PROCESS MANAGEMENT

Thread

- A thread is a single sequence stream within in a process.
- Because threads have some of the properties of processes, they are sometimes called lightweight process.
- In a process, threads allow multiple executions of streams.
- In many respect, threads are popular way to improve application through parallelism.

Thread

- The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.
- Like a traditional process i.e., process with one thread, a thread can be in any of several states **(Running, Blocked, Ready or Terminated).**

Thread

- Each thread has its own stack.
- Stack contains temporary data such as
  - Function parameters
  - Return Addresses
  - Local Variables etc.
- Since thread will generally call different procedures and thus a different execution history this is why thread needs its own stack.

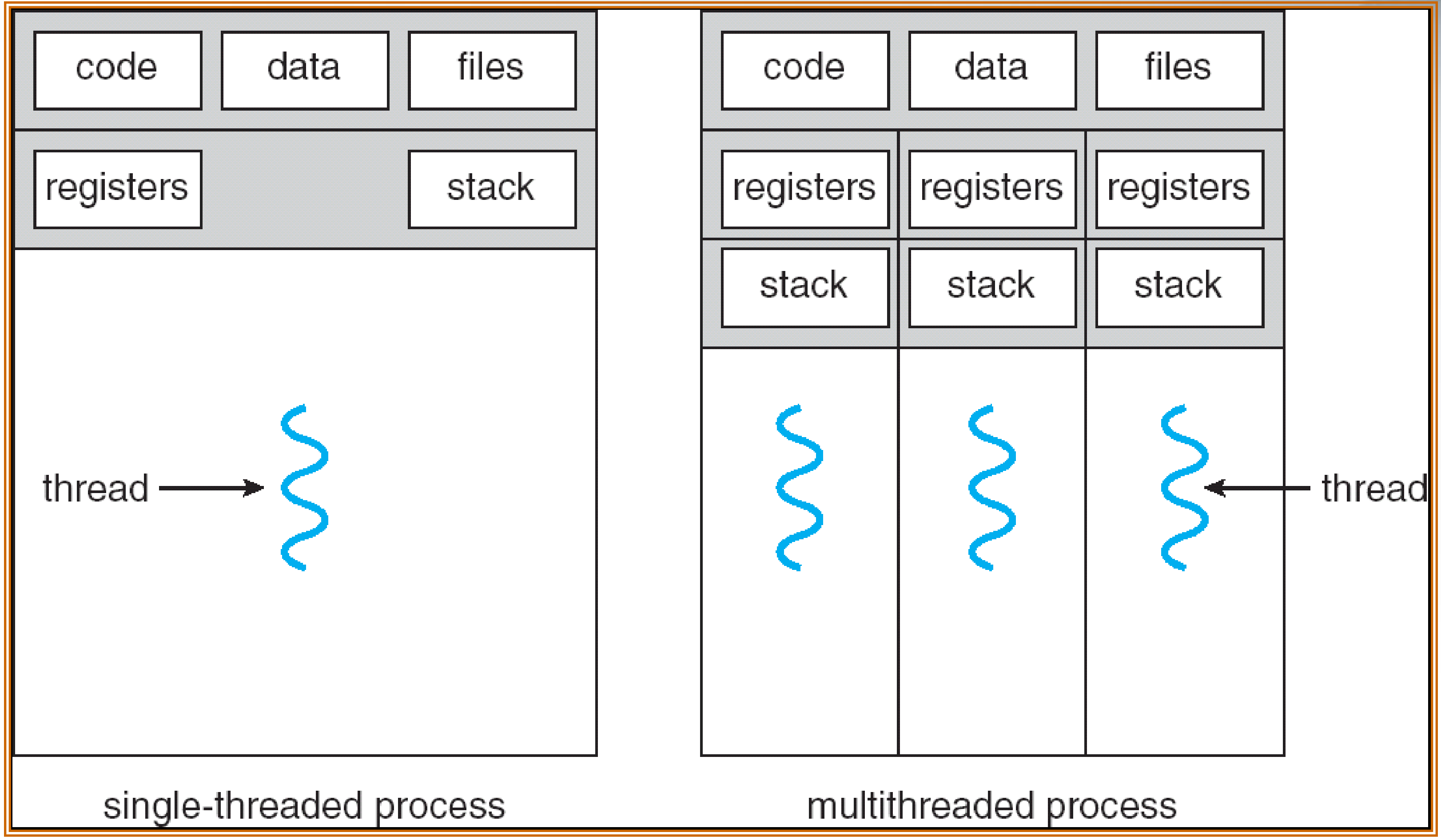
Thread

- An operating system that has thread facility, the basic unit of CPU utilization is a thread.
- A thread has or consists of a
  - program counter (PC),
  - a register set, and
  - a stack space.

Thread

- Threads are not independent.
- Threads shares with other threads their
  - code section,
  - data section,
  - OS resources also known as task, such as open files and signals.

Single and Multithreaded Processes



Benefits of Threads

- Responsiveness
- Resource Sharing
- Economy
- Utilization of Massive Parallel Architectures

Processes Vs Threads

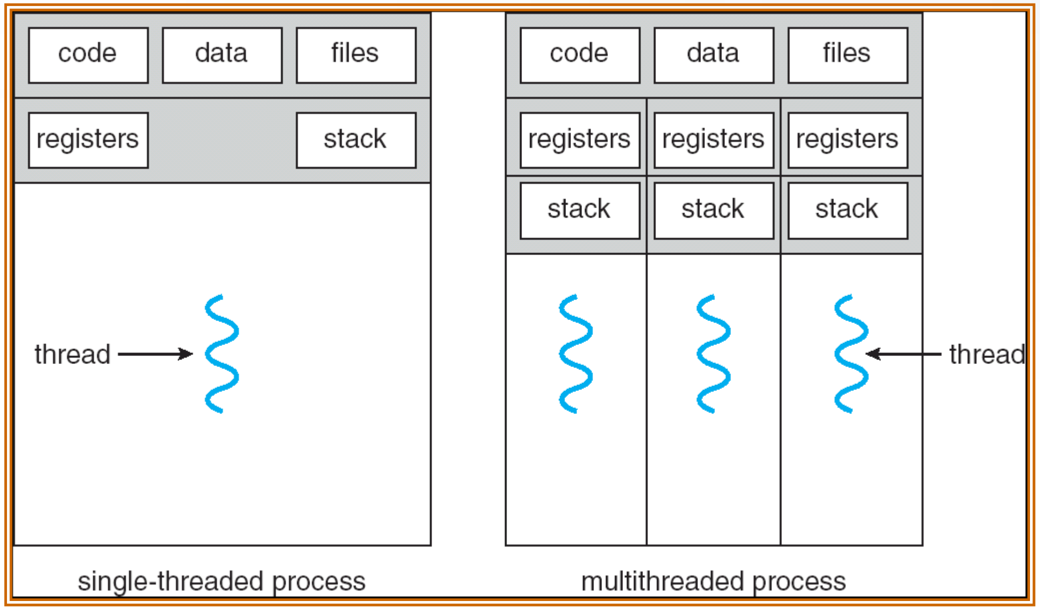
Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes, execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Processes Vs Threads

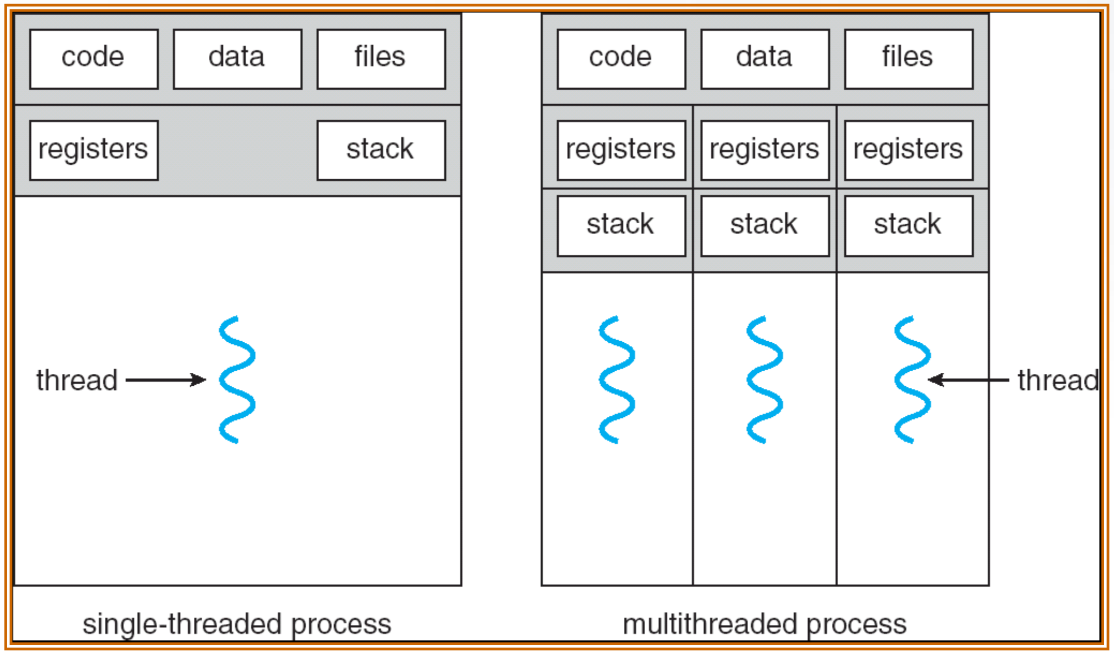
Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .



Processes Vs Threads

- Unlike processes, thread are design to assist one other.
- Note that processes might or might not assist one another because processes may originate from different users.



Why Threads?

- A process with multiple threads make a great server for example printer server.
- Because threads can share common data, they do not need to use inter-process communication.

Why Threads?

- Threads are cheap in the sense that they only need a stack and storage for registers therefore, threads are cheap to create.
- Threads use very little resources of an operating system in which they are working.
- That is, threads do not need new address space, global data, program code or operating system resources.

Why Threads?

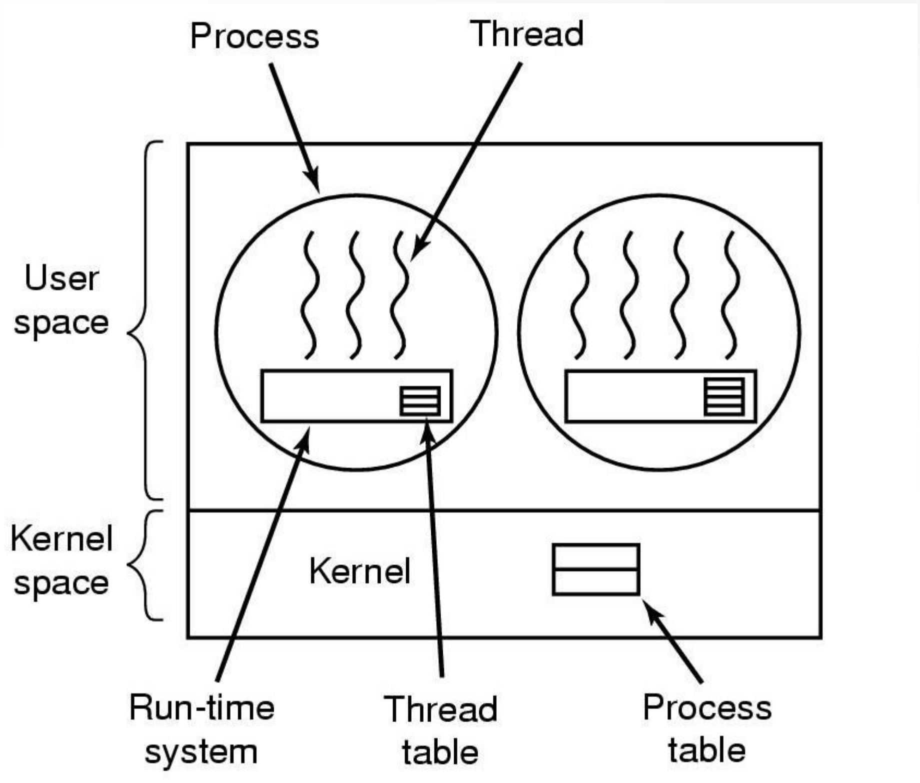
- Context switching are fast when working with threads.
- The reason is that we only have to save and/or restore
  - PC,
  - SP &
  - registers.

Types of Thread

- User Thread
- Kernel Thread

Implementing Threads in User Space

The kernel is not aware of the existence of threads;



- Run-time system (thread-library in execution) is responsible for bookkeeping, scheduling of threads
- allows for customised scheduling
- can run on any OS
- But: problem with blocking system calls (when a thread blocks, the whole process blocks; i.e other threads of the same process cannot run)

User Threads

- Thread management done by user-level threads library.
- So thread switching does not need to call operating system and to cause interrupt to the kernel.
- In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages: User Threads

- A user-level threads package can be implemented on an Operating System that does not support threads.
- User-level threads does not require modification to operating systems.
- Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.

Advantages: User Threads

- Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient: Thread switching is not much more expensive.

User Threads: Disadvantages

- There is a lack of coordination between threads and operating system kernel.
- Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within.
- It is up to each thread to give up control to other threads.

User Threads: Disadvantages

- For example, if one thread causes a page fault, the process entire process will blocked in the kernel, even if there are runable threads left in the processes.

User Threads

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

What is POSIX

- POSIX or "Portable Operating System Interface [for Unix]" is the name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system .

What is Win32

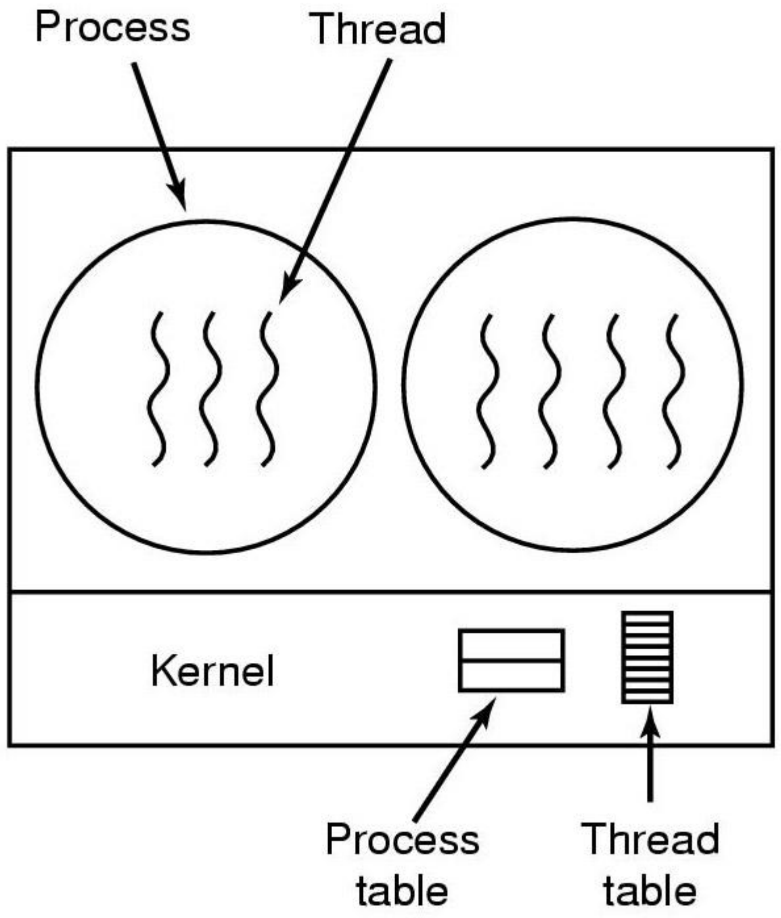
- The Windows API, informally WinAPI, is Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems.

