Great Place To Work® Certified FEB 2023 – FEB 2024 INDIA

UPES

UNIVERSITY OF TOMORROW

nirf RANKING 2023

#RANKED 52 IN INDIA

#University Category

QS WORLD UNIVERSITY RANKINGS

NO.1 PVT. UNIVERSITY IN ACADEMIC REPUTATION IN INDIA

ACCREDITED WITH GRADE A NAAC

ACCREDITED **GRADE 'A'** BY NAAC

E-LEAD

PERFECT SCORE OF **150/150** AS A TESTAMENT TO EXCEPTIONAL E-LEARNING METHODS

Unit 3 : Inter Process Communication and Synchronization

Dr. Abhijit and Mr. Deepak

School of Computer Science
UPES, Dehradun
India

Unit 3: Inter-Process Communication

L1	**Inter Process Communication (IPC), IPC mechanisms: Shared Memory and Message Passing**
L2	Message Passing: Shared Memory, Pipes and Named pipes in Linux
L3	Critical Section Problem, Race Condition,
L4	Producer Consumer Problem, Solution to Critical section Problem: Hardware and Software Solutions,
L5	Software Solutions: Semaphores: Counting semaphore, Binary semaphore,
L6	Software Solutions: Monitors
L7	Software Solutions: Algorithm 1, Algorithm 2,
L8	Software Solutions: Algorithm 3/Peterson Solution, Bakery Algorithm
L9	Classic process synchronization problems (case studies).

UPES
UNIVERSITY OF TOMORROW

Unit 3: Inter-Process Communication

Lecture 1	**Inter Process Communication (IPC), IPC mechanisms: Shared Memory and Message Passing**

Objective:

- To discuss the communication mechanism among processes.
- To explore different approach to achieve inter process communication

Interprocess Communication

- Interprocess Communication (IPC) refers to the mechanisms provided by the operating system that allow processes to communicate with each other and synchronize their actions.
- IPC is essential for the development of complex software systems where multiple processes need to work together and share data.
- Here are some common IPC mechanisms:

  i) Pipes

  ii) Message Passing

  iii) Shared Memory

  iv) Socket

  v) Semaphores

  vi) Remote Procedure Call (RPC)

