Chapter : Threads

Chapter : Threads

- Overview
- Benefits of Threads vs Processes
- Single and Multithreaded Processes
- Types of Threads
- Multithreading Models
- Thread Libraries

Threads

- It is single sequential (flow of) execution of tasks of process
- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - Thread id
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources

Threads (Cont.)

- In a multiple threaded task, while one thread is blocked and waiting, a second thread in the same task can run.
  - In web browser, one thread displays images, other text, other fetches data from network.
  - Word processor, graphics, response to keystrokes, spell check.

Benefits of Threads vs Processes

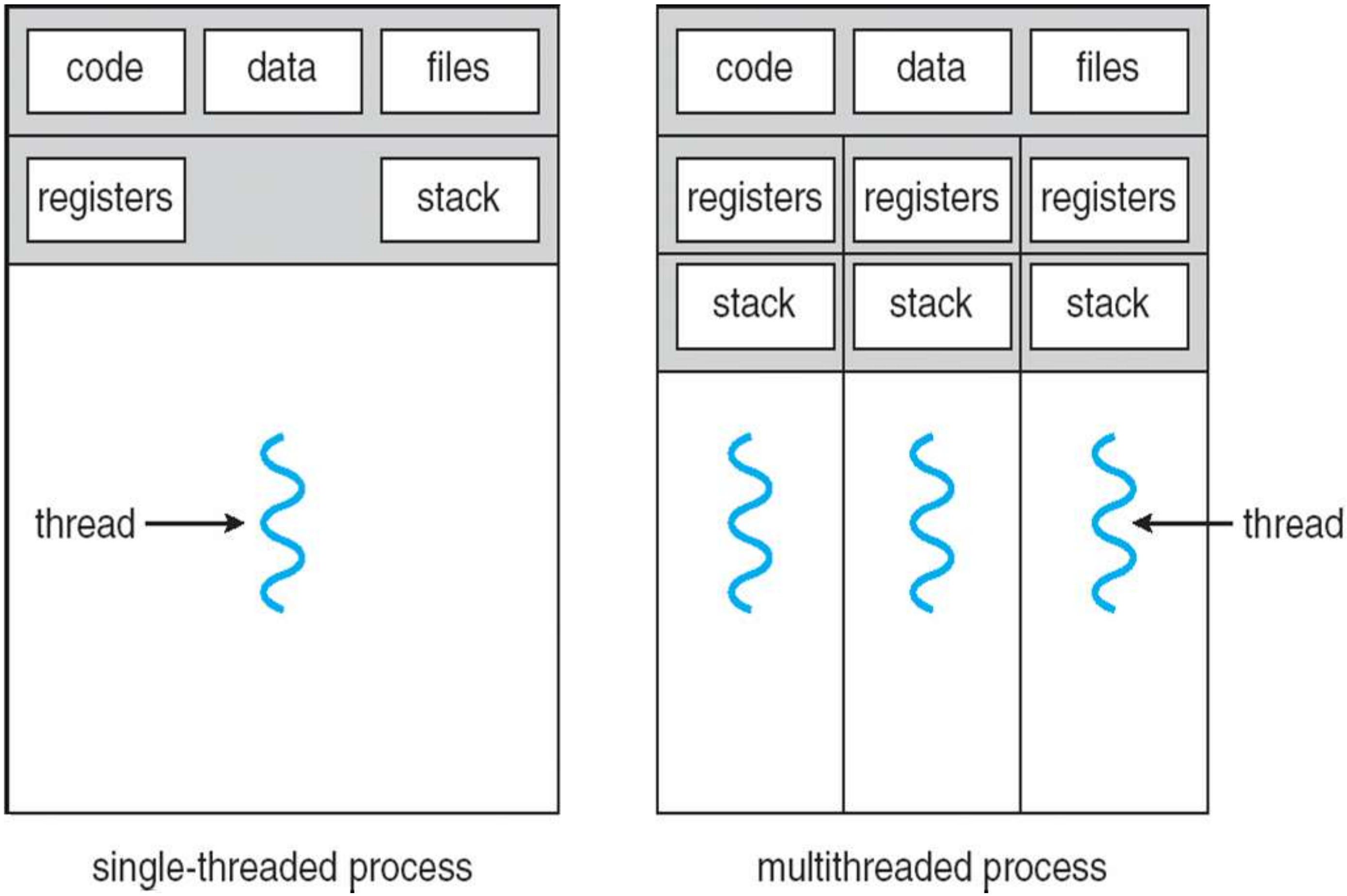
- **Less time to create a new thread** than a process, because the newly created thread **uses the current process address space**.