Kernel Threads

- In this method, the kernel knows about and manages the threads.
- No runtime system is needed in this case.
- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system.
- In addition, the kernel also maintains the traditional process table to keep track of processes.

Implementing Threads in the Kernel

Kernel maintains context information for the process and the threads



- Scheduling is done on a thread basis
- Does not suffer from "blocking problem"
- Less efficient than user-level threads (kernel is invoked for thread creation, termination, switching)

Kernel Thread Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Kernel Thread Advantages:

- **Sharing:** Treads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

Kernel Thread Disadvantages:

- Since kernel must manage and schedule threads as well as processes.
- It require a full thread control block (TCB) for each thread to maintain information about threads.
- As a result there is significant overhead and increased in kernel complexity.
- The kernel-level threads are slow and inefficient.

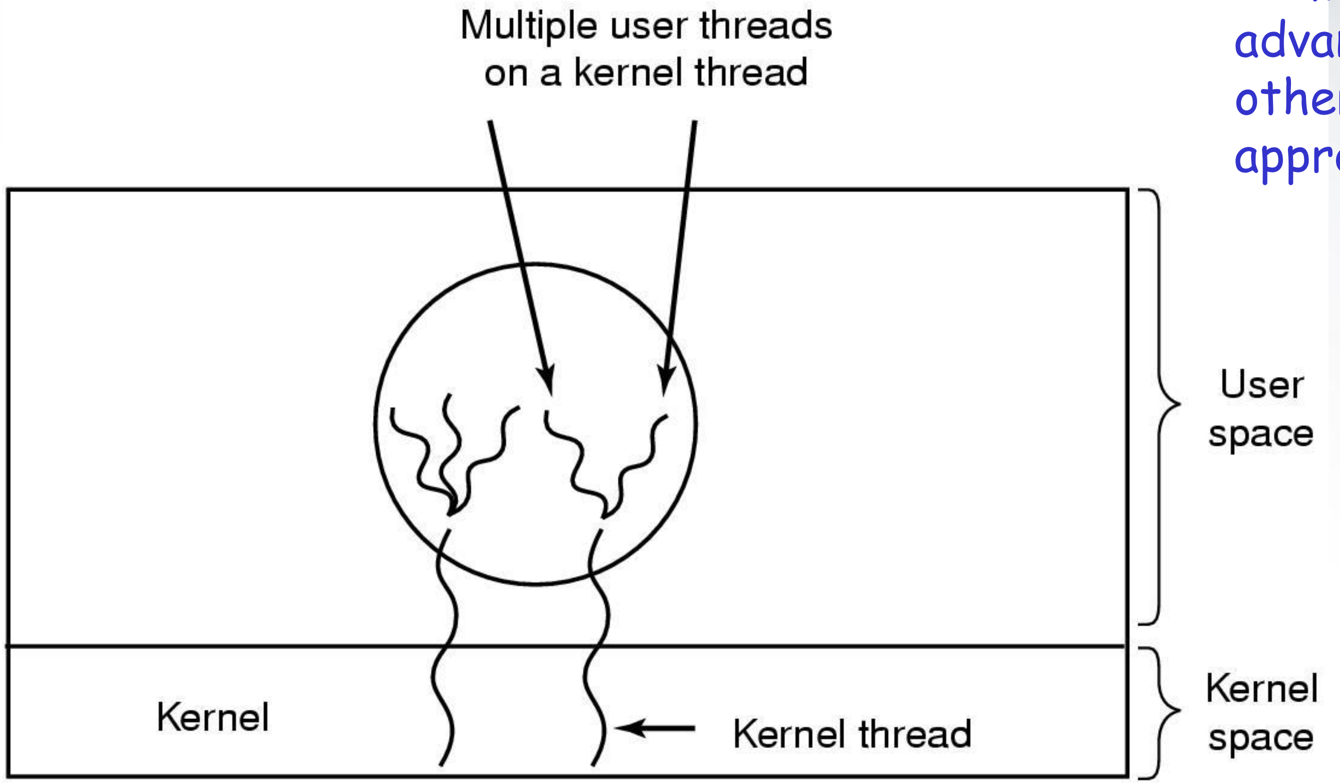
Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Hybrid Implementations

Multiplexing user-level threads onto kernel- level threads

To combine the advantages of the other two approaches



Advantages of Threads over Multiple Processes

- **Context Switching**   Threads are very inexpensive to create and destroy, and they are inexpensive to represent.
- For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc.

Advantages of Threads over Multiple Processes

Context Switching

- With so little context, it is much faster to switch between threads.
- In other words, it is relatively easier for a context switch using threads.

Disadvantages of Threads over Multiprocesses

- **Blocking** The major disadvantage if that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

Disadvantages of Threads over Multiprocesses

- **Security**   Since there is, an extensive sharing among threads there is a potential problem of security.
- It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

Application that Benefits from Threads

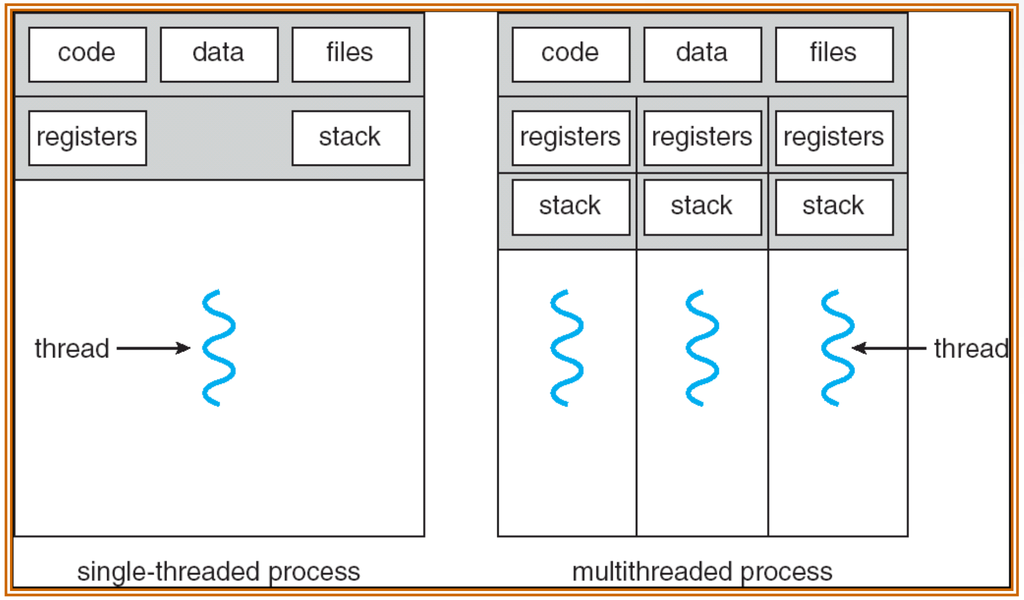
- A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi-threaded process.
- In general, any program that has to do more than one task at a time could benefit from multitasking.
- For example, a program that reads input, process it, and outputs could have three threads, one for each task.

Application that cannot Benefit from Threads

- Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes.
- For example, a program that displays the time of the day would not benefit from multiple threads.

Resources used in Thread Creation and Process Creation

- When a new thread is created it shares its code section, data section and operating system resources like open files with other threads.
- But it is allocated its own stack, register set and a program counter.



Resources used in Thread Creation and Process Creation

- The creation of a new process differs from that of a thread mainly in the fact that all the shared resources of a thread are needed explicitly for each process.
- So though two processes may be running the same piece of code they need to have their own copy of the code in the main memory to be able to run.

Resources used in Thread Creation and Process Creation

- Two processes also do not share other resources with each other.
- This makes the creation of a new process very costly compared to that of a new thread.

Context Switch

- Each switch of the CPU from one process to another is called a context switch.

Major Steps of Context Switching

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

Major Steps of Context Switching

- In a multiprogrammed uniprocessor computing system, context switches occur frequently enough that all processes appear to be running concurrently.
- If a process has more than one thread, the Operating System can use the context switching technique to schedule the threads so they appear to execute in parallel.

Major Steps of Context Switching

- Threads can also be implemented entirely at the user level in run-time libraries. Since in this case no thread scheduling is provided by the Operating System, it is the responsibility of the programmer to yield the CPU frequently enough in each thread so all threads in the process can make progress

Action of Kernel to Context Switch Among Threads

- The threads share a lot of resources with other peer threads belonging to the same process.
- So a context switch among threads for the same process is easy.
- It involves switch of register set, the program counter and the stack.
- It is relatively easy for the kernel to accomplished this task.

Action of kernel to Context Switch Among Processes

- Context switches among processes are expensive.
- Before a process can be switched its process control block (PCB) must be saved by the operating system.
- The PCB consists of the following information:
- The process state.
- The program counter, PC.
- The values of the different registers.
- The CPU scheduling information for the process.

Action of kernel to Context Switch Among Processes

- Memory management information regarding the process.
- Possible accounting information for this process.
- I/O status information of the process.
- When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.

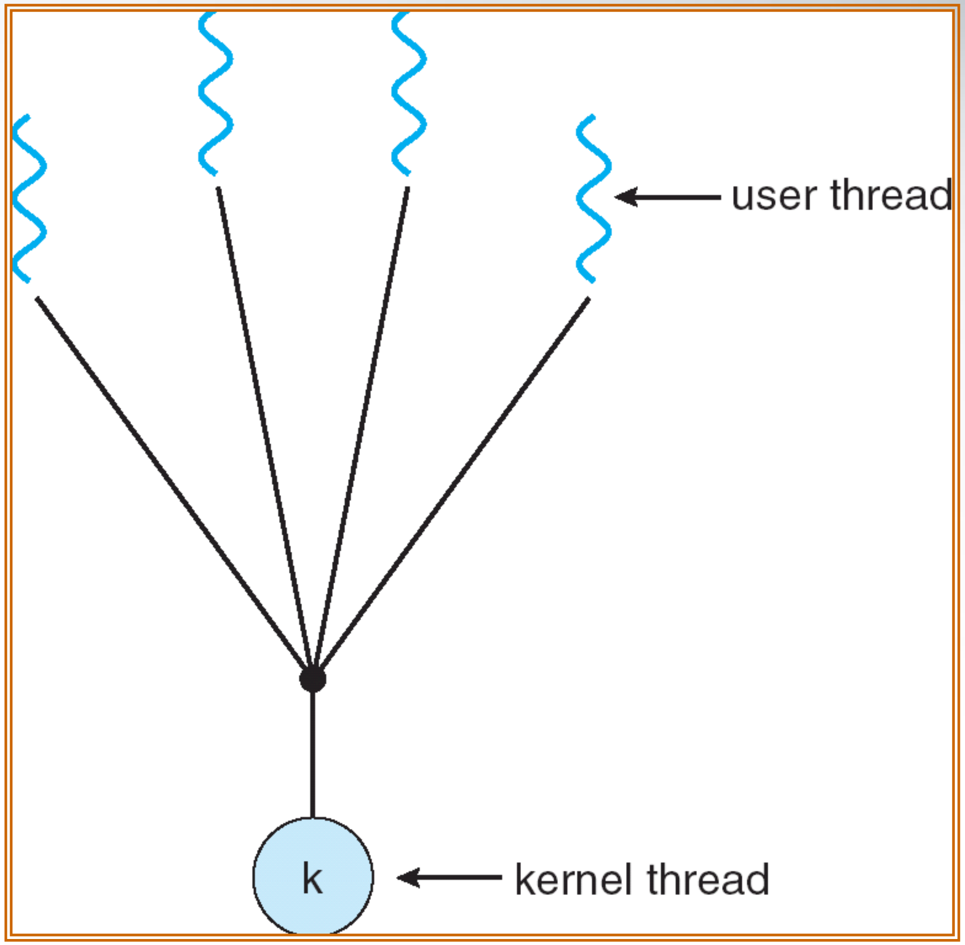
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

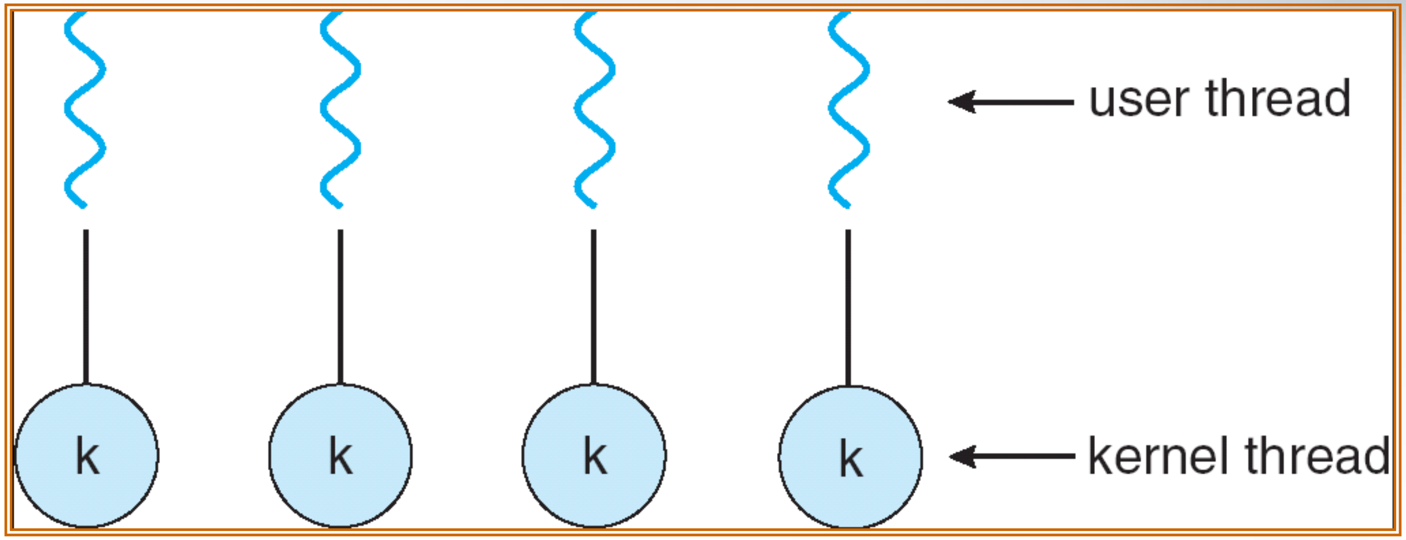
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

