Inter Process Communication

Bhupendra Pratap Singh

ACTS, CDAC, Pune

Revisiting Process

Program loaded in memory for CPU execution is a process

Types –

- Independent Process
- Co-Operating Process

Independent Process:

❖ Not affected by the execution of other processes (resources are not shared).

Co-Opearating Process: affected by the execution of other processes (resources are shared)

Need/Requirements: Increasing computational speed, convenience & modularity.

What is Inter Process Communication (IPC)?

- Mechanism which allows processes to communicate each other and synchronize their events/actions.
- Processes can communicate with each other using two ways:

Shared Memory – Eg. Global variables/buffers shared between two processes(threads within process).

(Producer-Consumer Problem)

Message Passing: (without shared memory) – Establish a communication link (no need if already exists) for examples sockets for over TCP/UDP communication **(send and receive functions)**

✓ **pipe is also an example of Message passing**

Shared Memory

- A total of four copies of data are required (2 read and 2 write).
- Shared memory provides a way by letting two or more processes share a memory segment.
- The data is only copied twice – from input file into shared memory and from shared memory to the output file.

Some useful system calls

- ftok(): is use to generate a unique key.
- shmget():
- int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.
- shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat().

  void *shmat(int shmid ,void *shmaddr ,int shmflg);
- shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

Cont…..

- shmdt(): When you’re done with the shared memory segment, your program should detach itself from it using shmdt().

  Syntax: int shmdt(void *shmaddr);

- shmctl(): when you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

  Syntax: shmctl(int shmid,IPC_RMID,NULL);

Associated Terminologies

Critical Section

- The critical section in a code segment where the shared variables can be accessed.
- Atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.

Critical Section Example

```
do
{
    Entry Section

    Critical Section

    Exit Section

    Remainder Section

} while (TRUE);
```

Solution to Critical Section Problem

- **Mutual Exclusion**

Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

- **Progress**

Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

- **Bounded Waiting**

Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

Some more Terminologies

Race Conditions:

concurrent access to shared resources may lead to inconsistency/
/corrupted resources

True atomicity:

Critical section must be executed by single process/thread.

Support pseudo/logical atomicity:- custom user code with busy waiting principle - semaphores, mutex, spinlocks etc.

Some more Terminologies

Spin Lock:

Spinlock is a **lock** which causes a thread trying to acquire it to simply wait in a loop ("**spin**") while repeatedly checking if the **lock** is available. Since the thread remains active but is not performing a useful task, the use of such a **lock** is a kind of busy waiting.

Some more Terminologies

Semaphore:

A **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

RTOS context : It is an object.

Types:

- Binary Semaphore
- Counting Semaphore

Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.

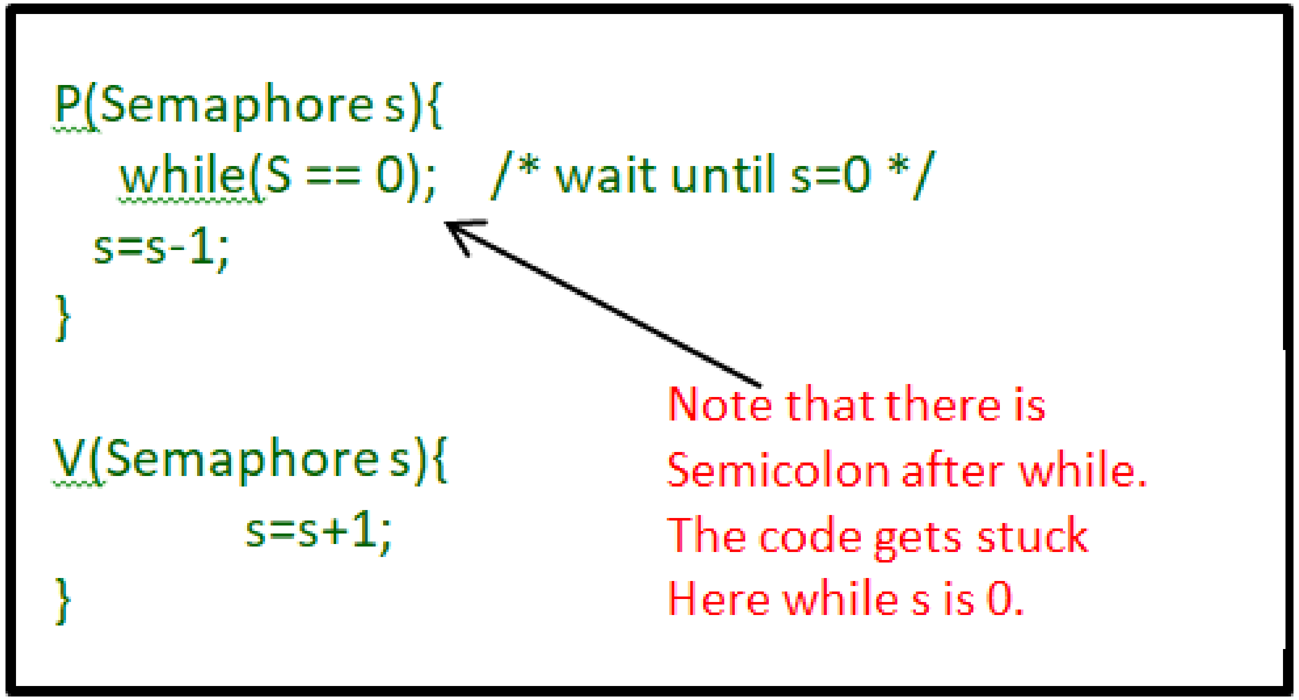
Semaphores cont..

