OSL LAB MANUAL

**Assignment No. 1 :**

A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, seed etc.

B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

**Assignment No. 2:**

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states

1. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.
2. B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order

**Assignment No. 3:**

Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

**Assignment No. 4:**

A. Thread synchronization using counting semaphores. Application to demonstrate: producerconsumer problem with counting semaphores and mutex.

B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: ReaderWriter problem with reader priority

**Assignment No. 5:**

Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

**Assignment No. 6:**

Implement the C program for Page Replacement Algorithms: FCFS, LRU, andOptimal for frame size as minimum three.

**Assignment No. 7:** Inter process communication in Linux using following.

A. FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

**Assignment No. 8:**

Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

Study Assignment: Implement a new system call in the kernel space, add this new system call in the Linux kernel by the compilation of this kernel (any kernel source, any architecture and any Linux kernel distribution) and demonstrate the use of this embedded system call using C program in user space.

**Assignment No: 01**

**Title: Process Control**

**Assignment No. 1 :**

A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

**AIM:**

* **Implement addition, subtraction, multiplication and division using required control  structures**
* **Convert a string from upper case to lower case or vice versa**

**OBJECTIVE:**

Understanding of Linux shell commands & shell programming.

**THEORY:**

* **Linux Shells**

Linux comes with various command interpreters called as shells in the Unix terminology. Actually shell sits in between the kernel of an operation system and the user. So whatever user wants to do through kernel it is available in terms of shell commands. Once you provide valid command for the required operation it hands over the request to the operating system kernel and finally job will be done by the system.

There are various shells available to use in the Linux environment but following shells are the standard shells.

**Linux Shells**

**Bourne Shell**

**C Shell**

**Korn Shell**

**Bash**

**tcsh**

The shells used in the Linux operating system has dual capability, in one had it is used as a tool which accepts commands interpret it and hands over it to the operating system kernel. Due to this capability it is called as command – line interpreter, another use of shell is it can be used as a programming language. Shell programming is interpretive by nature and mostly it is used to assist in system administration tasks.

* **Steps to interpret a shell script**

Assume a written shell script is stored in a script file named as **Example-1.** To execute this script we have two approaches: ***Make it as an executable*** and ***execute it as an argument to bash*** command.

Make it as an executable :

* By the use of **chmod** command one can set execute permission on it. It is shown below for our first script.

**chmod +x Example-1**

* This script can be executed like any other shell command after associating the execute permission.
* The following command shows the execution step for our **Example-1** file.

**./Example-1**

**Running it as an argument of bash :**

This approach of script execution can be done to specify its name as the argument of the bashcommand.

* For example, the script named as **Example-1** can be executed by the use of following

command.

**bash Example-1**

* **Basic Shell Commands**
* **read**

The command read reads one line from the standard input (or from a file) and assign the word to the variable.

**Example**

read var\_year   
echo "The year is: $var\_year"

echo -n "Enter your name and press [ENTER]: "   
read var\_name  
echo "Your name is: $var\_name"

* **echo**

echo command in the bash shell writes its arguments to standard output**.**

The syntax for echo is

echo [option(s)] [string(s)]

The items in square brackets are optional. A *string* is any finite sequence of *characters* (i.e., letters, numerals, symbols and punctuation marks).

When used without any options or strings, echo returns a blank line on the display screen followed by the command prompt on the subsequent line.

For example, a variable named *x* can be created and its value set to 5 with the following command:

x=5

The value of x can subsequently be recalled by the following:

echo The number is $x.

* **expr**

It is an old Unix program that can evaluate math is **expr**. Since it is a command, command substitution is needed.

