Introduction to System Programming

Threads and Concurrency

Minal Shah

Outline

- Introduction to Threads
- Principles of Concurrency
- Semaphores
- Monitors
- Producer/Consumer Problem
- Reader/Writer Problem

What is a Computer Process?

- Traditionally, a process or task is considered an instance of a computer program that is being executed.
- A process contains
  - Unit of allocation (resources)
  - Unit of execution (context)
  - Unit of external input (data)
- Of the above, which are independent of the others?
  - Unit of allocation (resources) → process
  - Unit of execution (context) → thread or lightweight process
- Can a process have more than one context?
- How would switching processes compare to switching threads?

Processes (Heavyweight)

- What resources might be owned by a process?
  - Code, memory, heap
  - Tables (files, signals, semaphores, buffers, I/O,…)
  - Privileges
- What is owned by a unit of execution (context)?
  - CPU registers (PC, SR, SP), stack
  - State
- How is information shared between processes?
  - Messaging, sockets,
  - IPC, shared memory,
  - Pipes
- How would you describe inter-process communication?
  - Expensive: need to context switch.
  - Secure: one process cannot corrupt another process.

Properties of a Process

- Creation of each process includes system calls for each process separately.
- A process is an isolated execution entity and does not share data and information.
- Processes use IPC (Inter-process communication) mechanism for communication which significantly increases the number of system calls.
- Process management consumes more system calls.
- Each process has its own stack and heap memory, instruction, data and memory map.

Threads (Lightweight)

- What is a thread?
  - A Thread is an independent program counter and stack operating within a process - sometimes called a lightweight process (LWP)
  - Execution trace.
  - Smallest unit of processing (context) that can be scheduled by an operating system
- What do all process threads have in common?
  - Process resources
  - Global variables
  - Reduced overhead by sharing the resources of a process.
  - Switching can happen more frequently and efficiently.

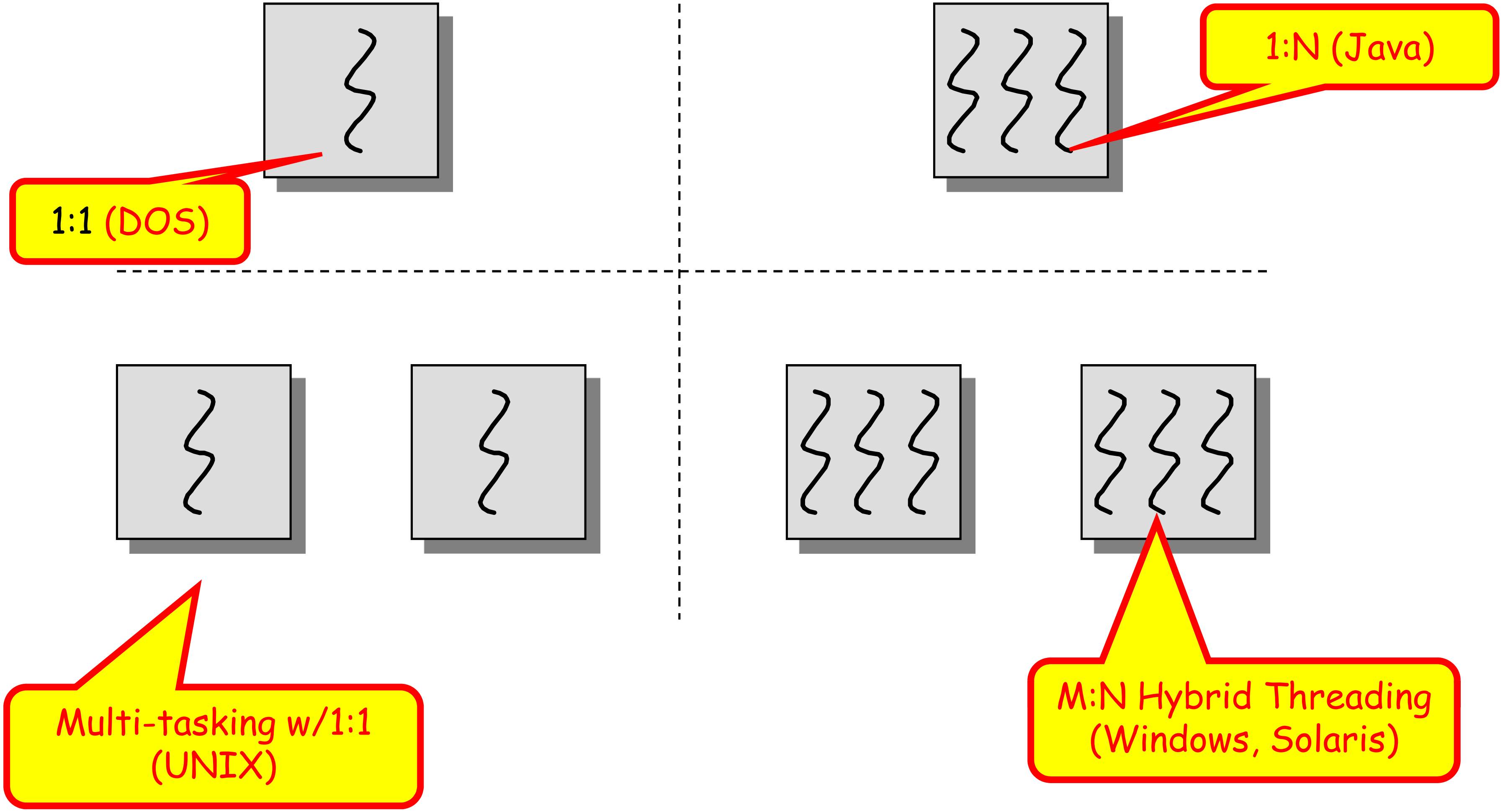
Threads (Lightweight)

- How would you describe inter-thread communication?
  - Cheap: can use process memory without needing a context switch.
  - Not Secure: one thread can write to memory in use by another thread.

Properties of a Thread

- Only one system call can create more than one thread (Lightweight process).
- Threads share data and information.
- Threads shares instruction, global and heap regions but has its own individual stack and registers.
- Thread management consumes no or fewer system calls as the communication between threads can be achieved using shared memory.
- The isolation property of the process increases its overhead in terms of resource consumption.

Examples of Threads/Processes



Types of Threads

- A thread consists of:
  - a thread execution state (Running, Ready, etc.)
  - a context (program counter, register set.)
  - an execution stack.
  - some per-tread static storage for local variables.
  - access to the memory and resources of its process (shared with all other threads in that process.)
  - OS resources (open files, signals, etc.)
- Thus, all of the threads of a process share the state and resources of the parent process (memory space and code section.)
- There are two types of threads:
  - User-space (ULT) and
  - Kernel-space (KLT).

What are the Benefits of Threads?

- Threads of a process share the instructions (code) and process context (data).
  - Far less time to create/terminate.
  - Switching between threads is faster than switching between processes.
  - No memory management issues (segmentation, paging).
  - Can enhance communication efficiency.
  - Simplify the structure of a program.
- Examples
  - Foreground/Background – editing while checking spelling / grammar.
  - Keep CPU busy (asynchronous Processing) – spreadsheet updates, read one set of data while processing another set.
  - Organization – For a word processing program, may allow one thread for each file being edited.

