**INTER PROCESS COMMUNICATION**

**Introduction to IPC**

* **IPC:** Inter Process Communication
* A mechanism enabling **communication** among processes.

Types of concurrent processes:-

* Independent processes
* Co-operating processes

Advantages of IPC:-

* Faster computation
* Modularity
* Convenience

IPC models:-

* **Message passing:** Communication with **kernel** as an intermediate.
* **Shared memory:** Communication with a **small piece of buffer memory** as intermediate.

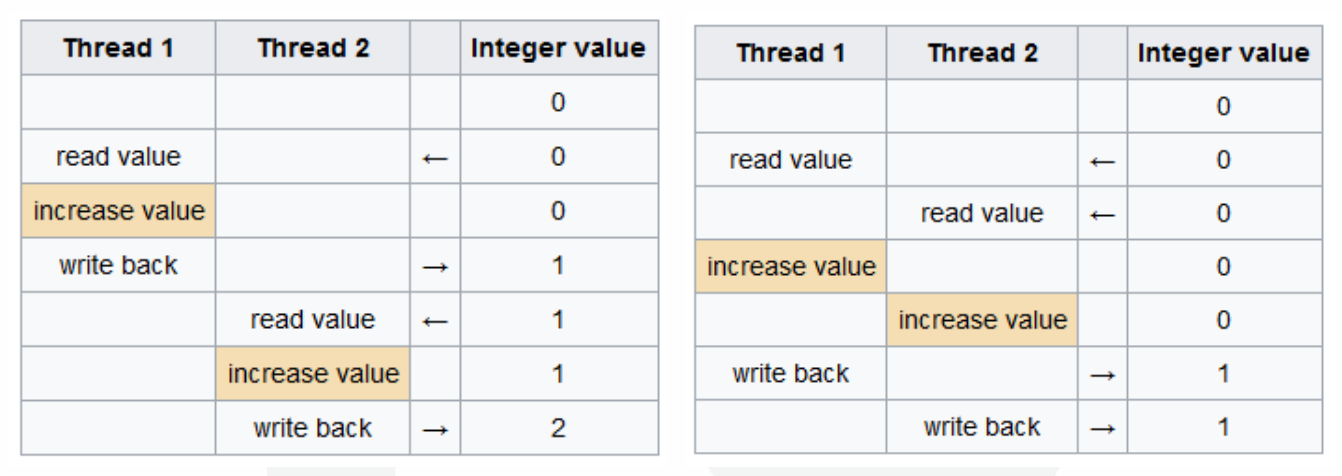
**Race Condition**

* A **situation** occurred when multiple processes try to access same piece of data.
* Generally, occurs among threads.

Reasons:-

* One instruction is executed **before** other(s).
* One instruction executes **faster** than other(s).
* Common **shared resources** like memory/data/file.

Example:-



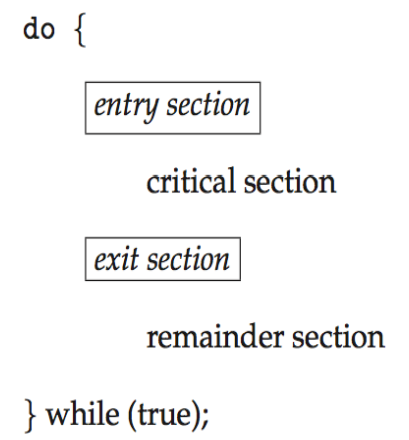
**Critical Section**

* Piece of **code** **shared** by multiple processes.
* It is **protected** to be accessed by only **one process at a time**.
* A process trying to access it when another process is already on it, is **blocked**.
* It is also known as **critical region**.
* Attempt for its concurrent access results in both processes executing **one after other**.

**Mutual Exclusion**

* A program written to **avoid concurrent access** to a shared resource.
* Actually, it is that code in critical section which **protects** it.
* Also known as **mutex**.

**Critical Section Problem**



* **Entry section:** Part of critical section **managing** access requests.

Solution must satisfy these:-

* **Mutual exclusion**
* **Bounded waiting:** All processes that requested get chance to enter the critical section.
* **Progress:** No process interferes another.
* **Arbitrary speed:** There is nothing like relative speed.

