Operating System
CS 2006
Lecture 8

Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Process Synchronization

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner

Background

- Processes can execute concurrently or parallel
- May be interrupted at any time, partially completing execution
  - Process scheduler switches among processes; concurrency
  - Multiprogramming distributes tasks among cores; parallelism
- May be interrupted at any time, partially completing execution
  - How to preserve the integrity of data shared by several processes..?
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Race Condition

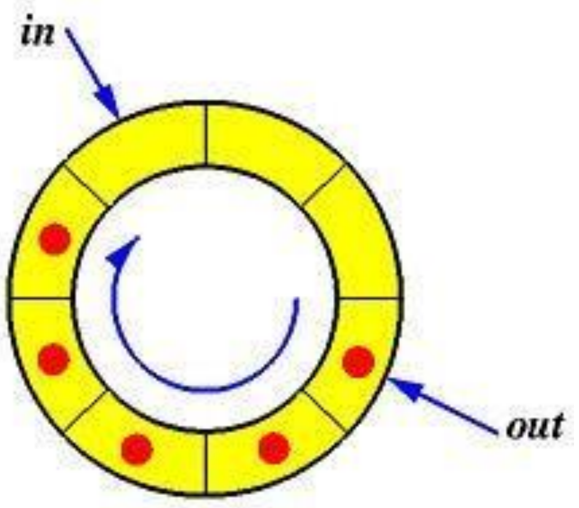
- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called race condition.

Race Condition

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers; the original solution in Chap-3 allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.

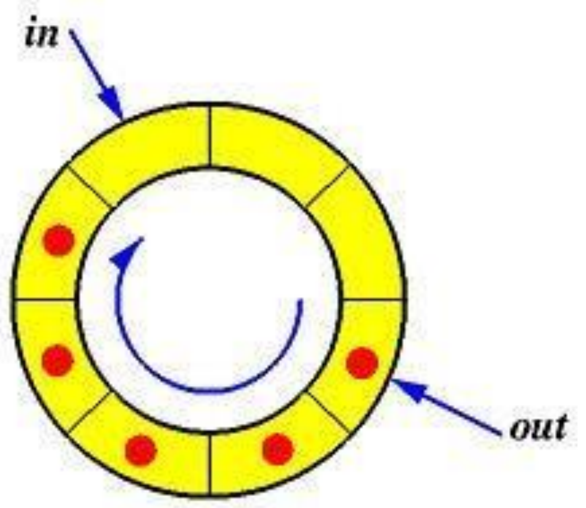
```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```



Race Condition

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers; the original solution in Chap-3 allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.

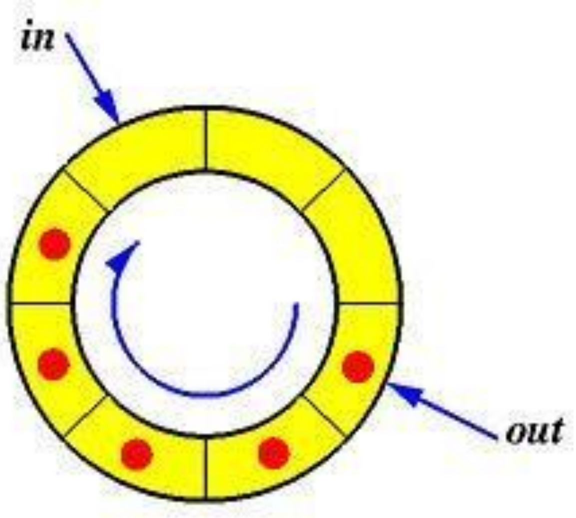



```
item next_consumed;
while (true) {
        while (in == out)  //Buffer is empty
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```



Race Condition

- **However, Producer and consumer may function incorrectly when executed concurrently**
- **counter++** could be implemented as

```
register1 = counter            //load
register1 = register1 + 1      //Increment
counter = register1            //store
```

- **counter--** could be implemented as

```
register2 = counter            //load
register2 = register2 - 1      //decrement
counter = register2            //store
```