**Example**

z=`expr $z + 1`

echo $z

* **Control Structures in shell**
* The bash shell provides many options to control the flow of script executions, typically known as control structures. The Table below list out these structures.

|  |  |
| --- | --- |
| **If** | If statement is used to execute commands based on certain conditions are met. It can be customized by the use of else to indicate what should happen if the condition does not satisfies. |
| **Case** | This is used to handle multiple options in the script. It is same like switch statements used in C programming language. |
| **For** | This is a loop statement. The *for* statement is used to run a command for a given number of items. |
| **While** | Use while statement to execute the included statements as long as the specified condition is met. |
| **Until** | This statement works exactly opposite of while statement. This can be used to execute a command until a certain condition is met. |

If...Then...Else :

* The if…then…else construct is used for flow control facility in shell scripting. This is usually used in conjunction with test command.
* This can be used for various activities like to find out the existence of a file, whether a variable currently has a value etc. The basic construction of if…then…else… statement is shown in below program code :

General description of if…then construct

if condition expression

then

command(s)

fi

* This means one can use it to check one specific condition, and in case of true the command or group of commands associated will be executed.
* The if…then…fi construct can be extended by including else statement to handle all cases where the condition was not met. The Fig. below shows the actual syntax of the construct along with test command.

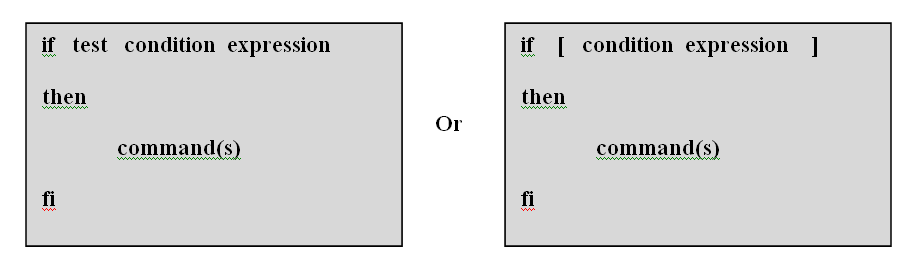


Fig. If…then…fi statement

* **Syntax of if…else statement :**

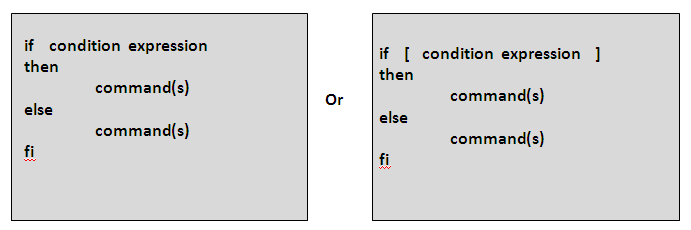
 The else can be included in the basic if statement in the fashion shown Fig

Fig.If…then…else statement

* **Syntax of multiple else blocks using elif :**

The if statement can have elif construct in case of multiple else statements included along with the if statement.

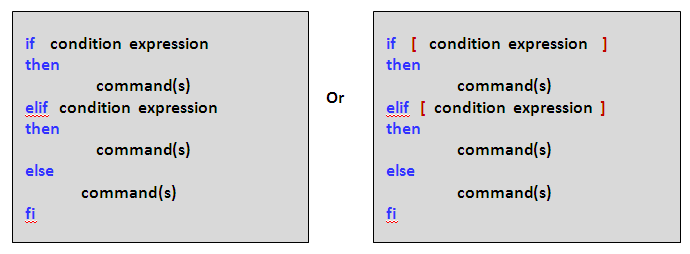


Fig.: elif statement

Case Statement :

* The case statement is a multiple branch decision mechanism. The syntax of case statement is described in the following figure as well as its working is shown in the below   
  Fig.

|  |
| --- |
| case **word** in  pattern1)  **command(s)**  ;;  pattern2)  **command(s)**  ;;  .............  patternN)  **command(s)**  ;;  \*)  esac |

Fig. Syntax of case statement

While Statement :

* The while statement is described with the nature says : As long as the condition is true, the commands between the do and the doneare executed.
* The syntax of while statement is described in the Fig. below.