Thread Design

- Thread states:
  - *Ready* – ready to run.
  - Running – currently executing.
  - ***Blocked* – Waiting for resources.**
- Thread design considerations:
  - Blocking – generally, it is desirable that a thread can block without blocking the remaining threads in the process.
  - Multi-threading – allow the process to start two operations at once, each thread blocks on the appropriate event.
  - Shared resources:
    - Synchronization
    - System calls
    - Thread-safe subroutines
    - Locks

Thread Review

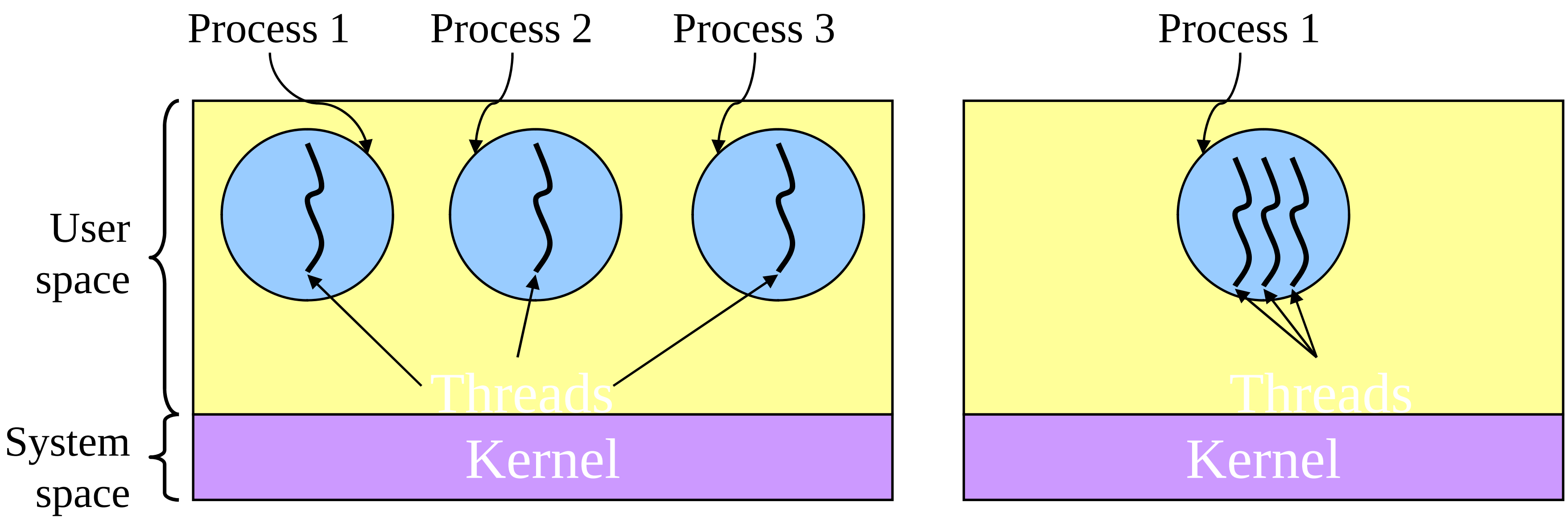
- How does a thread differ from a process?
  - Resource ownership
  - Smallest unit of processing that can be scheduled by an operating system
- What are the implications of having an independent program counter?
  - Each thread has its own stack.
  - Code and global data belong to the process and are shared among threads.
  - Threads “own” local data.
- Thread state is defined by processor registers and the stack.

Difference between Process and Thread

S.No.	Process	Thread
1	Process means any program is in execution.	Thread means segment of a process.
2	Process takes more time to terminate.	Thread takes less time to terminate.
3	It takes more time for creation.	It takes less time for creation.
4	It also takes more time for context switching.	It takes less time for context switching.
5	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6	Process consume more resources.	Thread consume less resources.
7	Process is isolated.	Threads share memory.

Threads: “processes” sharing memory

- Process == address space
- Thread == program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads



Processes Relationship

- In the concurrent environment basically processes have two relationships, ***competition*** *and* ***cooperation.***
- We distinguish between ***independent process*** *and* ***cooperating process.*** A process is independent if it cannot affect or be affected by other processes executing in the system.
- ***Independent process: These type of processes have following features:***
  - Their state is not shared in any way by any other process.
  - Their execution is deterministic, i.e., the results of execution depend only on the input values.
  - Their execution is reproducible, i.e., the results of execution will always be the same for the same input.
  - Their execution can be stopped and restarted without any negative effect.

Processes Relationship

- ***Cooperating process:*** In contrast to independent processes, cooperating processes can affect or be affected by other processes executing the system. They are characterized by:
  - Their states are shared by other processes.
  - Their execution is not deterministic, i.e., the results of execution depend on relative execution sequence and cannot be predicted in advance.
  - Their execution is irreproducible, i.e., the results of execution are not always the same for the same input.

Cooperation

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- A **race condition** occurs when the outcome depends on the particular order of execution, most often associated with shared resources.
  - Share logical space (threads)
  - Files or messages
  - Concurrent or parallel execution
- **Mutual exclusion** prevents race conditions by synchronizing resource access.

Interprocess Communication

- Related processes communicates with each other and synchronize their activity. Such communication is called IPC.
- Processes needs to communicate with each other.
- cat f1 f2 | grep hello

- 3 issues when processes communicates:

1) how one process can pass info to another process.
2) making such that 2 processes do not come in each other way.
3) proper sequencing when dependencies are there

- ex: cat f1 f2 | grep hello

- *Race Condition*
  - Two process A and B sharing some common resource. E.g A and B sharing common memory of variable count.
  - Process A increment the count.
  - Process B decrement the count.
  - CPU switching is done between process A and B.

Interprocess Communication

- Process A (count++)could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- Process B (count--) could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with “count = 5” initially:
  S0: Process A execute register1 = count {register1 = 5}
  S1: process A execute register1= register1 + 1 {register1 = 6}
  S2: Process B execute register2 = count {register2 = 5}
  S3: Process B execute register2 = register2 - 1 register2 = 4}
  S4: Process A execute count = register1 {count = 6 }
  S5: Process B execute count = register2 {count = 4}

Interprocess Communication

- Value of count is depending on which process execute at last.
  → race condition
  - Situations, where two or more processes are reading or writing , some shared data and the data result depends on who runs precisely, when are called race conditions.
- To avoid race condition, when one process is reading/writing to shared resource, other process should be prohibited/excluded
  → called as mutual exclusion.
- In above case, problem arise because process B started using variable in before process A has finished.
- When process is busy doing internal computations that do not lead to race condition.

Interprocess Communication

- When processes have to access shared memory/files lead to race condition.
- Hence, the part of program where the shared memory is accesses is called the critical region or critical section.
- Race condition doesn’t occur, if both processes are not in critical region at same time.
- In above example process A was in critical region and still process B entered in its critical region

Critical Regions

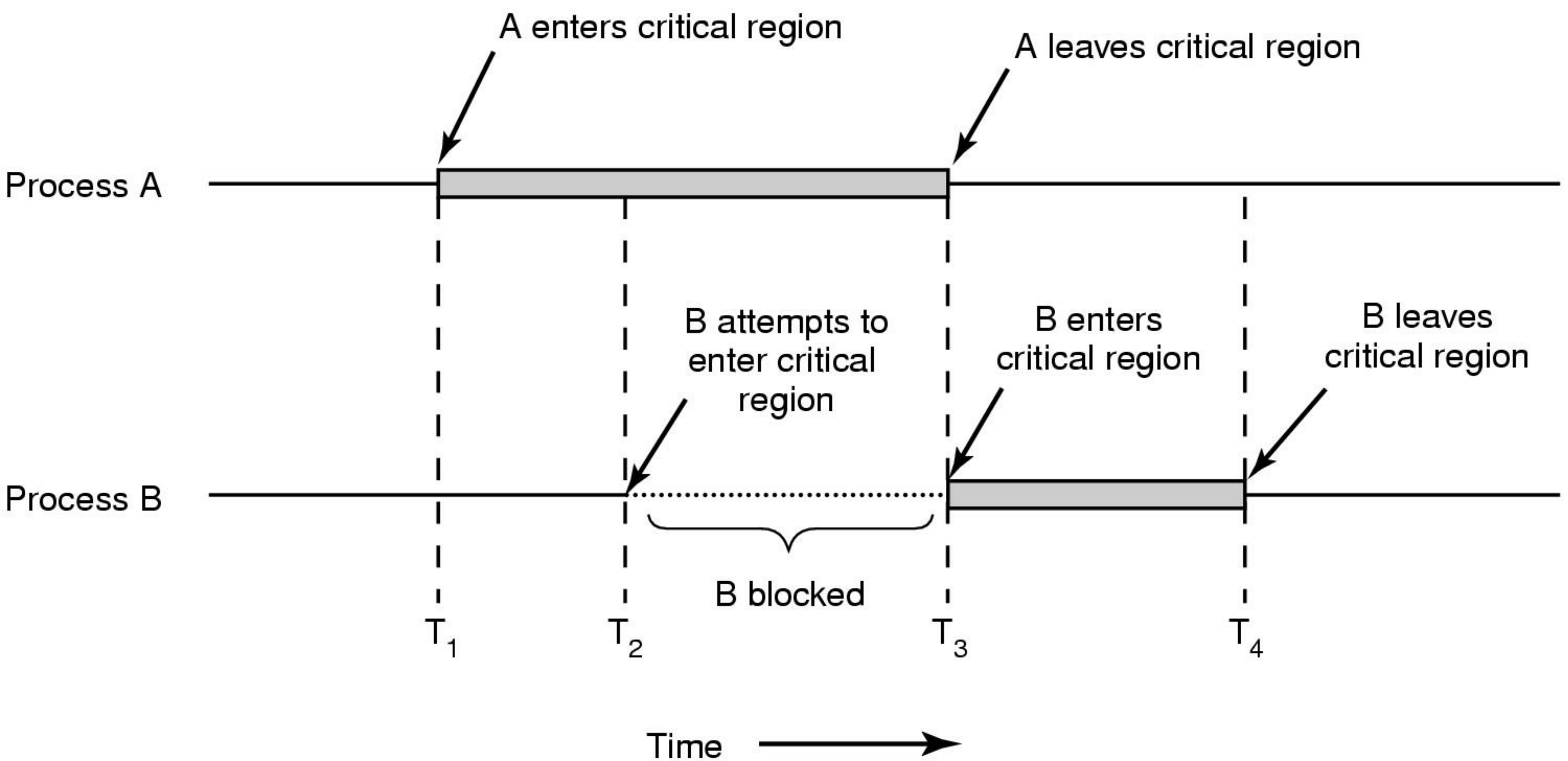
- Mutual Exclusion : if process Pi is executing in its critical section, then no other processes can be executing in their critical section
- conditions to provide mutual exclusion or for avoiding race condition.