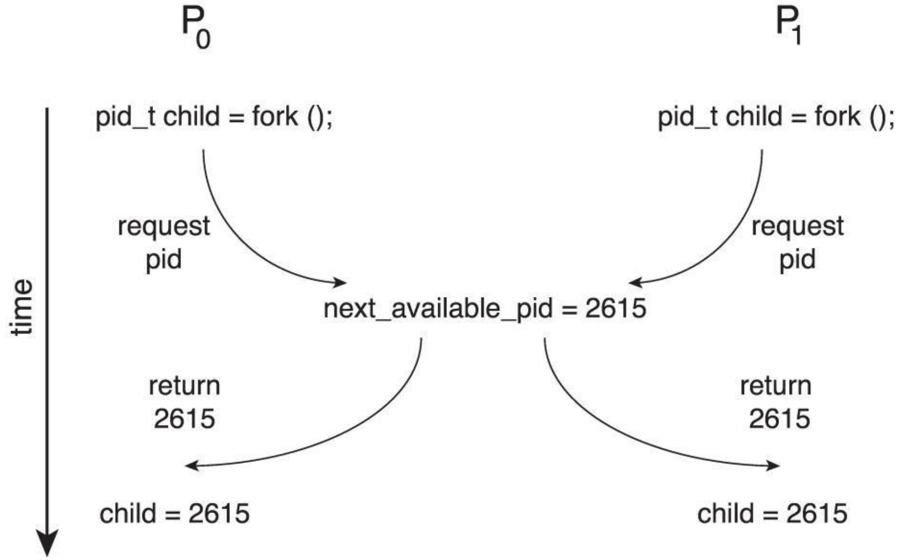
- Consider this execution interleaving with "count = 5" initially:

S0: **producer** execute `register1 = counter` {register1 = 5}
S1: **producer** execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: **producer** execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {**counter = 4**}

- ***Race condition*** = concurrent access to variable + result depends on order
  - ***Solution:*** synchronization; only one process at a time accesses data

Race Condition

- Processes P_0 and P_1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

Critical Section Problem

Critical Section

A critical section is a **code segment** that can be accessed by only one process at a time.

- The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables.
- Critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