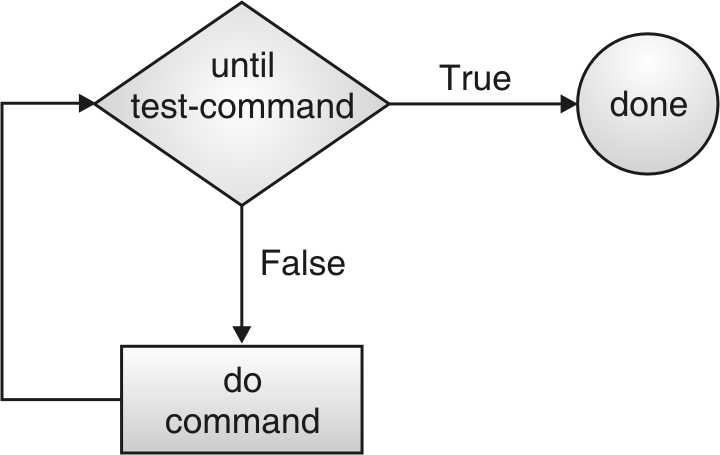
|  |
| --- |
| while **condition**  do  **command(s)**  done |

Fig. Syntax of while statement

Until Statement :

* The syntax of until statement is described in the following figure as well as its working is shown in the Fig
* This loop continues to execute the command(s) between the do and done until the condition is true.

|  |
| --- |
| until **condition**  do  **command(s)**  done |

Fig.: Syntax of until statement

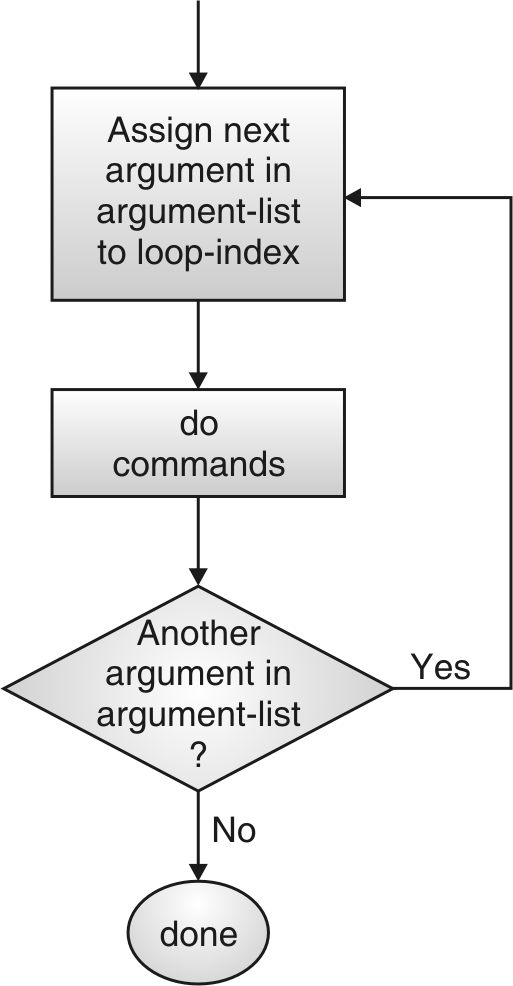
for…in Statement :

* The for…in control structure has the following syntax :

|  |
| --- |
| for **index** in **argument-list**  do  **command(s)**  done |

Fig: Syntax of for…in statement

* The **for…in** structure gets a value from the argument-list and assigns it into loop-index in every iterations and executes the command(s) between do and done statements. The do and done marks the beginning and end of the for loop statement.



Using for Control Statement :

* The for control structure has the following syntax :

|  |
| --- |
| for index  do  command(s)  done |

Fig. Syntax of for control structure

* In this for control structure the index variable takes value of the each of the command line arguments, one at a time.
* This structure is same as for…in structure; the only difference is here the loop index gets value from command line arguments instead of the separate argument list provided along with for.

Using for((…)) Loop Statement :

* The for loop is used to execute a series of commands, either for a limited number of times or for an unlimited number of times. This works like high level programming languages for loops.

|  |
| --- |
| for ((**init ; condition ; increment**))  do  **command(s)**  done |

Fig: Syntax of for((…)) statement

* **INPUT: please specify input test case**
* **OUTPUT: please specify the expected output for the input test case**
* **FAQS:**

1. **Why does #!/bin/sh have to be the first line of my scripts?**

#!/bin/sh must be the first line in your script to ensure that the correct shell is used to execute your script. This line must be the first line in your shell script because of the underlying mechanism used by a shell to execute commands.