**Ways to Achieve Mutual Exclusion**

* Disabling interrupts
* Shared lock variable
* Strict alteration
* TSL (test & set lock) instruction
* Exchange instruction
* Peterson’s solution

***\*For any question asked about particular method, draw its example timeline and write pseudo code.\****

**Disabling Interrupts**

* **Interrupt:** **Signals/process** produced by a device connected to computer.
* Disabling interrupts would result in lesser chances of another process **entering critical section** induced by interrupts.
* Any process **entering** critical section **disables** all interrupts & then **re-enables** them when **leaving** it.

Problems:-

* **Unwise** to provide user process power to control interrupts.
* **Hazardous** if user forgets to include code for **re-enabling** interrupts.
* In multiprocessor systems, user have to program for each processor **separately**.

**Shared Variable Lock (SVL)**

* **SVL** has a value of **0 or 1**.
* Value of **SVL** is **0** usually, but set to **1** when a process is **inside the critical section**.
* Another process trying to access it has to **wait** if value of **SVL is 1**.
* So, 1 basically means that critical section is **occupied** by another process.

Problems:-

* Context may switch while a process is setting **SVL to 1**.
* **Unsafe** as sometimes after context switching a process may read **SVL as 0**, and enter the critical section despite another process being there.

**Strict Alteration**

* Integer variable **turn** tells **which** process will enter critical section.
* **turn** is initially **0**.
* If a process finds turn to be different than their **process number**, then it **waits in the loop**, testing each iteration if it is its turn.
* **Busy waiting:** Continuous **testing** for something to happen.

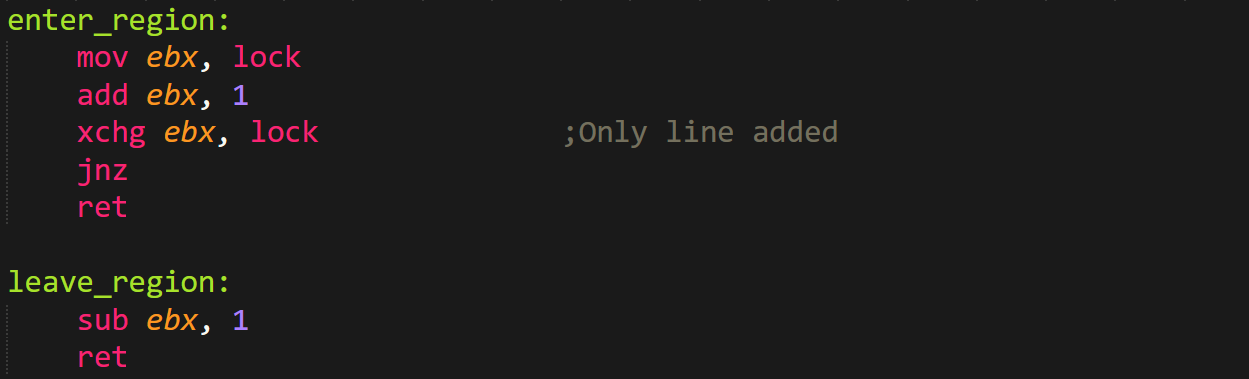
Disadvantages:-

* Sometimes a process **may not** enter critical section after being into it once.
* It then resides in **non-critical region** and another process ready to enter critical section is **unnecessarily blocked** due to it, **wasting** CPU’s time.
* Turns may **not** be **ordered well**.