- **Less time to terminate a thread**

- **Less time to switch between two threads** - newly created thread uses the current process address space.

- Less communication overheads - **threads share everything**: address space, So, **data produced by one thread is immediately available to all the other threads.**

Single and Multithreaded Processes



Threads States

- Three key states: running, ready, blocked
- Termination of a process, terminates all threads within the process

Types of Threads

1. User Level Threads: (ULT)

- Threads of user application process.
- ULT are supported above the kernel and managed without kernel support
- These are implemented in user space in main memory. And managed bu user level library.
- The kernel is not aware of the existence of threads
- User Level Library is used for – Thread Creation, Scheduling and Management

Threads (Cont.)

- ULT require a kernel system call to operate
- It only takes care of the execution part.
- The lack of cooperation between user level threads and the kernel is a known disadvantage.
- In this case, the kernel may not favor a process that has many threads.
- User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call.
- If one thread blocks cause the entire process to block.

User Level Threads

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads(KLT)

- Threads of processes defined by OS itself
- KLT are supported and managed directly by OS.
- Kernel performs Thread Creation, Scheduling and Management in kernel space.
- No thread library but system calls to the kernel facility exists.

Kernel Level Threads(KLT)

- Kernel level threads are managed by the OS
- thread operations (ex. Scheduling) are implemented in the kernel code.
- Kernel level threads may favor thread heavy processes.
- they can also utilize multiprocessor systems by splitting threads on different processors or cores.
- If one thread blocks it does not cause the entire process to block.
- KLT are not portable because the implementation is operating system dependent.

Kernel Level Threads(KLT)

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Combined ULT/KLT Approaches

- Thread creation done in the user space
- In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process
- The programmer may adjust the number of KLTs

- Example is Solaris

Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

Multithreading Models

- One-to-One
- Many-to-One
- Many-to-Many

One-to-One

- Each user-level thread maps to one kernel thread
- The one-to-one model associates a single user-level thread to a single kernel-level thread.
- kernel level threads follow the one to one model

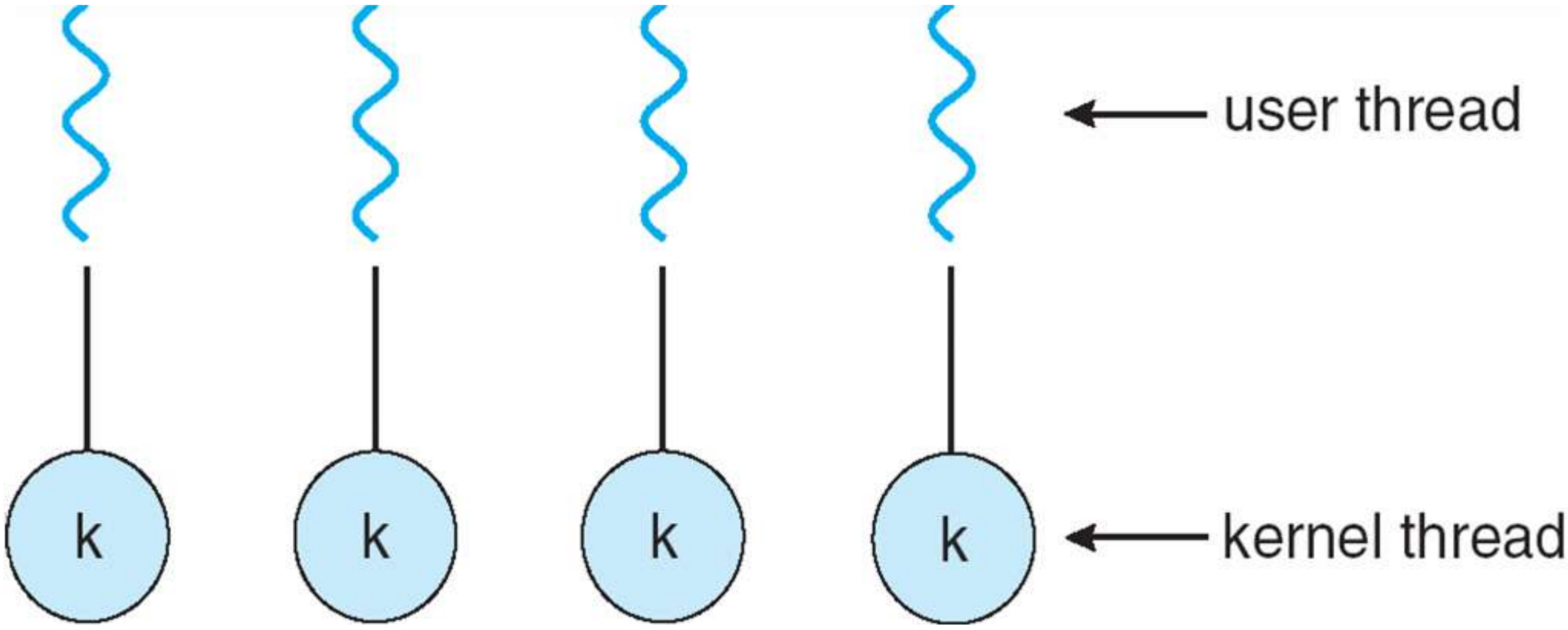
Advantage:

facilitates the running of multiple threads in parallel.

Drawback:

Generation of every new user thread must include the creation of a corresponding kernel thread causing an overhead

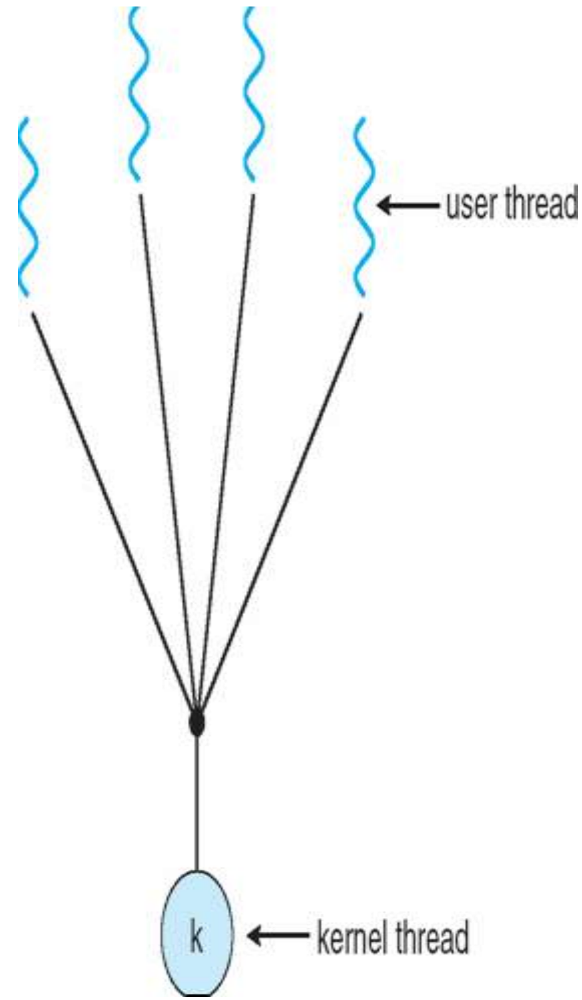
One-to-one Model



The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application

Many-to-One

- The many-to-one model associates all user- level threads to a single kernel-level thread.
- User level threads follow the many to one threading model.
- This means multiple threads managed by a library in user space but the kernel is only aware of a single thread of the process owning these threads.