#### How can I access the name of the current shell in my initialization scripts?

In your shell initialization scripts, the name of the current shell is stored in the variable $0.

Users who have a single .profile that is shared by sh, ksh, and bash use this variable in conjunction with a case statement near the end of this file to execute additional shell–specific startups.

For example, you can use the following case statement in your .profile to set up the prompt, PS1, differently depending on the shell:

case "$0" in

\*bash) PS1="\t \h \#$ " ;;

\*ksh) PS1="´uname -n´ !$ " ;;

\*sh) PS1="´uname -n´$ " ;;

esac

export PS1

#### How can I determine whether a command executed successfully?

You can determine whether a command executed successful by checking the command's exit code, which the shell stores in the variable $?. By convention, the exit code of a successful command is 0. A nonzero exit code indicates a failure.

An if statement of the following form is often used to check whether a command executed successfully

cmd

if [ $? -eq 0 ] ; then

: # cmd successful

else

: # cmd failed

fi

* **CONCLUSION**

The Linux shell provides several commands and programming language constructs to implement various operations.

**Assignment No: 02**

**Title: Process Control**

**Problem statement:**

**Assignment No. 2:**

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states

1. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.
2. B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order

Write a program using fork to create a child process. The parent process should sort elements in ascending order and child process should sort elements in descending order.

**Objectives**:

* To understand the concept of Process, Process Image
* To understand System calls
* To use system call to create a new process

**Theory:**

A **process** is an instance of running program. A process is said to be born when the program starts execution, it remains alive as long as the program is active. After the execution is complete, the process is said to die or terminate. Program and process are two different entities, when two users run the same program; there is only one program on disk but two processes in memory.

**Kernel** is responsible for the management of the processes. It determines the time and priorities that are allocated to processes so that multiple processes are able to share CPU resources. It provides a mechanism by which a process is able to execute for a finite period of time and then relinquish control to another process. Kernel allocates space memory not only for process image but also for control information that enables it to switch the processes.

Every process has the various attributes associated with it, and they are maintained in memory by kernel, in a separate structure called process table. Some important attributes are

Process\_id (PID) : Each process is identified by a unique integer called Process\_id, that is allocated by the kernel when process is born. We need PID to control the process, e.g. to kill the process.

Parent\_id (PPID) : The PID of the parent is also available as a process attribute. It is also required to control the process. E.g. if several processes have same PPID, we just kill the parent, rather than to kill children separately.

State of Process: whether running, ready, zombie, waiting etc.

**System Call**: A routine defined in the kernel which performs the basic operations of the computers, like opening a file and creating a process. All UNIX commands and library functions are written in terms of system calls. Processor switches from user mode to kernel mode when executing a system call.

Mechanism to create a process: Creation of a new process, makes use of three important system calls or functions – fork, exec and wait.

**Fork**: Fork system call creates a copy of the process that invokes it. The process image is identical to that of calling process, except for few parameters like PID. When a new process is created, it gets new PID. PIDs, 0,1,2,and 3 are reserved and created by kernel only. After fork, parent and child have different PID and PPIDs. At this point there are two processes (parent and child), with practically identical constituents. They continue execution at the statement following fork (code before fork ignored by child). When a fork is invoked, kernel replicates the address space of the current process ( its text, data, stack etc.) It also creates a separate entry in the process table containing several fields copied from the entry of the parent. This includes the file descriptors, current directory etc. Because child runs into its own address space, changes made to these parameters don’t affect the parent. The size of the process table places a restriction on the total number of processes that a machine can support. If an attempt to fork a process violates this restriction, fork returns -1.

**Exec**:

It is a system call that gives a call to a brand new existing executable program.

eg: Program A (parent process) creates a child process B, and child process B gives a call to program C through exec. Thus program C overlays the address space of child process B. Thus, exec never returns any value.

**Wait**: The parent executes wait system call to wait for the child process to complete. Kernel clears the slot in the process table that was allocated to the child. Parent then picks up the exit status of the child and continues with statements following wait. A pointer to integer is passed to the wait system call, it .not only contains the exit status but other thing such as process state. Exit status is stored in least eight significant bits and WEXITSTATUS is the macro used to fetch the exit status.

