

CT 656

OPERATING SYSTEM

LECTURE – 3

Process Communication and Synchronization

WHERE ARE WE ?

1. Introduction
2. Process Management
3. Process Communication and Synchronizing
4. Memory Management
5. File Systems
6. I/O Management and Disk Scheduling
7. Deadlock
8. Security
9. System Administration

Introduction

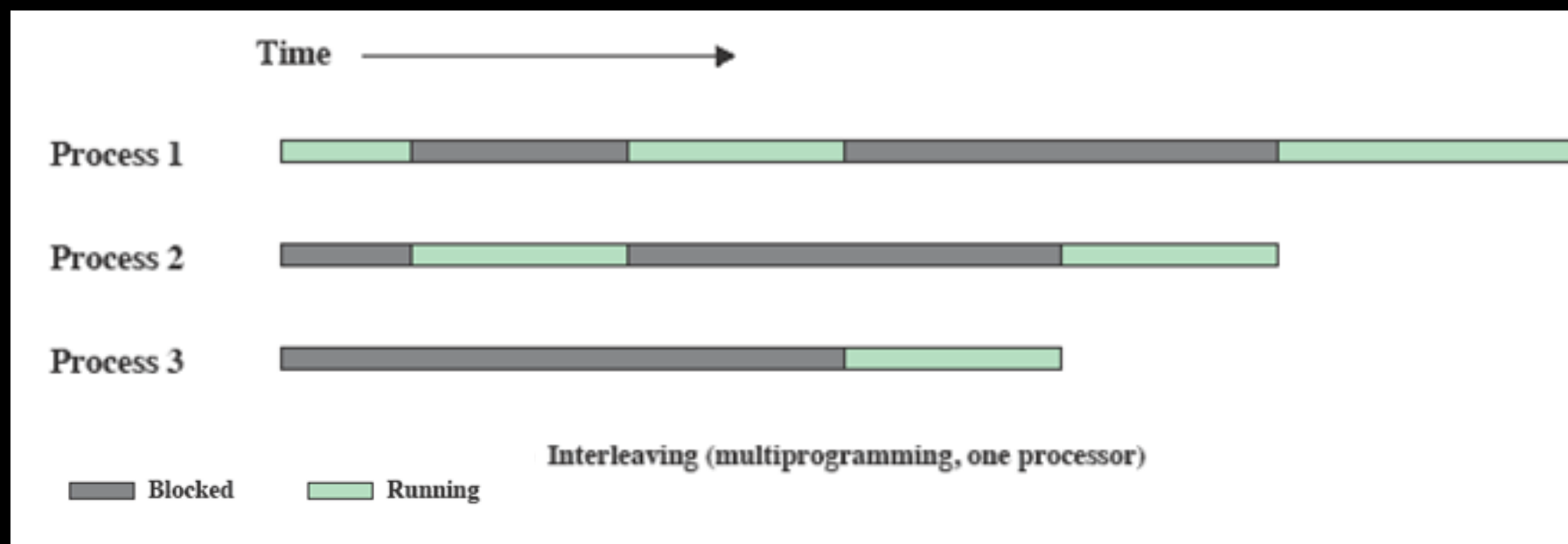
- ❖ Concurrency refers to **simultaneous execution of multiple processes**
- ❖ Central to the design of modern Operating Systems is **managing multiple concurrent processes** in the system
- ❖ Multiprogramming, Multiprocessing and Distributed Processing is not possible without proper **communication and synchronization** of concurrent processes

Principle of Concurrency

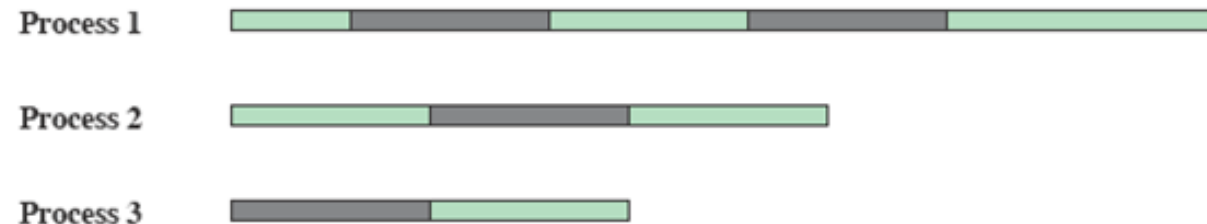
- ❖ Processes are concurrent if they exist at the same time
- ❖ Concurrent processes are capable of **functioning** independent of each other
- ❖ Relative speed of the execution of the processes is not predictable. It depends on activities of other processes, the way in which OS handles interrupts, and scheduling policies of OS.
- ❖ System interrupts are not predictable

Interleaving and Overlapping

- ❖ **Interleaving** and **overlapping** are two different concurrent processing
- ❖ we saw that processes may be interleaved on **uniprocessors**
- ❖ And not only interleaved but overlapped on **multi-processors**



Interleaving and Overlapping



(b) Interleaving and overlapping (multiprocessing; two processors)

Blocked Running

Concurrency Problems

❖ Concurrency has following difficulties:

1. Sharing of **global variables**
2. Optimal **allocation of system resources**
3. Difficult **to locate programming error**

Concurrency Problem Example

❖ Consider the following procedure shared by many processes

```
void echo(){  
    chin = getchar();  
    chout = chin;  
    putchar(chout);  
}
```