  1) No two processes simultaneously in critical region

  2) No assumptions made about speeds or numbers of CPUs

  3) No process running outside its critical region may block another process

  4) No process must wait forever to enter its critical region



Mutual exclusion using critical regions

Process A enters CR at T_1 time. At time T_2, process B attempts to enter CR but fails because another process is already in CR and only one process is allowed at a time in CR.

Process B is suspended till time T_3 when A leaves CR, B enters CR. At time T_4, B leaves CR and now no process is in CR.

Resource Allocation

- Shared resources are
  - **Produced**
  - **Consumed**
- Resource sharing presents problems of
  - Mutual Exclusion
    - Critical resource – a single nonsharable resource.
    - Critical section – portion of the program that accesses a critical resource.
  - Deadlock
    - Each process owns a resource that the other is waiting for.
    - Two processes are waiting for communication from the other.
  - Starvation
    - A process is denied access to a resource, even though there is no deadlock situation.

Semaphores

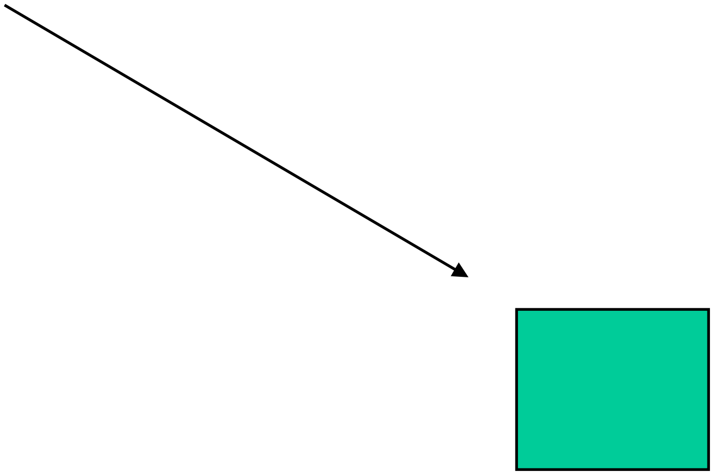
- What is a **semaphore**?
  - A semaphore is an autonomous synchronous abstract data type used for controlling access by multiple processes, to a common resource in a concurrent system.
- How are semaphores **produced**?
  - Semaphores are produced by SEM_SIGNAL.
- How are semaphores **consumed**?
  - Semaphores are consumed by SEM_WAIT and SEM_TRYLOCK.
- What are the major uses of semaphores?
  - Synchronization
  - Resource allocation
  - Mutual Exclusion

Semaphores

- What are the different types of semaphores?
  - Binary semaphore – 1 resource, 2 states
    - 0 = nothing produced, maybe tasks in queue
    - 1 = something produced, no tasks in queue
  - Counting semaphore – 1 resource, multiple copies
    - 0 = nothing produced, nothing in queue
    - -n = nothing produced, n tasks queued
    - +n = n items produced, no tasks in queue

I. Producer-Consumer

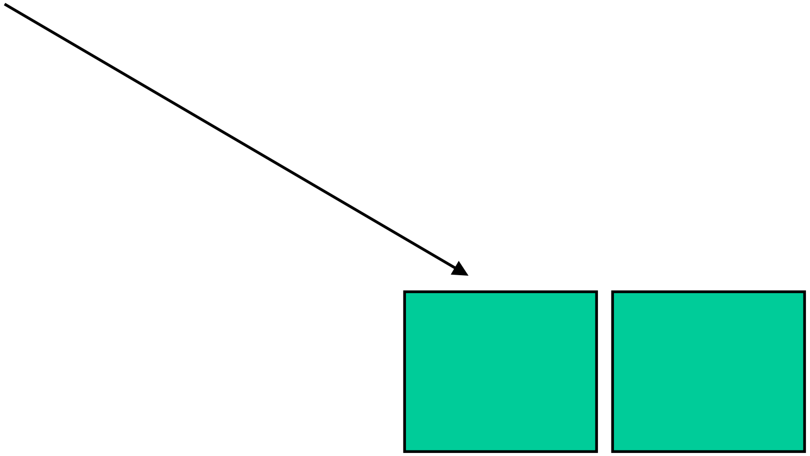
Imagine a chef cooking items
and putting them onto a conveyor



Producer-Consumer

Imagine a chef cooking items
and putting them onto a conveyor belt

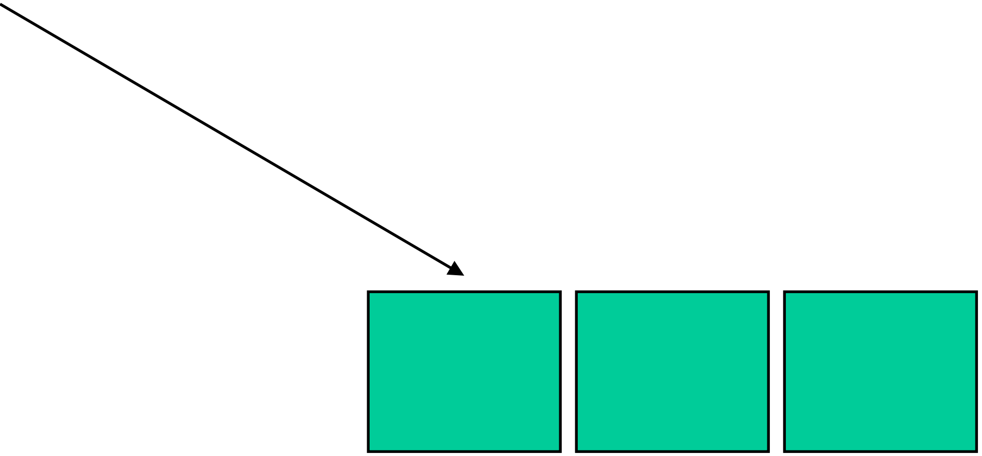
Now imagine many such chefs!



