

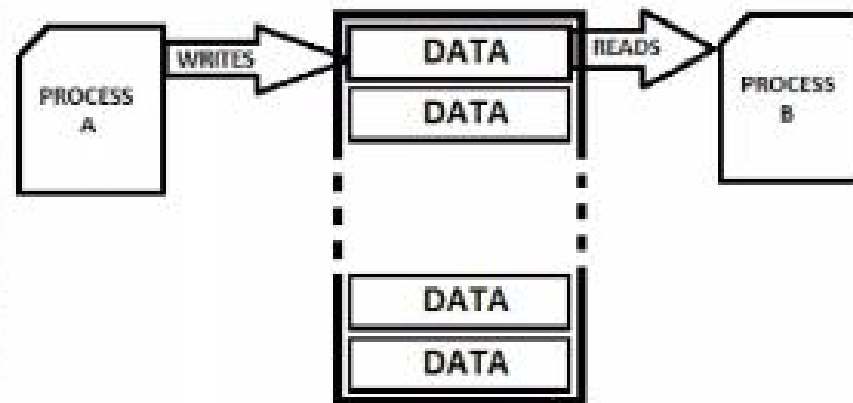
# Process Synchronization

Dr Raghunath Dey



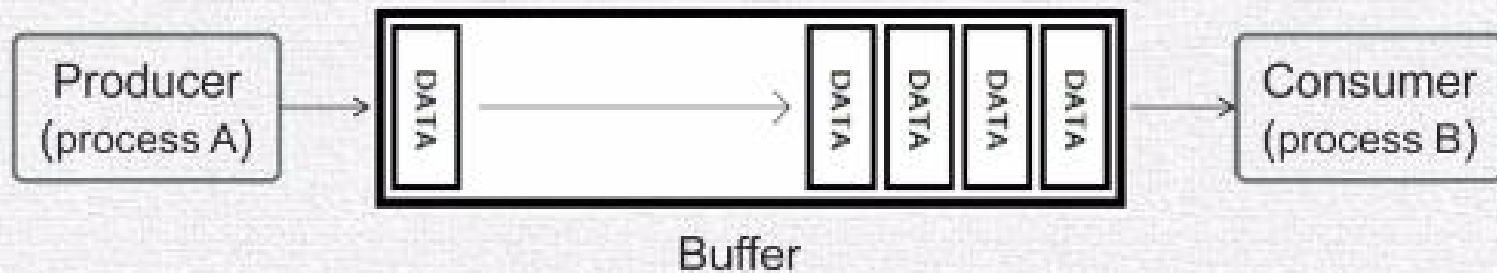
# WHAT IS PROCESS SYNCHRONIZATION?

- Several Processes run in an Operating System
- Some of them share resources due to which problems like data inconsistency may arise
- For Example: One process changing the data in a memory location where another process is trying to read the data from the same memory location. It is possible that the data read by the second process will be erroneous