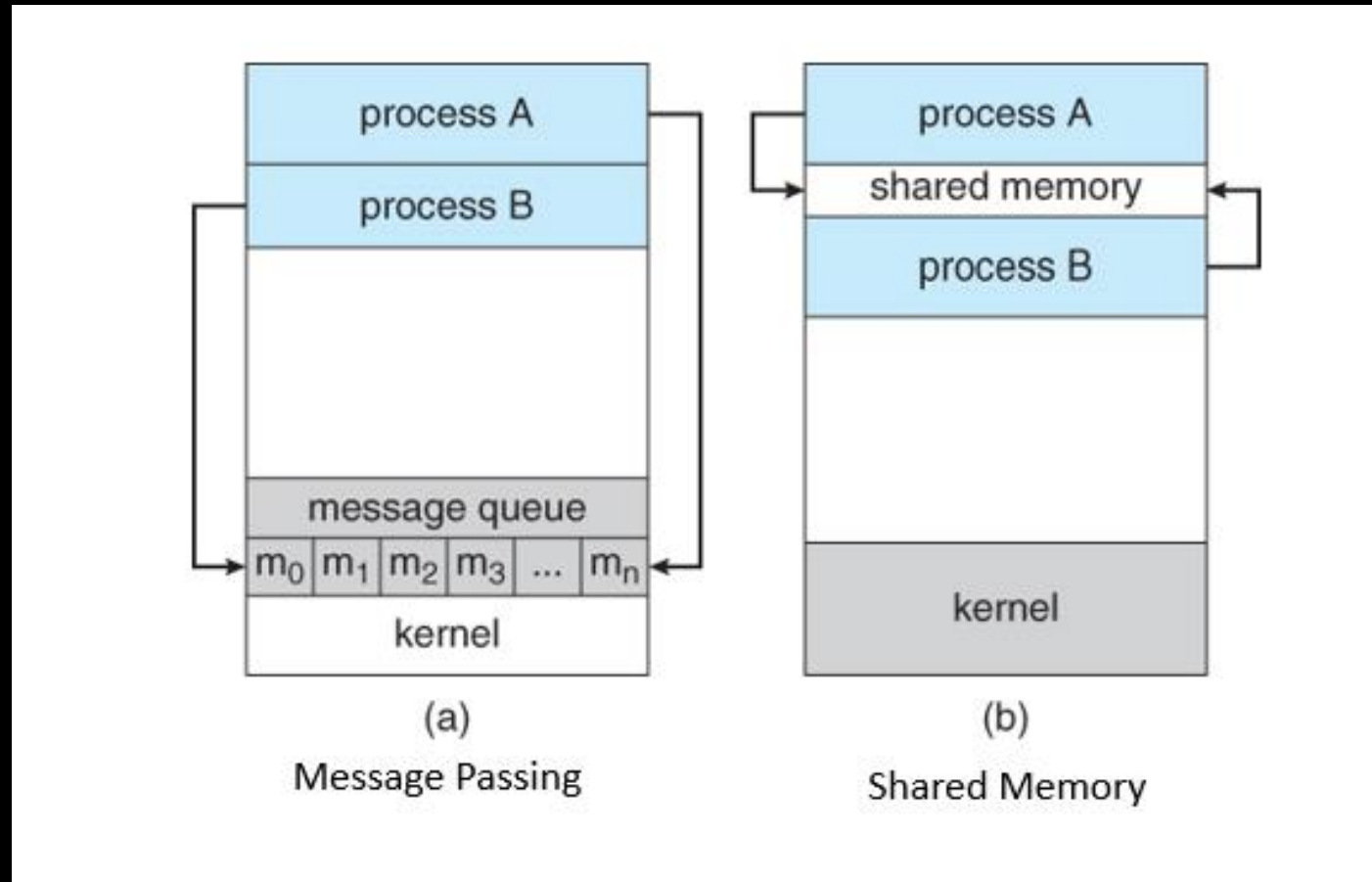

Concurrency Problem Example

- ❖ Since many application would want to invoke **echo** procedure, **echo** is a shared procedure that is loaded into a portion of **memory global to all application**.
- ❖ Consider the following sequence:
 - ❖ Process P1 invokes the **echo** procedure and is interrupted immediately after **getchar()** returns its value and stores in **chin**. Say 'x' is stored in **chin**
 - ❖ Process P2 is activated and invokes the **echo** procedure which runs to conclusion, inputting and displaying a single character on the screen say 'y'
 - ❖ Process P1 is resumed. But, the value 'x' has been overwritten in **chin** by 'y' and therefore the original value for P1 has been lost. Thus, P1 also displays 'y'
- ❖ This problem shows that **we need some synchronization scheme** when we are accessing shared variables and resources

Inter Process Communication (IPC)

- ❖ Cooperating processes frequently need to communicate with each other.
- ❖ Communication between processes is concerned with three main issues:
 1. Passing information from one process to another
 2. Making sure that two or more processes do not get in each other's way
 3. Proper sequencing
- ❖ IPC models:
 1. Shared Memory (Semaphore, Monitor etc.)
 2. Message Passing

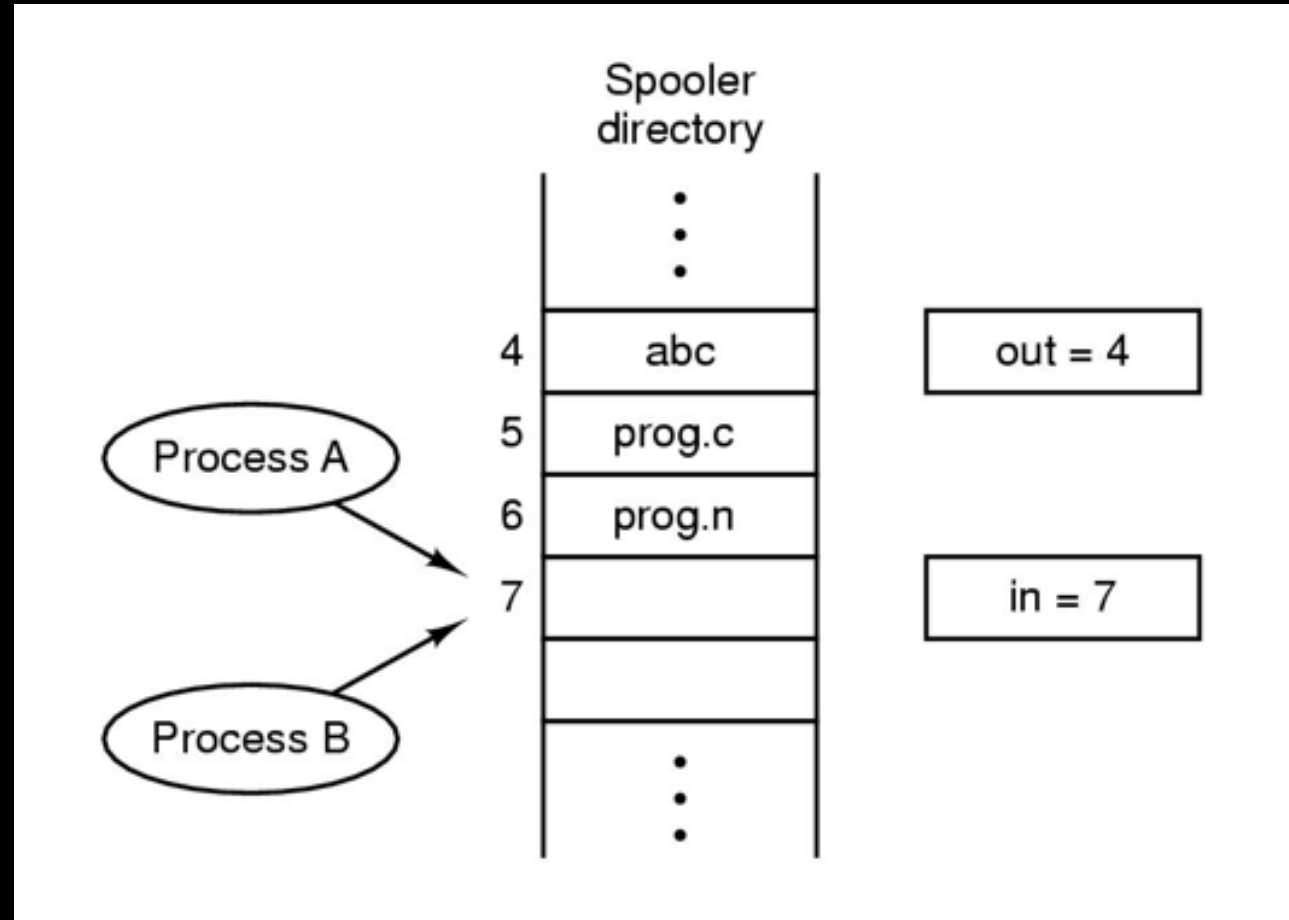
Inter Process Communication (IPC)