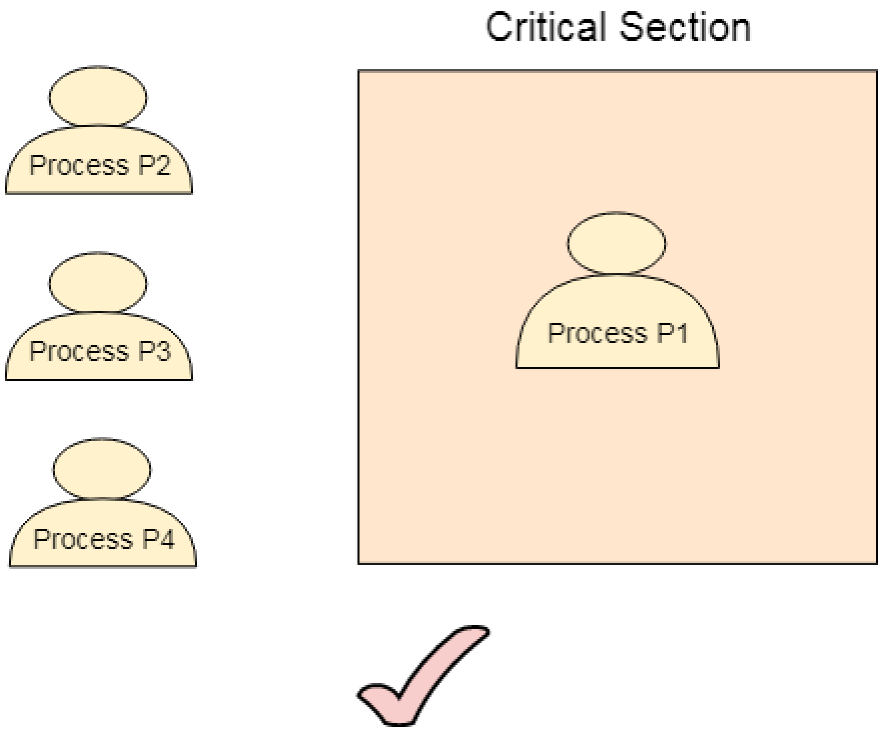
Critical Section Problem

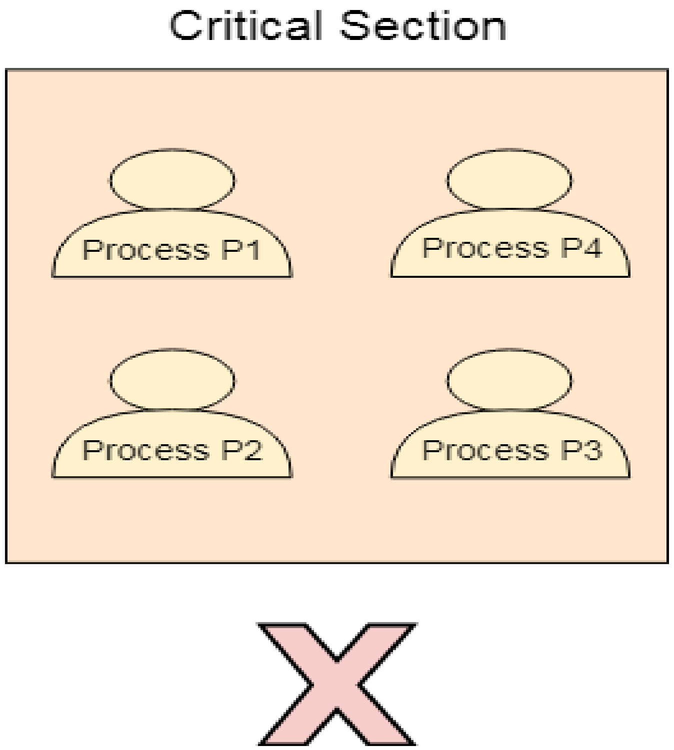
- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section Problem

Mutual Exclusion

- If process Pi is executing in its critical section, then no other processes can be executing in their critical sections





Critical Section Problem

Progress

Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section

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 process that will enter the critical section next cannot be postponed indefinitely.

Critical Section Problem

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

Critical Section

General structure of process P_i

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Request Permission to Enter

Housekeeping to let processes to enter other

Algorithm for Process Pi

Algorithm 1: Any Problem?.

```
bool flag[i] = false;
//Process intends to enter CS sets the flag true
do {
    flag[i] = True; //i is ready
        while (flag[j]);
            critical section
        flag[i] = False;
            remainder section
 } while (true);
```

```
bool flag[j] = false;
//Process intends to enter CS sets the flag true
do {
    flag[j] = True;//j is ready
        while (flag[i]);
            critical section
        flag[j] = False;
            remainder section
 } while (true);
```

Algorithm for process Pi

Algorithm 2: **Any Problem?.**

Turn = i

```
do {

while (turn == j); //if it is not my turn I wait

    critical section

    turn = j; //I am done, its your turn now

    remainder section

    } while (true);
```

Turn = j

```
do {

    while (turn == i);

    critical section

    turn = i;

    remainder section

     } while (true);
```

Critical Section Handling in OS

- Two approaches depending on if kernel is
  - preemptive
  - non-preemptive

Preemptive Kernels

allows preemption of process when running in kernel mode

- Preemption
  - The ability of the OS to interrupt a currently scheduled task in favor of a higher priority task.
  - It is normally carried out by a privileged task or part of the system known as a preemptive process scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system.
  - This **only applies to processes running in kernel mode.**

Preemptive Kernels

- More responsive to users, but
    - Shared kernel data may not be free from race conditions

- Preemptive kernel must be carefully designed

Non Preemptive Kernels

process runs until exits kernel mode, blocks, or voluntarily yields CPU

- free of race conditions in kernel mode as only one process is active in the kernel at a time

Solution to Manage the Critical Section

1. Mutex Lock
2. Peterson’s Solution
3. Semaphore

Petersons Solutions

Petersons Solution

- Peterson's solution is a **classical software-based method** for achieving **mutual exclusion** in concurrent programming.
- It allows two processes to share a single-use resource without conflict, using only shared memory for communication.