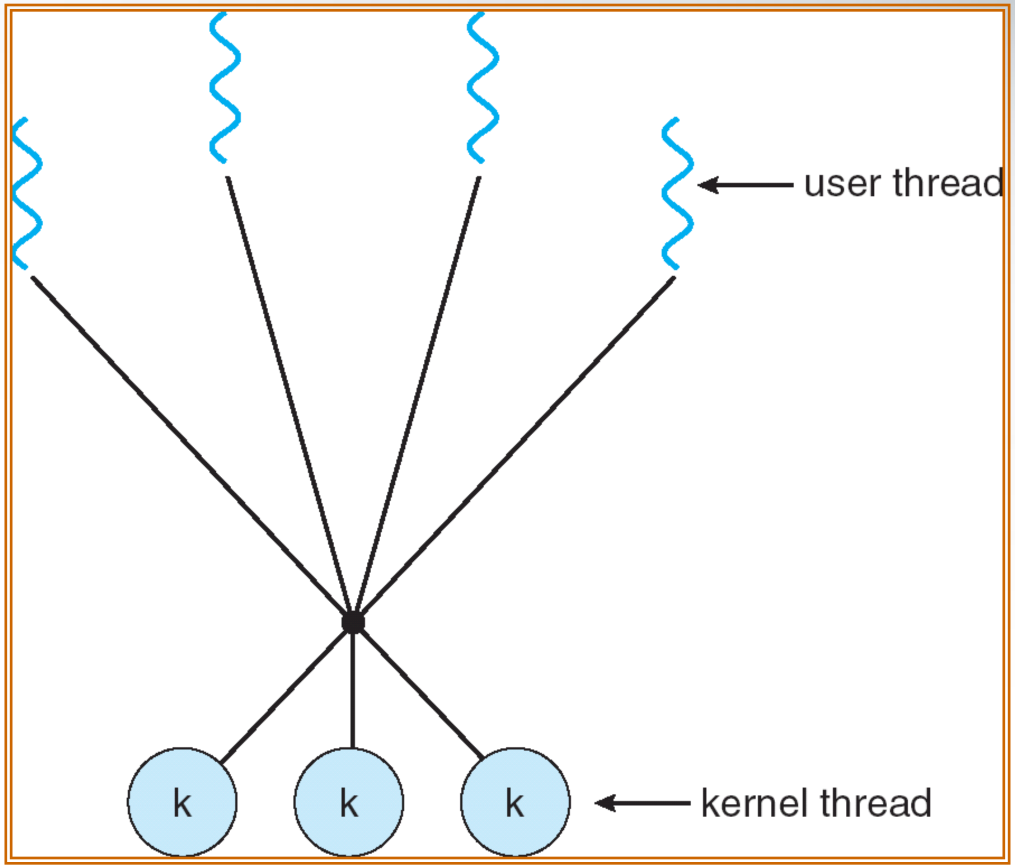
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

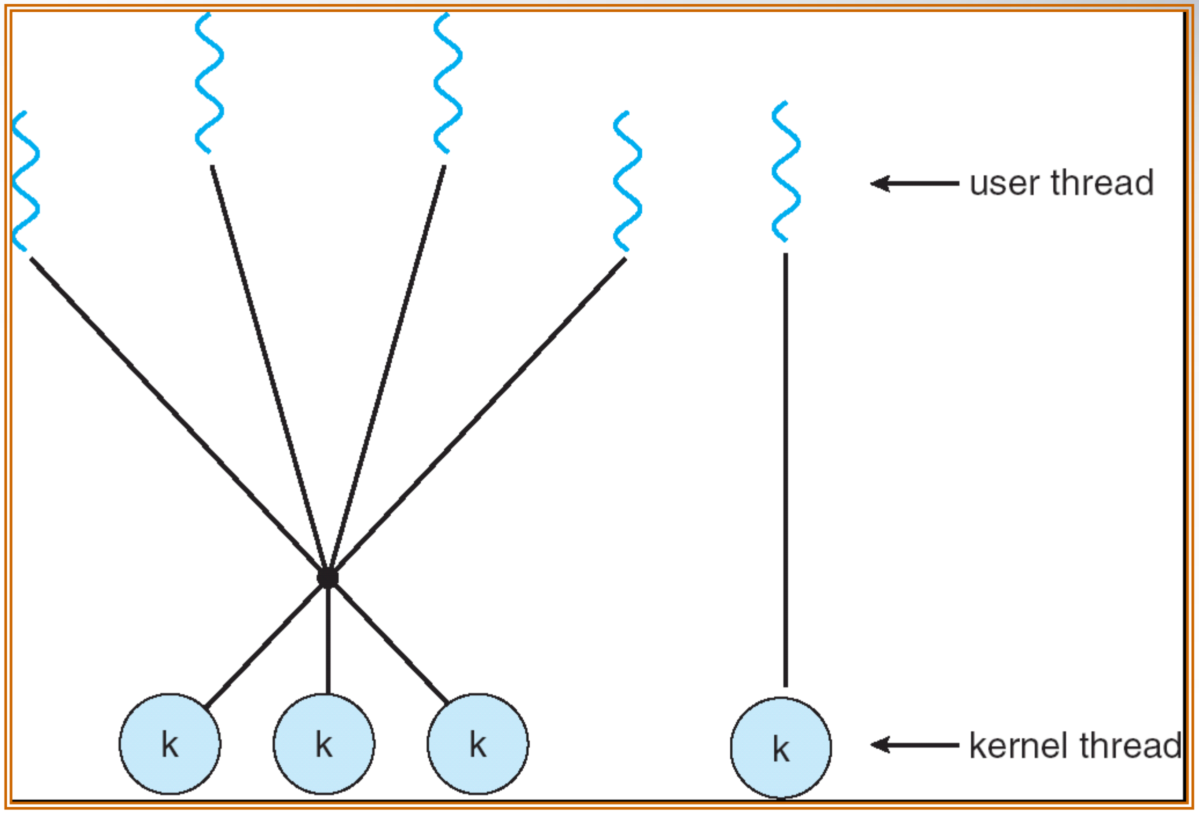
Many-to-Many Model



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Thread related operation

- Thread creation
- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Benefits of Threads

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel
- May allow parallelization within a process:
  - I/O and computation to overlap (remember the historical step from uniprogramming to multiprogramming?)
  - concurrent execution in multiprocessors
- Takes less time to
  - create/terminate a thread than a process
  - switch between two threads within the same process

Uses of Threads

- Overlap foreground (interactive) with background (processing) work
- Asynchronous processing (e.g. backup while editing)
- Speed execution (parallelize independent actions)
- Modular program structure (must be careful here, not to introduce too much extra overhead)

Q: can one achieve parallelism within a process but without using threads?

Examples:

- Posix Pthreads: (IEEE) standard:
  - Specifies interface
  - Implementation (using user/kernel level threads) is up to the developer(s)
  - More common in UNIX systems
- Win32 thread library:
  - Kernel-level library, windows systems
- Java threads:
  - Supported by the JVM (*VM: a run-time system, a general concept, with deeper roots and potential future in the OS world*)
  - Implementation is up to the developers –e.g. can use Pthreads API or Win32 API, etc

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:**

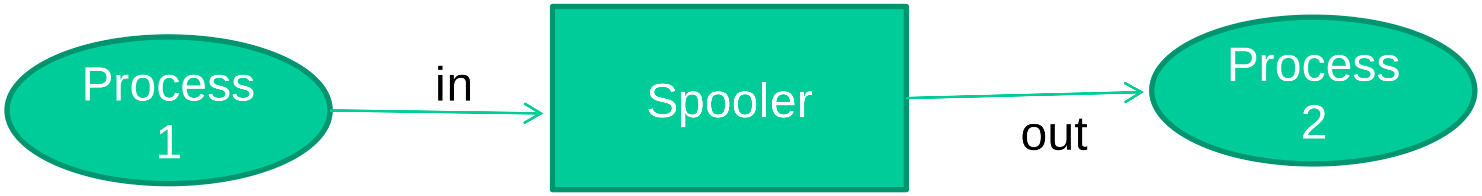
Interprocess communication

- 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.

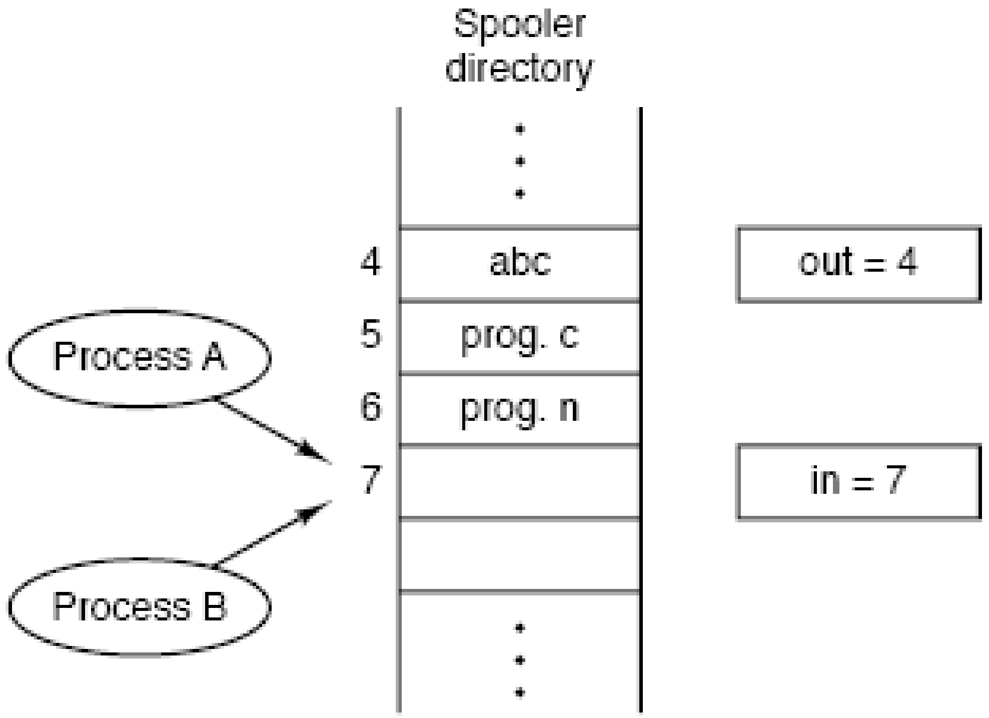


Example of IPC

- 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.