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models

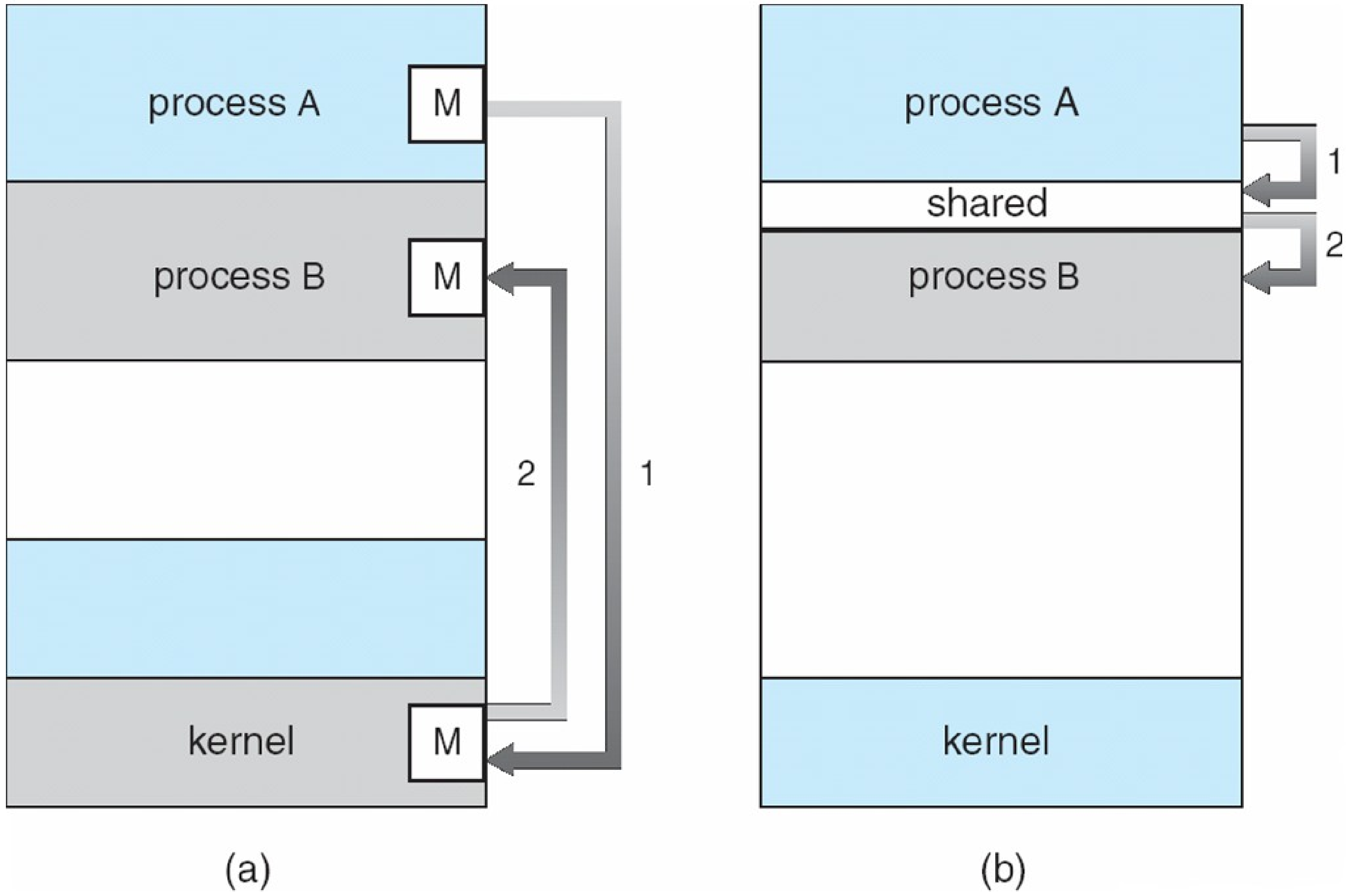


Fig 3.1 Interprocess communication model

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Examples of Cooperating Processes

- Web Servers: Multiple processes handle different client requests, sharing resources like memory and network connections.
- Database Systems: Processes collaborate to manage transactions, maintain data consistency, and ensure concurrency control.
- Distributed Systems: Processes running on different machines communicate and cooperate to perform distributed computations, share resources, and synchronize actions.
- Operating Systems: System processes cooperate to manage hardware resources, perform I/O operations, and execute user programs

UPES
UNIVERSITY OF TOMORROW

Benefits of Cooperating Processes

- Cooperating processes offer numerous benefits, including resource sharing, computation speedup, modularity, information sharing, improved reliability, enhanced communication, flexibility, and scalability.
- These advantages make cooperating processes essential in modern computing environments, particularly in multi-core, distributed, and real-time systems.

Lecture 2	Message Passing: Shared Memory, Pipes and Named pipes in Linux

Objective:

- To understand the concepts of Interprocess Communication through Message passing
- To understand the concepts of Interprocess Communication through Shared memory
- To understand the concepts of Interprocess Communication through Pipes

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Interprocess Communication – Message Passing

- Examples of Message Passing in Different Contexts

1. Message Passing Interface (MPI) in Parallel ComputingMPI is a standardized and portable message-passing system designed for parallel computing.

