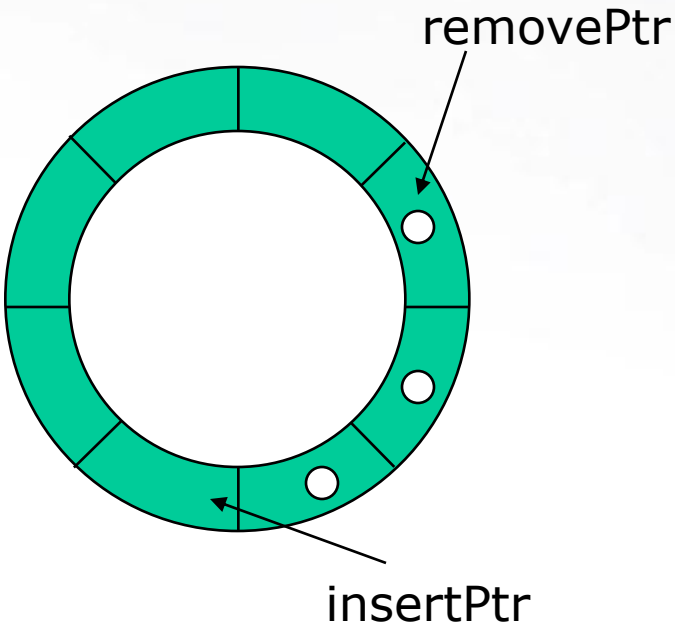
Producer-Consumer

Chef = Producer
Customer = Consumer



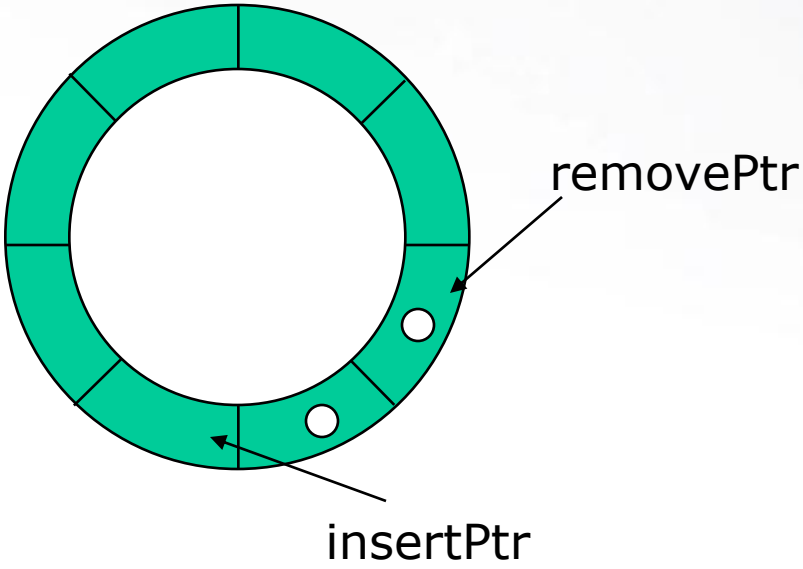
Producer-Consumer

Chef = Producer
Customer = Consumer



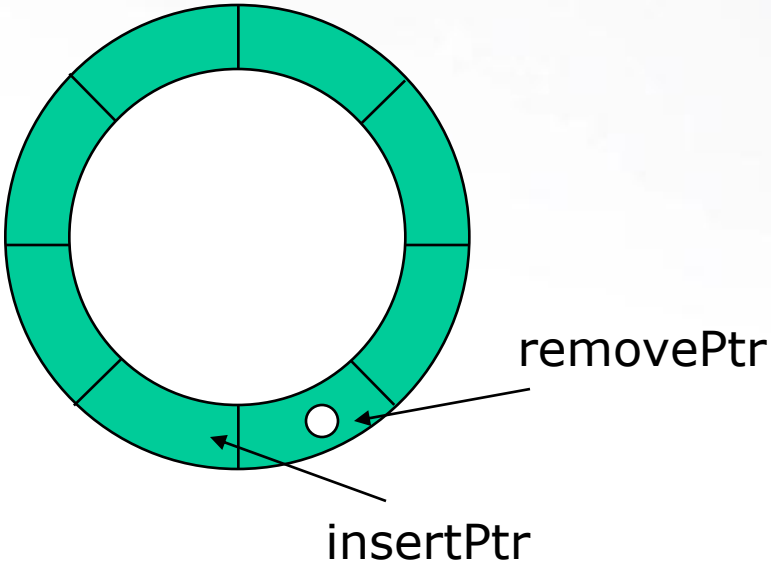
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

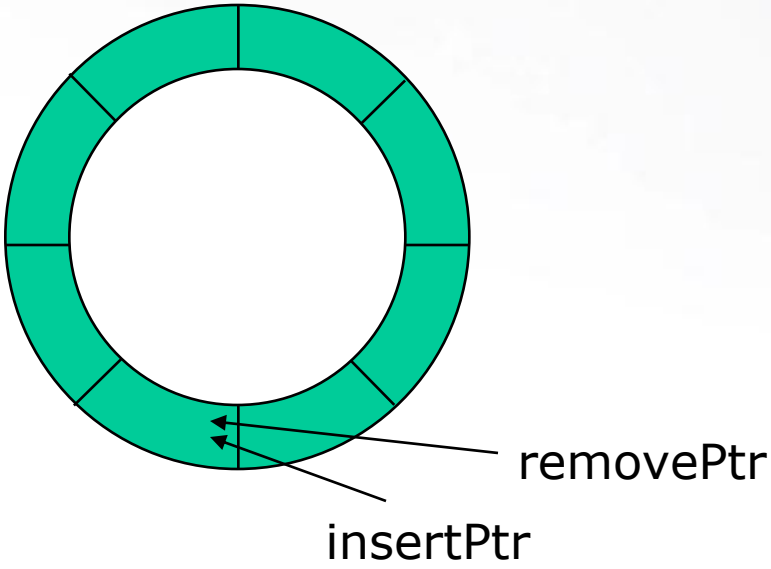
Chef = Producer
Customer = Consumer



Producer-Consumer

Chef = Producer
Customer = Consumer

BUFFER EMPTY: Consumer must be blocked!