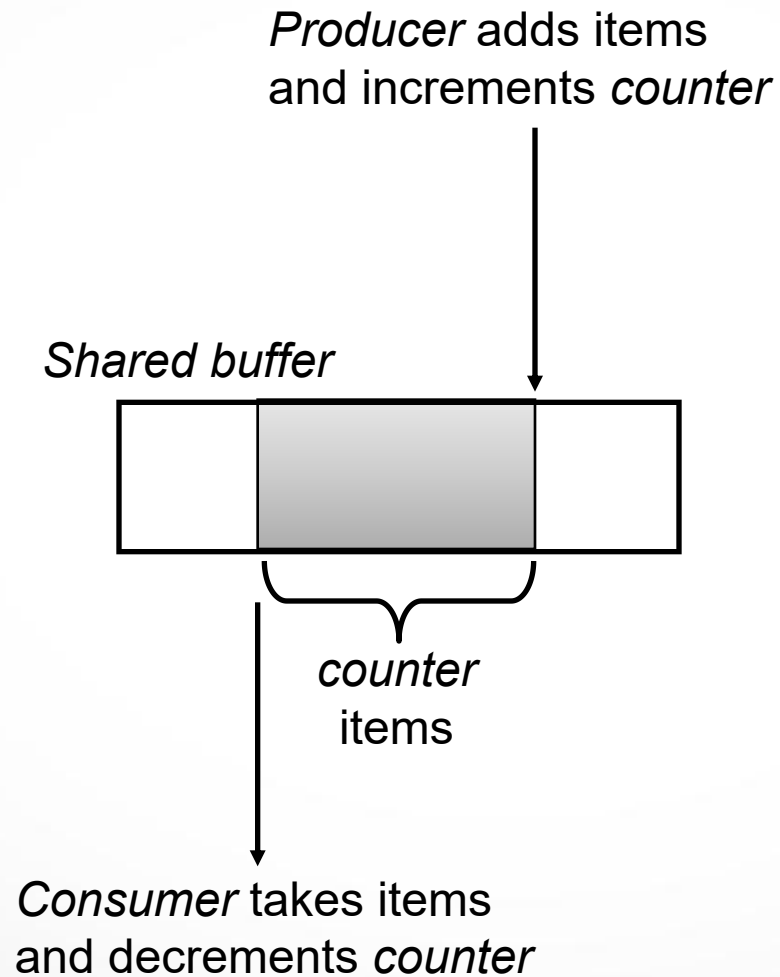


# WHAT IS PROCESS SYNCHRONIZATION?

- **PRODUCER CONSUMER PROBLEM**  
(or Bounded-Buffer Problem)



- Problem: To ensure that the **Producer** should not add DATA when the Buffer is *full* and the **Consumer** should not take data when the Buffer is *empty*



# WHAT IS PROCESS SYNCHRONIZATION?

- **SOLUTION PRODUCER CONSUMER PROBLEM**
- **Using Semaphores** (In computer science, particularly in operating systems, a **semaphore** is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system.)
- **Using Monitors** (In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true)
- **Atomic Transactions** (Atomic read-modify-write access to shared variables is avoided, as each of the two Count variables is updated only by a single thread. Also, these variables stay incremented all the time; the relation remains correct when their values wrap around on an integer overflow)

# RACE CONDITION

**Producer:**

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

**Consumer:**

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

# CRITICAL SECTION PROBLEM

- **A section of code, common to  $n$  cooperating processes, in which the processes may be accessing common variables.**
- A Critical Section Environment contains:
- **Entry Section** Code requesting entry into the critical section.
- **Critical Section** Code in which only one process can execute at any one time.
- **Exit Section** The end of the critical section, releasing or allowing others in.
- **Remainder Section** Rest of the code AFTER the critical section.

# CRITICAL SECTION PROBLEM

- The critical section must **ENFORCE ALL THREE** of the following rules:

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **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
3. **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





# CRITICAL SECTION PROBLEM

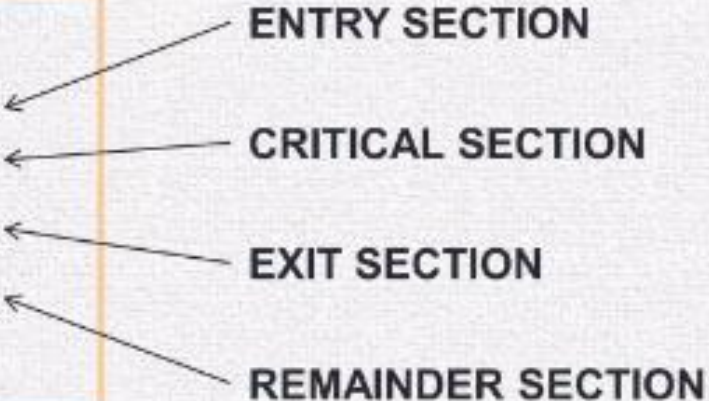
```
do {  
    while ( turn ^= i );  
    /* critical section */  
    turn = j;  
    /* remainder section */  
} while(TRUE);
```

ENTRY SECTION

CRITICAL SECTION

EXIT SECTION

REMAINDER SECTION



# PETERSON'S SOLUTION

- To handle the problem of Critical Section (CS), Peterson gave an algorithm with a bounded waiting
- Suppose there are  $N$  processes ( $P_1, P_2, \dots, P_N$ ) and each of them at some point need to enter the Critical Section
- A `FLAG[]` array of size  $N$  is maintained which is by default false and whenever a process need to enter the critical section it has to set its flag as true, i.e. suppose  $P_i$  wants to enter so it will set `FLAG[i]=TRUE`
- There is another variable called `TURN` which indicates the process number which is currently to enter into the CS. The process that enters into the CS while exiting would change the `TURN` to another number from among the list of ready processes

# Peterson's Solution



- Good algorithmic description of solving the problem
- Two process solution
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process  $P_i$  is ready!