Race Condition

- ❖ A situation where several processes/threads access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**
- ❖ For example:
 - P1 and P2 share global variables b and c with initial values $b=1$ and $c=2$
 - P1 has an assignment operation, $b=b+c$
 - P2 has an assignment operation $c=b+c$
 - If P1 and P2 are running concurrently, value of b and c would be 3 and 5 if P1 executes its assignment first or it could be 4 and 3 if P2 executes its assignment first

Race Condition: Printer Spooler



Race Condition: Printer Spooler

- ❖ there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory
- ❖ At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing)
- ❖ More or less simultaneously, processes A and B decide they want to queue a file for printing
- ❖ Process A reads *in* and stores the value, 7, in a local variable called *next-free slot*
- ❖ Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- ❖ Process B also reads *in*, and also gets a 7. It too stores it in its local variable *next-free slot*.
- ❖ At this instant both processes think that the next available slot is 7

Race Condition: Printer Spooler

- ❖ Process B now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things
- ❖ Eventually, process A runs again, starting from the place it left off.
- ❖ It looks at *next free slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.
- ❖ Then it computes *next free slot + 1*, which is 8, and sets *in* to 8
- ❖ The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output
- ❖ Debugging programs containing **race conditions** is no fun at all.
- ❖ The results of most test runs are fine, but once in a rare while something weird and unexplained happens

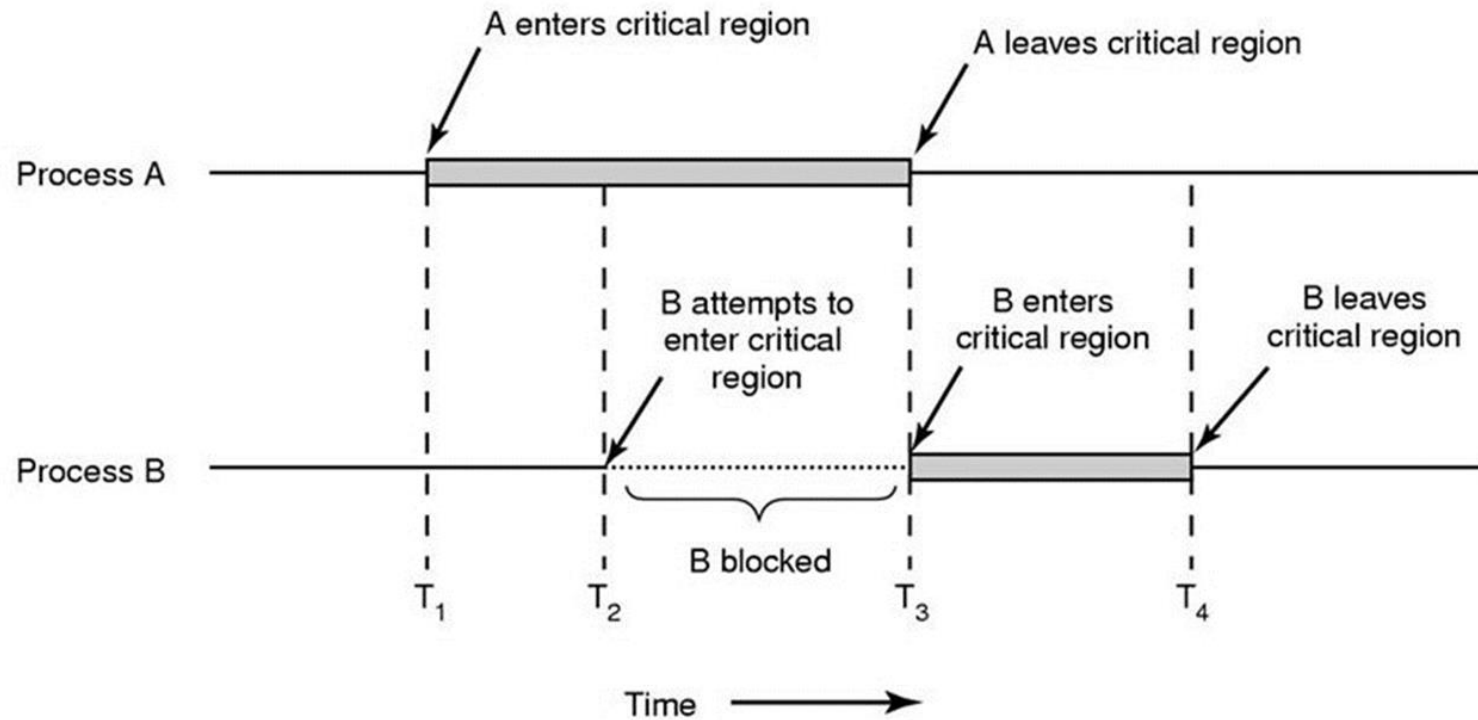
How to avoid race condition?

- ❖ To guard against the race conditions, we need to synchronize the processes in such a way that only one process at a time can be manipulating shared variable or file.
- ❖ This is called **mutual exclusion, ME**
- ❖ In the print spooler example, shared variable is *in*. If process B had not started using the shared variable before process A was finished with it, we could have avoided user of process B from not getting his/her output printed.

Critical Region/Section

- ❖ The part of the program **where shared memory is accessed** is called critical region
- ❖ We could arrange matters such that **when one process is executing in its critical section, no other process is to be allowed to execute in its critical section**. This would solve race conditions.
- ❖ **Mutual exclusion conditions:**
 1. No two processes may be simultaneously inside their critical region
 2. No assumptions may be made about the speeds or the number of CPUs
 3. **No process running outside its critical region may block other processes from entering the CS**
 4. No process should have to wait forever to enter its critical region