- It **Restricted to two processes**
- Good algorithmic description
  - Can show how to address the 3 requirements
- No guarantees on modern architectures
  - Instruction reordering

Shared Variables

- Flag
  - An array of boolean where flag[i] indicates if process Pi is interested in entering the critical section.

- Turn
  - An integer variable that indicates whose turn it is to enter the critical section.

int turn

boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section

initially, the value of turn is set to *i*

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!

Algorithm for Process Pi

```
while (true){

        while (turn = = j);

        /* critical section */
        turn = j;
        /* remainder section */
}
```

Petersons Solution for Process Pi

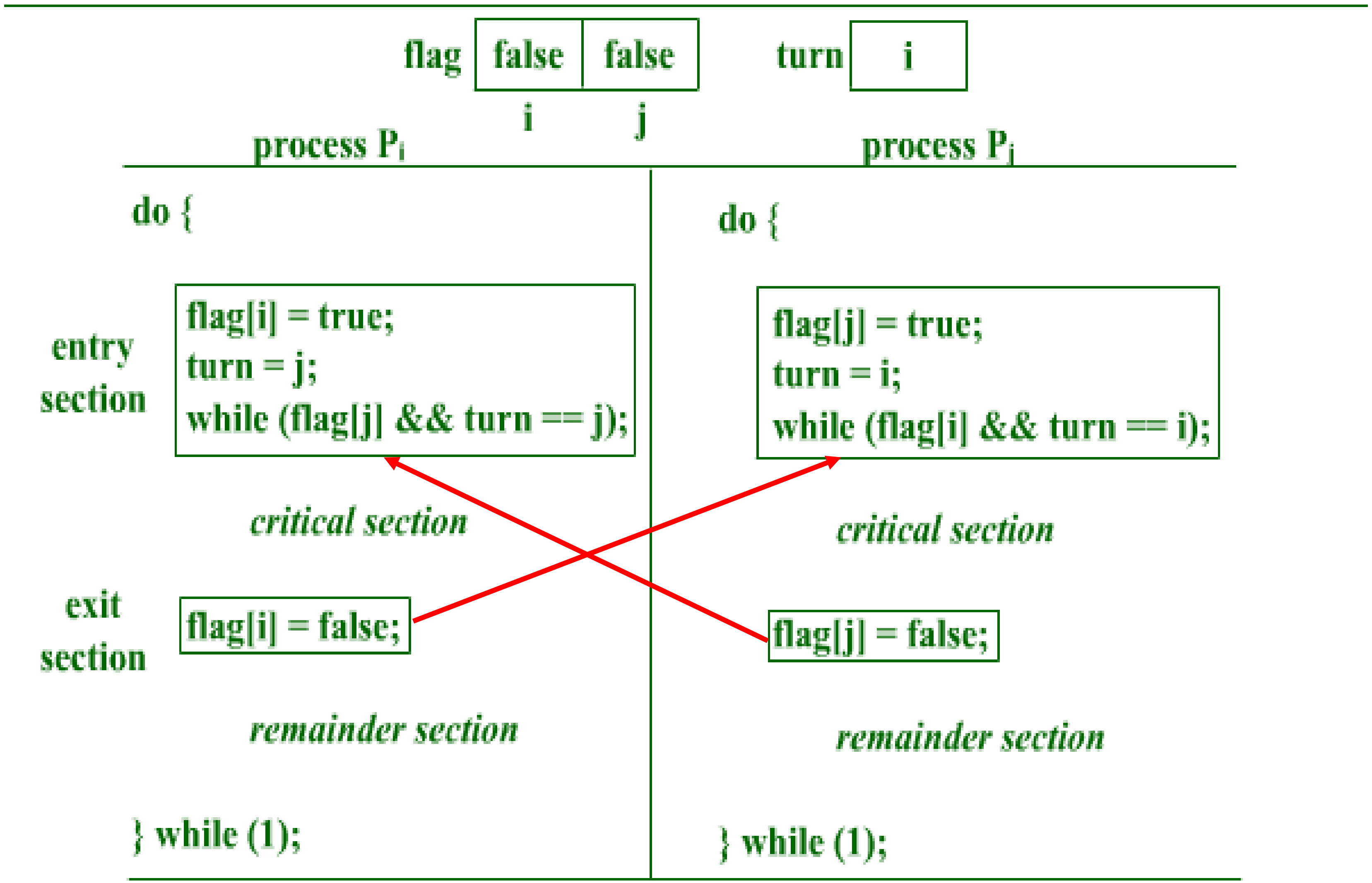
Flag[] set to false initially.