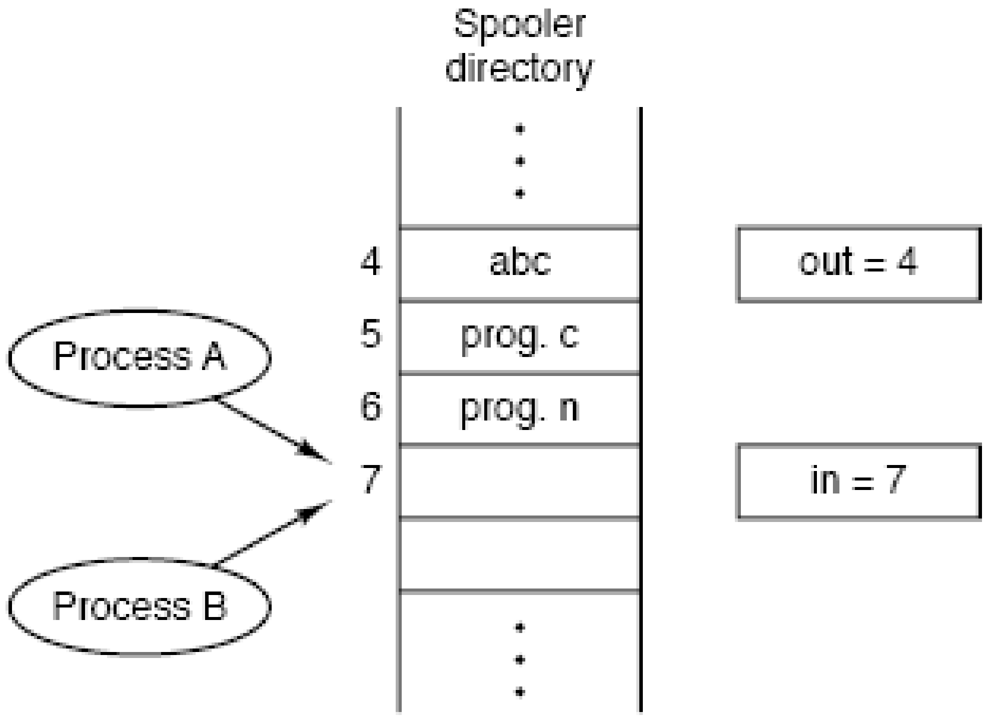
Example of IPC

- 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

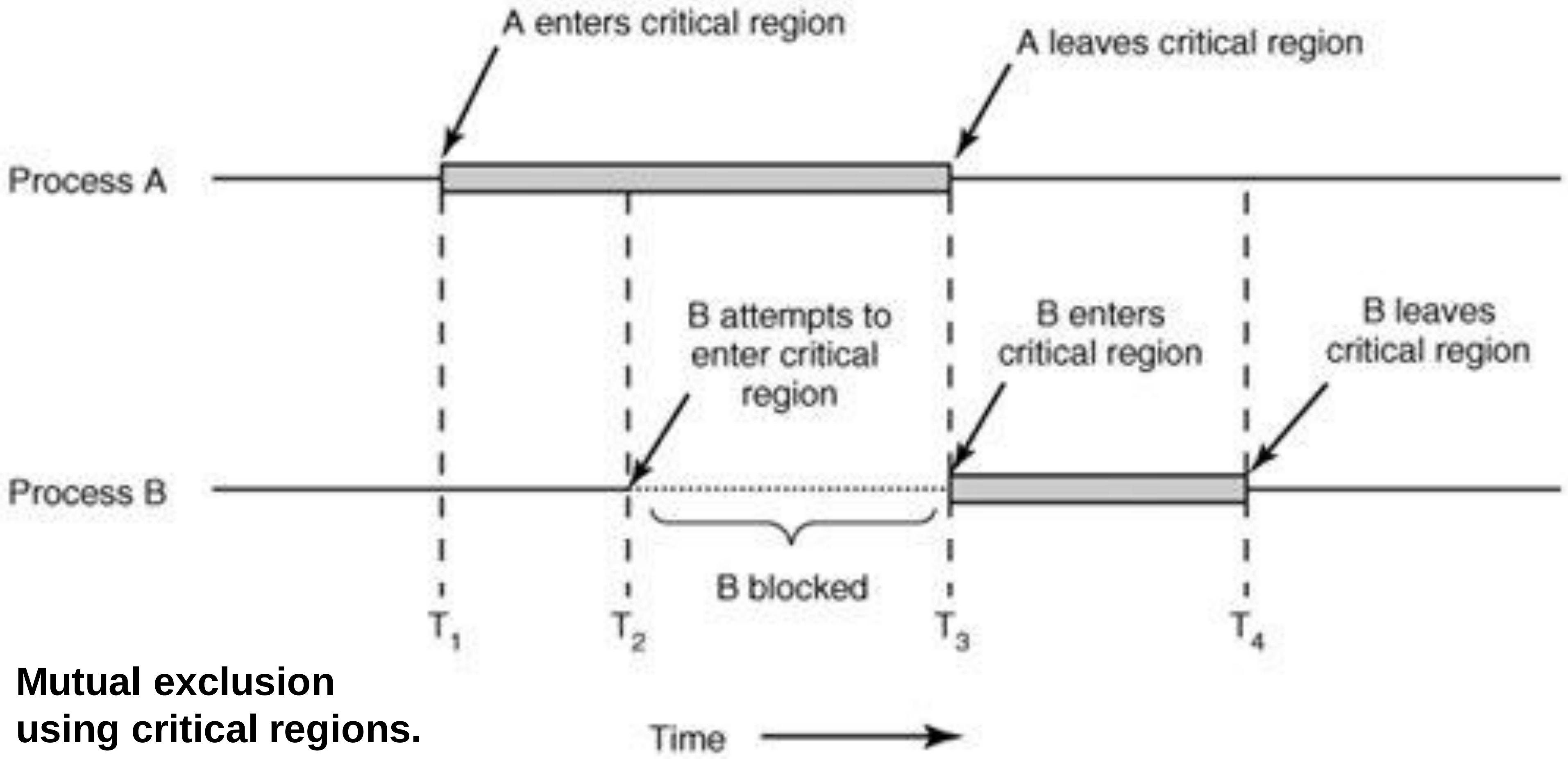
- How do we avoid race conditions?
- It is important to find some way to prohibit more than one process from reading and writing the shared data at the same time.
- **what we need is mutual exclusion**
- *The difficulty above occurred because process B started using one of the shared variables before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

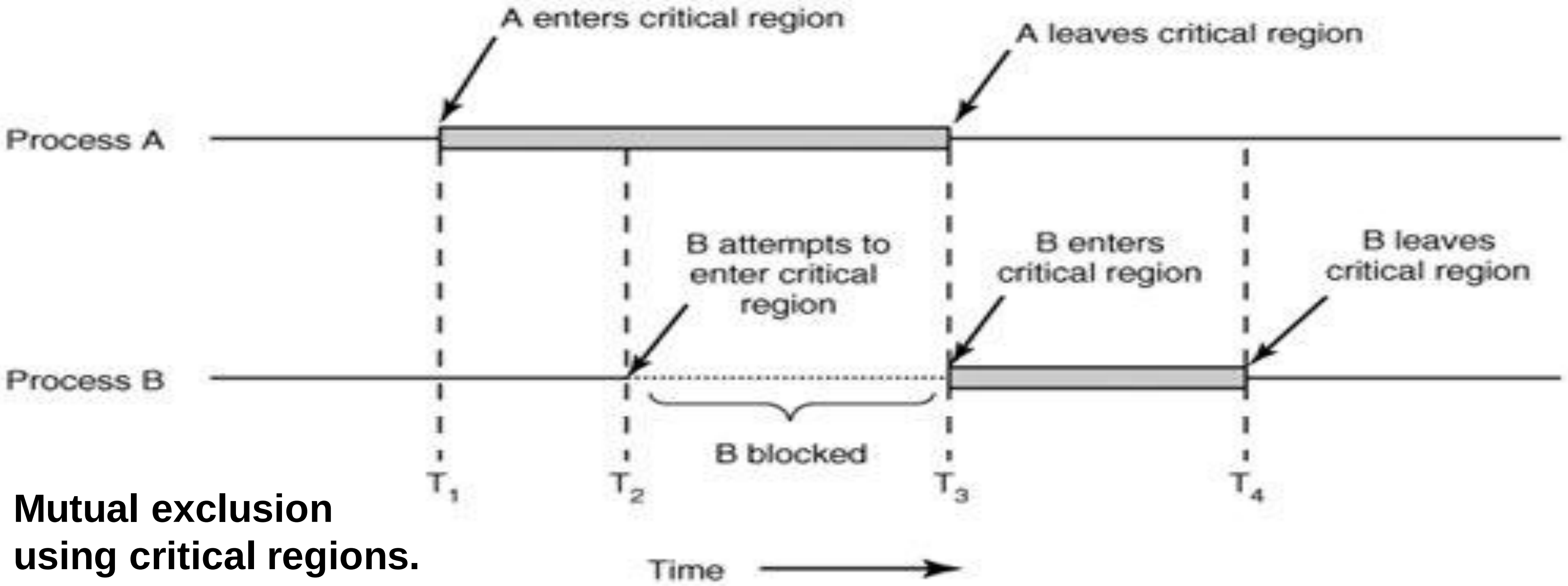
- Here process *A enters its critical region at time T_1*



Mutual exclusion using critical regions.

Critical Sections

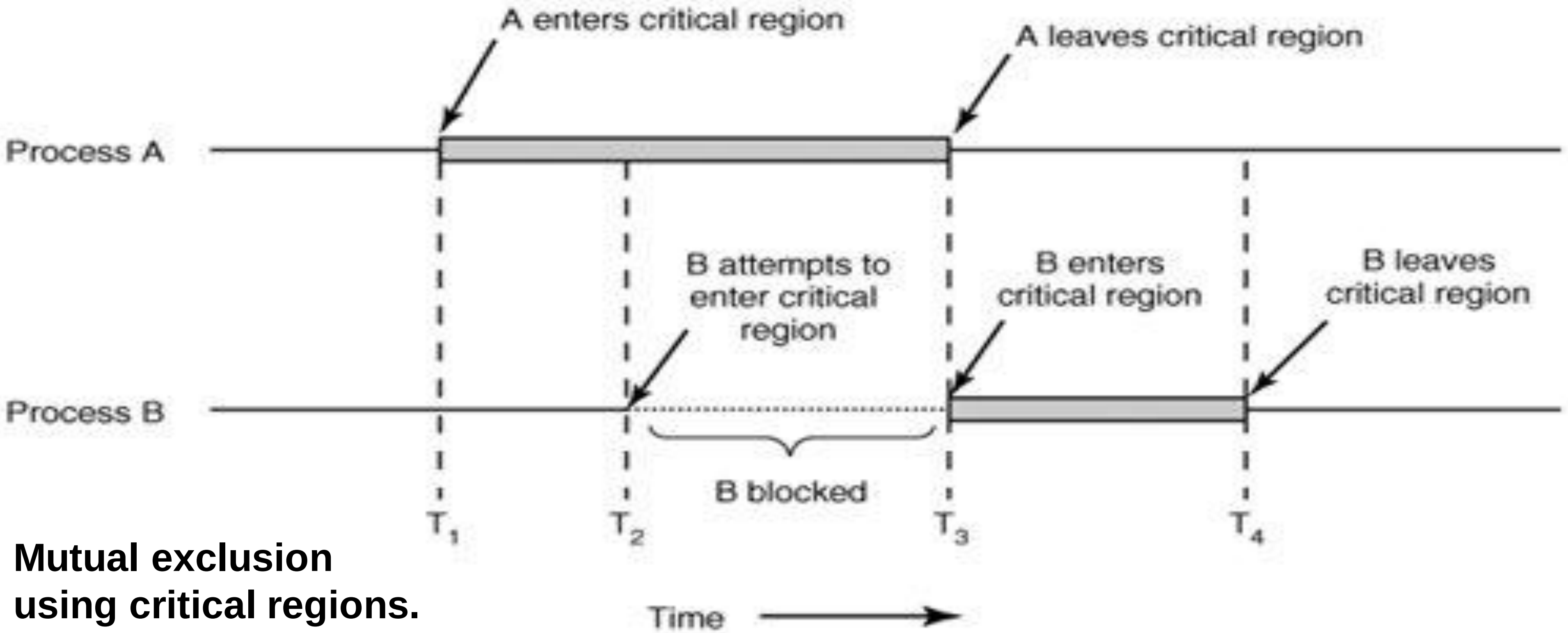
- At time *T_2 process B attempts to enter its critical region but fails because another* process is already in its critical region and we allow only one at a time.



Mutual exclusion using critical regions.

Critical Sections

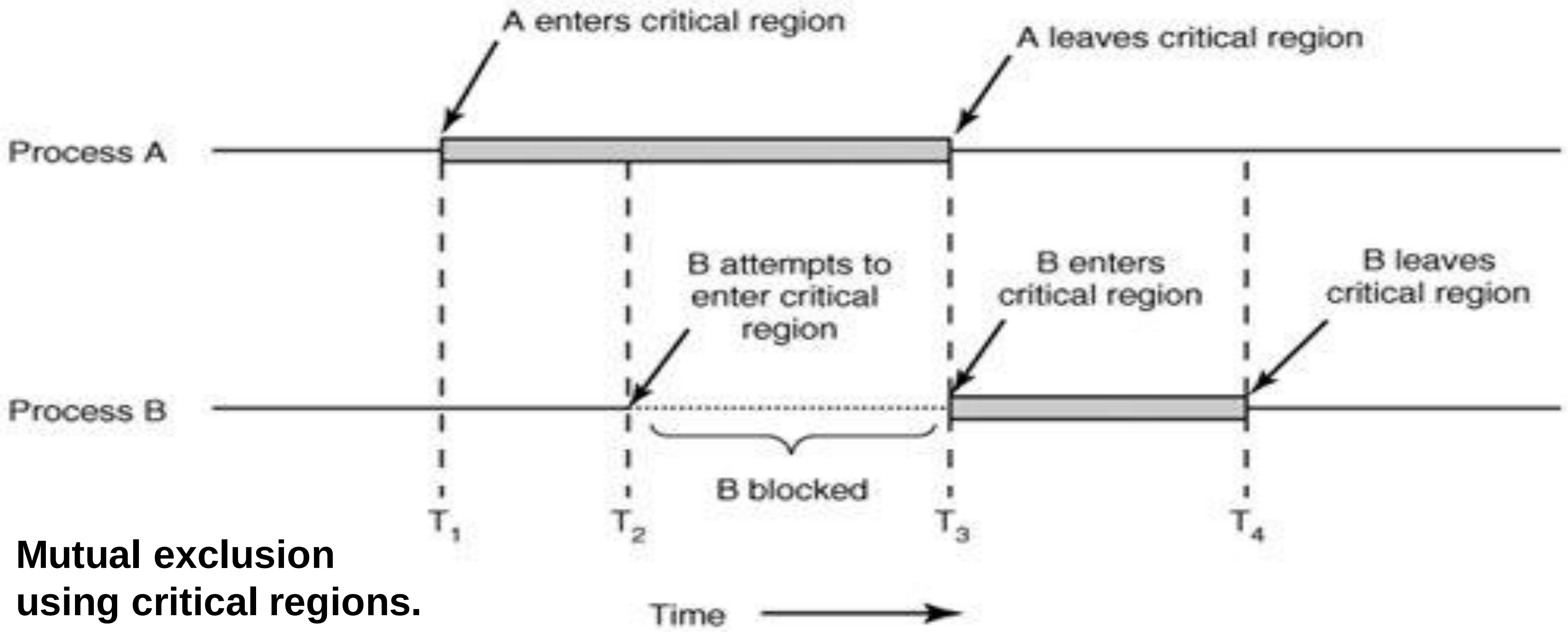
- Consequently, *B is* temporarily suspended until time *T 3 when A leaves its critical region, allowing B to enter*



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

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

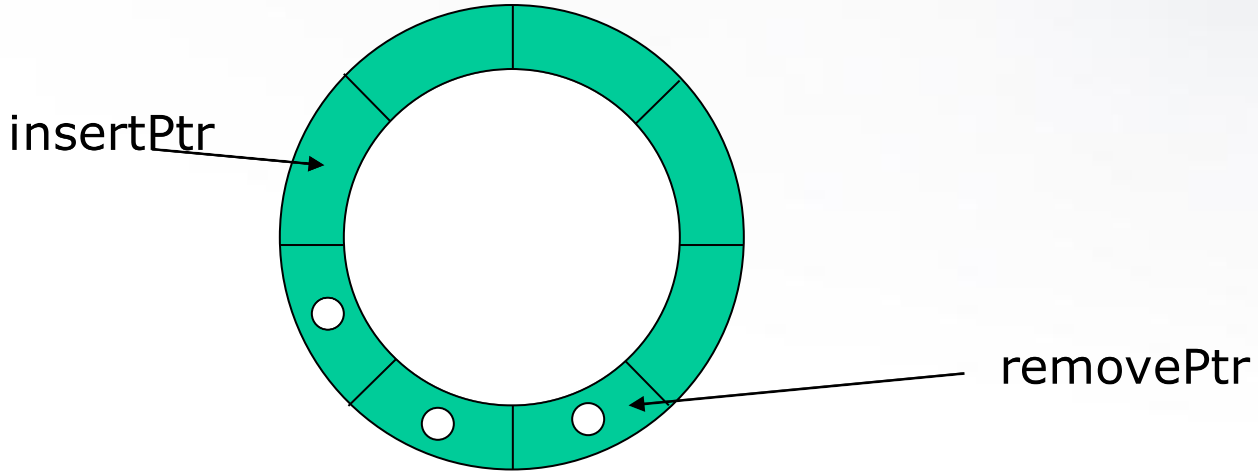
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- If the producer encounters a full buffer, or if the consumer encounters an empty buffer, the process blocks.

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.