Algorithm:-

* 1. Read the array size and read the array elements
  2. Create a child process using fork, which returns zero to child process and some nonzero positive integer to parent.
  3. In child process sort the array in descending order and print the sorted array
  4. In child process use exit to terminate child.
  5. In parent process use wait system call to collect the exit status of child and print the exit status of the child using a macro WEXITSTATUS.
  6. In parent process sort the array in ascending order and print the sorted array
  7. Terminate parent

Input :

Output :

**Conclusion**: Thus we have studied the concept of process control and different system calls

**FAQs**

1. Define Process
2. Define Kernel
3. Explain Address Space of process
4. Explain the term system call
5. Explain fork system call
6. Explain wait system call

**Problem statement:**

Write a program using fork to create a child process. The parent process should sort elements in ascending order and child process should sort elements in descending order.

**Objectives**:

* To understand the concept of Process, Process Image
* To understand System calls
* To use system call to create a new process

**Theory:**

A **process** is an instance of running program. A process is said to be born when the program starts execution, it remains alive as long as the program is active. After the execution is complete, the process is said to die or terminate. Program and process are two different entities, when two users run the same program; there is only one program on disk but two processes in memory.

**Kernel** is responsible for the management of the processes. It determines the time and priorities that are allocated to processes so that multiple processes are able to share CPU resources. It provides a mechanism by which a process is able to execute for a finite period of time and then relinquish control to another process. Kernel allocates space memory not only for process image but also for control information that enables it to switch the processes.

Every process has the various attributes associated with it, and they are maintained in memory by kernel, in a separate structure called process table. Some important attributes are

Process\_id (PID) : Each process is identified by a unique integer called Process\_id, that is allocated by the kernel when process is born. We need PID to control the process, e.g. to kill the process.

Parent\_id (PPID) : The PID of the parent is also available as a process attribute. It is also required to control the process. E.g. if several processes have same PPID, we just kill the parent, rather than to kill children separately.

State of Process: whether running, ready, zombie, waiting etc.

**System Call**: A routine defined in the kernel which performs the basic

operations of the computers, like opening a file and creating a process. All

UNIX commands and library functions are written in terms of system calls.

Processor switches from user mode to kernel mode when executing a system

call.

Mechanism to create a process: Creation of a new process, makes use of three important system calls or functions – fork, exec and wait.

**Fork**: Fork system call creates a copy of the process that invokes it. The process image is identical to that of calling process, except for few parameters like PID. When a new process is created, it gets new PID. PIDs, 0,1,2,and 3 are reserved and created by kernel only. After fork, parent and child have different PID and PPIDs. At this point there are two processes (parent and child), with practically identical constituents. They continue execution at the statement following fork (code before fork ignored by child). When a fork is invoked, kernel replicates the address space of the current process ( its text, data, stack etc.) It also creates a separate entry in the process table containing several fields copied from the entry of the parent. This includes the file descriptors, current directory etc. Because child runs into its own address space, changes made to these parameters don’t affect the parent. The size of the process table places a restriction on the total number of processes that a machine can support. If an attempt to fork a process violates this restriction, fork returns -1.

**Exec**:

It is a system call that gives a call to a brand new existing executable program.

eg: Program A (parent process) creates a child process B, and child process B gives a call to program C through exec. Thus program C overlays the address space of child process B. Thus, exec never returns any value.

**Wait**: The parent executes wait system call to wait for the child process to complete. Kernel clears the slot in the process table that was allocated to the child. Parent then picks up the exit status of the child and continues with statements following wait. A pointer to integer is passed to the wait system call, it .not only contains the exit status but other thing such as process state. Exit status is stored in least eight significant bits and WEXITSTATUS is the macro used to fetch the exit status.