In layman terminology - it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P(S) and V(S) respectively. The classical definitions of wait and signal are:

Wait: Decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).

Signal: Increments the value of its argument S, as there is no more process blocked on the queue.

look at two operations which can be used to access and change the value of the semaphore variable.



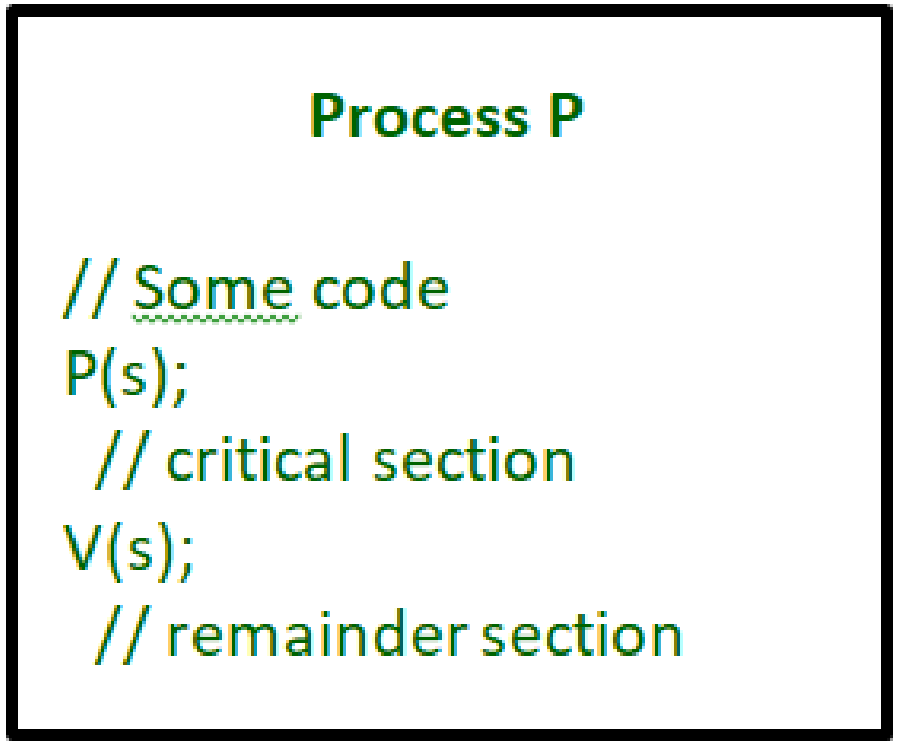
```
P(Semaphore s){
    while(S == 0);    /* wait until s=0 */
  s=s-1;
}

V(Semaphore s){
            s=s+1;
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

P(wait) and V (Signal) Operation

- P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
- Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
- A critical section is surrounded by both operations to implement process synchronization.

Critical Section: In view of P and S

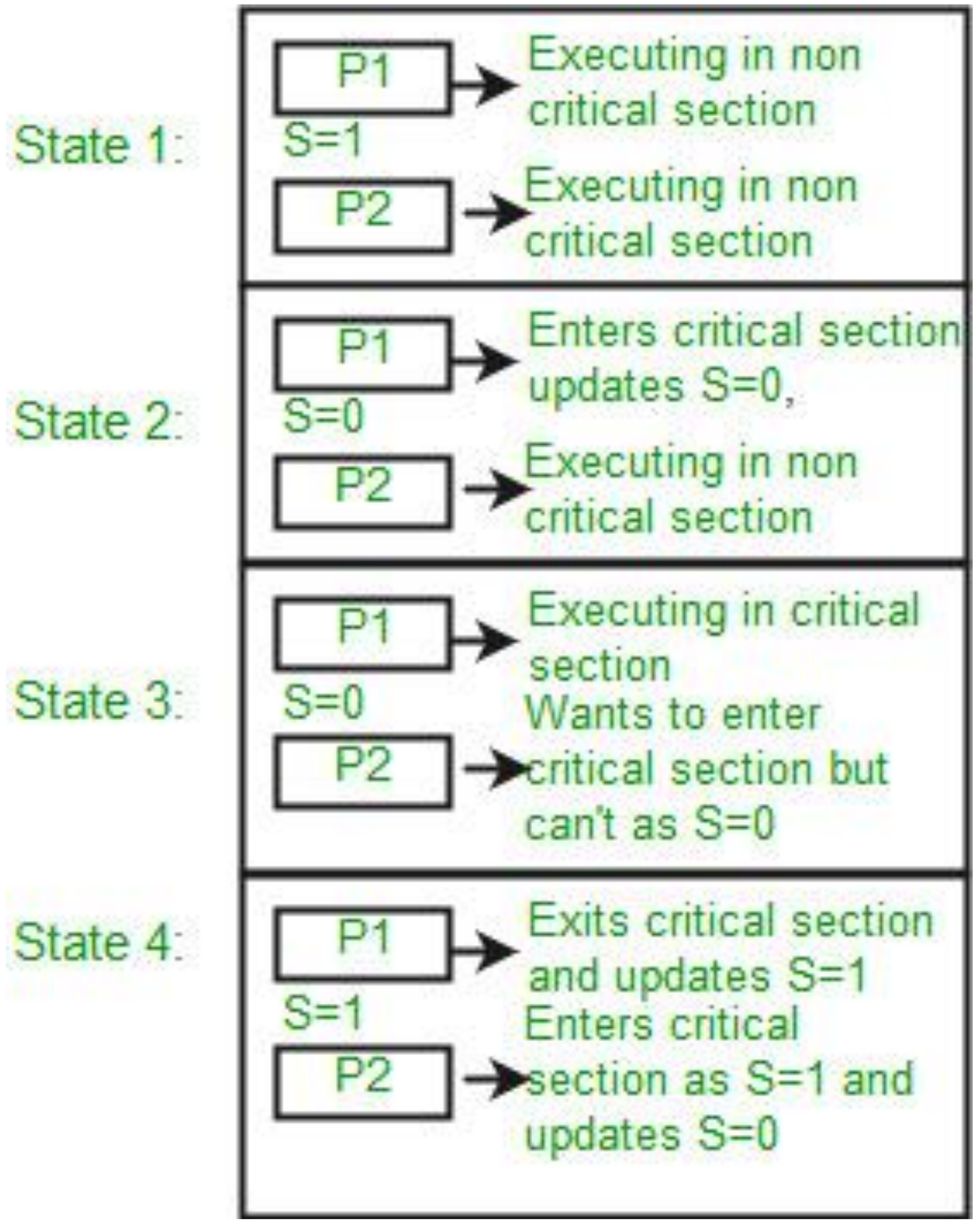


Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Mutual Exclusion

- Let there be two processes P1 and P2 and a semaphore s is initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0.
- Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
- This way mutual exclusion is achieved.


State 1:
P1
Executing in non critical section
S=1
P2
Executing in non critical section
State 2:
P1
Enters critical section updates S=0,
S=0
P2
Executing in non critical section