Many-to-Many Model

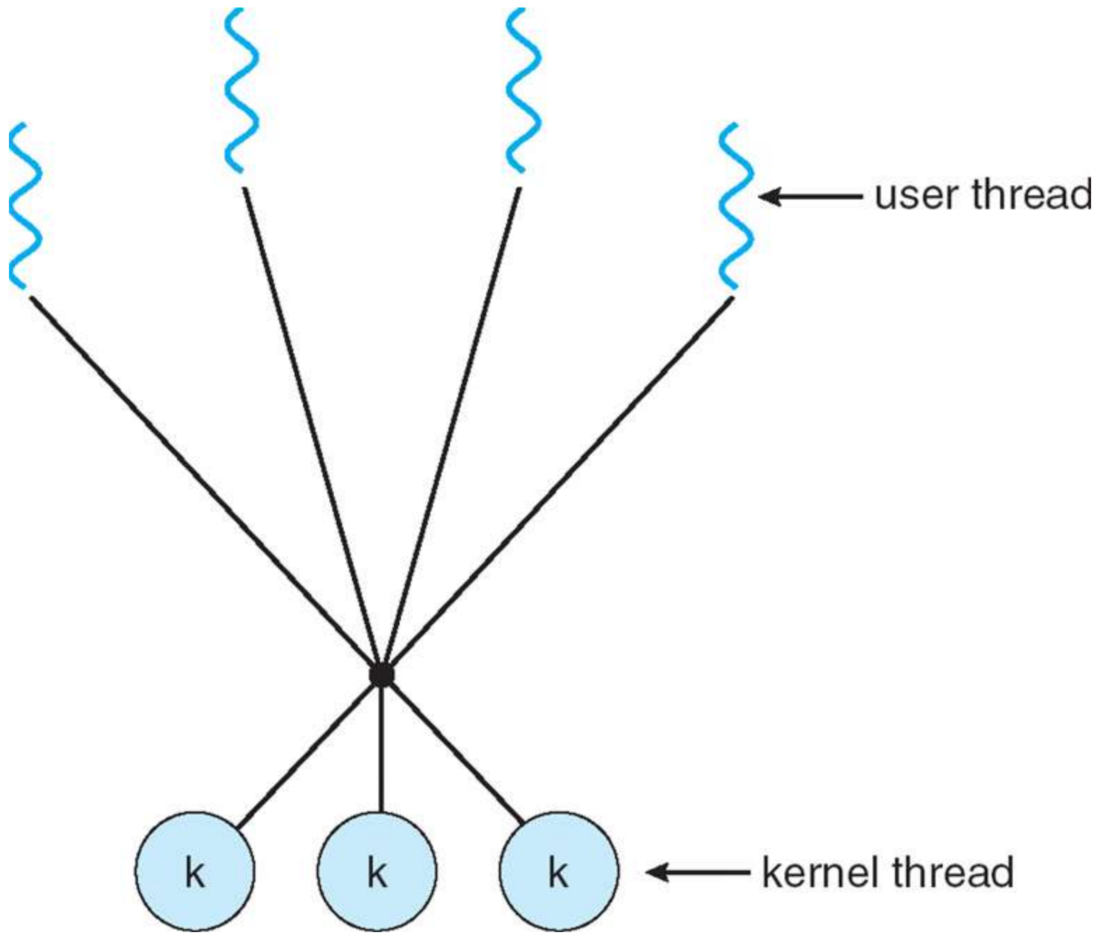
- In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.
- This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Many-to-Many Model

- A number of user-level threads are associated to an equal or smaller number of kernel-level threads.

- Allows many user level threads to be mapped to many kernel threads

Many-to-Many Model



Thread Cancellation

- **Thread cancellation** is the task of terminating a thread before it has completed.
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.

End of Chapter

Chapter : Process Synchronization

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Background

- **Co-Operating Process: that can affect or be affected by other processes executing in system**
- Concurrent access to shared data may result in data inconsistency
- **Process Synchronization: Ensures coordination among processes and maintains Data Consistency**

- **Process P1** **Process P2**

1. X=5

2. X=5+2

1. read(x);

2. x=x+5;

3. Printf(x);

Producer Consumer Problem

There can be two situations:

1. **Producer Produces Items at Fastest Rate Than Consumer Consumes**

2. **Producer Produces Items at Lowest Rate Than Consumer Consumes**

Producer Consumer Problem

1. **Producer Produces Items at Fastest Rate Than Consumer Consumes:**

If Producer produces items at fastest rate than Consumer consumes Then Some items will be lost