**TSL Instruction**

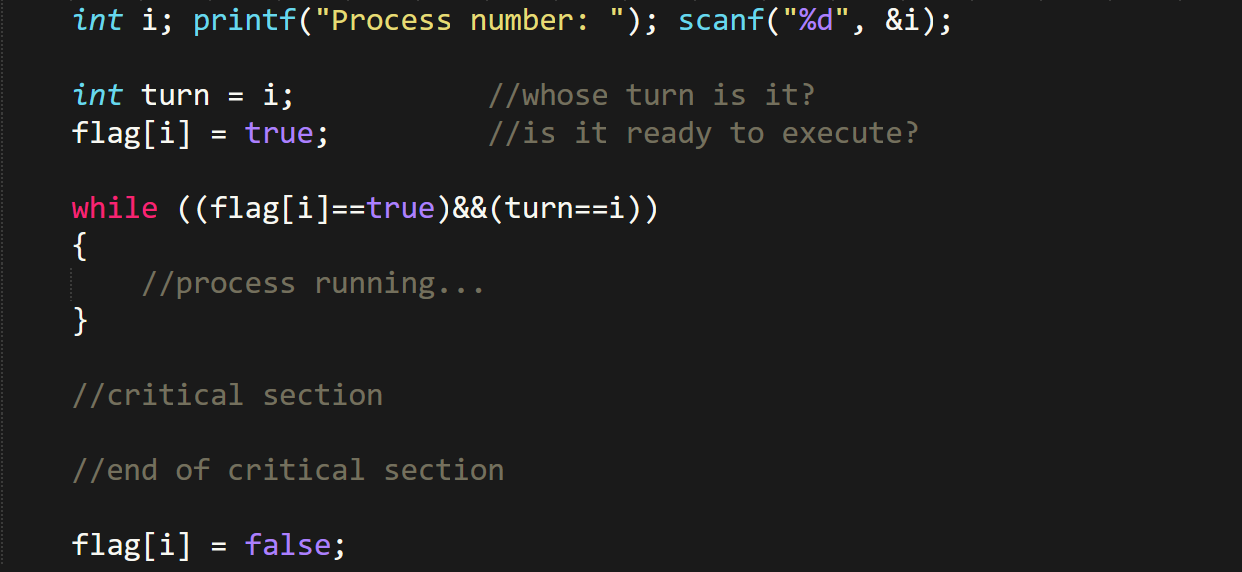


**Exchange Instruction**



**Peterson’s Solution**

* Allows multiple processes to access a **resource** (**not** critical section) **without** conflict.
* Processes communicate through **shared memory**.



Disadvantages:-

* Involves too much **busy waiting**.
* Can be applied on only **2 processes** at a time.

**Producer Consumer Problem**

* It’s a problem regarding **synchronization of multiple processes**.
* Also known as **bounded buffer problem**.
* In this problem, we have a **producer** & a **consumer**.
* They communicate through **shared memory** buffer.
* Producer’s job is to **produce** **information** & **store** **it in buffer**.
* Consumer’s job is to **utilize that information** & **remove it from buffer**.
* We have to make sure that the producer **doesn’t** produce anything if buffer is **full** & the consumer **doesn’t** try to consume anything if buffer is **empty**.

Solution:-

* In case of **full buffer**, producer either **discards its data** or **goes to sleep**.
* Producer is **notified** when buffer becomes **empty**.
* In case of **empty buffer**, consumer **goes to sleeps** & **notified** when...
* Buffer could also be **partially** filled.

Problem:-

* It has a **race condition** leading to **deadlock**.
* **Deadlock:** A process in which **no progress** is made.