Entry Section

```
do {
        flag[i] = true; //Process i is ready
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

Exit Section

Petersons Solution for Pi and Pj



Solution Requirements

- The solution to the critical section problem must satisfy the following three requirements

1. Mutual exclusion
2. Progress
3. Bounded Waiting

Example of Petersons Solution

Step 1

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
int flag[2]={0,0};
int turn=0;
int count;
```

Step 2

```
void* increment (void* ptr)
{
for(int i=0;i<1000000;i++)
{ flag[0] = 1;
turn = 1;
while(flag[1]==1 && turn==1);
count++;
flag[0]=0;
}
pthread_exit(0);
}
```

Step 3

```
void* decrement (void* ptr)
{
for(int i=0;i<1000000;i++)
{
flag[1] = 1;
turn = 0;
while(flag[0]==1 && turn==0);
count--;
flag[1] = 0;
}
pthread_exit(0);
}
```

Step 4

```
int main()
{
count = 100;
pthread_t pid1,pid2;
printf("\nInitial value of Count is %d ",count);
pthread_create(&pid1,NULL,&increment,NULL);
pthread_create(&pid2,NULL,&decrement,NULL);
pthread_join(pid1,NULL);
pthread_join(pid2,NULL);
printf("\nFinal value of Count is %ld ",count);
printf("\nIs it correct? \n");
return 0;
}
```

Advantages of Petersons Solution

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met.
- Multiple processes can access and share a resource without causing any resource conflicts.
- Every process has a chance to be carried out.
- It uses straightforward logic and is easy to put into practice.
- eliminates the chance of a deadlock.

Disadvantages of Petersons Solution

- Waiting for the other processes to exit the critical region may take a long time.
- it busy waiting.
- On systems that have multiple CPUs, this algorithm might not function.
- The Peterson solution can only run two processes concurrently.

Mutex Lock

Mutex Lock

- A mutex lock has a Boolean variable available whose value indicates if the lock is available or not.
- If the lock is available, a call to acquire()succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Mutex Lock

- Protect a critical section by

  First acquire() a lock

  Then release() the lock
- Calls to acquire() and release() must be atomic
- Usually implemented via hardware atomic instructions such as compare-and-swap.

Syntax of Mutex Lock

```
while (true) {
   acquire lock

      critical section

   release lock

remainder section
}
```

Example of Mutex in C

Program create two threads: one to increment the value of a shared variable and second to decrement the value of shared variable. Both the threads make use of locks so that only one of the threads is executing in its critical section

Step 1:

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
pthread_mutex_t l; //mutex lock
```

Step 2:

```
int main()
{
pthread_mutex_init(&l, NULL); //initializing mutex locks
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
}
```

```
void *fun1()
{
int x;
printf("Thread1 trying to acquire lock\n");
pthread_mutex_lock(&l);
printf("Thread1 acquired lock\n");
x=shared;//thread one reads value of shared variable
printf("Thread1 reads the value of shared variable as
%d\n",x);
x++; //thread one increments its value
printf("Local updation by Thread1: %d\n",x);
sleep(1); //thread one is preempted by thread 2
shared=x; //thread one updates the value of shared
variable
printf("Value of shared variable updated by Thread1 is:
%d\n",shared);
pthread_mutex_unlock(&l);
printf("Thread1 released the lock\n");
}
```