An example using MPI in C

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);  // Initialize MPI environment
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  // Get rank of the current process
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // Get total number of processes
    int number;
    if (world_rank == 0) {
        // Process 0 initializes the number
        number = -1;
        // Process 0 sends the number to process 1
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (world_rank == 1) {
        // Process 1 receives the number from process 0
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Process 1 prints the received number
        printf("Process 1 received number %d from process 0\n", number);
    }
    MPI_Finalize();  // Finalize MPI environment
    return 0; }
```

UPES
UNIVERSITY OF TOMORROW

Pipes

- Acts as a conduit allowing two processes to communicate
- **Issues**
  - **Is communication unidirectional or bidirectional?**
  - **In the case of two-way communication, is it half or full-duplex?**
  - **Must there exist a relationship (i.e. parent-child) between the communicating processes?**
  - **Can the pipes be used over a network?**

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

Ordinary Pipes

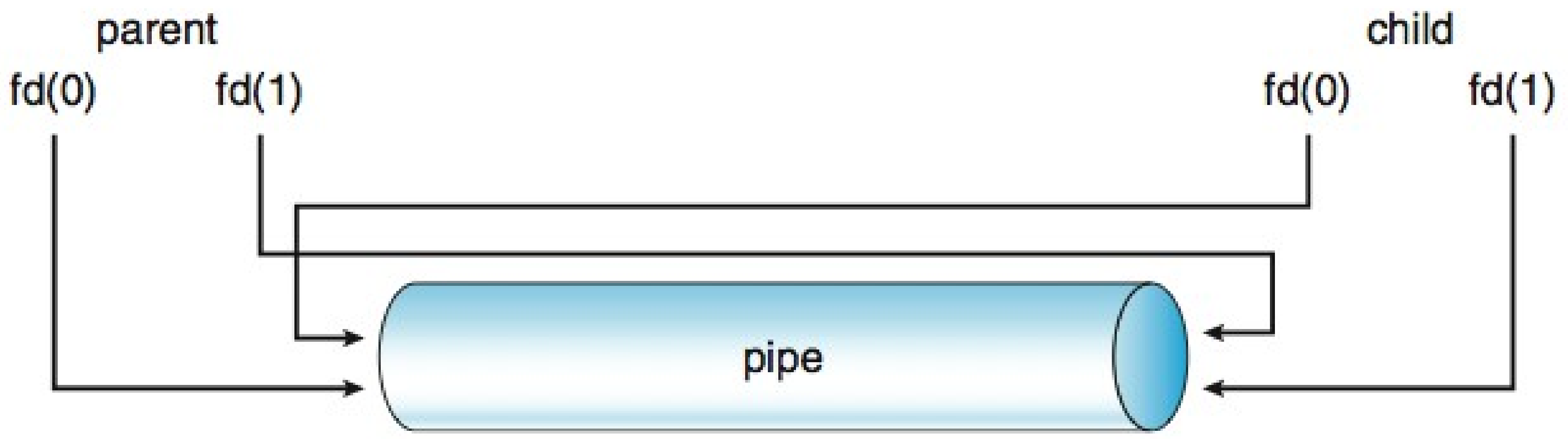


Fig: Ordinary Pipes

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

IPC mechanism in Pipes

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int fd[2];
    pid_t pid;
    char buffer[30];
    pipe(fd);
    pid = fork();

    if (pid == 0) { // Child process
        close(fd[0]); // Close unused read end
        write(fd[1], "Hello, parent!", 14);
        close(fd[1]); // Close write end
    } else { // Parent process
        close(fd[1]); // Close unused write end
        read(fd[0], buffer, 14);
        printf("%s\n", buffer);
        close(fd[0]); // Close read end
    }
    return 0;
}
```

L3	Critical Section Problem, Race Condition

Objectives:

- To understand the concepts of Critical Section problem
- To understand the race condition

Introduction

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Example: The producer-consumer problem.
    - Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race Condition:

•A race condition occurs when multiple processes access and manipulate shared data concurrently, leading to unexpected results.

•Example: Incrementing and decrementing a shared counter.

UPES
UNIVERSITY OF TOMORROW

Race Condition

- count++ could be implemented as

    register1 = count
    register1 = register1 + 1
    count = register1

- count-- could be implemented as

    register2 = count
    register2 = register2 - 1
    count = register2

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

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

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)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
              count--;

                  /*  consume the item in
nextConsumed
}
```

Solution to Critical-Section Problem

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
   - No assumption concerning relative speed of the N processes

L4	Producer Consumer Problem, Solution to Critical section Problem: Hardware and Software Solutions,

Objectives:

- To understand the classical problems of Critical Section
- To device and develop the solution related to classical problems of Critical Section

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- 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!

UPES UNIVERSITY OF TOMORROW

Algorithm for Process P_i

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
} while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

TestAndSet Instruction

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

Solution using TestAndSet

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