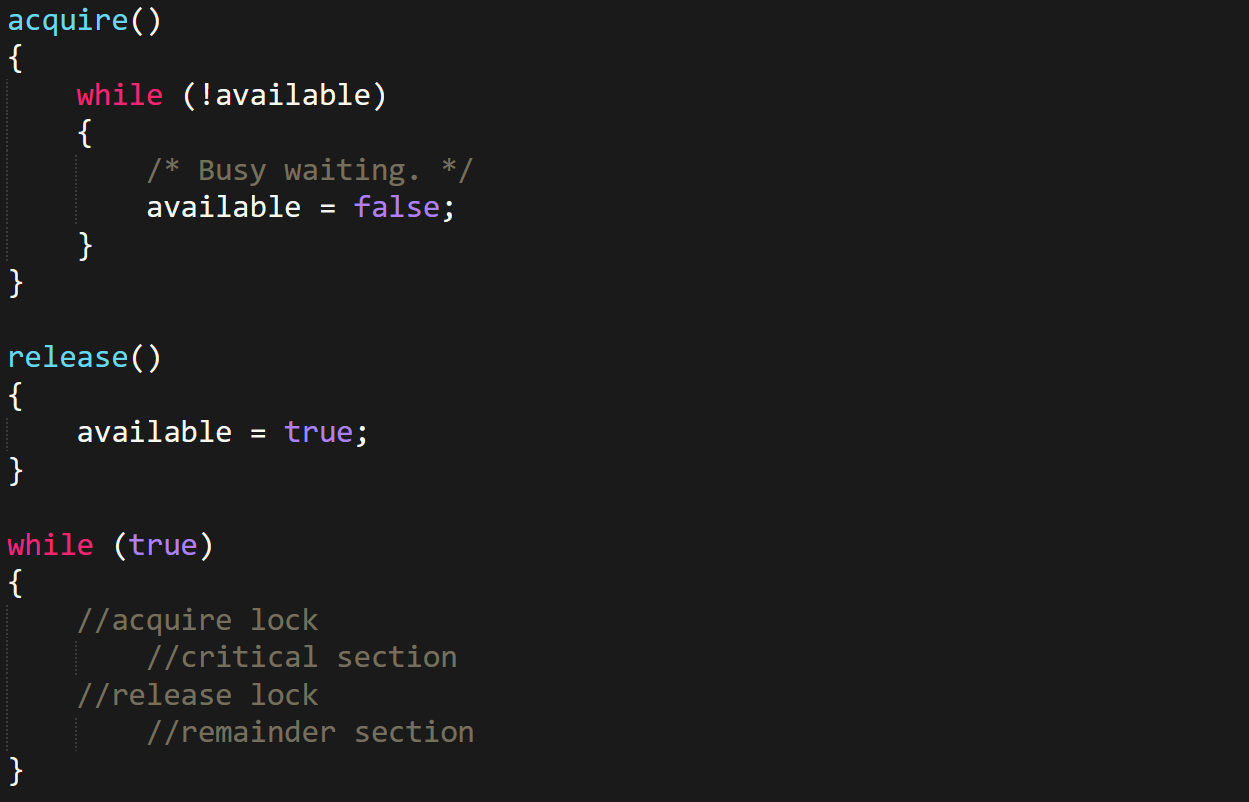
What happens:-

* We know that **consumer** **goes to sleep** when **count is 0** (empty buffer).
* So in case of a **empty buffer**, the consumer attempts to **sleep**.
* But **before** it could sleep, it is **interrupted** by **producer** who has new data to fill the **buffer**, turning the count to **1**.
* Now the producer will **try waking up the consumer**, but it is **already awake**.
* And when consumer is **resumed**, they **resume** the process of **getting into sleep** which was **suspended** earlier.
* After falling asleep, it **won’t** wake up as producer tries waking it up **only once**.
* Then when buffer is **full**, producer also goes to **sleep**.
* Thus, both of them will **continue sleeping** forever resulting in a **deadlock**.

**Mutex Lock**

* There is a **boolean variable** saying if critical section is **available or not**.
* There are also 2 keys: ***acquire()*** and ***release()***
* ***acquire()*** makes our boolean variable ***false*** if critical section is **not** available.
* ***release()*** makes it ***true*** if available.
* It involves **busy waiting**, and thus also known as **spin lock**.

Code:-



**Semaphore**

* **Semaphores:** Variables that are constantly checked and modified.
* It solves the problem we faced in **producer-consumer** model.
* Two types of semaphores: **binary semaphores** & **counting semaphores**
* A function ***insert\_item()*** is similar to producer.
* And so is ***remove\_item()*** to consumer.
* No busy waiting.

Binary semaphores:-

* Can be either **0** or **1**.
* Has two methods: ***up* & *down***, or ***lock* & *unlock***, or ***signal* & *wait*** etc.
* These methods are used for **acquiring** or **releasing** lock.

Counting semaphores:-

* More than two possible values.

Function ***insert\_item()*** & ***remove\_item()***:-

* They get **mutually exclusive** access to **critical section**.
* ***insert\_item()*** only works when buffer is **not** overflowing.