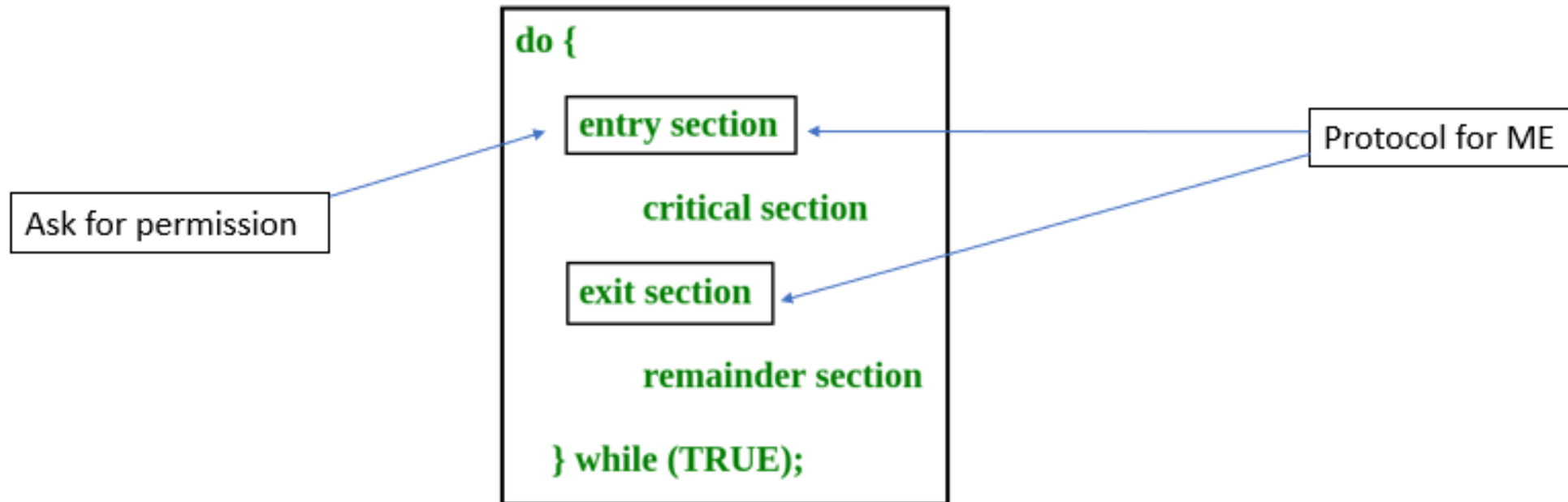
Mutual Exclusion (ME)



Critical section problem

- ❖ The **critical-section problem** is to design a **protocol** that the processes can use to cooperate and **avoid race conditions**
- ❖ Each process **must request permission to enter its critical section**. The section of code implementing this request is the **entry section**.
- ❖ The **critical section** may be followed by an **exit section**.
- ❖ The remaining code is the **remainder section**.

General Structure of Typical Process



Solution to critical section problem

❖ A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion

- Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

- If no process is in its critical section, and if one or more process want to execute their critical section then only those processes that are not executing in their remainder sections can participate in the decision on which will critical section, and the selection cannot be postponed indefinitely. In short, **a process which doesn't want to enter in the critical section should not stop other processes to get into it**

3. Bounded Waiting (no starvation)

- After a process makes a request for getting into its critical section, **there is a limit for how many other processes can get into their critical section, before this process's request is granted**. So after the limit is reached, system must grant the process permission to get into its critical section

Mutual exclusion with busy waiting

- ❖ We will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble
 1. Disabling interrupts
 2. Mutex Lock
 3. Strict Alternation
 4. Peterson's solution
 5. The TSL Instruction

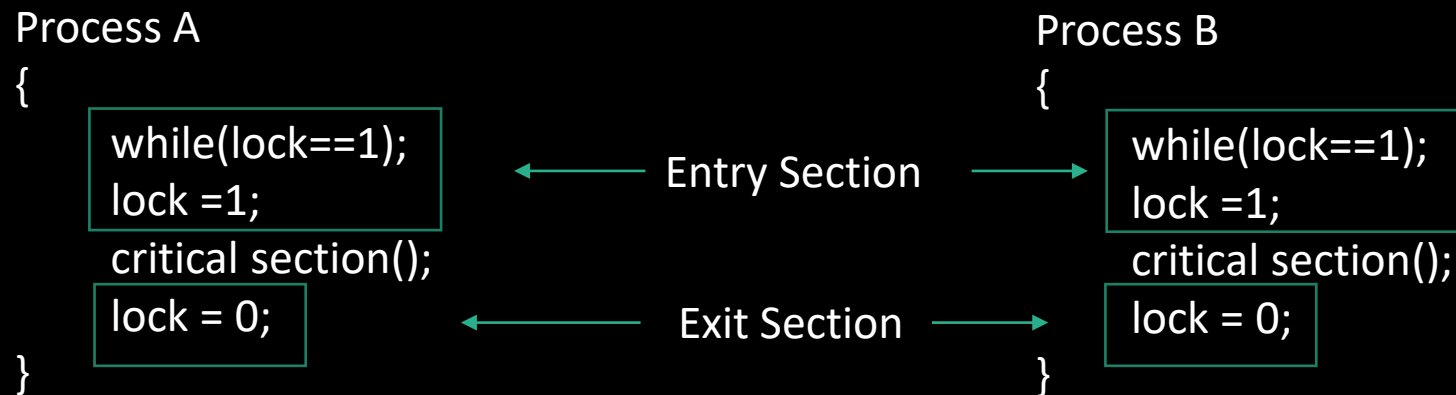
1. Disabling Interrupt

- ❖ Each process **disable its interrupts just after entering its critical section**
- ❖ Process does its work in the critical section
- ❖ Process re-enables the interrupts just before leaving its critical section
- ❖ **Problem??**
- ❖ This approach is **unattractive** as it is **unwise to give user processes the power to turn off interrupts.**

2. Mutex Lock

- ❖ In this approach, we will have a single shared variable called **lock**
- ❖ If $\text{lock}=0$, it means that no process is in the critical region while $\text{lock}=1$ means that a process is currently working in the critical region
- ❖ When a process wants to enter into its critical section, it first tests the **lock**
- ❖ If the lock is 0, the process sets it to 1 and enters the critical section
- ❖ If the lock is already 1, the process just waits until it becomes zero

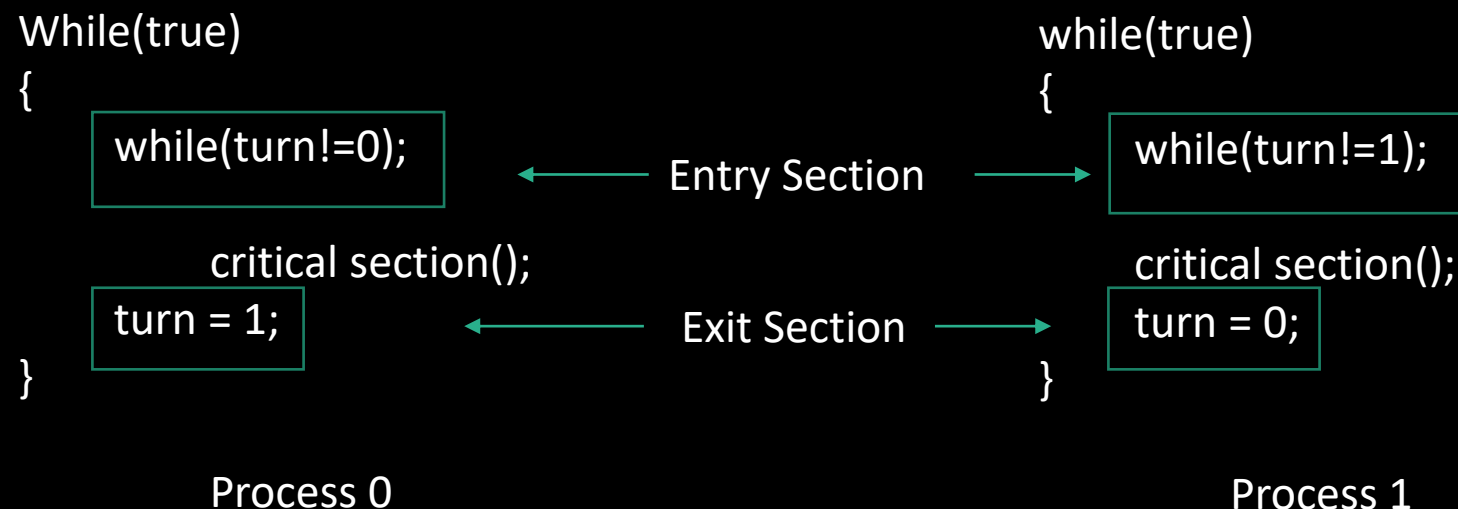
Mutex Lock (Contd...)