Eg. Computer → Producer

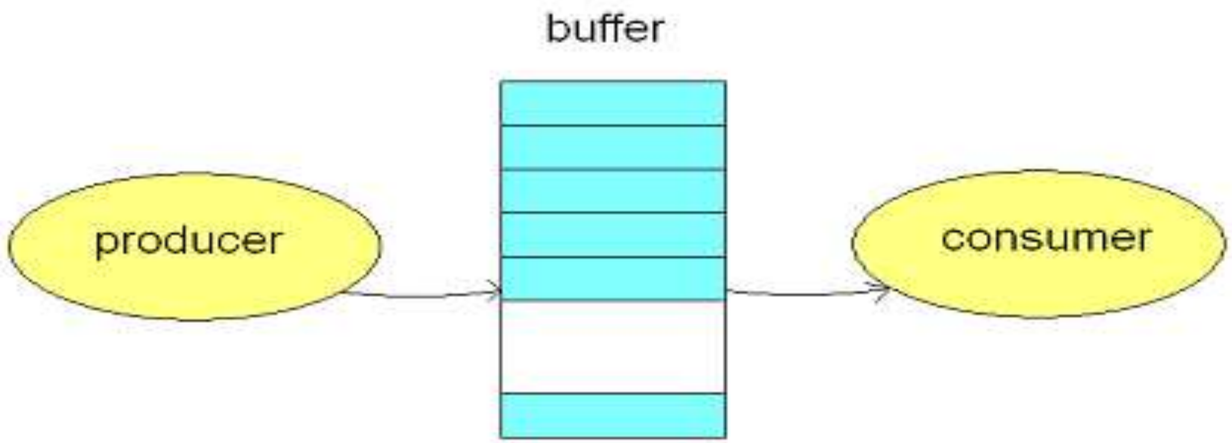
Printer → Consumer

Producer Consumer Problem

Solution:

To avoid mismatch of items Produced or Consumed → Take Buffer

Idea is: Instead of sending items from Producer to Consumer directly→ Store items into buffer



Producer Consumer Problem

Buffer Can be:

1. Unbounderd Buffer:
   1. No buffer size limit
   2. Any no. of items can be stored
   3. Producer can produce on any rate, there will always be space in buffer

2. Bounderd Buffer:
   1. Limited buffer size

Producer Consumer Problem

Bounderd Buffer:

If rate of Production > rate of Consumption:

Some items will be unconsumed in buffer

If rate of Production < rate of Consumption:

At some time buffer will be empty

Producer

```
while (true) {

    /*  produce an item and put in nextProduced  */

        while (count == BUFFER_SIZE)
                ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)              // buffer empty
                {
                        ; // do nothing
                 }
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count- -;

                        /*  consume the item in
 nextConsumed
 }
```

count=0 → buffer is empty

Producer

count=count+1

1. R1=count
2. R1=R1+1
3. Count = R1

Consumer

count=count-1

4. R2=count
5. R2=R2-1
6. Count = R2

Order of execution is not defined: May be

1→ 2→ 4→ 5→ 3→ 6

Race Condition

- When multiple processes access and manipulate the same data at the same time, they may enter into a race condition.
- **Race Condition: When output of the process is dependent on the sequence of other processes.**
- Race Condition occurs when processes share same data

- Process P1

1. reads i=10
2. i=i+1 =11

PI got INTERRUPTED

3. Stores i=11 in memory

Process P2

1. P2 reads i=11 from memory
2. i=i+1 = 12
3. Stores 12 in memory

Critical Section Problem

- Section of code or set of operations, in which process may be changing shared variables, updating common data.
- A process is in the critical section if it executes code that manipulate shared data and resources.
- Each process should seek permission to enter its critical section→ **Entry Section**
- **Exit Section**
- **Remainder section:** Contains remaining code

Structure of a process

```
Repeat
{
    // Entry Section

    Critical Section

    // Exit Section

Remainder Section
}   until false.
```

Solution to: Critical Section

1. Mutual Exclusion:

It states that if one process is executing in its critical section, then no other process can execute in its critical section.

2. Bounded Wait:

It states that if a process makes a request to enter its critical section and before that request is granted, there is a limit on number of times other processes are allowed to enter that critical section.

Pi made request for Critical Section

Pj is already executing in its Critical Section…… Pi has to wait

Solution to: Critical Section

3. Progress:

It states that process cannot stop other process from entering their critical sections, if it is not executing in its CS.