State 3:
P1
Executing in critical section
S=0
P2
Wants to enter critical section but can't as S=0
State 4:
P1
Exits critical section and updates S=1
S=1
P2
Enters critical section as S=1 and updates S=0

Mutual Exclusion through Binary semaphore
POSIX standard solutions:
semwait()
sempost()

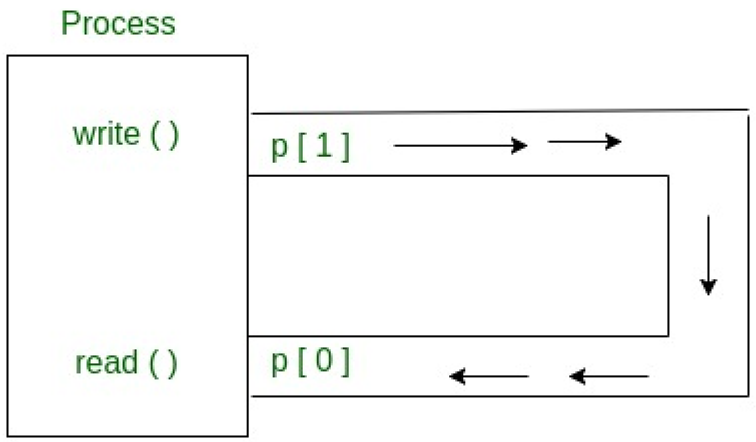
Mutex

- **Mutex** is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.
- When a program is started a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource.
- The mutex is set to unlock when the data is no longer needed or the routine is finished.
- **POSIX standard solutions:**

  pthread_mutex_lock() and pthread_mutex_unlock()

PIPE

- Connection between two processes, such that the standard output from one process becomes the standard input of the other process.
- Pipes are useful for communication between related processes(inter-process communication).
- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a ***"virtual file".***



PIPE (Unnamed pipe)

int pipe(int fds[2]);

Parameters :

fd[0] will be the fd(file descriptor) for the

read end of pipe.

fd[1] will be the fd for the write end of pipe.

Returns : 0 on Success.

-1 on error.

Named pipe (FIFO)

- Extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Syntax:

```
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with name ***pathname***. Here ***mode*** specifies the FIFO’s permissions.