- What is the problem with this approach ??
 - two process may be in critical region at the same time
 - a process sees lock = 0 and enters critical region, but before it can set lock to 1, another process is scheduled, it sees lock=0 and enters critical region. Both process enter critical section, both assigning 1 to lock

3. Strict Alternation

- ❖ In this approach, we have an integer variable named **turn**, which keeps track of whose turn is it to enter critical section
- ❖ Say, in a two process scenario, Process 0 runs when $\text{turn}=0$ and process 1 runs when $\text{turn}=1$



Strict Alternation (Contd...)

- ❖ Turn is initially set to 0
- ❖ Process 0 examines turn to be 0 and enters the critical region.
- ❖ Process 1 also finds turn to be 0 so it sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called busy waiting
- ❖ When process 0 leaves the critical region, it sets turn to 1 so that process 1 can now enter the critical region.
- ❖ Problem??
 - violates the condition number 3 in slide no. 15
 - if process 0 finishes its work in non critical region fast enough after it set turn to 1 and process 0 needs to enter critical section again before process 1 then it cannot do so. So, process 1 which is running outside critical region is blocking process 0

4. Peterson's Algorithm

- ❖ is a solution to critical section problem
- ❖ Peterson's solution is **restricted to two processes** that alternate execution between their critical sections and remainder sections.
- ❖ Uses two shared variables : **turn** and **flag**

```
int turn;  
boolean flag [2];
```

- ❖ The variable **turn** indicates **whose turn it is to enter its critical section**. That is, if $turn == i$, then process P_i is allowed to execute in its critical section.
- ❖ The **flag array** is used to indicate if a process is ready to enter its critical section. For example, if **flag[i] is true, this value indicates that P_i is ready to enter its critical section.**

Peterson's Algorithm (Contd...)

- ❖ To enter the critical section, process P_i first sets $flag[i]$ to be true and then sets $turn$ to the value j , thereby asserting that if the other process wishes to enter the critical section, it can do so.
- ❖ If both processes try to enter at the same time, $turn$ will be set to both i and j at roughly the same time.
- ❖ Only one of these assignments will last; the other will occur but will be overwritten immediately.
- ❖ The eventual value of $turn$ decides which of the two processes is allowed to enter its critical section first.

Peterson's Algorithm

do{

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

critical section

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

remainder section

}while(TRUE);

Process P_i

do{

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

critical section

```
flag [j] = FALSE;
```

remainder section

}while(TRUE);

Process P_j

Correctness of Peterson's Algorithm

❖ To show that Peterson's Algorithm is correct, we need to prove that using this algorithm:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

Correctness of Peterson's solution

❖ Mutual exclusion

- ❖ If one process is executing their critical section when the other wishes to do so, **the second process will become blocked by the flag of the first process.**
- ❖ If both processes attempt to enter at the same time, the last process to execute "turn = j or i" will be blocked.

❖ Progress

- ❖ Each process can only be blocked at the **while** if the other process **wants** to use the critical section ($\text{flag}[j] == \text{true}$), **AND it is the other process's turn** to use the critical section ($\text{turn} == j$).
- ❖ If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set $\text{flag}[j]$ to false, releasing process i.
- ❖ The shared variable **turn assures** that **only one process at a time can be blocked**, and the **flag variable allows one process to release the other when exiting their critical section.**

Correctness of Peterson's solution

❖ Bounded Waiting

- ❖ As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

5. TSL Instruction

- ❖ This proposal to solve critical section problem requires a little help from hardware, it uses the following instruction

TSL Rx, LOCK

- ❖ It reads the content of memory word lock into register RX and stores a non zero value at the memory address lock
- ❖ This is an atomic instruction, that is operation of reading a word and storing into it are guaranteed to be indivisible
- ❖ When lock is 0, any process may set it to 1 through TSL instruction and read or write the shared memory
- ❖ When it is done, the process sets lock back to 0 using an ordinary MOVE instruction

TSL Instruction (Contd..)

enter_region:

TSL REGISTER,LOCK

CMP REGISTER,#0

JNE enter_region

RET

| copy lock to register and set lock to 1
| was lock zero?
| if it was nonzero, lock was set, so loop
| return to caller; critical region entered

leave_region:

MOVE LOCK,#0

RET

| store a 0 in lock
| return to caller

TSL Instruction (Contd..)

❖ Entering and exiting Critical Region using XCHG instruction

```
enter_region:
    MOVE REGISTER,#1          | put a 1 in the register
    XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
    CMP REGISTER,#0           | was lock zero?
    JNE enter_region          | if it was non zero, lock was set, so loop
    RET                       | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0              | store a 0 in lock
    RET                       | return to caller
```

❖ All Intel x86 CPUs use XCHG instruction for low-level synchronization

Priority Inversion Problem

- ❖ Consider two process

 - H = higher priority process

 - L = low priority process

- ❖ Scheduling rules say that H must run whenever it is in ready state

- ❖ Consider L is in critical region, H becomes ready to run.

- ❖ So, H now begins busy waiting

- ❖ But, since L never gets scheduled while H is running, L never gets chance to leave the critical section, so, H loops forever.