Decision of entry into Critical Section is made by→ Processes in **Entry Section**

How? →→→ Set Lock in Entry Section

Once process leaves C.S, Lock is released (i.e in Exit Section)

Solution to: Critical Section

3. Progress:

- N processes share same Critical Section

- Decision about which process can enter the Critical Section is based on Processes which are interested to enter the C.S

(i.e. Processes of Entry Section)

- Processes that do not want to execute in CS should not take part in decision

S/w Approaches/Solutions For C.S

- **Consider a Shared Variable,** that can take 2 values 0 and 1

  If (shared variable == 0)

  P0 can enter to Critical Section

  If (shared variable == 1)

  P1 can enter to Critical Section

S/w Approaches/Solutions For C.S

- Let **turn=0**

P0	**P1**

While (1)
{
While(turn !=0);// if true process will be stuck in the loop
CS
turn= 1;
}
	While (1)
	{
	While(turn !=1);// if true process will be stuck in the loop
	CS
	turn= 0;
	}

S/w Approaches/Solutions For C.S

- **Mutual Exclusion satisfied**
- **Progress**

Now, Turn =1 i.e. process P1 can enter C.S.

But

Suppose P1 does not want to enter C.S.

So.. turn =1 is never set

Strict alteration is there.

Progress is not satisfied

2nd S/W Approach

Instead of one shared variable.

Assume a flag can be true or false.

- If p0 wants to enter into CS it will set its flag true and when it doesn’t want to enter flag will be false.

2nd S/W Approach

P0	P1

While(1)
{
Flag[0]=true;
while(Flag[1]);
CS
Flag[0]=false;
}
	While(1)
	{
	Flag[1]=true;
	while(Flag[0]);
	CS
	Flag[1]=false;
	}

2nd S/W Approach

- Follows mutual exclusion
- Doesn’t follow progress
- System may go into deadlock state.

Peterson's Solution

- 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!

Algorithm for Peterson's solution

P0	P1

While(1)
{
Flag[0]=true;
turn=1;
while(turn==1 &&
flag[1]==true);
CS
Flag[0]=false;
}
	While(1)
	{
	Flag[1]=true;
	turn=0;
	while(turn==0 &&
	flag[0]==true);
	CS
	Flag[1]=false;
	}

- Mutual exclusion satisfied.
- Progress satisfied.
- Bounded wait satisfied.

Synchronization Hardware

Hardware Solution to C.S.

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
- Disabling interrupt in multiprocessor environment can be time consuming , as message is passed to all the processors.
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Solution to C.S.

- **Interrupt Disabling**

1. **In Single CPU system, processes do not execute in parallel**
2. **Process leaves control of CPU when it is interrupted.**
3. **Solution is:**
   1. **To have each process disable all interrupts just after entering to the critical section.**
   2. **Re-enable interrupts after leaving critical section**

Hardware Solution to C.S.

- **Interrupt Disabling**

Repeat

Disable interrupts

C.S

Enable interrupts

Remainder section

Hardware Solution to C.S.

- **Hardware instructions**

1. Machines provide inst. That can read, modify and store memory word
2. Common inst. are:
    1. **Test and Set**

This inst. Provides action of testing a variable and set its value

It executes atomically

Test and Set instructions operates on single Boolean variable.

TestAndndSet Instruction

- Definition:

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

Solution using TestAndSet

- Shared Boolean variable lock., initialized to false.
- Solution:

Test and Set definition	

boolean TestAndSet (boolean *lock)
{
boolean init = lock;
lock= TRUE;
return init;
}
	while (true)
	{
	while (TestAndSet (&lock));
	
	// critical section
	
	lock = FALSE;
	
	// remainder section
	
	}

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

Definition	

void Swap (boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp:
}
	while (true) {
	key = TRUE;
	while (key == TRUE)
	{
	Swap (&lock, &key);
	}
	// critical section
	
	lock = FALSE;
	
	// remainder section
	
	}

Bounded waiting mutual exclusion with TestAndSet

- do

```
{ waiting[i] = true;
key = true;
while (waiting[i] && key)
 key = test_and_set(&lock);
waiting[i] = false;
/* critical section */
 j = (i + 1) % n;
 while ((j != i) && !waiting[j])
j = (j + 1) % n;
if (j == i)
lock = false;
else waiting[j] = false;    /* remainder section */
}
while (true);
```

Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

```
while S <= 0
    ; // no-op
  S--;
}
```

  - signal (S) {

```
  S++;
}
```

- We can use semaphore to solve various synchronization problem.
- Let P1 and P2 are two process
- P1 wants to execute S1 statement. And P2 wants to execute S2 statement.
- S1 is to be executed before S2

S1;

Signal(synch);

Wait(synch);

S2

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

    Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short

Semaphore Implementation with no 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.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block();  }
}
```

- Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P);  }
}
```

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

P_0	P_1
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.