Producer-Consumer Problem

Chef = Producer
Customer = Consumer

- Producers insert items
- Consumers remove items
- Shared bounded buffer *

* Efficient implementation is a circular buffer with an insert and a removal pointer.

Producer-Consumer Problem

- Producer inserts items. Updates insertion pointer.
- Consumer executes destructive reads on the buffer. Updates removal pointer.
- Both update information about how full/empty the buffer is.
- Solution should allow multiple producers and consumers

Challenges

- Prevent buffer overflow
- IF NO FREE SLOT= BLOCK PRODUCER
- Prevent buffer underflow
- IF ALL FREE SLOT= BLOCK PRODUCER
- Proper synchronization

Semaphore

- A new variable type, called a **semaphore** , was introduced.
- A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Semaphore

- Dijkstra proposed having two operations, down and up (which are generalizations of sleep and wakeup , respectively).
- The down operation on a semaphore checks to see if the value is greater than 0.
- If so, it decrements the value (i.e., uses up one stored wakeup) and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment.
- Same with Up.

Sleep and Wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Producer-consumer problem with fatal race condition

Semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

The producer-consumer problem using semaphores

Binary semaphores

- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores .**
- If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Mutexes

- A **mutex is a variable that can be in one of two states: unlocked or locked.**
- **Consequently, only 1 bit** is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

Mutexes

- Two procedures are used with mutexes.
- When a process (or thread) needs access to a critical region, it calls *mutex_lock . If the mutex is currently unlocked (meaning* that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

Mutexes

- On the other hand, if the mutex is already locked, the caller is blocked until the process in the critical region is finished and calls mutex_unlock .
- If multiple processes are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Monitor

- 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

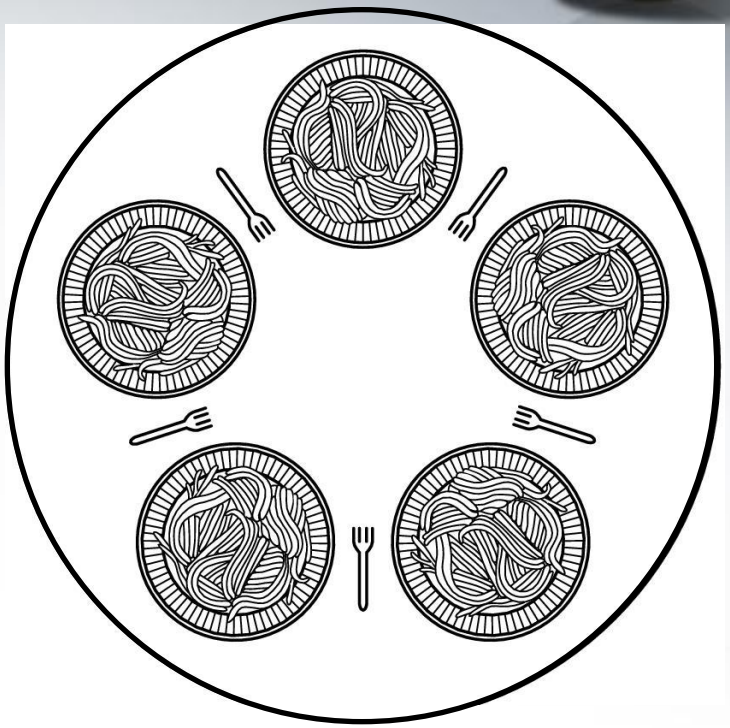
Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Dining Philosophers (1)

- The key question is: can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

Dining Philosophers Deadlock

- The procedure take_fork waits until the specified fork is available and then seizes it.
- Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously.
- None will be able to take their right forks, and there will be a deadlock.

Dining Philosophers Deadlock

- We could modify the program so that after taking the left fork, the program checks to see if the right fork is available.
- If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process.

Dining Philosophers Starvation

- This proposal too, fails, although for a different reason.
- With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever.

Dining Philosophers Starvation

- A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called **starvation .**

- "If the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small."

Dining Philosophers (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers (3)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}
```

Solution to dining philosophers problem (part 1)

Dining Philosophers (4)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
      test(i);                          /* try to acquire 2 forks */
      up(&mutex);                       /* exit critical region */
      down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = THINKING;              /* philosopher has finished eating */
      test(LEFT);                       /* see if left neighbor can now eat */
      test(RIGHT);                      /* see if right neighbor can now eat */
      up(&mutex);                       /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```


Solution to dining philosophers problem (part 2)

The Readers and Writers Problem

- Another famous problem is the readers and writers problem which models access to a database.
- Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader.

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.

The Readers and Writers Problem

- The question is how do you program the readers and the writers?
- Allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

The Sleeping Barber Problem

- Another classical IPC problem takes place in a barber shop.
- The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on.
- If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig.

The Sleeping Barber Problem

The Sleeping Barber Problem

- When a customer arrives, he has to wake up the sleeping barber.
- If additional customers arrive while the barber is cutting a customer’s hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full).
- The problem is to program the barber and the customers without getting into race conditions.