Advantages:-

* No producer starvation (***insert\_item()*** doesn’t wait forever).
* No consumer starvation (***remove\_item()*** doesn’t wait forever).

Down operation:-

* If value is **1** then makes it **0**.
* If value is **0** then **goes to sleep**.
* Done as an **atomic** action, thus **no busy waiting**.
* When a function operation has started, it **can’t** be interrupted.

Up operation:-

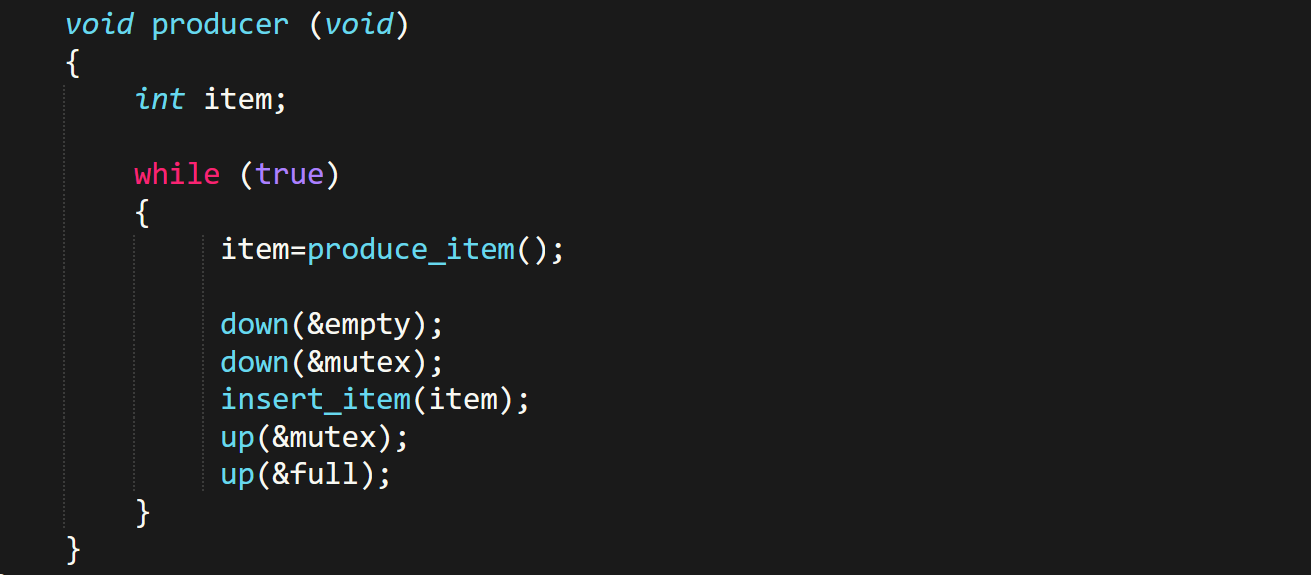
* If value is **0** then makes it **1**.
* If value is **1** then **wakes up** sleeping semaphores.
* Also contains **atomic operations** in functions.
* Also has **uninterruptable** operations.

**Producer-Consumer Using Semaphores**

Common starting:-



insert\_item:-



remove\_item:-

***\*Now you know\****

**Monitor**

* Same definition as **code library**.
* Processes call **other procedures** in monitors (**not** monitor into **user procedure** like libraries).
* Monitors’ internal is somehow **encapsulated**.
* It is created for specially **working with processes**.
* Monitors achieve **mutual exclusion** by allowing only one process at a time in it.
* When a process is called into monitor, monitor’s first few lines of code checks if any other process is present in it or **not**.
* If yes, then this new process is **suspended** until the old one leaves the monitor.

**Producer-Consumer Using Monitor**

* It uses: **Conditional variables**, **wait operation** & **signal operation**.
* Sometimes the process might be **unable** to be continued (like **buffer overflow**).
* When such situations are faced, the monitor allows any other blocked process to execute, if it could.
* This is done using a **conditional variable**, which tells if a process could **continue or not**.