Classical Problems of Synchronization

- 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
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {

        //   produce an item

     wait (empty);
     wait (mutex);

        //  add the item to the  buffer

     signal (mutex);
     signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

                //  consume the removed item

}
```

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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

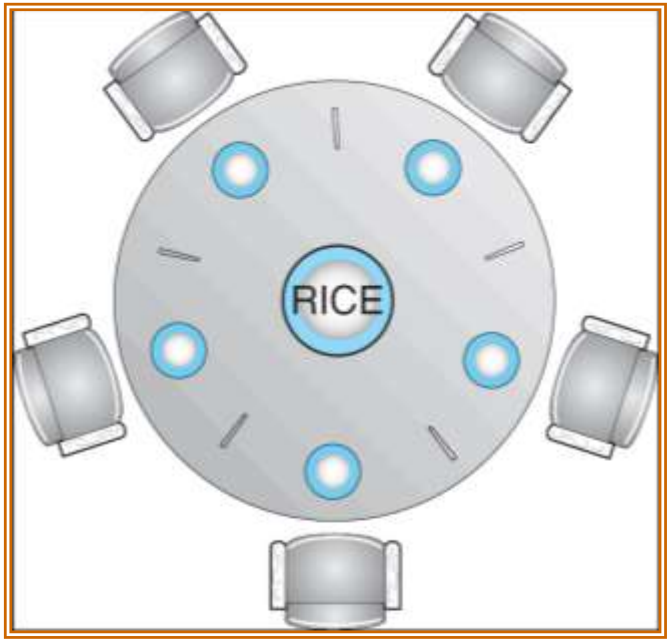
- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

                // reading is performed

         wait (mutex) ;
         readcount  - - ;
         if (readcount  == 0)  signal (wrt) ;
         signal (mutex) ;
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

              //  think

}
```

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)

Monitors

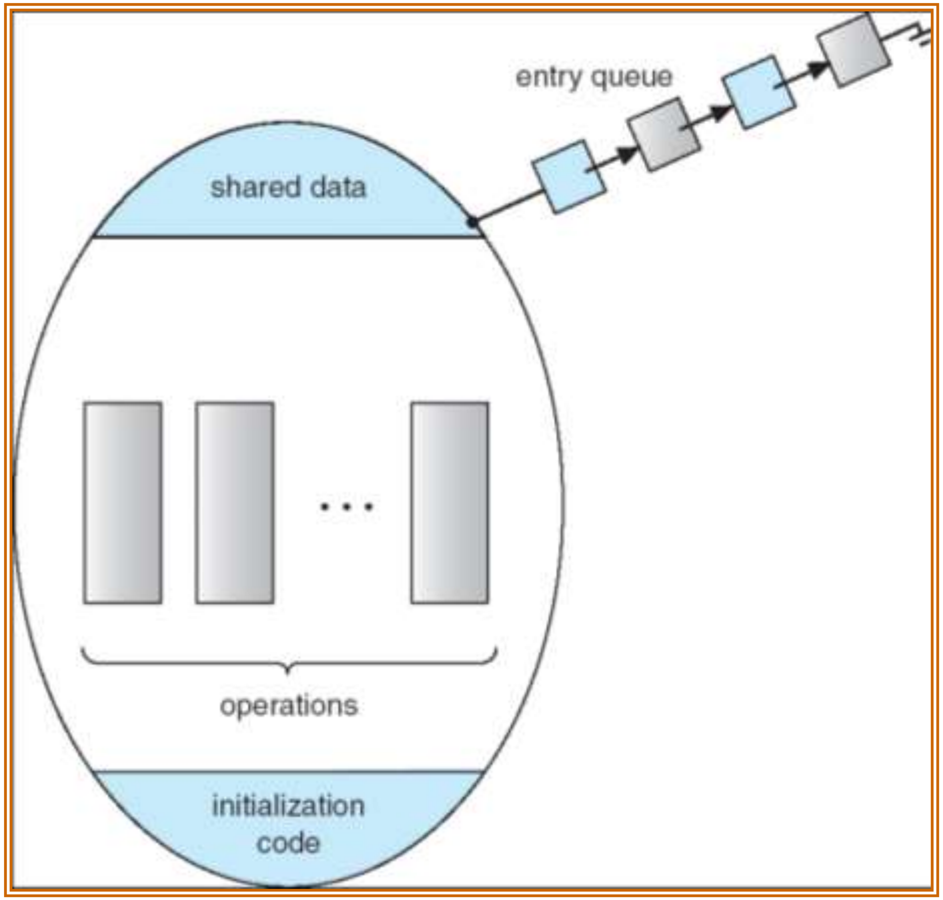
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