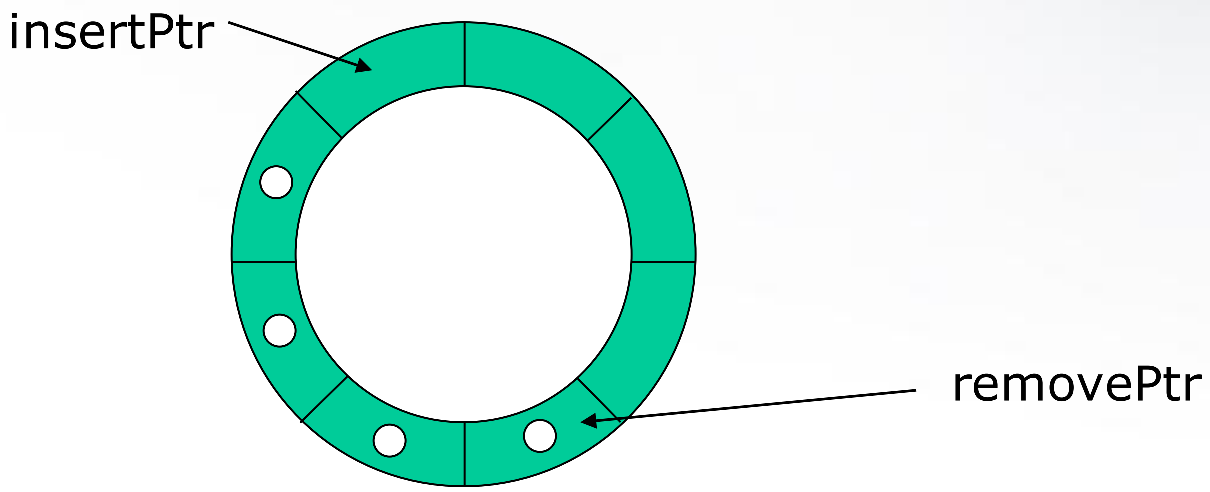
Producer-Consumer

Chef	= Producer
Customer	= Consumer



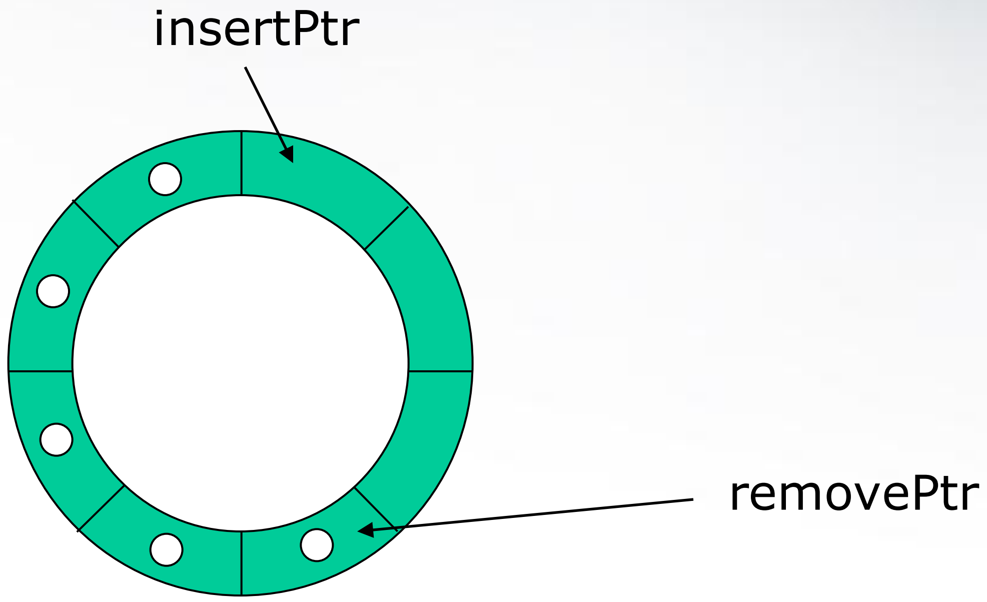
Producer-Consumer

Chef	= Producer
Customer	= Consumer



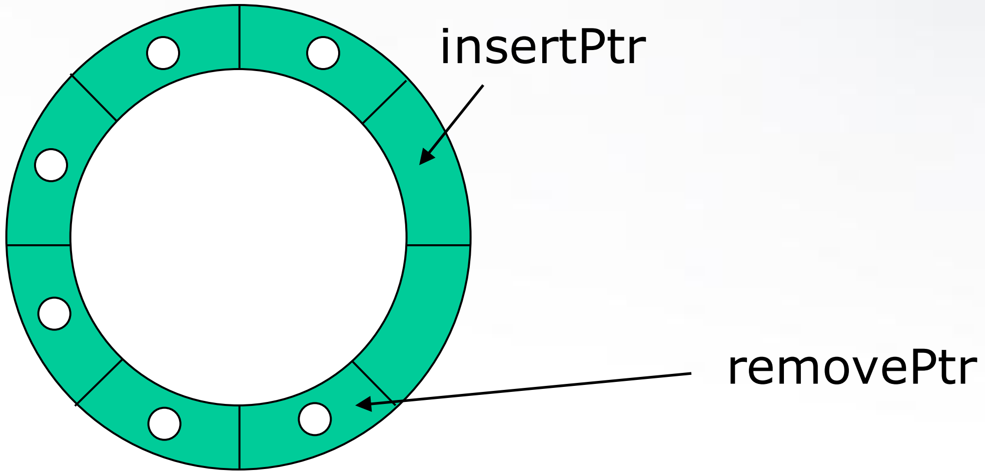
Producer-Consumer

Chef	= Producer
Customer	= Consumer



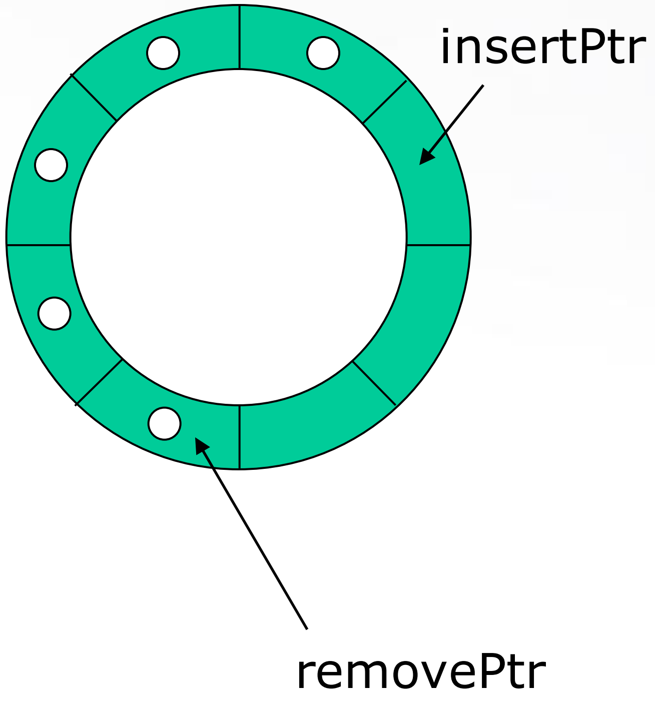
Producer-Consumer

Chef	= Producer
Customer	= Consumer



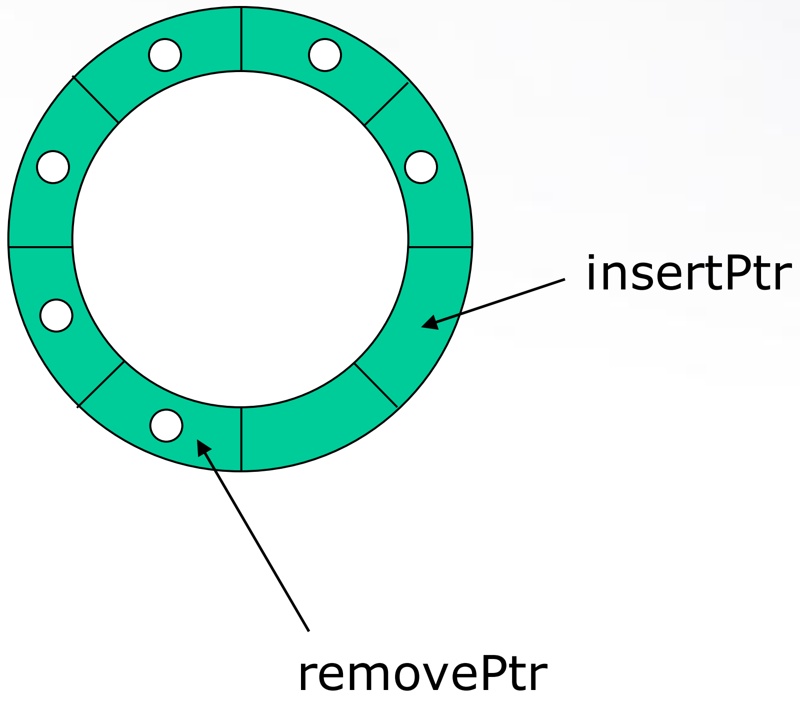
Producer-Consumer

Chef	= Producer
Customer	= Consumer



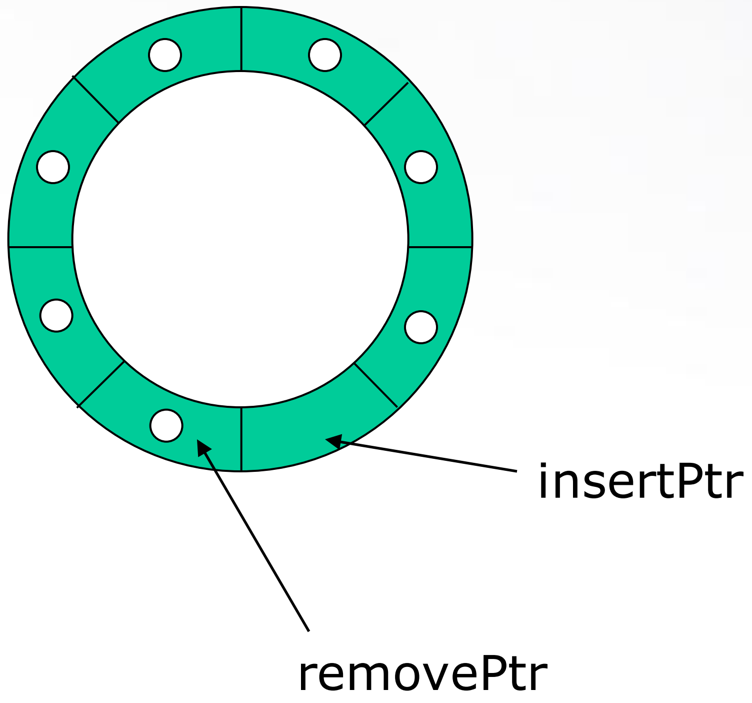
Producer-Consumer

Chef	= Producer
Customer	= Consumer



Producer-Consumer

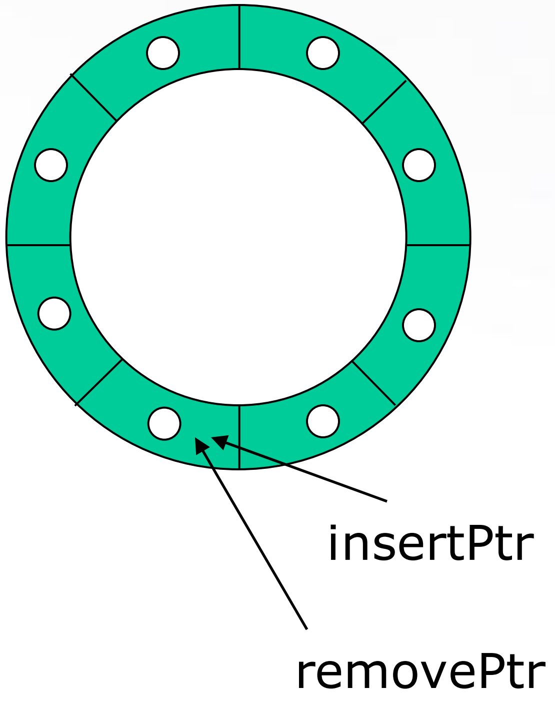
Chef	= Producer
Customer	= Consumer



Producer-Consumer

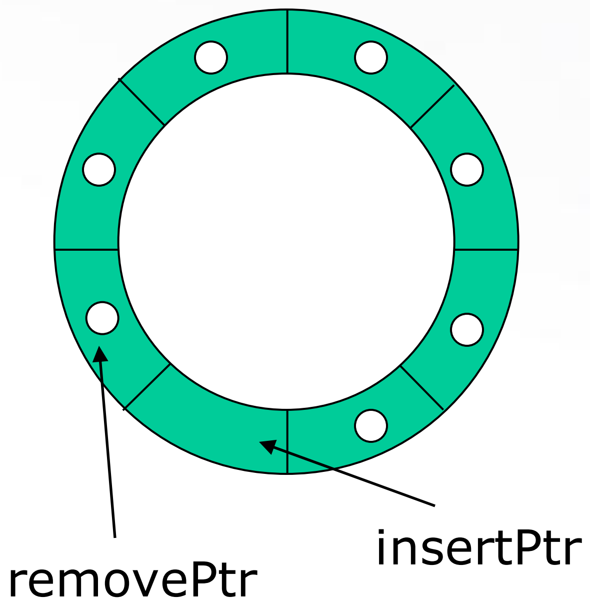
Chef	= Producer
Customer	= Consumer

BUFFER FULL: Producer must be blocked!