Producer-Consumer

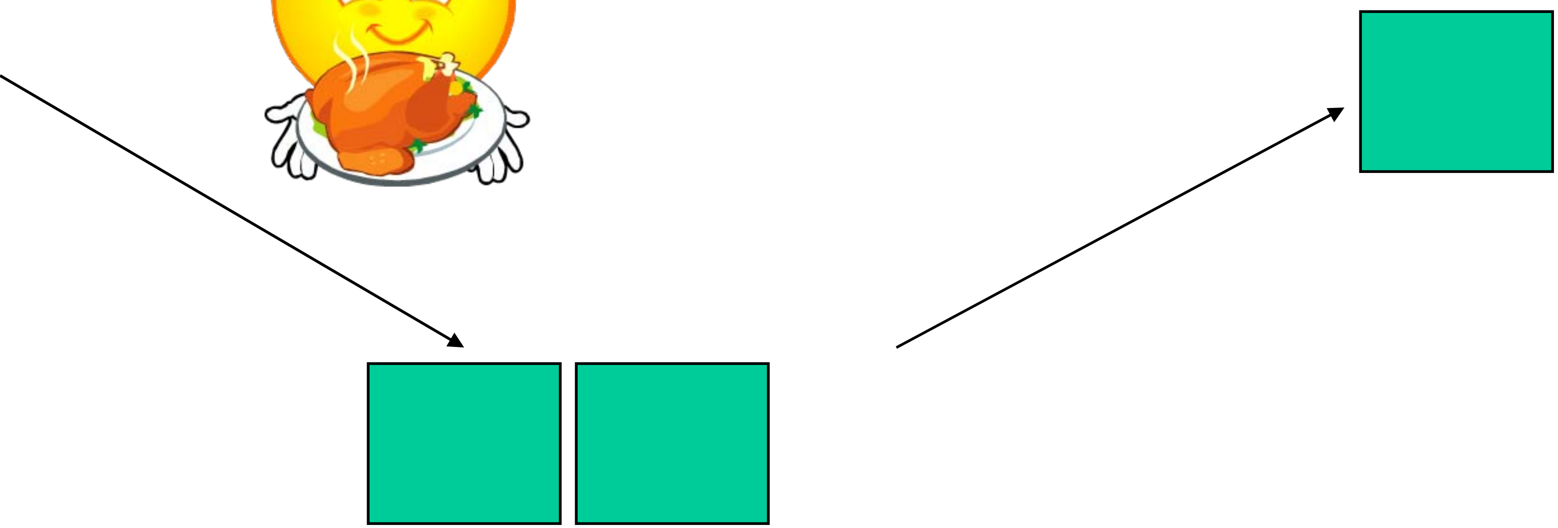
Imagine a chef cooking items
and putting them onto a conveyor belt

Now imagine many such chefs!



Producer-Consumer

Imagine a chef cooking items
and putting them onto a conveyor belt
Now imagine many such chefs!
Now imagine a customer picking items off the conveyor belt



Producer-Consumer

Imagine a chef cooking items
and putting them onto a conveyor belt

Now imagine many such chefs!

Now imagine <u>many such</u> customers
picking items off the conveyor belt

Producer-Consumer

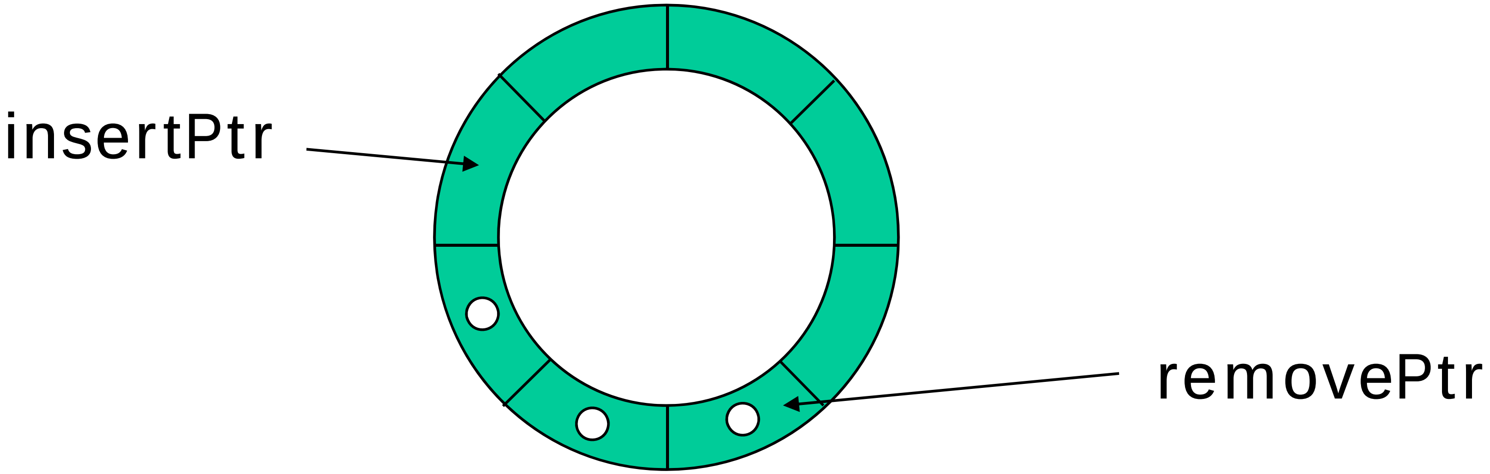
Imagine a chef cooking items
and putting them onto a conveyor belt

Now imagine many such chefs!

Now imagine many such customers picking items off the conveyor belt!

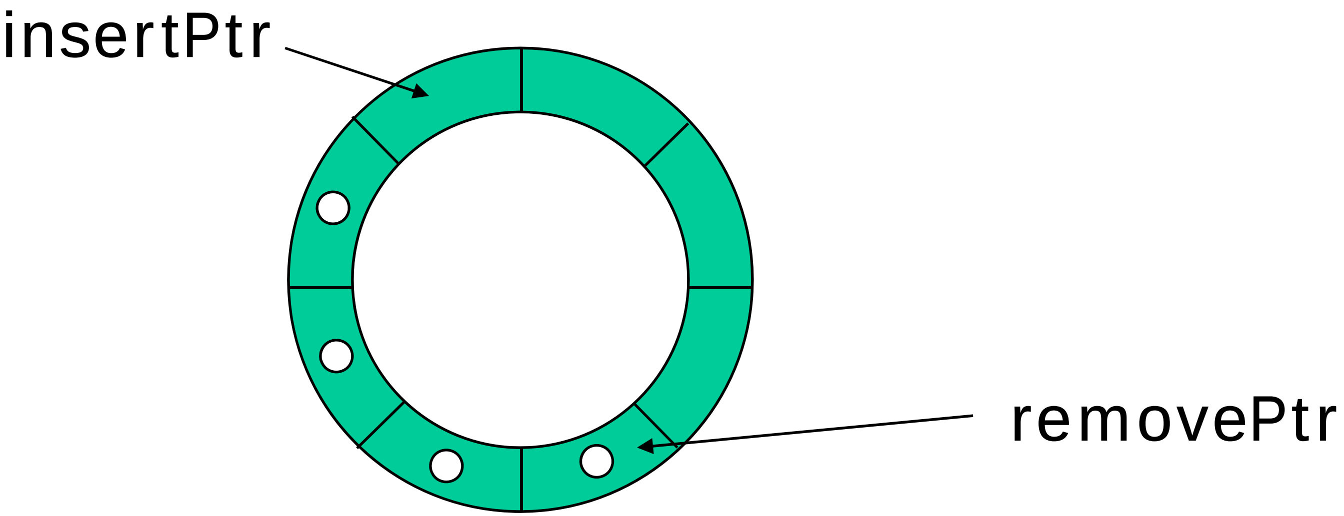
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

