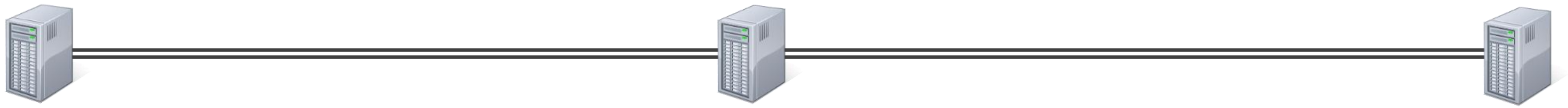
Process Synchronization

(Operating System)



Deepika H V

C-DAC Bengaluru

(deepikahv@cdac.in)

- **Process**
  – Independent Process : It cannot affect or be effected by the other processes executing in the system ;Does not share any data with any other processes
  – Cooperating Process: It can affect or be affected by the other processes executing in the system.
- **Cooperating Process**
  – Share variable, memory, buffer, code, resources
- **Synchronization**
  – Several processes access and manipulate the same data **concurrently**
  – Outcome of the execution depends on the particular **order** in which the access takes place
  – **data consistency** is must in co-operating processes

- **What is Synchronization?**
  - Process Synchronization means **sharing system resources by processes** in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.
  - Maintaining **data consistency demands mechanisms** to ensure synchronized execution of cooperating processes.

Producer Consumer Problem

- 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()

Producer

```
int count = 0;
Producer() {
while (TRUE)
        produce item();
        if (count == MAX_SIZE)   sleep();
        enter item();
        count ++;
        if (count == 1) wakeup(Consumer);
}
```

register1 = count
register1 = register1 + 1
count = register1

Consumer

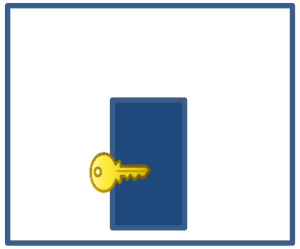
```
Consumer() {
while(TRUE)
        if (count == 0) sleep();
        remove item();
        count --;
        if (count == MAX_SIZE - 1)  wakeup(Producer);
        consume item();
}
```

- **Race Condition**
  - It is an undesirable situation which occurs when 2 computer program processes/threads attempt to access the same resource at the same time and cause problems in the system.
  - A race condition can be difficult to reproduce and debug because the end result is nondeterministic and depends on the relative timing between interfering threads.
- **Critical Section**
  - Critical Section is the part of a program which tries to access shared resources.
  - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder** section

- **The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise**
- In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.
  - **Primary conditions :**
    - Mutual Exclusion : one process is executing inside critical section then the other process must not enter in the critical section
    - Progress : one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.
  - **Secondary conditions** :
    - Bounded Waiting : able to predict the waiting time for every process to get into the critical section.
    - Architectural Neutrality : our solution is working fine on one architecture then it should also run on the other ones as well.

CSP Methods - Mutex

- **Simplest is mutex lock**
- **Product critical regions with it by first `acquire()` a lock then `release()` it**
  - Boolean variable indicating if lock is available or not
- **Calls to `acquire()` and `release()` must be atomic**
  - Usually implemented via hardware atomic instructions
- **But this solution requires busy waiting**
- This lock therefore called a **spinlock**



```
acquire() {
   while (!available); /* busy wait */
   available = false;;
}
```

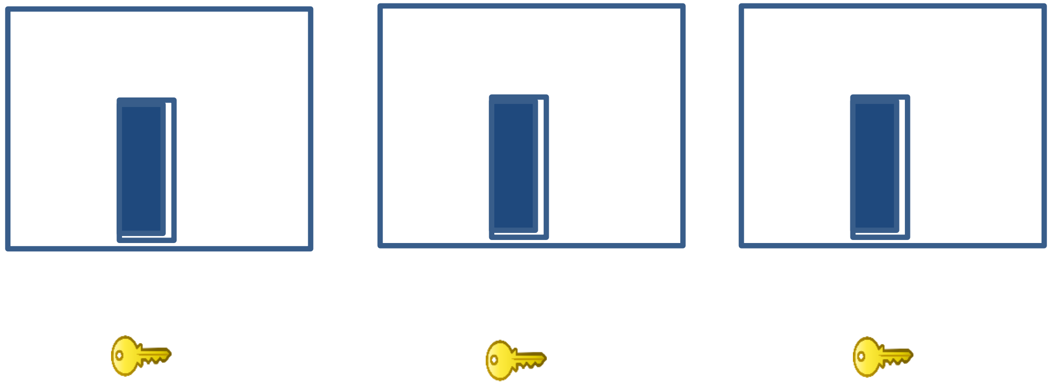
```
do{
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

```
release() {
   available = true;
}
```

CSP Methods - Semaphore

- Integer variable used in a mutual exclusive manner by various cooperating processes in order to achieve synchronization.
- tool that does not require busy waiting
- Semaphores are like integers, except NO negative values
- Accessed only through two standard operations **P()/acquire()/wait()/down() and V()/release()/ signal()/up()**.

```
Semaphore s; // initialized to no of instances of the CS
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```



- **Two Types**
  - Counting Semaphores: value can range over an unrestricted domain; varies from

-∞ to +∞. Initialize to number of CS.

  - Binary Semaphores: integer value can range only between 0 and 1; also known as mutex locks. Initialize to 1

```
Semaphore s; // initialized to no of instances of the CS
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```

```
Down(Semaphore S){
        S.value= S.value -1;
        if(S.value<0){
                put process(PCB) in suspended list sleep();
        }
        else{
                return ;
        }
}
```

```
Up(Semaphore S){
        S.value = S.value +1;
        if(S.value <=0){
                select
process from sleep list
                wakeup();
        }
}
```

Binary Sempahore

```
Down(Semaphore S){
        if(S.value ==1){
                S.value=0;
        }
        else{
                Block this process and put in suspended list
                sleep() ;
        }
}
```

```
Up(Semaphore S){
    if( suspended list is Empty){
            S.value=1;
    }else{
            select process from sleep list
            wakeup();
    }
}
```

```
Semaphore s; // initialized to 1
do {
        down(s);
        // Critical Section
        up(s);
        // remainder section
} while (TRUE);
```

Producer Consumer problem

DEADLOCK	STARVATION
A situation that occurs with a set of processes in which every process is waiting for an event that can only be caused by another process in the set	A situation in which a process is perpetually denied necessary resources to process its work
Occurs when each process is holding a resource and waits for a resource held by another process	Occurs when a process waits for a resource for a long period of time
All the related processes cannot proceed	Some processes wait for resources but others can proceed

Thank You