- ❖ This situation is sometimes called priority inversion problem

Sleep and Wakeup

- ❖ solutions so far have the defect of busy waiting.
- ❖ that is, when a process wants to enter its critical region, it checks to see if entry is allowed, if it is not, the process just sits in a tight loop waiting until it is
- ❖ This approach wastes CPU time
- ❖ We could use *sleep* and *wakeup* system calls instead of wasting CPU time when they are not allowed to enter into their critical section.
- ❖ Essentially, when a process is not permitted to access its critical section, it uses a system call known as *sleep*, which causes that process to block.
- ❖ The process will not be scheduled to run again, until another process uses the *wakeup* system call. Wakeup is called by a process when it leaves its critical section if any other processes have blocked

Semaphore

- ❖ **Software solution** to the critical section problem
- ❖ A semaphore is a special kind of **integer variable**, usually **stored in shared memory**, so all processes can access it
- ❖ Semaphores are (usually greater than ZERO) associated with a number which denotes the **total number of resources controlled by the semaphore variable**
- ❖ It is used as a flag that is **only accessed through two system calls**
 1. down/decrement /wait/P()
 2. up/increment/signal/V()
- ❖ Both of these methods are **atomic in nature** (It is guaranteed that once a semaphore operation has started, no other process can access semaphore until the operation has completed or blocked)

Note:

P (from Dutch word **proberen** meaning to test)

V (from Dutch word **verhogen** meaning to increment)

Semaphore

❖ Acquiring a semaphore

To acquire a semaphore, process executes `down()`. Every time an access is granted the count parameter of the semaphore is decremented by 1.

❖ Releasing the semaphore

To acquire a semaphore, process executes `up()`. When the resource released the count is incremented by 1.

❖ Properties of Semaphore:

- ❖ Machine independent.
- ❖ Works with any no of processes.
- ❖ Can have different semaphores for different critical sections.
- ❖ A process can acquire multiple needed resources by executing multiple downs.

Semaphore

❖ Values of semaphore:

1. Binary (only 0 and 1)

A Semaphore which controls access to only one resources is also referred to as Binary Semaphore or LOCKS

2. Counting (any numerical value)

A semaphore which controls access to a number of resources (>1) is referred to as counting Semaphores

❖ Types of semaphore:

1. Standard – busy waiting (can be binary or counting)

2. Complex – block while waiting (can be binary or counting)

Standard Semaphore (Binary)

```
wait(S) {  
    while(s<=0); //no-op  
    S--;  
}
```

← Entry
Section

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


← Exit
Section



Complex Semaphore (Binary)

```
Wait(S){  
    If(semaphore S==0){  
        //block(sleep) and queue  
    }else{  
        semaphore S--; //decrement semaphore  
    }  
}
```

Entry Section



```
signal(S){  
    if(semaphore S==0&&there is a process in queue){  
        //wake up the first process in the queue  
    }else{  
        semaphore S++; //increment semaphore  
    }  
}
```

Exit Section



Counting Semaphore: An example

INITIALIZATION

Semaphore $S_r = 3$; //complex, counting semaphore

Semaphore $S = 1$; // standard, binary semaphore

Flag[3] = {0}; //flag for each resource to indicate busy or available

Semaphore S_r

3

Semaphore S

1



0



0



0

Counting Semaphore

```
Wait( Sr ); //Semaphore to indicate if there is an available resource
wait( S ); //binary semaphore to let only one process check the flags
int resourceFree; //local variable of first resource that is available
int i = 0; //counter
do while (flagR[ i ] != 0 ) //loop to find the available resource
    i ++;
end while;
flagR [ i ] = 1; //set resource as busy
resource Free = i; //assign a value to the resourceFree local variable
signal ( S ); //allow another process to check resource flags
useResource (resourceFree); //use the resource
flagR [ resourceFree ] = 0; //set the resource available when done
Signal (Sr ); //signal, wake up next process or increment semaphore
```

Counting Semaphore

- Wait And Signal

```
wait(Sr){  
    if (semaphore Sr == 0) { // block and queue}  
    else { semaphore Sr --; } // decrement semaphore}
```

```
signal(Sr){  
    if (semaphore Sr == 0 && there is a process  
        { //wake up the first process in the queue }  
    else { semaphore Sr ++; } //increment semaphore}
```

```
wait(S){  
    while (semaphore S == 0) ; //no op  
    S --; } // decrement semaphore
```

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

What's wrong with semaphore?

- ❖ They are essentially shared global variables.
- ❖ There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
- ❖ Access to semaphores can come from anywhere in a program.
- ❖ There is no control or guarantee of proper usage
 - ❖ Suppose that a process interchanges the order in which the wait() and signal () operations on the semaphore are executed.
 - ❖ In such case, two processes can be simultaneously be in critical section, violating the mutual exclusion criteria
 - ❖ Suppose that a process replaces signal () with wait (). In such case, a deadlock will occur
 - ❖ Suppose that a process omits the wait (), or the signal (), or both. In this case, either mutual exclusion is violated or a deadlock will occur.
- ❖ Solution: Use a higher level primitives called **Monitors**

Monitors

- ❖ Another software solution for critical section problem is monitor
- ❖ Monitor is a **high-level abstraction** that provides a convenient and effective mechanism for process synchronization
- ❖ A monitor is similar to a *class* that encapsulates
 - ❖ Shared data structure
 - ❖ Procedure that operates on shared data
 - ❖ Synchronization between concurrent procedure invocation
- ❖ Unlike classes,
 - ❖ Monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor's method at a time
 - ❖ Monitors require all data to be private

Monitors

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

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

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

Syntax of Monitor

```
Monitor account
{
    double balance;

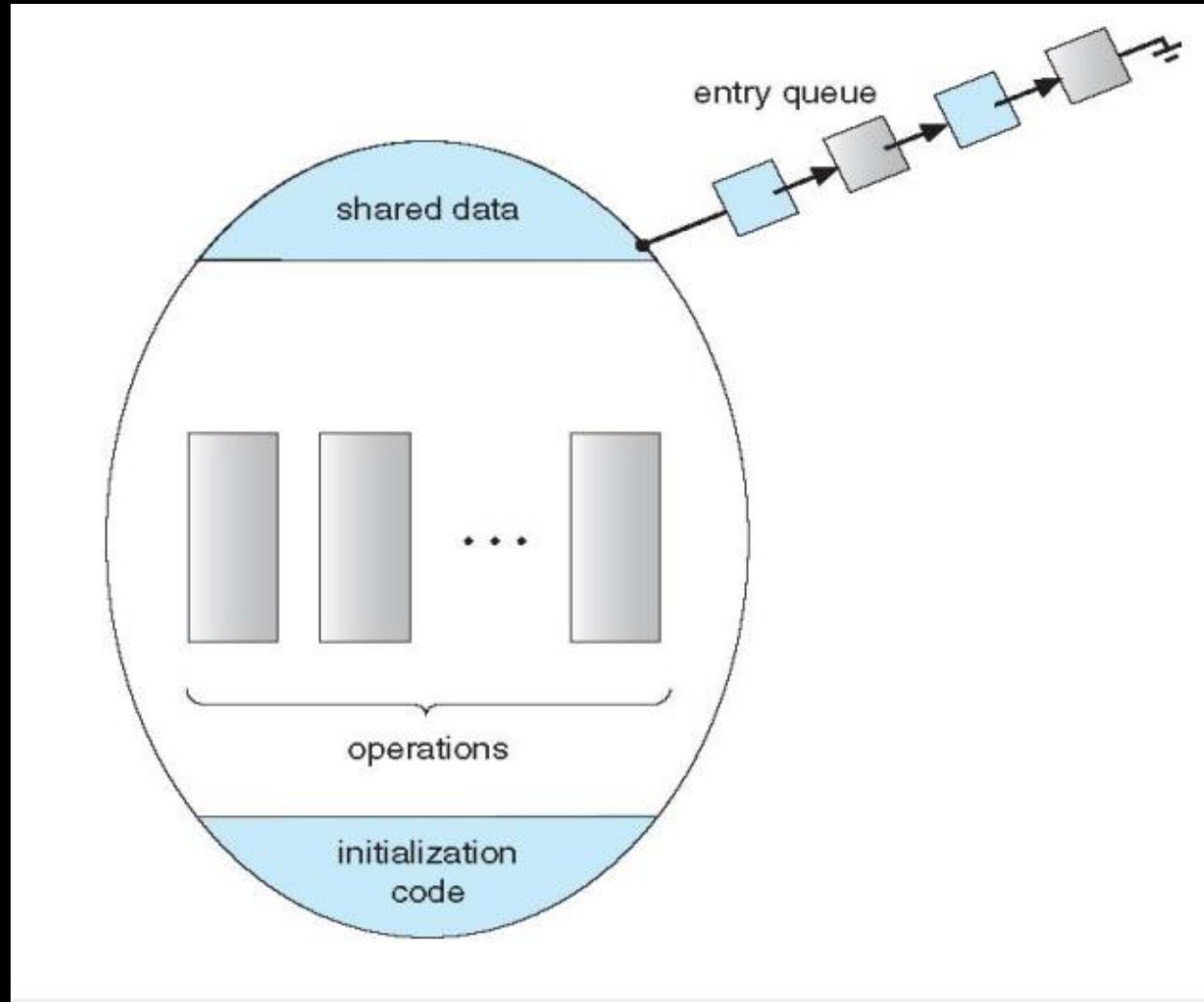
    withdraw(amount)
    {
        balance = balance-amount;
        return balance;
    }
}
```