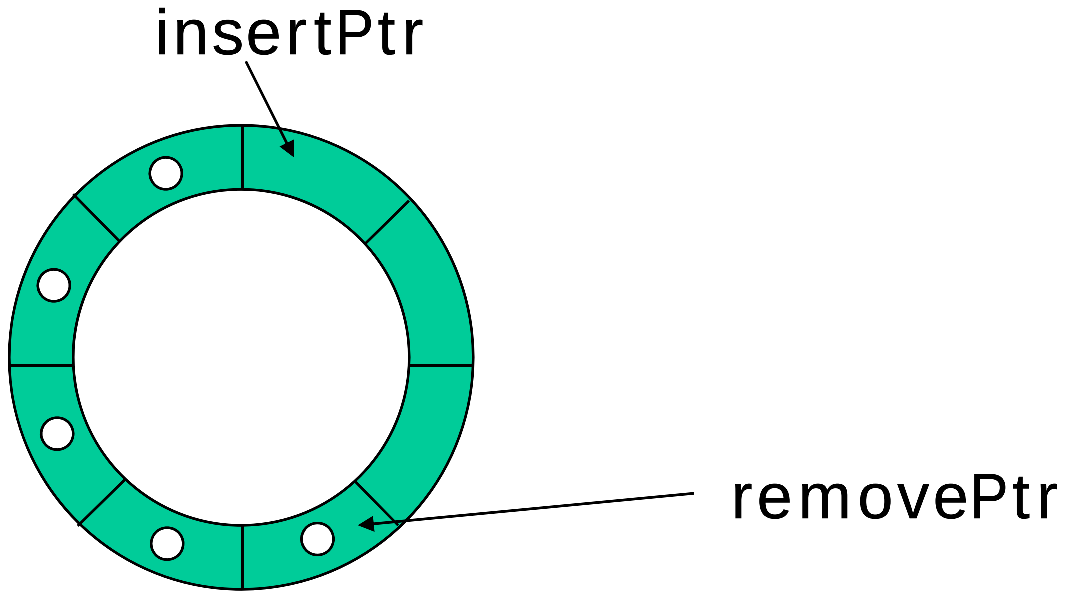
Chef = Producer
Customer = Consumer



Producer-Consumer

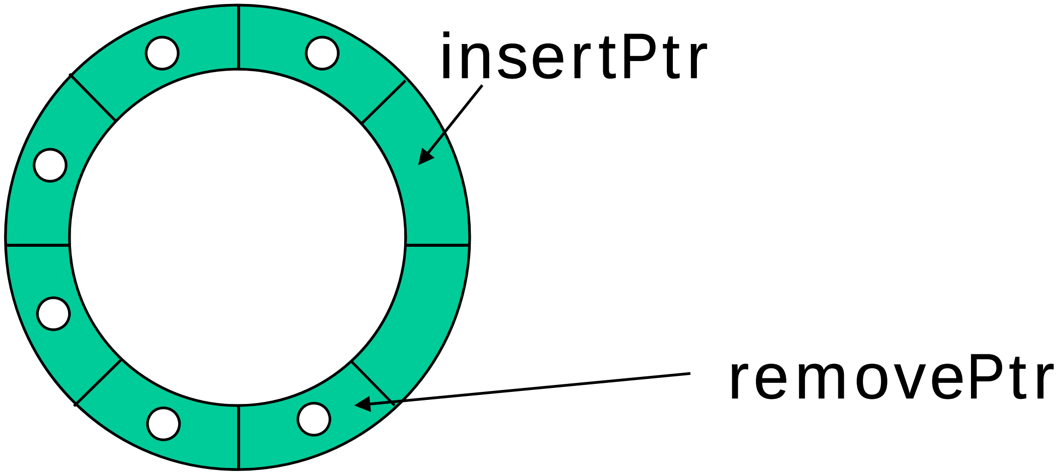
Chef = Producer

Customer = Consumer



Producer-Consumer

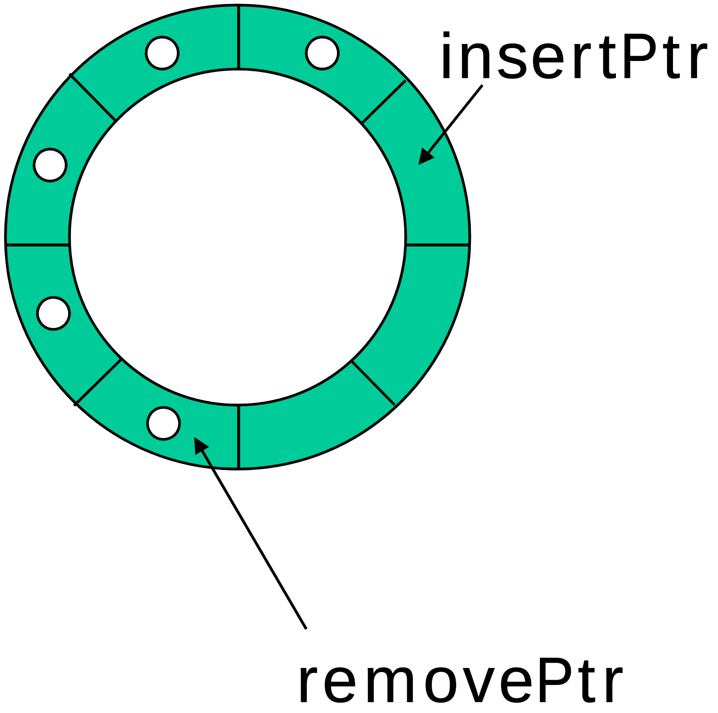
Chef	= Producer
Customer	= Consumer



Producer-Consumer

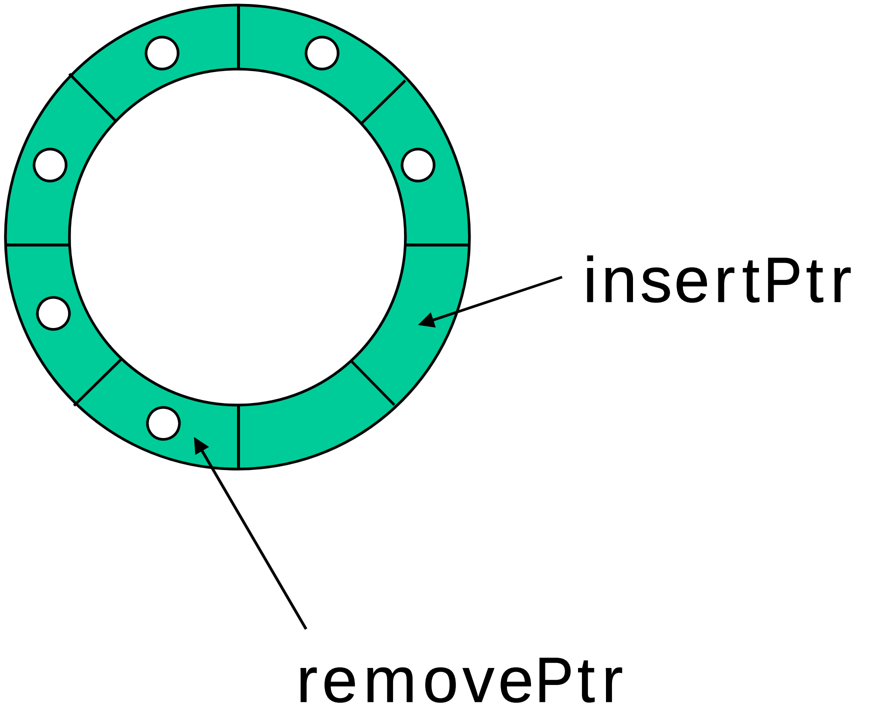
Chef = Producer

Customer = Consume



Producer-Consumer