```
do {
while ( TestAndSet (&lock ))
        ;   // do nothing

      //    critical section

lock = FALSE;

      //     remainder section

} while (TRUE);
```

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:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                  //    critical section

         lock = FALSE;

                 //     remainder section

 } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```

L5	Software Solutions: Semaphores: Counting semaphore, Binary semaphore,

Objective

- To understand the concept of semaphores and their types
- To implement the different types of Semaphores

Semaphore

- Semaphores are a synchronization mechanism used to control access to a common resource in concurrent programming and operating systems.
- They are particularly useful for managing access to shared resources and ensuring that multiple processes or threads do not simultaneously access a critical section, leading to race conditions or inconsistent data.

Basic Operations: Semaphores

Support two atomic operations:

- wait (P or down): Decreases the semaphore value. If the value is already 0, the process or thread is blocked until the semaphore value becomes positive.
- signal (V or up): Increases the semaphore value. If there are any processes or threads waiting for the semaphore, one of them is unblocked.

```
wait(S):
    while S <= 0:
        // Busy wait (or block the process/thread)
    S = S - 1


signal(S):
    S = S + 1
```

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 mutex;    //  initialized to 1
do {
   wait (mutex);
        // Critical Section
    signal (mutex);
                // remainder section
} while (TRUE);
```

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
    - 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 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(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Use of 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 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
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

L6	Software Solutions: Monitors

Objective:

- To explore the software solutions using Monitors

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

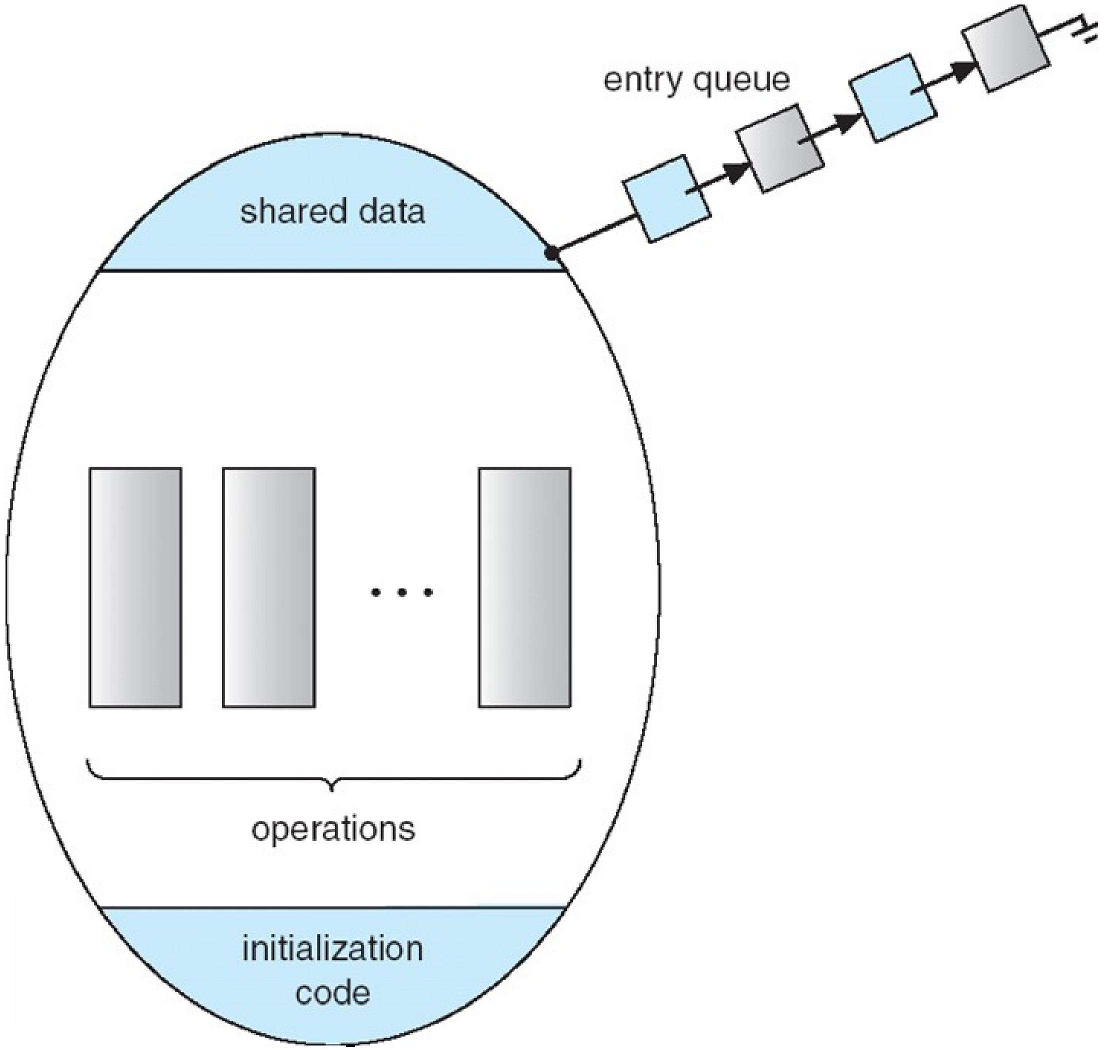


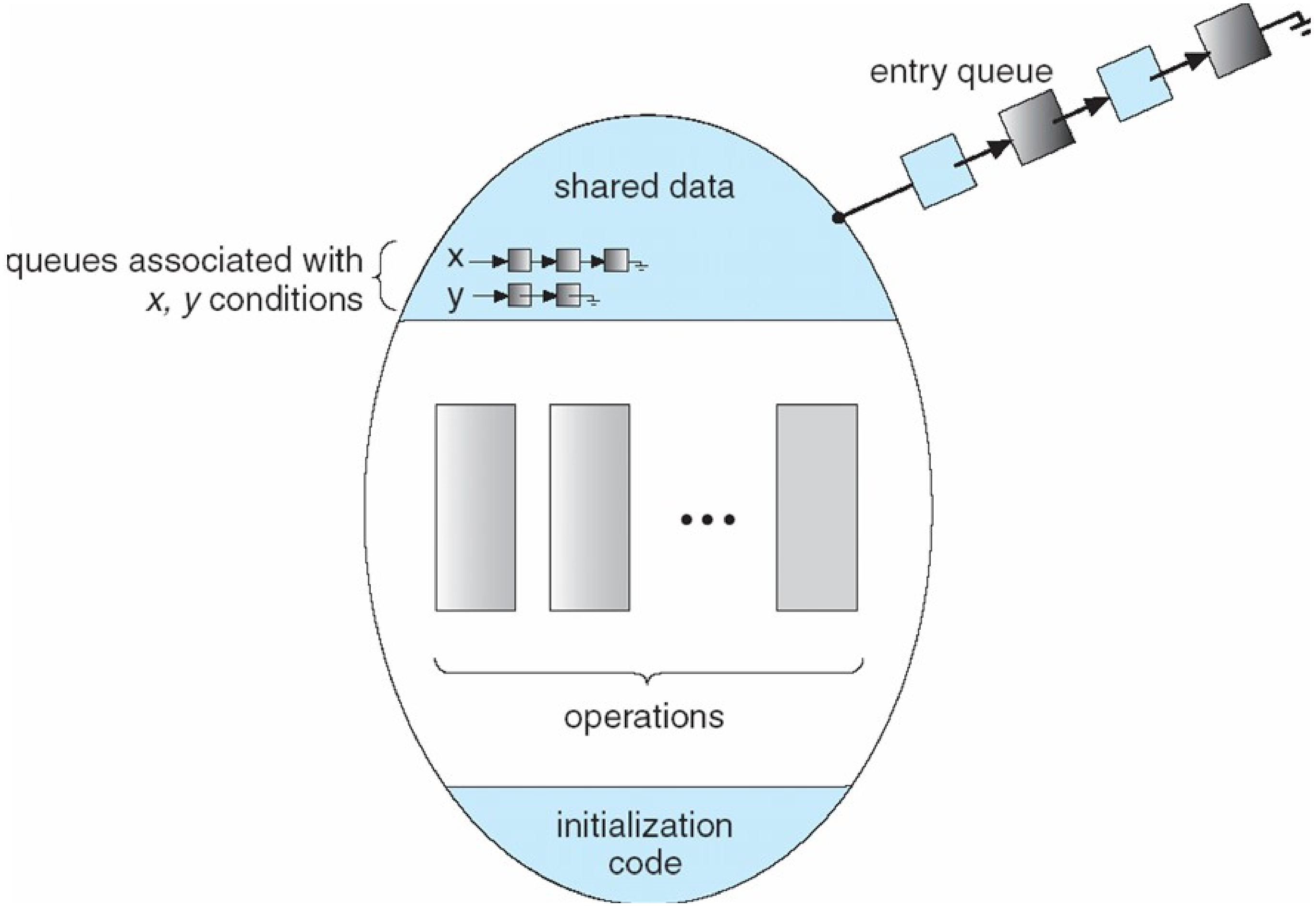
Fig: Schematic View of Monitor

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 using Monitor

```
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);
       }
```

UPES
UNIVERSITY OF TOMORROW

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:

DiningPhilosophters.pickup (i);

EAT

DiningPhilosophers.putdown (i);

UPES
UNIVERSITY OF TOMORROW

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--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
                if (busy)
                        x.wait(time);
                busy = TRUE;
    }
    void release() {
                busy = FALSE;
                x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

L7 & 8 Software Solutions: Peterson Solution, Bakery Algorithm

Objective:

- To understand the software solutions using various algorithms

Peterson’s Solution

- Two-process solution
- Assumptions:
  - LOAD and STORE instructions are atomic (cannot be interrupted).
- Shared variables:
  - int turn;
  - boolean flag[2];
- Explanation:
  - turn indicates whose turn it is to enter the critical section.
  - flag array indicates if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready.

Algorithm for Process Pi

```
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  // Critical section
  flag[i] = false;
  // Remainder section
} while (true);
```

Bakery Algorithm

- N-process solution
- Assumptions:
  - The integers are assumed to be unbounded.
- Shared variables:
  - boolean choosing[N];
  - int number[N];
- Explanation:
  - choosing[i] indicates if process Pi is in the process of choosing a number.
  - number[i] holds the number chosen by process Pi. A higher number indicates a higher priority.

Bakery Algorithm Cont..

```
do {
  choosing[i] = true;
  number[i] = 1 + max(number[0], number[1], ..., number[N-1]);
  choosing[i] = false;
  for (j = 0; j < N; j++) {
    while (choosing[j]);
    while ((number[j] != 0) && ((number[j], j) < (number[i], i)));
  }
  // Critical section
  number[i] = 0;
  // Remainder section
} while (true);
```

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

UPES
UNIVERSITY OF TOMORROW

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

UPES
UNIVERSITY OF TOMORROW

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

UPES
UNIVERSITY OF TOMORROW

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i) <$ W-timestamp(Q), Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq$ W-timestamp(Q)
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If $TS(T_i)$ < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i)$ < W-tiimestamp(Q), T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

L9	Classic process synchronization problems (case studies)

Objective:

- To understand the classic Synchronization examples and case studies

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- Linux provides:
  - semaphores
  - spin locks