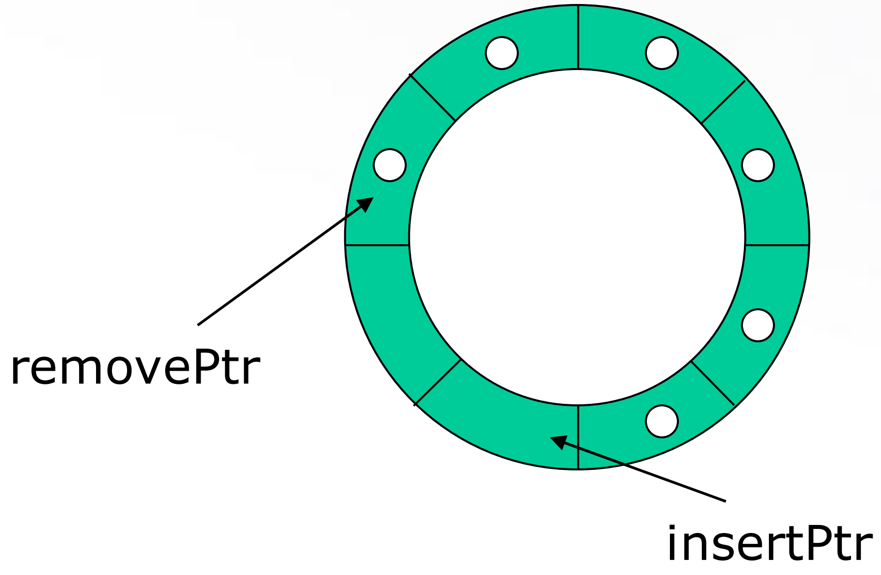
Producer-Consumer

Chef	= Producer
Customer	= Consumer



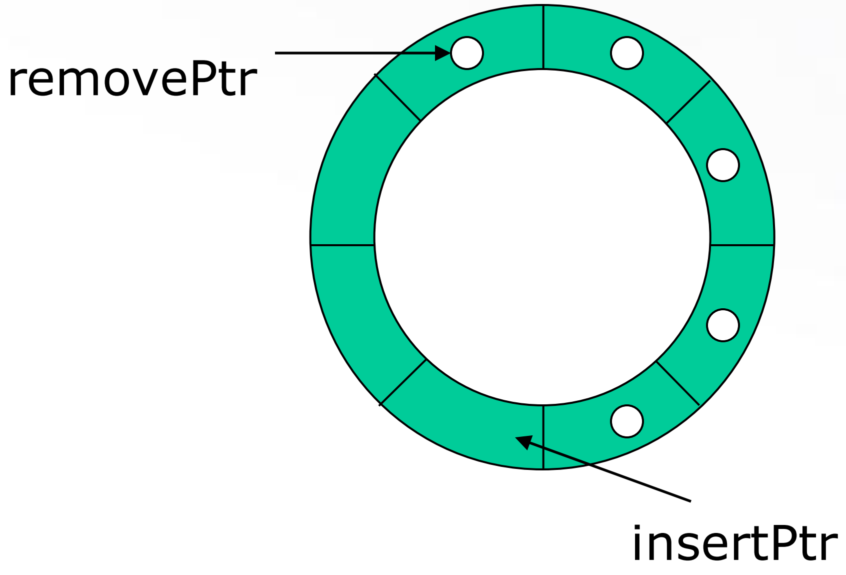
Producer-Consumer

Chef	= Producer
Customer	= Consumer



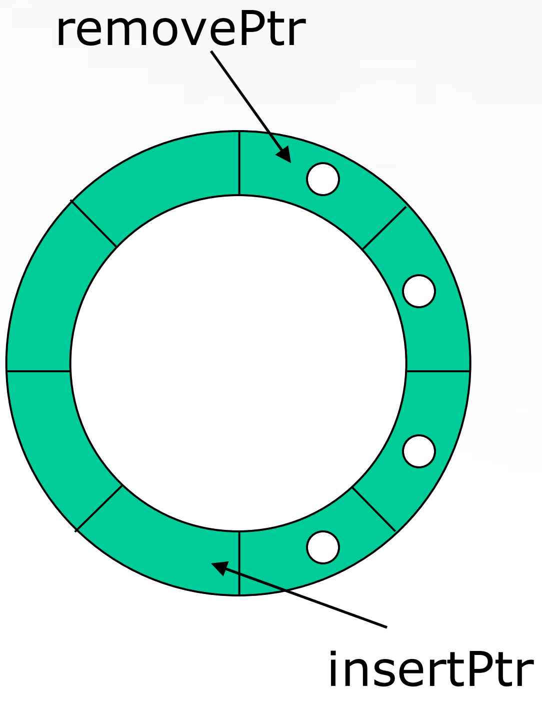
Producer-Consumer

Chef	= Producer
Customer	= Consumer



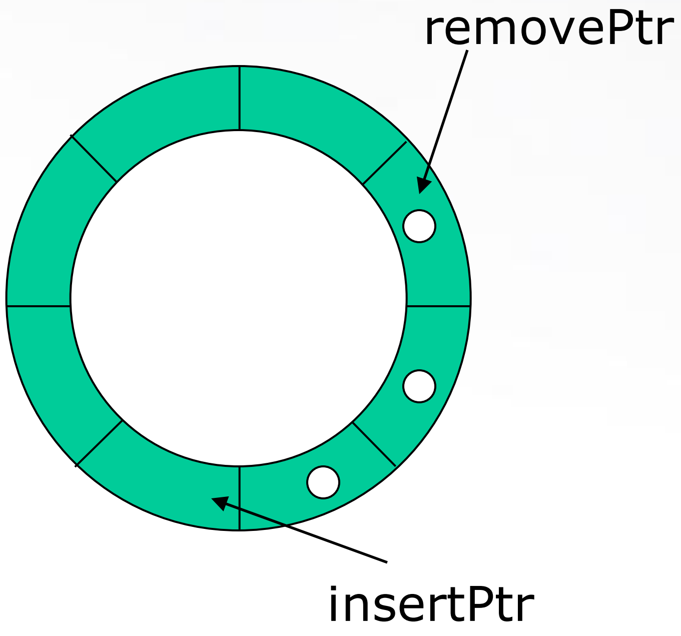
Producer-Consumer

Chef	= Producer
Customer	= Consumer



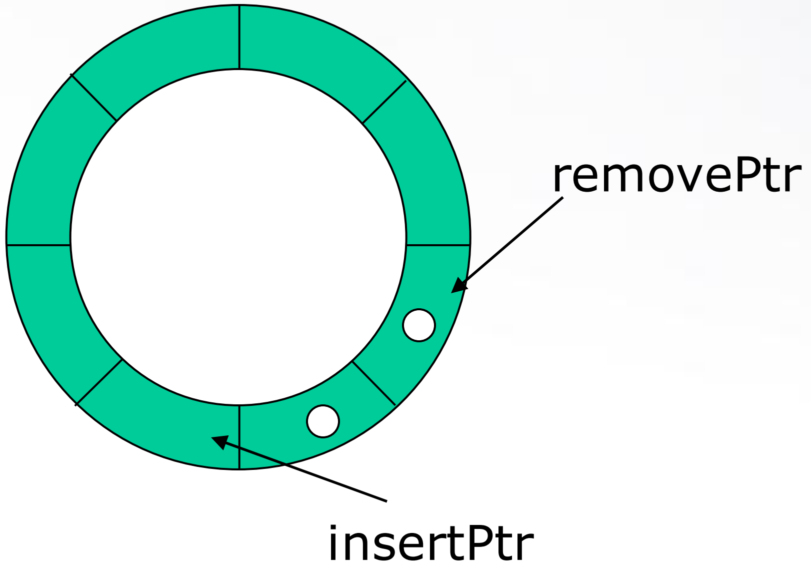
Producer-Consumer

Chef	= Producer
Customer	= Consumer



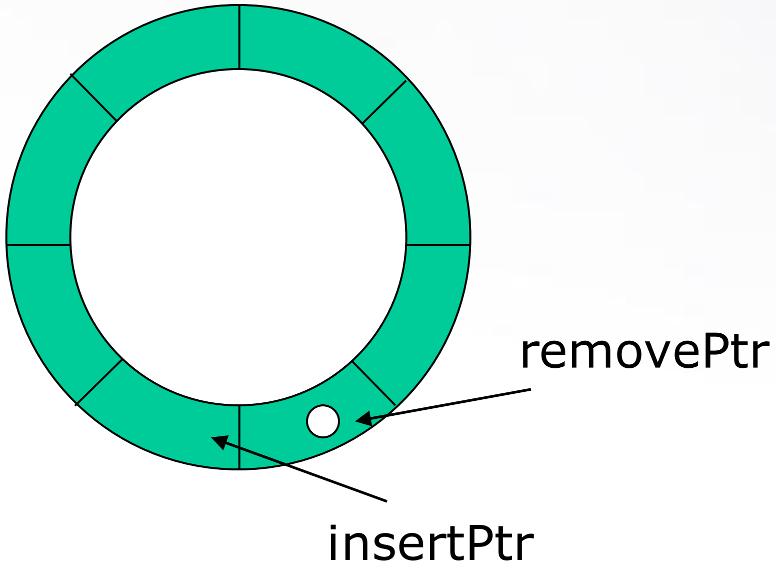
Producer-Consumer

Chef	= Producer
Customer	= Consumer



Producer-Consumer

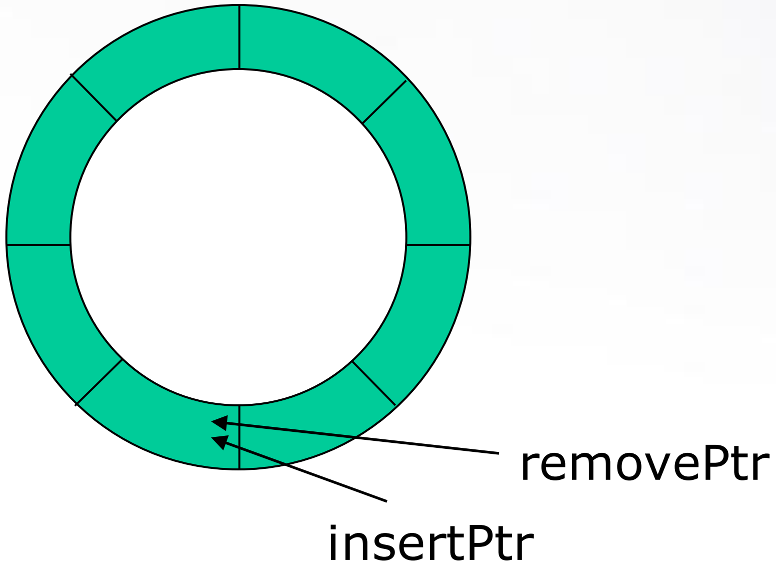
Chef	= Producer
Customer	= Consumer



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

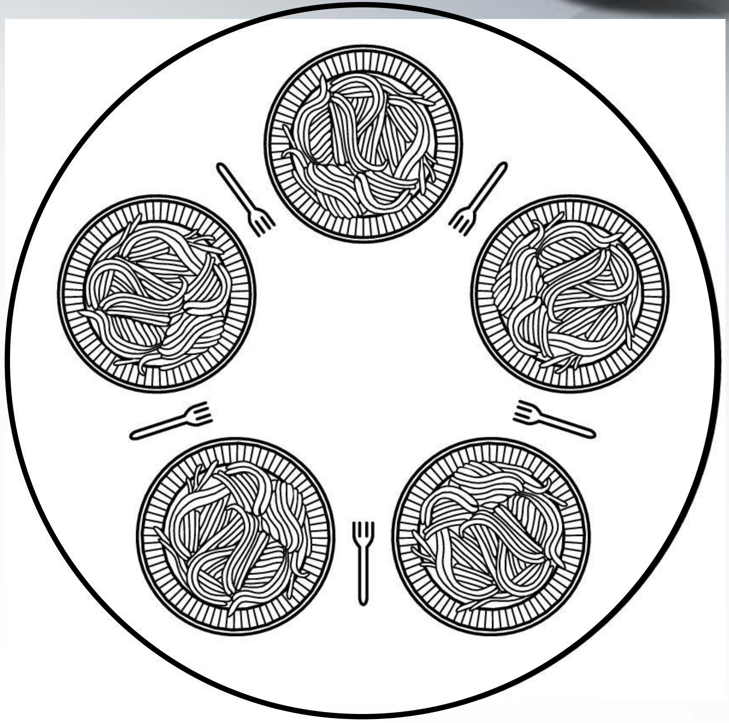
- Producer inserts items. Updates insertion pointer.
- Consumer executes destructive reads on the buffer. Updates removal pointer.
- Both update information about how full/empty the buffer is.
- Solution should allow multiple producers and consumers

Challenges

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

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."

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.

Drawback with Lock Variables

- Error as same as Spooler Directory.
  - Process 1 reads the lock = **0**.
  - Process 1 is going to set lock =1
  - On the Same time Process 2 read lock =0 .
  - Process 2 set the lock= **1**.
  - Process runs again, o **1**, and two processes will be in their critical regions at the same time.

Idea No. 3 : Turn Variable.

- Turn is a Shared Variable.
- Initially turn=0
- P0 Wants to enter in the CR found turn =0.
- Enter into CR.
- P1 Also waiting for its turn.
- Checking again n again
- It is Busy Waiting.

Drawbacks of 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**

Idea no. 4 :Peterson's solution for achieving mutual exclusion.

- To calls are introduced.
  - – Enter Region (For showing interest of CR)
  - – Leave Region (For Leaving the CR.)

Idea no. 4 :Peterson's solution for achieving mutual exclusion.

- **STEP 1:** Enter_region is call for all interested process for Critical Region.
- **STEP 2:** Before using the shared variables each process calls enter_region with its own process number, 0 or 1, as the parameter.
- **STEP 3:** This call will cause it to wait, if need be, until it is safe to enter.
- **STEP 4:** 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.

Idea no. 4 :Peterson's solution for achieving mutual exclusion..

- Let us see how this solution works.
- Step 1: Initially, neither process is in its critical region.
- Step 2: Now process 0 calls enter_region .It indicates its interest by setting its array element and sets turn to 0.
- Step 3: Since process 1 is not interested, enter_region returns immediately.

Idea no. 4 :Peterson's solution for achieving mutual exclusion.

- Step 4: If process 1 now calls enter_region, it will hang there until interested [0] goes to FALSE ,
- Step 5: an event that only happens when process 0 calls leave_region to exit the critical region.

Idea no. 4 :Peterson's solution for achieving mutual exclusion.

1. Now consider the case that both processes call enter_region almost simultaneously.
2. Both will store their process number in turn .
3. Whichever store is done last is the one that counts; the first one is lost.
4. Suppose that process 1 stores last, so turn is 1.
5. So it creates conflicts.

Idea no. 4 :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 */
```

Idea no. 4 :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 */


}
```

Idea no 5: Test and Set Lock

- Hardware based approach….
- Use an Instruction
- **TSL RX, LOCK**
- TSL= Test and Set Lock
- RX= Register
- Set indicates read operation.
- Lock = memory word (a small memory portion)

The TSL Instruction

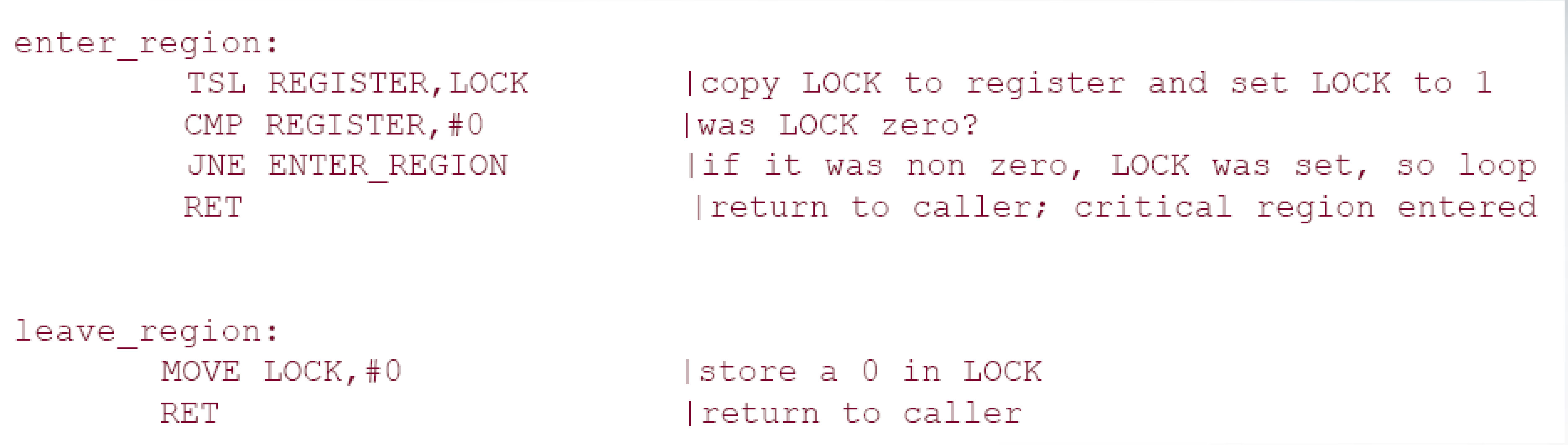
- Working of the insturuction:
- See the contents of the memory word LOCK
- Copy into register RX
- Stores a nonzero value at the memory address LOCK .