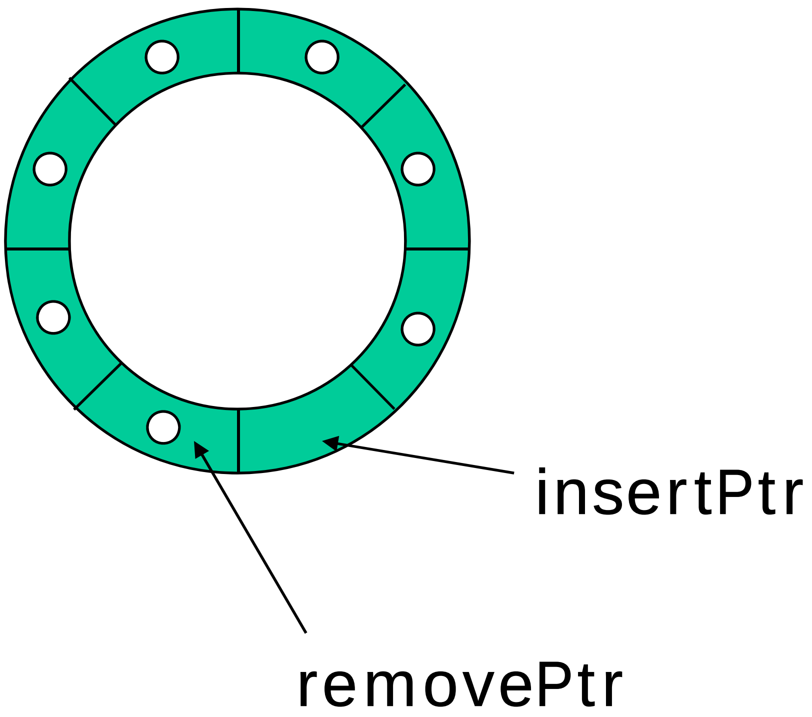
Chef = Producer
Customer = Consumer



Producer-Consumer

Chef = Producer

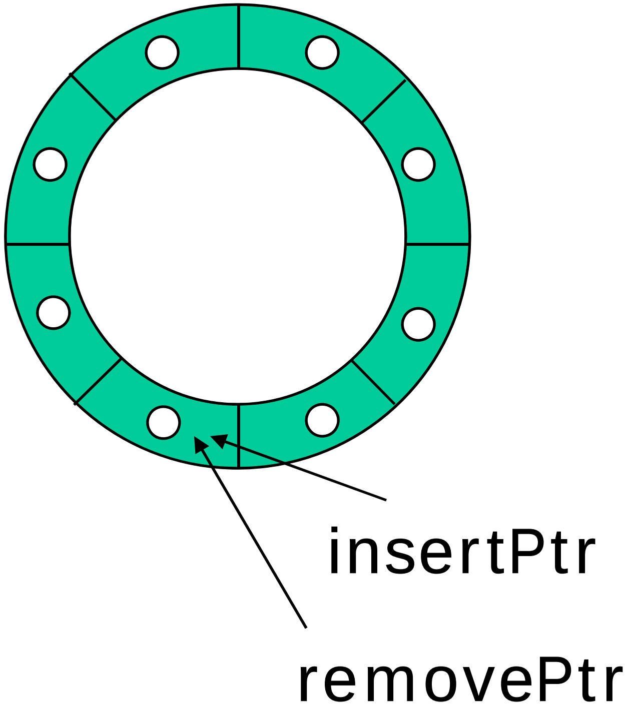
Customer = Consumer



Producer-Consumer

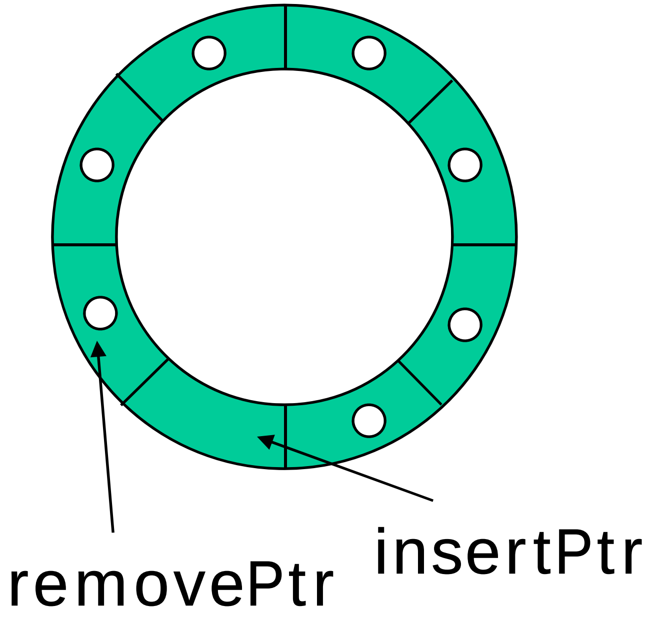
Chef = Producer
Customer = Consumer

BUFFER FULL: Producer must be blocked!



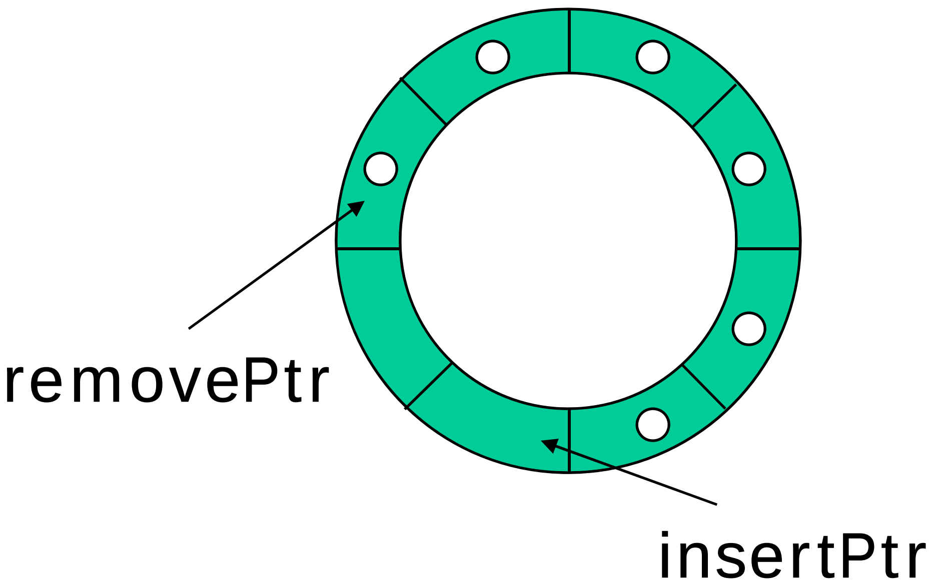
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