Example of Monitor

Monitors

- ❖ A monitor defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data
 - ❖ The monitor uses *lock* to ensure that only a thread is active in the monitor at a given instance
 - ❖ The *lock* also provides mutual exclusion for shared data
 - ❖ Conditional Variables enable threads to go to sleep inside of the critical sections, by releasing their lock at the same time it puts the thread to sleep
- ❖ Monitor operations
 - ❖ Encapsulates the shared data you want to protect
 - ❖ Acquires the lock at the start
 - ❖ Operates on the shared data
 - ❖ Temporarily release the lock if can't complete
 - ❖ Reacquires the lock when it can continue
 - ❖ Release the lock at the end

Monitors



Monitors with Condition Variables

- ❖ The monitor construct, defined so far, is not sufficiently powerful for modeling some synchronization schemes
- ❖ For this we have the mechanism of *condition variables*
- ❖ Condition variable provides synchronization inside the monitor
- ❖ To allow a process to wait within the monitor , a condition variable must be declared as:
condition x, y;
- ❖ Three operations can be invoked on a condition variable : *wait()* , *signal()* and *broadcast()*
- ❖ The operation *x.wait()* means that the process invoking this operation is suspended until another process invokes *x.signal()*

Monitors with Conditional Variables

- ❖ The `x.signal()` operation resumes exactly one suspended process on condition variable `x`. If no process is suspended on condition `x`, then the signal has no effect
- ❖ Suppose that, when the `x. signal ()` operation is invoked by a process `P`, there is a suspended process `Q` associated with condition `x`.
- ❖ Clearly, if the suspended process `Q` is allowed to resume its execution, the signaling process `P` must wait. Otherwise, both `P` and `Q` would be active simultaneously within the monitor.

Monitors with Condition Variables

- Three possibilities exist:
 - **Signal and wait:** P either waits until Q leaves the monitor or waits for another condition.
 - **Signal and continue:** Q either waits until P leaves the monitor or waits for another condition.
 - **Signal and Leave:** P has to leave the monitor after signaling

Message Passing

- ❖ Message passing is mechanism for processes to communicate and to synchronize their actions **without resorting to a shared variable** like a semaphore
- ❖ This method of Inter Process Communication (IPC) uses two primitives :
 - send (destination, message);**
 - receive (source, message);**
- ❖ **send** call sends a message to a given destination
- ❖ **receive** receives a message from a given source. **If no message is available, receiver can block until one arrives, or it can return immediately with error code**
- ❖ If processes P and Q needs to communicate, they need to
 1. establish a **communication link** between them
 2. exchange message via send/receive

Message Passing : Direct Communication

❖ Processes must name each other explicitly:

1. send(P, message) : send a message to process P
2. receive(Q, message) : receive a message from process Q

❖ Properties of direct communication link

1. A link is associated with exactly one pair of communicating processes
2. Between each pair there exists exactly one link
3. The link may be unidirectional, but is usually bidirectional

Message Passing : Indirect Communication

- ❖ Messages are directed and received through mailboxes (also referred as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- ❖ Properties of indirect communication link:
 1. Link established only if process share a common mailbox
 2. A link may be associated with many processes
 3. Each pair of processes may share several communication links
 4. Link may be unidirectional or bidirectional

Message Synchronization

- ❖ Message passing may be **blocking or non-blocking**
- ❖ *Blocking* is considered **synchronous**
 - *Blocking send* has the sender blocked until the message is received
 - *Blocking receive* has the receiver blocked until message has arrived
- ❖ *Non-blocking* is considered **asynchronous**
 - *Non-blocking send* has the sender send the message and continue
 - *Non-blocking receive* has the receiver receive a valid message or null

Classical Synchronization Problems

- ❖ Producer Consumer Problem (Bounded buffer problem)
- ❖ The Dining Philosophers problem
- ❖ The Readers and writers problem

Producer Consumer Problem

- ❖ two processes share a common fixed sized (bounded) buffer
- ❖ one of the process is producer and other is consumer
- ❖ producer puts information in buffer and consumer takes it out
- ❖ To keep track the number of items in the buffer, we have a shared variable '*count*'

Producer

- ✓ Has to stop when buffer is full
- ✓ If producer is producing , don't allow consumer to consume data

Consumer

- ✓ Has to stop when buffer is empty
- ✓ If consumer is consuming, don't allow producer to produce data

- ❖ Devise a synchronization protocol to solve this problem!!