- 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.
- If LOCK = 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.

The TSL Instruction



```
enter_region:
       TSL REGISTER,LOCK           |copy LOCK to register and set LOCK to 1
       CMP REGISTER,#0             |was LOCK zero?
       JNE ENTER_REGION            |if it was non zero, LOCK was set, so loop
       RET                         |return to caller; critical region entered


leave_region:
      MOVE LOCK,#0                 |store a 0 in LOCK
      RET                          |return to caller
```

Summary

- Before entering its critical region, a process calls enter_region , which does busy waiting until the lock is free; then it acquires the lock and returns.
- After the critical region the process calls leave_region , which stores a 0 in LOCK .
- Correct timing is important.. If a process cheats, the mutual exclusion will fail.

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.

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.
- WEB SERVER = PRODUCER PROCESS
- WEB BROWSER=CLIENT PROCESS

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

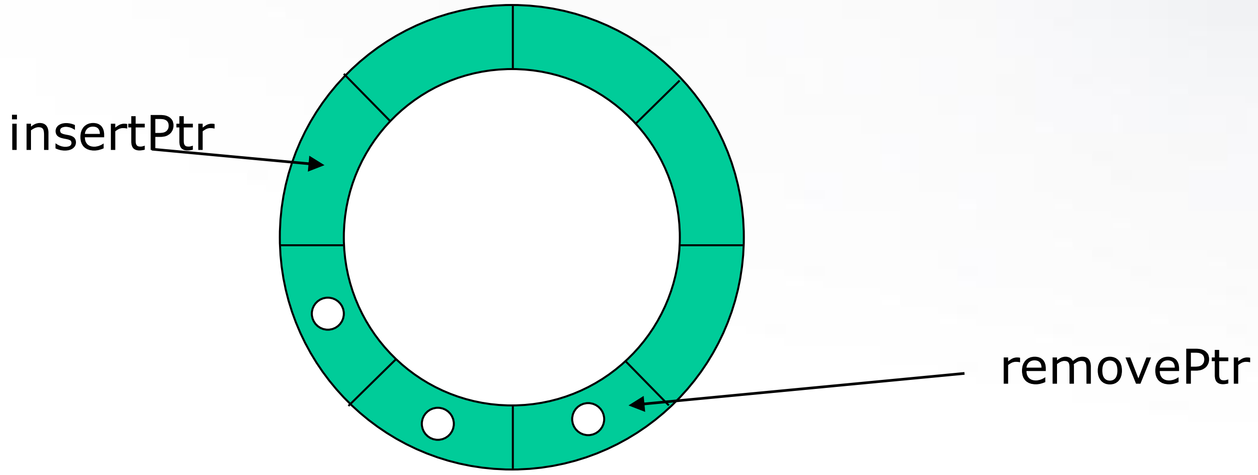
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- If the producer encounters a full buffer....Error
- or if the consumer encounters an empty buffer, ...Error the process blocks.

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.

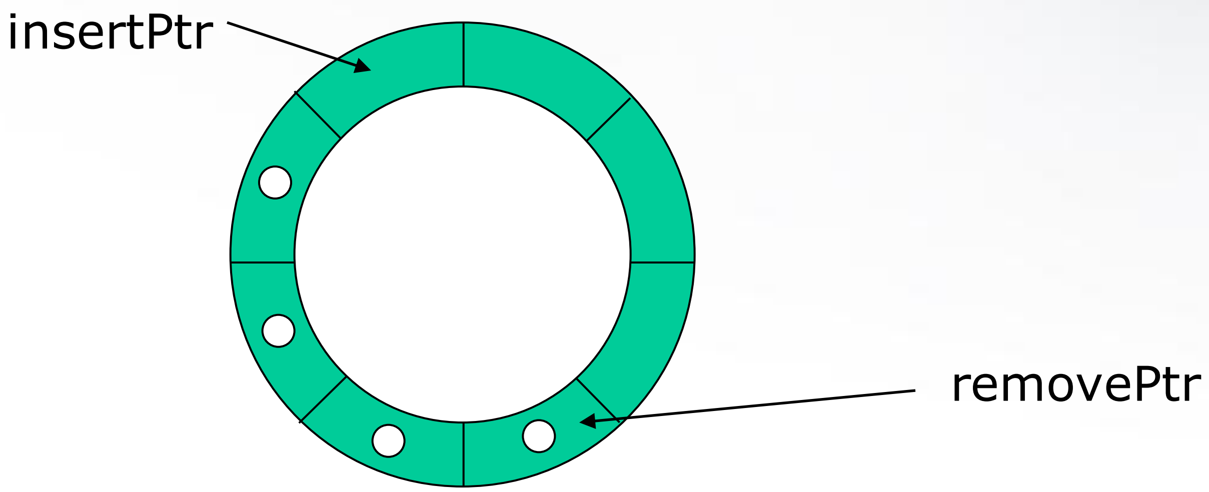
Producer-Consumer

Chef	= Producer
Customer	= Consumer



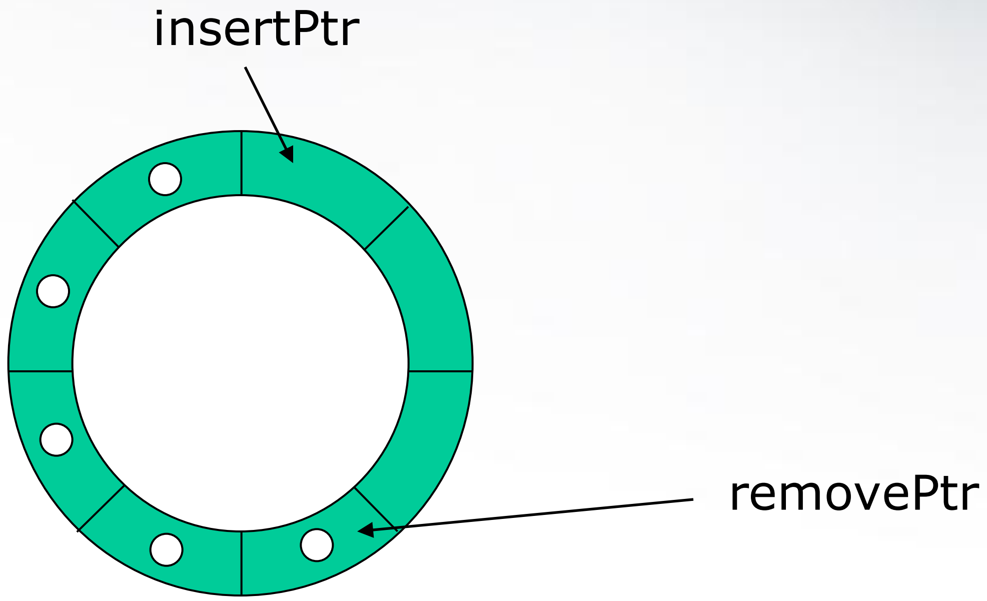
Producer-Consumer

Chef	= Producer
Customer	= Consumer



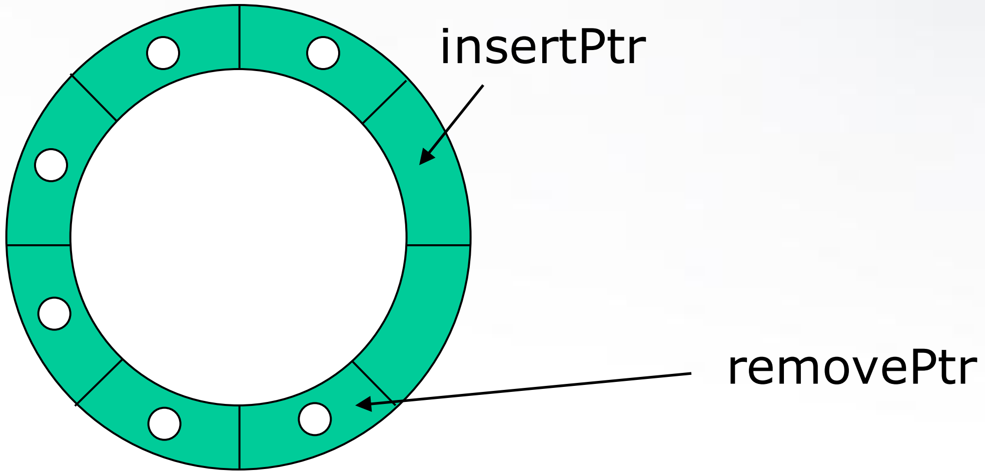
Producer-Consumer

Chef	= Producer
Customer	= Consumer



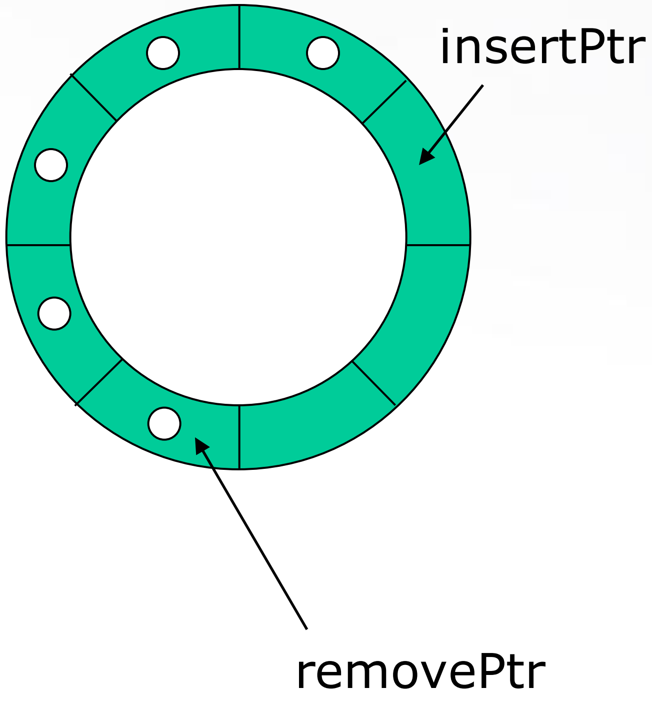
Producer-Consumer

Chef	= Producer
Customer	= Consumer



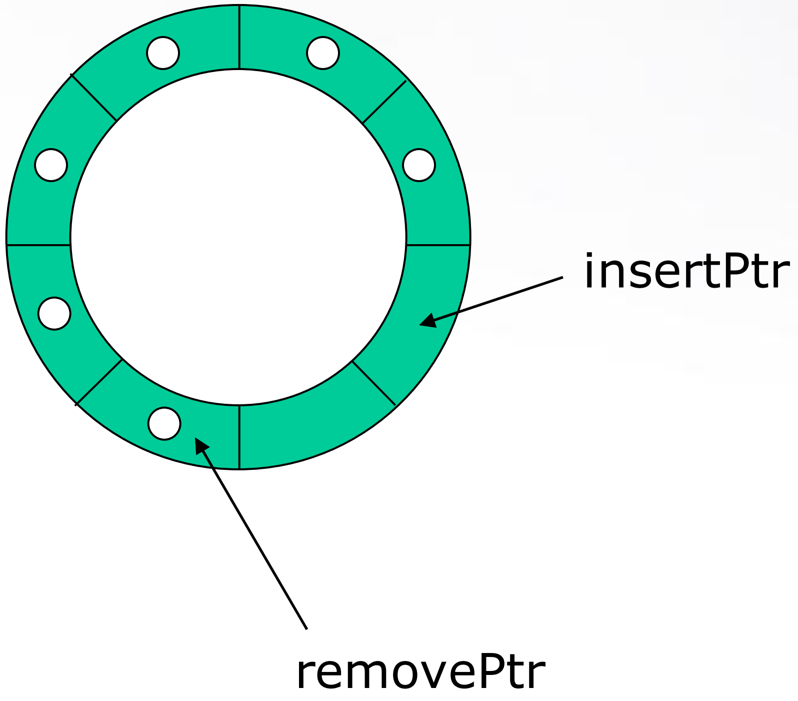
Producer-Consumer

Chef	= Producer
Customer	= Consumer



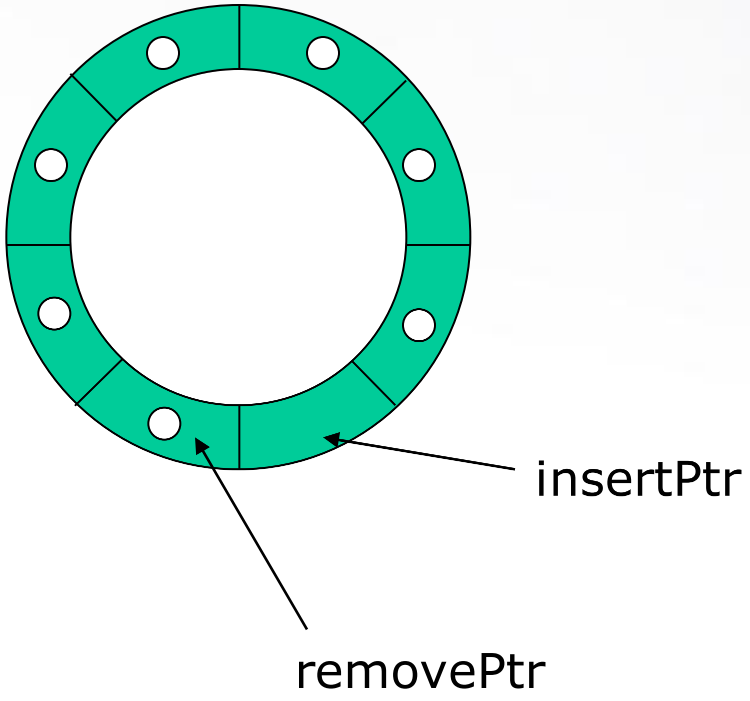
Producer-Consumer

Chef	= Producer
Customer	= Consumer



Producer-Consumer

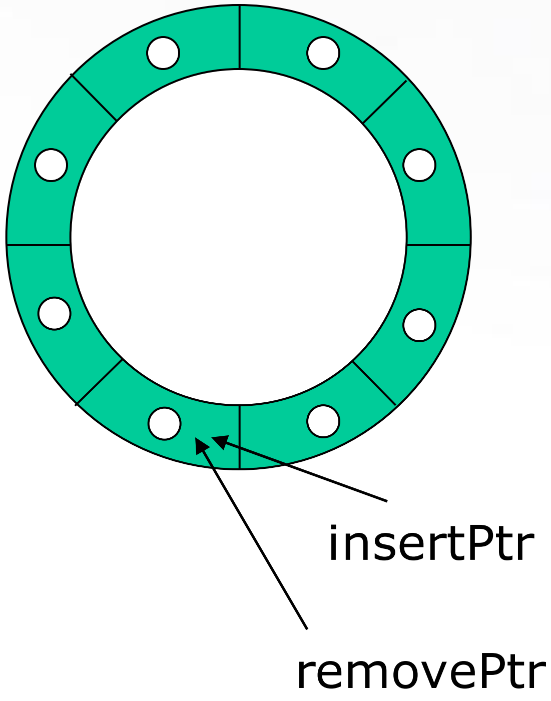
Chef	= Producer
Customer	= Consumer



Producer-Consumer

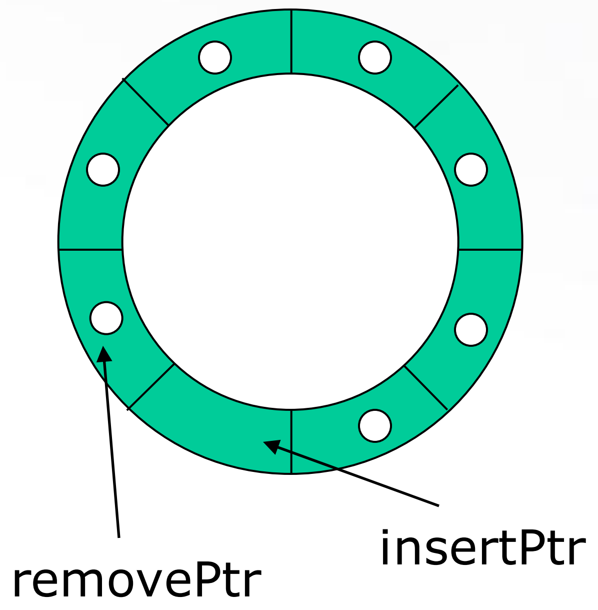
Chef	= Producer
Customer	= Consumer

BUFFER FULL: Producer must be blocked!