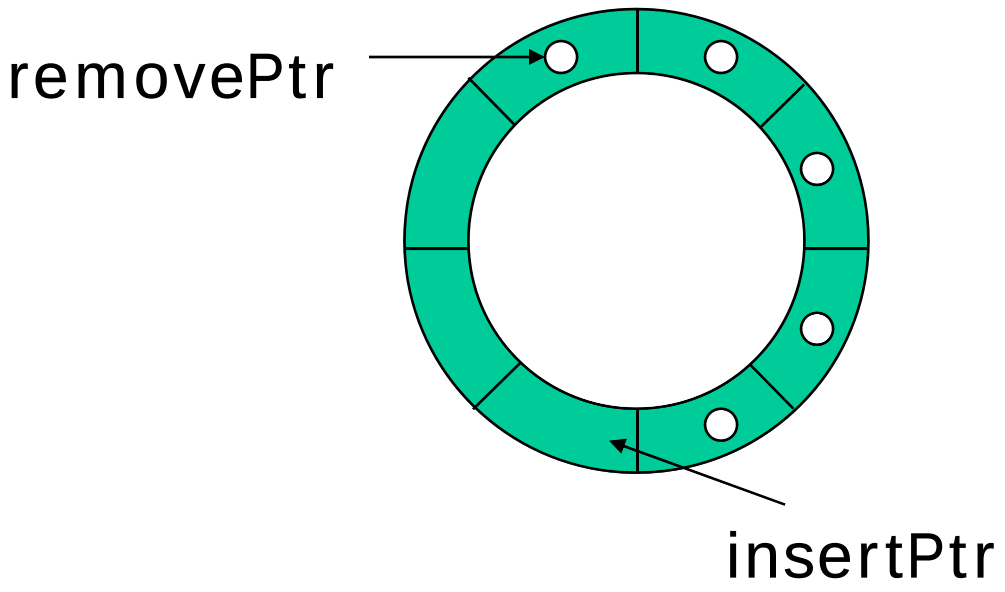
Chef = Producer
Customer = Consumer



Producer-Consumer

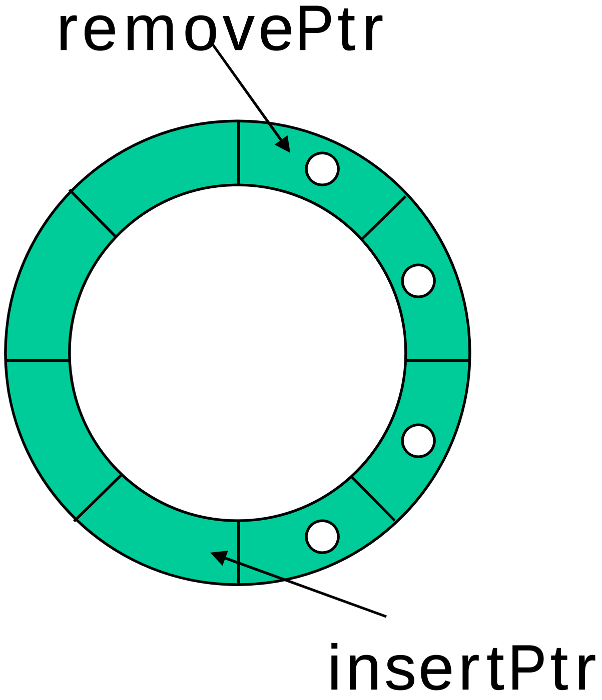
Chef = Producer

Customer = Consumer



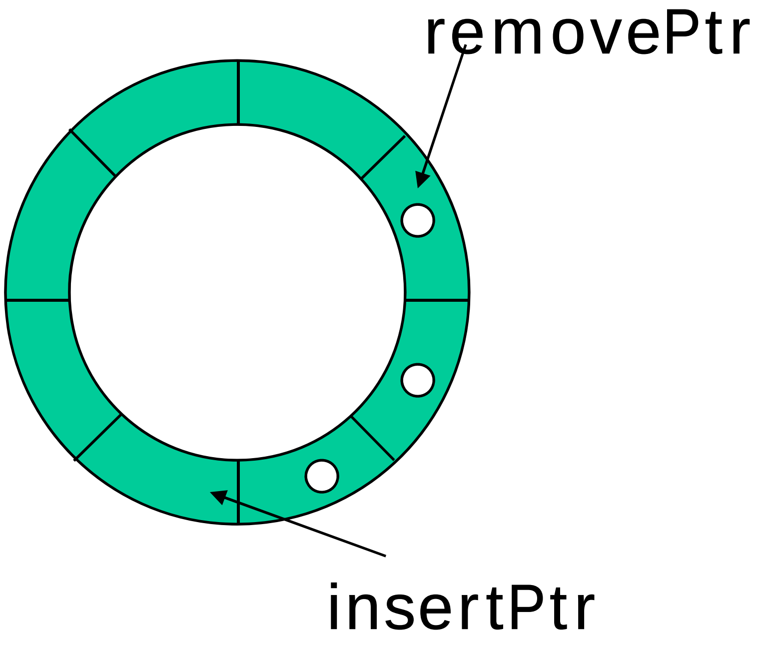
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

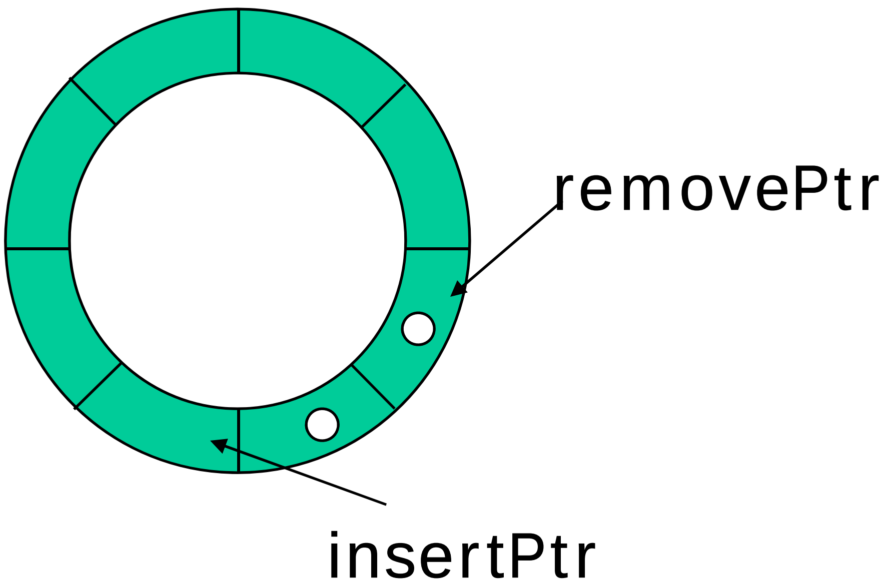
Chef	= Producer
Customer	= Consumer



Producer-Consumer

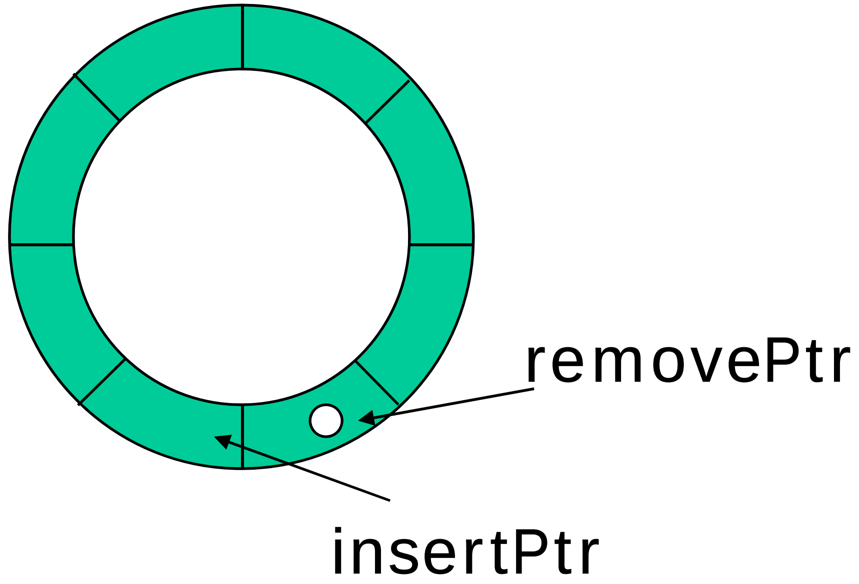
Chef = Producer

Customer = Consumer



Producer-Consumer

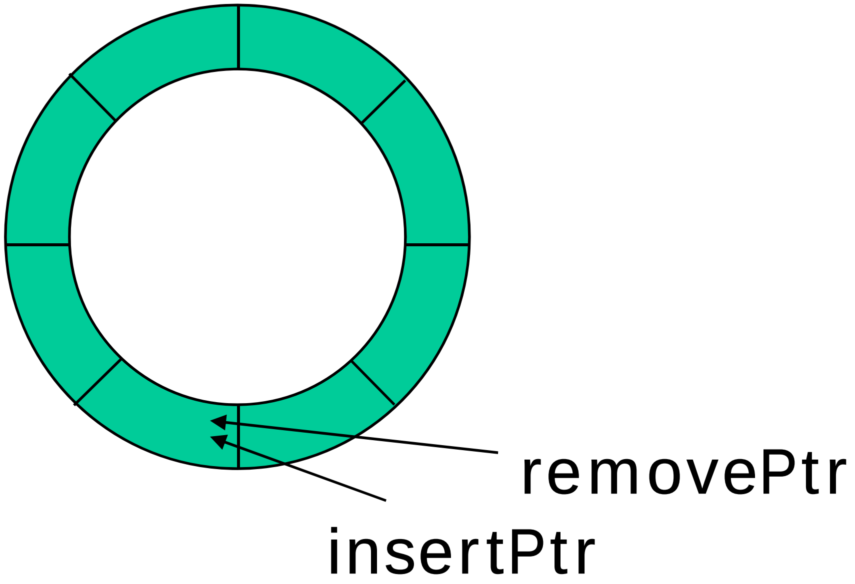
Chef = Producer
Customer = Consumer



Producer-Consumer

Chef = Producer
Customer = Consumer

BUFFER EMPTY: Consumer must be blocked!

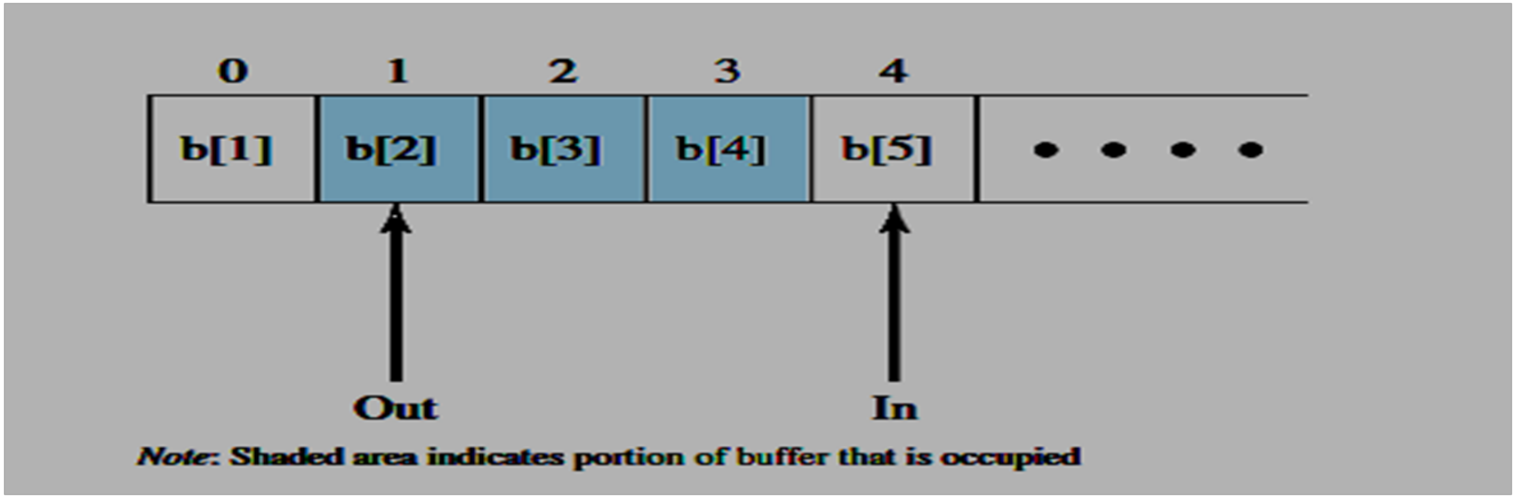


Producer Consumer problem

Infinite buffer

```
producer:
while (true) {
   /* produce item v */;
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /* do nothing */;
   w = b[out];
   out++;
   /* consume item w */;
}
```



The Producer - Consumer Problem

- Producer pushes items into the buffer.
- Consumer pulls items from the buffer.
- Producer needs to wait when buffer is full.
- Consumer needs to wait when the buffer is empty.