Schematic view of a Monitor

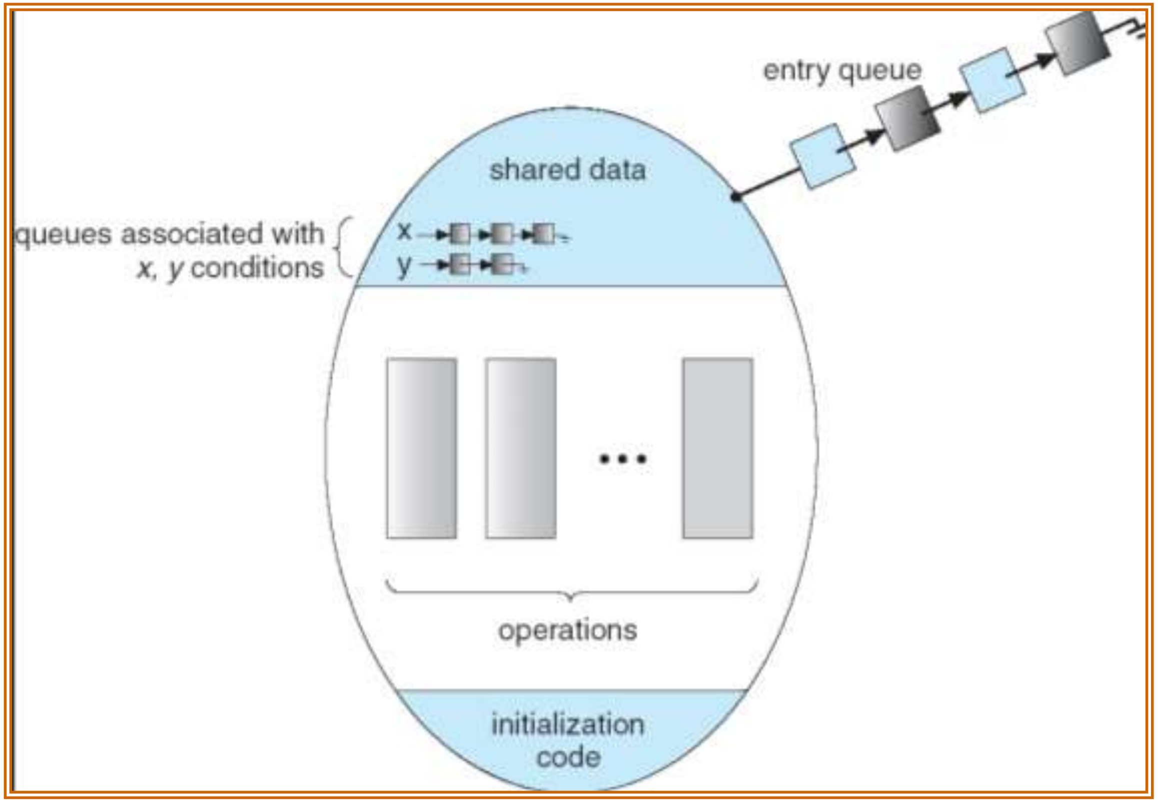


Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
  {
      enum { THINKING; HUNGRY, EATING) state [5] ;
      condition self [5];

      void pickup (int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING) self [i].wait;
      }

       void putdown (int i) {
            state[i] = THINKING;
                  // test left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
        }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
              self[i].signal () ;
         }
 }

 initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
}
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

dp.pickup (i)

EAT

dp.putdown (i)

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
        body of F;

    …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

End of Chapter 6