# Software approach- Peterson's Solution



```
do {
```

```
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = FALSE;
```

remainder section

```
} while (TRUE);
```

# Peterson's Solution

do {

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

} while (TRUE);

EXAMPLE 1	
Process 0	Process 1
i = 0, j = 1	i = 1, j = 0
flag[0] := TRUE turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is false because flag[1] = FALSE - Since condition is false, no waiting in while loop - Enter the critical section - <del>Process 0 happens to lose the processor</del>	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Since condition is true, it keeps busy waiting until it loses the processor
- Process 0 resumes and continues until it finishes in the critical section - Leave critical section flag[0] := FALSE - Start executing the remainder (anything else a process does besides using the critical section) - Process 0 happens to lose the processor	
	check (flag[0] = TRUE and turn = 0) - This condition fails because flag[0] = FALSE - No more busy waiting - Enter the critical section



# Synchronization Hardware

- Problems of Critical Section are also solvable by hardware
- Uniprocessor systems disables interrupts while a Process  $P_i$  is using the CS but it is a great disadvantage in multiprocessor systems
- Some systems provide a lock functionality where a Process acquires a lock while entering the CS and releases the lock after leaving it. Thus another process trying to enter CS cannot enter as the entry is locked. It can only do so if it is free by acquiring the lock itself
- Another advanced approach is the **Atomic Instructions** (Non-Interruptible instructions).

# hardware solutions

- There are three algorithms in the hardware approach of solving Process Synchronization problem:
  - Test and Set
  - Swap
  - Unlock and Lock



# Test and Set:

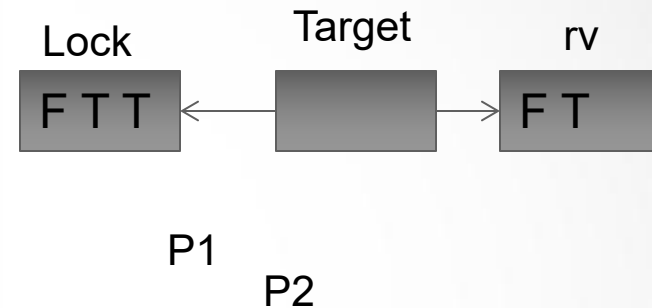
- Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured. Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. **So bounded waiting is not ensured.**







```
do {  
    while (test_and_set(&lock));  
        /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```



Definition:

```
boolean test_and_set (boolean *target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

# Swap





```
while(1){  
    key=true;  
    while(key){  
        swap(lock,key);  
    }  
  
    CRITICAL SECTION CODE  
    lock = false;  
    REMAINDER SECTION CODE  
}
```

**NB: Initial values of lock and key is false**



- In the code above when a process P1 enters the critical section of the program it first executes the while loop.
- As key value is set to true just before the for loop so swap(lock, key) swaps the value of lock and key. Lock becomes true and the key becomes false. In the next iteration of the while loop breaks and the process, P1 enters the critical section.
- The value of lock and key when P1 enters the critical section is lock = true and key = false.

- 
- Let's say another process, P2, tries to enter the critical section while P1 is in the critical section. Let's take a look at what happens if P2 tries to enter the critical section.
  - **key is set to true** again after the first while loop is executed i.e while(1). Now, the second while loop in the program i.e while(key) is checked. As key is true the process enters the second while loop. swap(lock, key) is executed again. as both key and lock are true so after swapping also both will be true. So, the while keeps executing and the process P2 keeps running the while loop until Process P1 comes out of the critical section and makes **lock false**.