Producer

- producer:

```
while (true) {
   /* produce item v */
   b[in] = v;
   in++;
}
```

Consumer

- consumer:

```
while (true) {
   while (in <= out)
      /*do  nothing */;
   w = b[out];
   out++;
   /* consume item w */
}
```

Readers and Writers Problem

- Data object is **shared** (file, memory, registers)
  - many processes that only read data (readers)
  - many processes that only write data (writers)
- Conditions needing to be satisfied:
  - **many can read** at the same time (patron of library)
  - **only one writer** at a time (librarian)
  - no one allowed to read while someone is writing
- Different from producer/consumer (general case with mutual exclusion of critical section) – possible for more efficient solution if only writers write and readers read.
- Solutions result in **reader or writer priority**

Shared Data

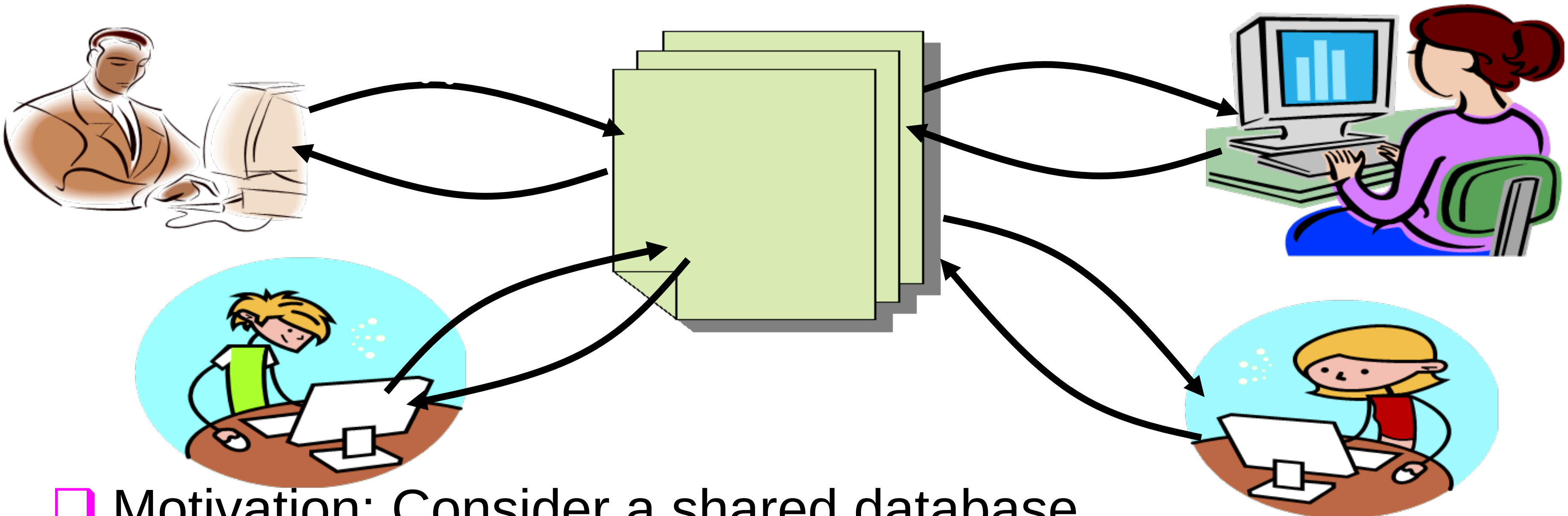
- What are some characteristics of shared data objects (files, data bases, critical code)?
  - Many processes only need mutual exclusion of critical sections (producer/consumer, mutexes)
  - many processes only read data (readers)
  - many processes only write data (writers)
- What conditions / advantages / problems arise when there is concurrent reading and writing?
  - many can read at the same time (patrons of library)
  - only one writer at a time (librarians)
  - no one allowed to read while someone is writing
  - possible for more efficient solution than producer / consumer if only writers write and readers read.
- Who should have priority, readers or writers?

Reader-Writer Problem

- readers: read data
- writers: write data
- Rule:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot in critical section together.
- Locking table: whether any two can be in the critical section simultaneously
- Writers should have mutual exclusion on shared object.
- ex: online reservation system -> shared database is there.

	Reader	Writer
Reader	OK	No
Writer	NO	No

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

The Readers and Writers Problem

- First reader – writer solution
  - Priority is given to reader.
  - First reader will get access to database. Now, if some reader comes, it is allowed to access simultaneously.
  - Any reader is allowed until there is at least one reader inside CR.
  - But if writes comes it is suspended because it will need mutual access.
- Limitation – As long as there is at least one reader, other readers are allowed. Suppose before last reader completes other arrive.
- So, writer will never get chance.

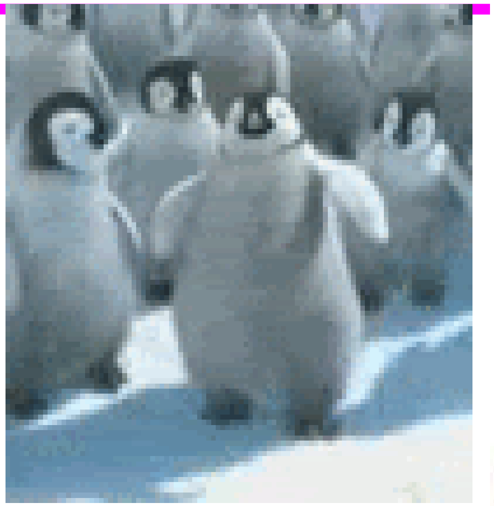
The Readers and Writers Problem Solution

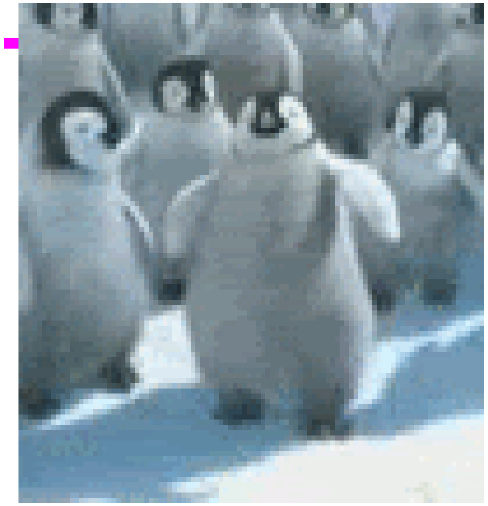
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                    /* repeat forever */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc + 1;                /* one reader more now */
            if (rc == 1) down(&db);     /* if this is the first reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            read_data_base();           /* access the data */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc − 1;                /* one reader fewer now */
            if (rc == 0) up(&db);       /* if this is the last reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            use_data_read();            /* noncritical region */
      }
}

void writer(void)
{
      while (TRUE) {                    /* repeat forever */
            think_up_data();            /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base();          /* update the data */
            up(&db);                    /* release exclusive access */
      }
}
```

The Readers and Writers Problem Solution

- Second solution:
  - When reader comes and writer is waiting, reader is suspended behind writer.
  - Because writer has to wait only for active reader.
- Disadvantage: less concurrency and low performance.








Minal Shah