Algorithm:-

* 1. Read the array size and read the array elements
  2. Create a child process using fork, which returns zero to child process and some nonzero positive integer to parent.
  3. In child process sort the array in descending order and print the sorted array
  4. In child process use exit to terminate child.
  5. In parent process use wait system call to collect the exit status of child and print the exit status of the child using a macro WEXITSTATUS.
  6. In parent process sort the array in ascending order and print the sorted array
  7. Terminate parent

Input :

Output :

**Conclusion**: Thus we have studied the concept of process control and different system calls

**FAQs**

1. Define Process
2. Define Kernel
3. Explain Address Space of process
4. Explain the term system call
5. Explain fork system call
6. Explain wait system call

**Assignment No: 03**

**Title: Process Control**

**Assignment No. 3:**

Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

**Assignment Statement:**

Write a menu driven program to implement following CPU scheduling algorithms

1) First come First Serve (FCFS)

2) Shortest Job First (SJF) (Preemptive)

3) Round Robin

**Objectives:**

* + To study theNeed for CPU scheduling
  + To do the comparative study of various CPU scheduling methods
  + Implementation of CPU Scheduling

**Theory:**

**Need:** In multiprogramming environment, multiple processes are maintained in the main memory. Each process alternates between the using a processor and waiting for I/O to be completed or for some other event to occur. The processor is kept busy by executing one process while others wait. **The key to multiprogramming is Scheduling**. The aim of Processor scheduling is to assign processes to be executed by the processor over time, in such a way that meets the system objectives, such as response time, throughput and processor efficiency.

Types of Scheduling:

Long Term Scheduling: It is invoked when new process is created. This decides whether to add a new process to the set/pool of processes that are currently active. **It controls the degree of multiprogramming**

Medium Term Scheduling: It is a part of swapping function. The decision is taken whether to add a process to those that are partially or fully available in memory and therefore available for execution.

Short Term Scheduling: It is the actual decision of which ready processes will be executed by the processor. CPU Scheduling comes under short term scheduling

**1) First Come First Serve (FCFS):-**

This is the simplest CPU scheduling algorithm. It is also known as first in first out (FIFO) or a strict queuing scheme. As each process becomes ready, it is added at the rear end of queue. When currently running process finishes, it is removed from the queue and the next process in the ready queue is selected for execution. FCFS performs much better for longer processes (having larger CPU burst time).

**Characteristics:**

**Selection Function:** max(w), selects the process which is waiting in the ready queuefor maximum time**.**

**Decision Mode: Non\_preemptive**

**Throughput: Not emphasized**

**Response Time:** May be high, especially if there is a large variance in the process Execution times.

**Overhead:** Minimum

**Effect on Processes:** Penalizes short and I/O bound processes.

**Starvation:** No

**2) Shortest Job First (SJF) Preemptive or Shortest Remaining Time:** It is a preemptive version of SJF. In this policy, scheduler always chooses the process that has the shortest expected remaining processing time. When a new process arrives in the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt whenever a new process becomes ready. Scheduler must have an estimate of processing time to perform the selection function.

**Characteristics:**

**Selection Function**: minimum total service time required by the process,

minus time spent in execution so far.

**Decision Mode : Preemptive ( At arrival time)**

**Throughput: High**

**Response Time:** Provides good response time

**Overhead: Can be high**

**Effect on Processes:** Penalizes long processes.

**Starvation:** Possible

**Data Structure Used:- Array of structures, having following fields in structure:**

**Process\_no, Arrival\_time, Burst\_Time, Waiting\_time, Turnaround\_time**

**Algorithm: FCFS**

1. Read total no of processes (n)
2. For reach process read arrival time and burst or service time.
3. Sort the processes on arrival time.
4. Initialize current\_time to arrival time of first process.
5. Initialize total turn\_around\_time and total\_waiting\_time to zero.
6. For each process compute waiting\_time and turn\_around\_time as below

for(i=0;i<n;i++)

{

curr\_time = curr\_time + p[i].bt; // add burst time of

//next process to current\_time

P[i].tat=curr\_time - p[i].at; // turn\_around\_time

p[i].wt = p[i].tat - p[i].bt; // waiting\_time

tot\_wt=tot\_wt+p[i].wt;

tot\_tat=tot\_tat+p[i].tat;

if(i<n-1 && curr\_time < p[i+1].at)//checks if CPU idle

{

curr\_time=p[i+1].at;