- 
- When Process P1 comes out of the critical section the value of lock is again set to false so that other processes can now enter the critical section.
  - When a process is inside the critical section than all other incoming process trying to enter the critical section is not maintained in any order or queue. So any process out of all the waiting process can get the chance to enter the critical section as the lock becomes false. So, there may be a process that may wait indefinitely. So, **bounded waiting is not ensured in Swap algorithm also.**

# Unlock and lock



- ***Unlock and lock algorithm*** uses the TestAndSet method to control the value of lock. Unlock and lock algorithm uses a variable `waiting[i]` for each process `i`. Here `i` is a positive integer i.e 1,2,3,... which corresponds to processes `P1`, `P2`, `P3`... and so on. **`waiting[i]` checks** if the process `i` is waiting or not to enter into the critical section.
- All the processes are maintained in a ready queue before entering into the critical section. The processes are added to the queue with respect to their process number. The queue is the circular queue.

# What we learned

- Process Synchronization problem is solved by Hardware locks.
- The three Hardware lock algorithms are **Test and Set, Swap, Unlock and lock Algorithm.**
- Test and set algorithm uses a boolean variable lock to regulate mutual exclusion of processes.
- Swap algorithm uses two boolean variable lock and key to regulate mutual exclusion of processes.
- Unlock and lock algorithm uses three boolean variable lock, key, and waiting[i] for each process to regulate mutual exclusion.
- Unlock and lock algorithm maintains a ready queue for the waiting and incoming processes and the algorithm checks the queue for the next process j when the first process i comes out of the critical section.
- **Unlock and lock algorithm ensures bounded waiting.**



# MUTEX LOCKS

- As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.
- As the resource is locked while a process executes its critical section hence no other process can access it





- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - 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() and release()



- ```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}
```
- ```
release() {  
    available = true;  
}
```
- ```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

# Semaphore



- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```

# Semaphore Usage



- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$   
Create a semaphore “**synch**” initialized to 0  
P1:  
     $S_1$ ;  
    **signal**(**synch**) ;  
P2:  
    **wait**(**synch**);  
     $S_2$ ;
- Can implement a counting semaphore  $S$  as a binary semaphore

# Deadlock and Starvation



- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $s$  and  $q$  be two semaphores initialized to

$P_0$

```
wait(S);  
wait(Q);  
...  
signal(S);  
signal(Q);
```

$P_1$

```
wait(Q);  
wait(S);  
...  
signal(Q);  
signal(S);
```

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

# Classical Problems of Synchronization



- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Bounded-Buffer Problem



- $n$  buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
  - (a binary semaphore which is used to acquire and release lock)
- Semaphore `full` initialized to the value 0
  - (a counting semaphore whose initial value is 0)
- Semaphore `empty` initialized to the value  $n$ 
  - (a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty)

# Bounded Buffer Problem (Cont.)



- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty); //wait while empty<=0 else empty--  
    wait(mutex); //acquire lock  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex); //release lock  
    signal(full); //increment full  
} while (true);
```



# Bounded Buffer Problem (Cont.)



- The structure of the consumer process

```
Do {  
    wait(full); //wait while full<=0 else decrement full  
    wait(mutex); //acquire lock  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex); //release lock  
    signal(empty); //increment empty  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```

# 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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

# Readers-Writers Problem (Cont.)



- The structure of a writer process

```
do {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
} while (true);
```

# Readers-Writers Problem (Cont.)



- The structure of a reader process

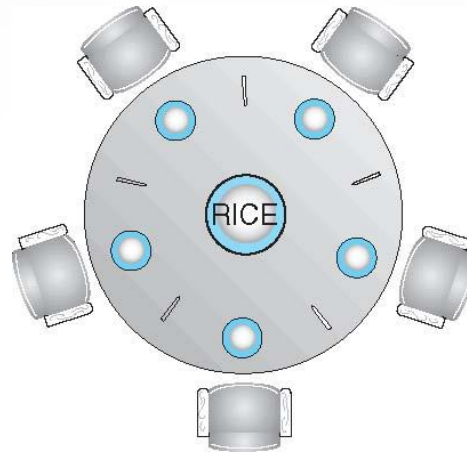
```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```

# Readers-Writers Problem Variations



- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore **chopstick** [5] initialized to 1

# Dining-Philosophers Problem Algorithm



- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
    // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    // think  
  
} while (TRUE);
```

- What is the problem with this algorithm?

## Dining-Philosophers Problem Algorithm (Cont.)



- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left.



# Problems with Semaphores



- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are

# Monitors



- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
}
```