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

```
do {

        //   produce an item in nextp

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer to nextc

    signal (mutex);
    signal (empty);

        //  consume the item in nextc

} 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
- 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

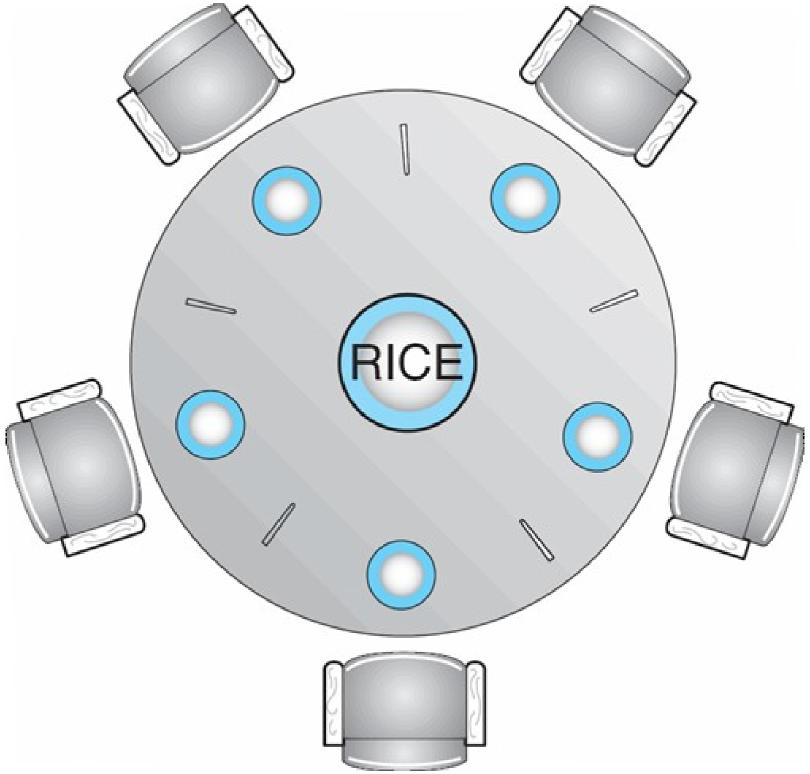
```
do {
    wait (wrt) ;

        //    writing is performed

    signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                    wait (wrt) ;
        signal (mutex)
           // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                    signal (wrt) ;
        signal (mutex) ;
   } while (TRUE);
```

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*:

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

            //  eat

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

            //  think

} while (TRUE);
```

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)

Multiple-Choice Questions (MCQs)

1) Which of the following is NOT a mechanism for Inter-Process Communication (IPC)?

A) Shared Memory

B) Message Passing

C) Pipelining

D) Remote Procedure Calls (RPC)

2) In message passing, what is the function of the send(message) operation?

A) To receive a message from another process

B) To send a message to another process

C) To create a shared memory space

D) To initialize message buffers

3) What is a race condition?

A) A condition where processes race to complete tasks

B) A scenario where two processes compete for CPU time

C) A condition where the outcome depends on the non-deterministic ordering of processes

D) A situation where processes run in a round-robin manner

4) Which of the following is NOT a requirement for the critical-section prob solution?

A) Mutual Exclusion

B) Progress

C) Bounded Waiting

D) Unbounded Execution

5) What is the primary role of a semaphore in process synchronization?

A) To allocate CPU time to processes

B) To manage memory allocation

C) To ensure mutual exclusion and synchronization

D) To handle I/O operations

6) Which synchronization mechanism uses wait() and signal() operations?

A) Locks

B) Semaphores

C) Monitors

D) Barriers

Multiple-Choice Questions (MCQs)

7) In Peterson's solution for the critical-section problem, what does the turn variable signify?

A) The process currently in the critical section
B) The process allowed to enter the critical section next
C) The number of processes waiting to enter the critical section
D) The priority of processes

8) What is a monitor in the context of process synchronization?

A) A low-level synchronization primitive
B) A high-level synchronization abstraction
C) A type of semaphore
D) A hardware-based synchronization mechanism

9) Which of the following problems is an example of a classic synchronization problem?

A) Memory Allocation Problem
B) Process Scheduling Problem
C) Dining Philosophers Problem
D) Disk Scheduling Problem

10) Which of the following operating systems uses adaptive mutexes for synchronization?

A) Windows XP
B) Solaris
C) Linux
D) MacOS

Answers:

1. C) Pipelining
2. B) To send a message to another process
3. C) A condition where the outcome depends on the non-deterministic ordering of processes
4. D) Unbounded Execution
5. C) To ensure mutual exclusion and synchronization
6. B) Semaphores
7. B) The process allowed to enter the critical section next
8. B) A high-level synchronization abstraction
9. C) Dining Philosophers Problem
10. B) Solaris

UPES

UNIVERSITY OF TOMORROW

Thank You

UPES

UNIVERSITY OF TOMORROW