```
void *fun2()
{
int y;
printf("Thread2 trying to acquire lock\n");
pthread_mutex_lock(&l);
printf("Thread2 acquired lock\n");
y=shared;//thread two reads value of shared

printf("Thread2 reads the value as %d\n",y);
y--; //thread two increments its value

printf("Local updation by Thread2: %d\n",y);
sleep(1); //thread two is preempted by thread 1
shared=y; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread2 is:
%d\n",shared);
pthread_mutex_unlock(&l);
printf("Thread2 released the lock\n");
}
```

Lock Contention

Locks are either contended or uncontended

- Contended
  - A lock is considered contended if a thread blocks while trying to acquire the lock.
- Uncontended
  - If a lock is available when a thread attempts to acquire it, the lock is considered uncontended

Contended locks can experience either **high contention (a relatively large number of threads attempting to acquire the lock)** or **low contention (a relatively small number of threads attempting to acquire the lock.)** Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

Spin Lock

- The type of mutex lock we have been describing is also called a **spin lock** because the process "**spins" while waiting for the lock to become available.**

- spinlocks do have an advantage, however, in that **no context switch is required when a process must wait on a lock**, and a context switch may take considerable time

On modern multicore computing systems, spinlocks are widely used in many operating systems.

Semaphore

Semaphore

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, **wait and signal** that are used for **process synchronization.**

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:
  - wait()
  - signal().

Semaphore

Wait

- The wait operation decrements the value of its argument S, if it is positive.
- If S is negative or zero, then no operation is performed.

Syntax

```
wait(S)
{
   while (S<=0);

   S--;
}
```

Signal

- The signal operation increments the value of its argument S.

Syntax

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

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically.

Types of Semaphore

- There are two types of semaphore

1. Binary Semaphore
2. Counting Semaphore

Binary Semaphore

- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
- The wait operation only works when the semaphore is 1
- The signal operation succeeds when semaphore is 0.
- It is sometimes easier to implement binary semaphores than counting semaphores

Thus, binary semaphores behave similarly to mutex locks.

Example

- Consider two concurrently running processes:P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed.

Solution

- We can implement this scheme readily by letting **P1 and P2 share a common semaphore synch, initialized to 0.**

- In process P1, we insert the statements
- P1:

  S1;

  signal(synch);

- In process P2, we insert the statements
- P2:

  wait(synch);

  S2;

Semaphore Implementation

- Must guarantee that no two processes can execute **the wait() and signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation without Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue

Example:

- Program creates two threads: one to increment the value of a shared variable and second to decrement the value of the shared variable. Both the threads make use of semaphore variable so that only one of the threads is executing in its critical section

```c
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1;
Sem_t s;    //semaphore
Variable
```

```c
int main()
{
sem_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of processes sharing semaphore, 3rd argument is the initial value of semaphore variable

pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);

pthread_join(thread1, NULL);
pthread_join(thread2,NULL);

printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
}
```

```
void *fun1()
{
int x;

sem_wait(&s); //executes wait operation on s
x=shared;//thread1 reads value of shared variable

printf("Thread1 reads the value as %d\n",x);

x++; //thread1 increments its value
printf("Local updation by Thread1: %d\n",x);

sleep(1); //thread1 is preempted by thread 2

shared=x; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread1 is: %d\n",shared);
sem_post(&s);
}
```

```
void *fun2()
{
int y;
sem_wait(&s);

y=shared;//thread2 reads value of shared variable
printf("Thread2 reads the value as %d\n",y);
y--; //thread2 increments its value
printf("Local updation by Thread2: %d\n",y);

sleep(1); //thread2 is preempted by thread 1

shared=y; //thread2 updates the value of shared variable
printf("Value of shared variable updated by Thread2 is: %d\n",shared);
sem_post(&s);
}
```

Working Detail

- The process initializes the semaphore variable s to ‘1’ using the sem_init() function.
- The initial value is set to ‘1’ because binary semaphore is used here.
    - If you have multiple instances of the resource then counting semaphores can be used . Next, the process creates two threads. thread1 acquires the semaphore variable by calling sem_wait().
    - Next, it executes statements in its critical section part. We use sleep(1) function to preempt thread1 and start thread2. This simulates a real-life scenario. Now,
- when thraed2 executes sem_wait() it will not be able to do so because thread1 is already in the critical section.
- Finally, thread1 calls sem_post() function.
- Now thread2 will be able to acquire s using sem_wait(). This ensures synchronization among threads.

Example 2:

Develop a C program where three threads (users) attempt to access a shared printer. Use a binary semaphore to ensure that only one user can use the printer at a time.

Solution:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
 sem_t printer;
```

```
 void* printer(void* arg) {
    int thread_id = *((int*)arg);
     printf("Thread %d is waiting to use the printer\n", thread_id);
    sem_wait(&printer);
    printf("Thread %d is using the printer\n", thread_id);
    sleep(2);
    printf("Thread %d is done using the printer\n", thread_id);
    sem_post(&printer);
    return NULL;
}
```