Producer Consumer Problem

SOLUTION 1 (sleep and wakeup)

- ❖ The solution is for the producer to go to sleep, to be awakened when consumer has removed one or more items.
- ❖ Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up

```

#define N 100    //no. of slots in the buffer
int count=0;    //no. of items in the buffer

void producer(void) {
    int item;
    while(1) {
        item=produce_item();    //generate next item
        if(count==N) sleep();    //if buffer is full, go to sleep
        insert_item(item);    //put item in the buffer
        count=count+1;    //increment count of items in buffer
        if(count==1) wakeup(consumer);    //if buffer was empty, wake up consumer now
    }
}

void consumer(void) {
    int item;
    while(1) {
        if(count==0) sleep();    //if buffer is empty, go to sleep
        item=remove_item();    //take item out of the buffer
        count=count-1;    //decrement count of items in the buffer
        if(count==N-1) wakeup(producer);    //if buffer was full, wakeup producer now
        consume_item(item);    //do stuff with item
    }
}

```

Producer Consumer Problem

Q. What is the problem with that solution ??

A. Race condition

- ❖ Race condition may occur because the access to the shared variable count is unconstrained
- ❖ Both may sleep forever because the wake up signal may be lost
- ❖ Let us now see how we can solve this problem **using Semaphore**

```
#define N 100
typedef int semaphore;
semaphore mutex=1; //mutex control access to critical region
semaphore empty=N; //to count empty buffer slots
semaphore full=0; //to count full buffer slot
void producer(void) {
    int item ;
    while(TRUE) {
        item = produce_item(); //generate sth to enter into buffer
        wait(empty); //or down(empty) to decrease empty count
        wait(mutex); //enter CS
        insert_item(item); //put new item in buffer
        signal(mutex); //leave CS
        signal(full);
    }
}
void consumer(void) {
    int item;
    while(TRUE) {
        wait(full);
        wait(mutex); //enter CS
        item = remove_item(); //take item from buffer
        signal(mutex);
        signal(empty);
    }
}
```


Producer Consumer Problem

❖ What happens when we reverse the order of two *wait()* in producer routine??

```
void producer ( )  
{  
    int item;
```

```
    while(TRUE) {  
        item=produce_item();  
        down(empty);  
        down(mutex);  
        insert_item(item);  
        up(mutex);  
        up(full);  
    }  
}
```

```
void producer()  
{  
    int item;
```

```
    while(TRUE) {  
        item=produce_item();  
        down(mutex);  
        down(empty);  
        insert(item);  
        up(mutex);  
        up(full);  
    }  
}
```

The Dining Philosophers Problem

- ❖ Five philosophers are seated around a circular table
- ❖ Each philosopher has a plate of spaghetti
- ❖ Between each pair of plate is a single fork
- ❖ The spaghetti is so slippery that a philosopher needs two fork to eat it
- ❖ The life of the philosopher consists of alternate period of eating and thinking
- ❖ When a philosopher gets hungry, s/he tries to acquire his/her left and right fork, one at a time, in either order.
- ❖ If successful in taking two forks s/he eats for a while and put down the fork, and continues to think.

Can you write a program for each philosopher that does what it is supposed to do and never get stuck (avoid deadlock)?

The Dining Philosophers Problem

Solution 1 :

- ❖ Take the left fork first then right fork
- ❖ wait if right fork is unavailable

Problem??

- ❖ if all pick up left fork at once,
there will be deadlock as
no one will get the right one

```
#define N 5
void philosopher(int i) {
    while (TRUE) {
        think();
        wait(fork[i]);
        wait(fork[(i+1)%N]);

        eat();

        signal(fork[i]);
        signal(fork[(i+1)%N]);
    }
}
```

The Dining Philosophers Problem

- ❖ We could modify the program so that after taking the left fork, the philosopher checks to see if right fork is available.
- ❖ If not, philosopher puts back the left fork, waits for some time and repeats the whole process.
- ❖ But what if all the philosophers pick up the left fork simultaneously , fail to obtain right fork simultaneously, put back the left fork simultaneously and try again simultaneously ?
- ❖ This situation in which the program continues to run indefinitely but fail to make any progress is called starvation
- ❖ Let us look at another solution to this problem !!

```
#define N 5          //number of philosopher
#define LEFT (i+N-1)%N //left neighbour of i
#define RIGHT (i+1)%N //right neighbour of i
#define THINKING 0    //philosopher is thinking
#define HUNGRY 1      //philosopher is trying to get forks
#define EATING 2      //philosopher is eating

typedef int semaphore; //semaphore is special kind of int
int state[N];          //array to keep track of what everyone is doing
semaphore mutex =1;    //for ME in critical region
semaphore s[N];        //one semaphore for each philosopher

void philosopher(int i){
    while(TRUE){        //repeat forever
        think();        //philosopher is thinking
        take_forks(i);  //acquire two fork or block
        eat();          //eating
        put_forks(i);   //put both fork back
    }
}
```

```
void take_forks(int i){
    wait(mutex);    //enter critical region
    state[i]=HUNGRY;    //ith philosopher is hungry
    test(i);    //try to acquire two forks
    signal(mutex);    //exit critical region
    wait(s[i]);    //block of forks were not obtained
}

void put_forks(int i){
    wait(mutex);    //enter CS
    state[i]=THINKING;    //finished eating, time to think
    test(LEFT);    //see if left neighbour can now eat
    test(RIGHT);    //see if right neighbour can now eat
    signal(mutex);    //exit CS
}

void test(int i){
    if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING){
        state[i]=EATING;
        up(s[i]);
    }
}
```

Readers Writers Problem

- Readers writers problem models access to database
- Many competing process may want to perform read or/and write operations in the database
- It is acceptable to have multiple processes reading the database at the same time, but if a process is updating (writing) the database, no other process may have access to the database, not even readers
- The question is how do you program the readers and writers?

```

typedef int semaphore;
semaphore mutex=1;           //control access to rc (read counter)
semaphore db=1;              //control access to database
int rc=0;

void reader(void) {
    while(TRUE) {
        wait(mutex);          //get exclusive access to rc
        rc=rc+1;              //one reader more now
        if(rc==1) wait(db);    //if this is the first reader.
        signal(mutex);        //release exclusive access to rc

        read_database();

        wait(mutex);          //get exclusive access to rc
        rc=rc-1;              //one reader fewer now
        if(rc==0) signal(db);  //if this is the last reader
        signal(mutex);        //release exclusive access to rc
        use_read_data();
    }
}

void writer(void) {
    while(TRUE) {
        get_data();           //get data to write
        wait(db);             //get exclusive access to db
        write_to_database();   //write the data
        signal(db);           //release exclusive access to db
    }
}

```


Readers Writers problem

- In the solution given in the previous slide, the first reader to get access to the database does a *down/wait* on semaphore *db*.
- Subsequent members merely increase the counter *rc*
- As readers leave, they decrement the counter, and the last one does an *up/signal* on the semaphore, allowing a blocked writer, if there is one, to get in
- **Problem??**
 - In this solution, the number of readers keep on increasing
 - But if a writer shows up, the writer won't get access to the database
 - As long as at least one reader is still active, subsequent readers are admitted. As a consequence, as long as there is a steady supply of readers, they will get all in as soon as they arrive.
 - Writer will get suspended until no reader is present

Readers Writers Problem

- To prevent this situation, the program could be written slightly differently:
- When a reader arrives and the writer is waiting, the reader is suspended behind the writer instead of being admitted immediately.
- In this way, the writer will have to wait for the readers which were active when it arrived but does not have to wait for the readers that came after it.

End of Chapter 3