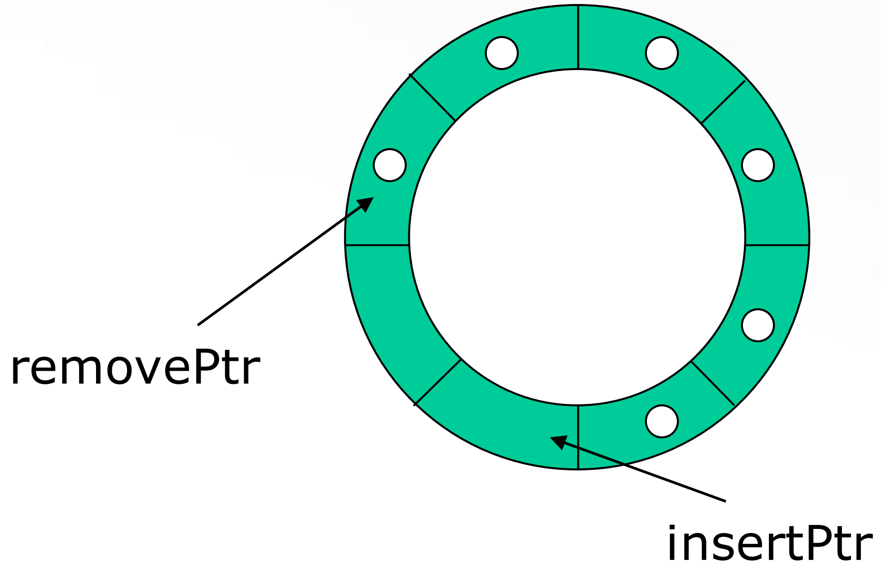
Producer-Consumer

Chef	= Producer
Customer	= Consumer



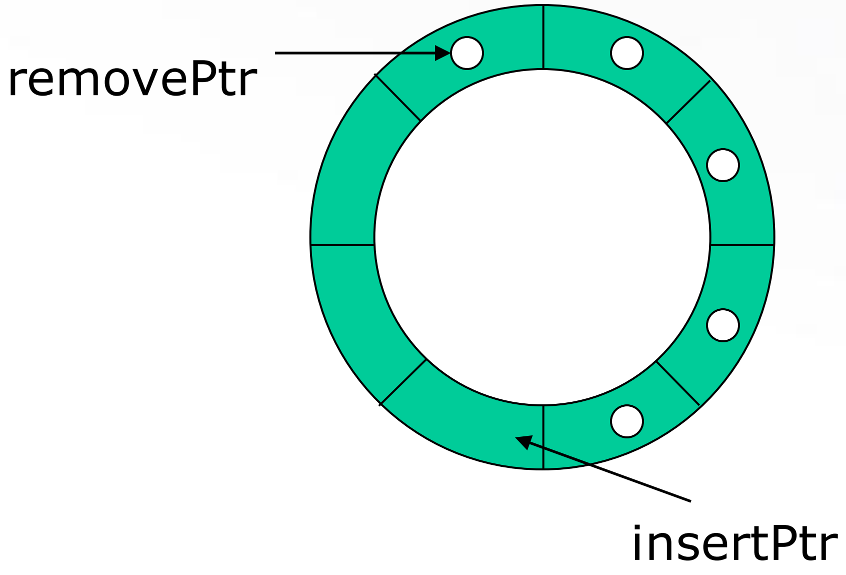
Producer-Consumer

Chef	= Producer
Customer	= Consumer



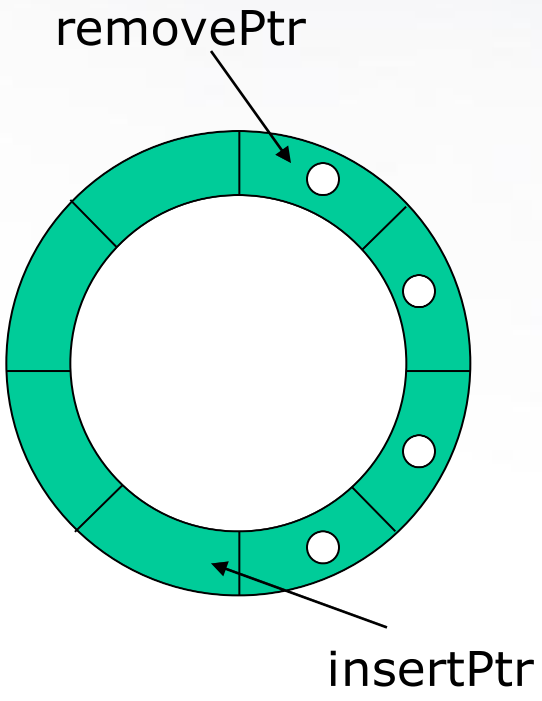
Producer-Consumer

Chef	= Producer
Customer	= Consumer



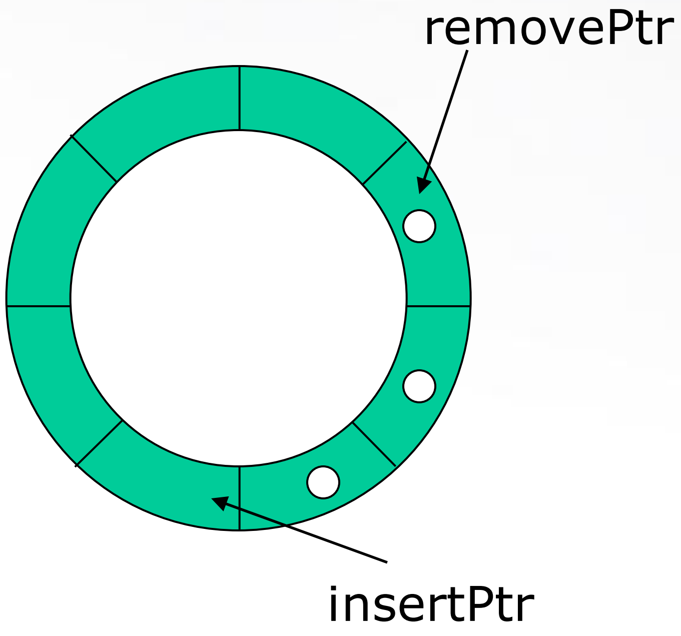
Producer-Consumer

Chef	= Producer
Customer	= Consumer



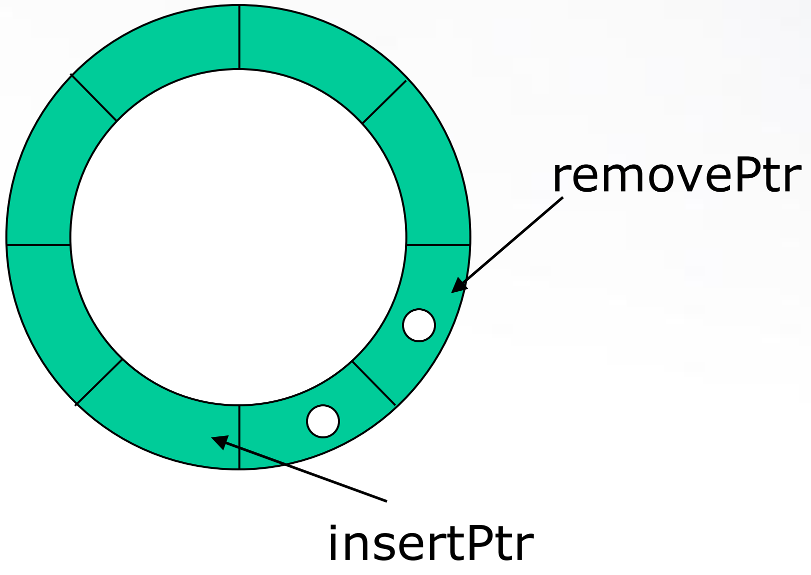
Producer-Consumer

Chef	= Producer
Customer	= Consumer



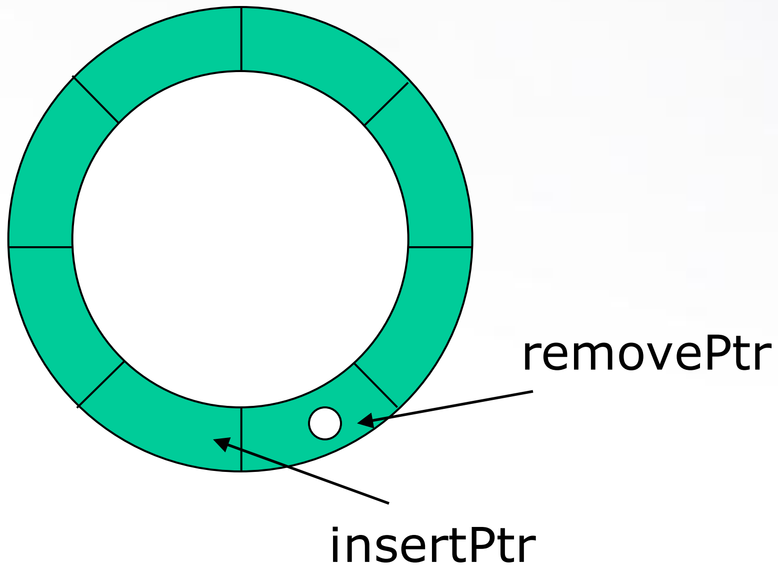
Producer-Consumer

Chef	= Producer
Customer	= Consumer



Producer-Consumer

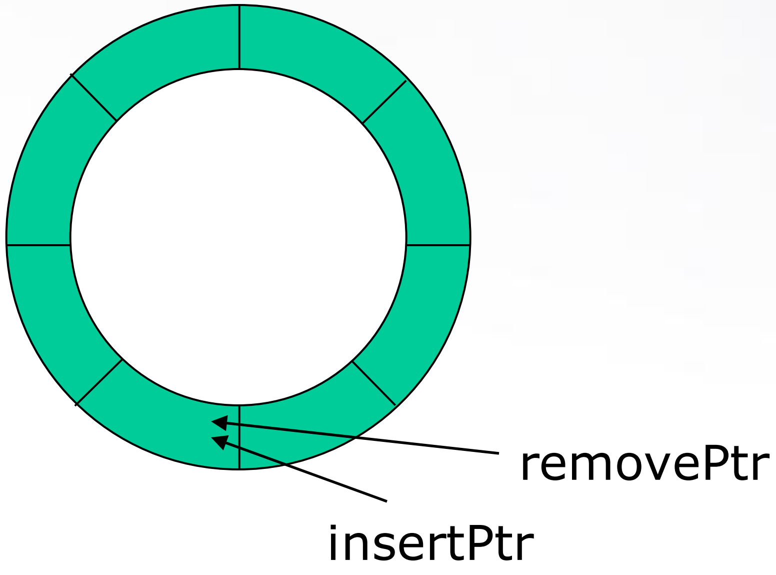
Chef	= Producer
Customer	= Consumer



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

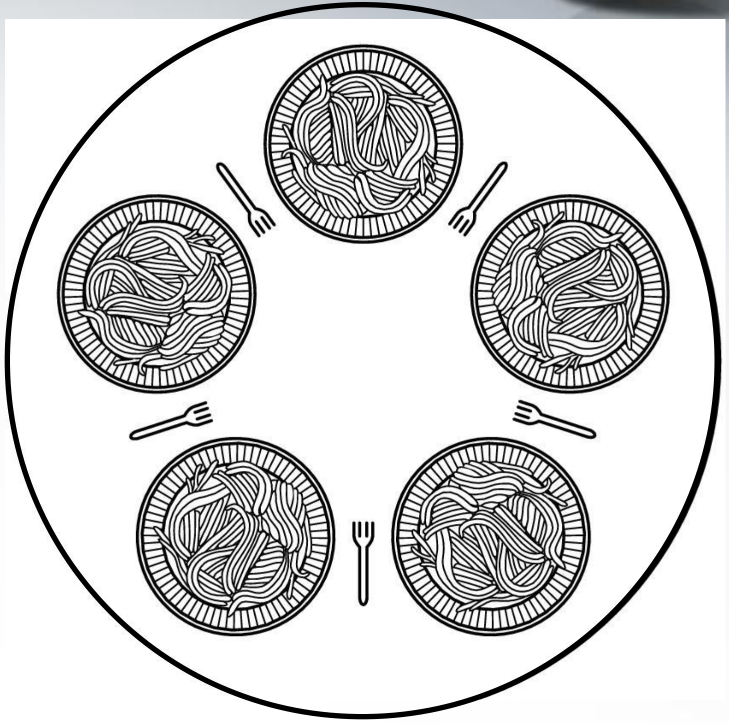
- Producer inserts items. Updates insertion pointer.
- Consumer executes destructive reads on the buffer. Updates removal pointer.
- Both update information about how full/empty the buffer is.
- Solution should allow multiple producers and consumers

Challenges

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

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."

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.