**Fooling Authorities in Exams (OS Codes)**

* Make ***typedefs*** for some **key variables**.
* Compulsorily write a ***while*** loop.
* Try converting **steps** into codes.
* **For example:** Adding to buffer, removing & the steps involved in it.
* And the codes must be **pseudo**, not serious.
* There are no rules, you can use ***Booleans*** & ***objects*** in **C** codes.

**Event Counters**

* A special integer value that can only be **incremented**.

Possible operations on event counters:-

* **Read(E):** **Returns** value of event counter ‘E’.
* **Advance(E):** **Increments** event counter ‘E’.
* **Await(E,V):** Process resumes until value of E becomes **V or more**.

Consumer-producer perspective:-

* Both contain a **serial number**.
* Producer contains the serial number of data to be **consumed** by consumer.
* And those data are **serially passed** to consumer for consuming.
* ***in*** is commonly used as **serial number** for **recently added item** to buffer.
* ***out*** is similarly…
* Rest of it, all **classical laws** are followed by producer & consumer.

**Readers Writers Problem**

* It is also a **synchronization** **problem**.
* A **piece of resource** has to be shared by **multiple processes**.
* **Reader** & **writer** are two types of processes.
* Processes categorized as **readers**, can **read** from the shared resource **simultaneously**.
* Whereas only **one** **writer** process can **write** to that resource at a time.
* And also, **writers** can write only when **no reader is reading** the shared resource.
* It shows that readers are given **higher priority**.

Solution:-

* To solve it, we use: **a mutex (*m*)**, **a semaphore (*s*)** & ***read\_count*** variable
* ***read\_count*** **tracks** the **number of readers** accessing the shared resource.
* ***read\_count*** is initialized to **0** & ***m*** and ***s*** are to **1**.

Solution implementation:-

* When a **reader** wants to access the resource, ***m*** makes a **lock** on the resource.
* Then ***read\_count*** is **incremented**, and then reader **accesses** the resource.
* After that, the **lock is removed** from the resource (**unlocked**).
* ***\*Similar process is followed when reader stops reading the resource\****
* The ***s*** is active when there are active readers (**atleast 1**).
* This **signals** writers to **not** try writing to the resource when any **reader is present**.
* And similarly allows them to write if **no** reader is accessing it.

**Message Passing**

* If each reader belongs to a **separate computer**, these computers can **parallelly communicate** with each other.
* Or another way is by **parallel communication among the processors** of same computer.
* Its easier when **OOP** is used.
* This method can be either **synchronous** or **asynchronous**.
* **Synchronous message passing:** Sender is **blocked** until receiver **receives and processes** the message.
* **Asynchronous message passing:** Sender **continues working** without waiting for the receiver.

Send & receive function:-



**Dining Philosopher’s Problem**



* **Five philosophers** sit in a dining, each with a bowl of rice.
* And there are **five chopsticks** on the table, but a philosopher requires **two chopsticks** when **eating**.
* At a time, a philosopher is either **eating** or **thinking**.
* A philosopher waits for **left chopstick** (if **not** available); then picks it up for **eating**, first.
* Then does the same with **right** chopstick.

Problem:-

* **Not enough** (2n) **chopsticks** available.
* The two **adjacent philosophers** of an eating philosophers **can’t** eat.
* When **all are hungry**, they will pick up their **left chopstick**.
* As now **no right chopstick** is available, this creates a **deadlock** condition.

Possible solutions:-

* A philosopher must be allowed to pick up the chopsticks if both **left and right** are available.
* **At max four philosophers** can eat at a time, leaving **at least one** chopstick on table.

Metaphors:-

* Here, **philosophers** represent **processes**.
* And **chopsticks** represent **resources**.
* So **philosophers using chopsticks** represent **processes accessing resources**.

Properties:-

* As each chopstick represent a resource, each chopstick has **one semaphore**.
* This semaphore is **Boolean semaphore**.
* The state where the philosopher is waiting for chopstick(s) to be available together is called **hungry**.
* Initially, all philosophers are in **thinking** state.
* ***Thinking -> Hungry -> Eating***
* Eating has to pass through ***Hungry*** state.