Solution:

```
int main() {
    sem_init(&printer, 0, 1);
     pthread_t t1, t2, t3;
    int t1_id = 1, t2_id = 2, t3_id = 3;
     pthread_create(&t1, NULL, use_printer, &t1_id);
    pthread_create(&t2, NULL, use_printer, &t2_id);
    pthread_create(&t3, NULL, use_printer, &t3_id);
     pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
     sem_destroy(&printer);
     return 0;
}
```

Counting Semaphore

The integer value S can range over an unrestricted domain

- These semaphores are used to **coordinate the resource access**, where the semaphore **count is the number of available resources.**
  - If the resources are added, semaphore count automatically incremented
  - if the resources are removed, the count is decremented.

Semaphore Implementation without Busy Waiting

```
  S->value--;
  if (S->value < 0) {



  }
}
```

```
  S->value++;
  if (S->value <= 0) {



  }
}
```

Counting Semaphore

Counting semaphore is a synchronization tool that is used in operating systems to control the access to shared resources.

It is a type of semaphore that allows more than two processes to access the shared resource at the same time.

A counting semaphore is represented by an integer value that can be incremented or decremented by the processes

Counting Semaphore Declaration Method

Three methods is used to initialize the counting semaphore

1. P( ) and V( )
2. Wait ( ) and Signal ( )
3. Down( ) and Up( )

Example:

Wait Operation

```
Wait(Semaphore S)
{
s.value = s.value – 1;
If(s.value<0)
{
//add the process into the
suspended list
}
Else
//process enter to the critical
Section
}
```

Signal Operation

```
signal(Semaphore S)
{
s.value = s.value + 1;
If(s.value <= 0)
{
//wakeup the process from
suspended list to wait list
}
Else
Return ;
}
```

Example:

A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution:

1. P operation also called as wait operation decrements the value of semaphore variable by 1.
2. V operation also called as signal operation increments the value of semaphore variable by 1.

Solution:

Initialized Value = 10
Wait Operation = 6
Signal Operation = 4

Wait Operation = 6						
10	**6**	**5**	**4**	**3**	**2**	**1**
	10-1 =9	9-1 = 8	8-1=7	7-1=6	6-1=5	5-1 =4

Wait Operation

```
Wait(Semaphore S)
{
s.value = s.value – 1;
If(s.value<0)
{
//add the process into the
suspended list
}
Else
//process enter to the critical
Section
}
```

Solution:

Initialized Value = 10
Wait Operation = 6
Signal Operation = 4

Signal Operation = 4				
4	**4**	**3**	**2**	**1**
	4+1 = 5	5+1=6	6+1=7	7+1=8

Signal Operation

```
signal(Semaphore S)
{
s.value = s.value + 1;
If(s.value <= 0)
{
//wakeup the process from
suspended list to wait list
}
Else
Return ;
}
```

Example 2:

A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?

Solution:
2

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 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
....	...
signal(S);	signal(Q);
signal(Q);	signal(S);

- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Deadlock and Starvation

- Priority Inversion
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Drawbacks of Semaphore

- Incorrect use of semaphore operations:

signal(mutex) …. wait(mutex)
wait(mutex) … wait(mutex)

Omitting of wait (mutex) and/or signal (mutex)

- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- Liveness refers to a set of properties that a system must satisfy to. ensure processes make progress
- Indefinite waiting is an example of a liveness failure.

Liveness

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let $\boldsymbol{S}$ and $\boldsymbol{Q}$ be two semaphores initialized to 1

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

Liveness

- Other forms of deadlock:
- **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