}

1. Print waiting\_time and Turn\_around\_time for all processes
2. Print average waiting time and average\_turn\_around\_time
3. Stop

**Algorithm: Shortest Job First (Preemptive)**

1. Read total no of processes (n)
2. For reach process read arrival time and burst or service time.
3. Sort the processes on arrival time.
4. Initialize current\_time to arrival time of first process.
5. Initialize total turn\_around\_time and total\_waiting\_time to zero.
6. While all processes not selected by the processor do
7. Select from arrived processes that process having least remaining time.
8. Let j be the index of that process.
9. If all processes arrived

Assign jth process to processor

Execute it totally, add burst time of process to current\_time

Set remaining time to zero.

Compute Waiting time and Turn\_around\_time for that process

1. If all processes not arrived

Execute jth process only for one unit of time, add 1 to current\_time

Reduce remaining time by 1 for that process.

If remaining time becomes zero, compute Waiting\_time and

Turn\_around\_time for that process

1. End while
2. Compute and print average waiting time and average Turn\_around\_time.
3. Print tabular output, Waiting\_time and Turn\_around\_time for each process
4. Stop

**Input-Output along with test cases**

**FCFS: Input**

**Process AT BT**

**P0 0 3**

**P1 1 1**

**P2 2 2**

**P3 7 4**

**P4 8 2**

**Output**

**Gant Chart**

**0ß----P0---à3ßP1à4ß--P2--à6ßIDLEà7ß-----P3----à11ß-P4-à13**

**Process AT BT WT TAT**

**P0 0 3 0 3**

**P1 1 1 2 3**

**P2 2 2 2 4**

**P3 7 4 0 4**

**P4 8 2 3 5**

**Average Waiting Time = 1.4**

**Average Turn Around Time = 3.8**

**SJF-Preemptive Input**

**Process AT BT**

**P0 0 3**

**P1 1 1**

**P2 2 2**

**P3 7 4**

**P4 8 2**

**Output**

**Gant Chart**

**0ßP0à1ßP1à2ßP0à4ßP2à6ßIDLEà7ßP3à8ß-P4-à10ßP3à13**

**Process AT BT WT TAT**

**P0 0 3 1 4**

**P1 1 1 0 1**

**P2 2 2 2 4**

**P3 7 4 2 6**

**P4 8 2 0 2**

**Average Waiting Time = 1**

**Average Turn Around Time = 3.4**

**Platform :-** LINUX

**Language**:- C - Programming Language / C++

**Conclusion**: FCFS and SJF Preemptive scheduling algorithms were successfully implemented.

**FAQ’s :-**

**(1)** What is meant by CPU scheduling?

(2) What are different types of schedulers?

(3) What is preemptive & non - preemptive scheduling?

(4) What is i/o burst and CPU burst?

(5) Define the following terms:

(1)Turnaround Time

(2)Waiting Time

(3) Throughput

**Assignment No. 4:**

**Title:** producer consumer problem and Reader Writer problem with reader priority

A. Thread synchronization using counting semaphores. Application to demonstrate: producer consumer problem with counting semaphores and mutex.

B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader Writer problem with reader priority

**Title:**

Implementing Producer Consumer problem and Reader Writer Problem using Multithreading using POSIX (Portable Operating Systems Interface) standard --pthread functionalities and Semaphores and Mutex.

**Aim:**

Implementing Producer Consumer problem and Reader Writer Problem using Multithreading using POSIX (Portable Operating Systems Interface) standard --pthread functionalities and Semaphores and Mutex.

**Objective:**

We write a program using the functionalities of pthread (POSIX threads) to implement the Producer Consumer problem and Reader Writer Problem. We also showcase those independent threads are responsible for incorporating parallelism during execution. We also learn and implement these applications using Semaphore and Mutex.

**Theory:**

**Semaphores:**

Semaphore is an integer value used for signalling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.

The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. It is known as a counting semaphore or a general semaphore. Semaphores are the OS tools for synchronization.

Two types:

1. Binary Semaphore.

2. Counting Semaphore

**Binary semaphore**

Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable / available) are called binary semaphores and are used to implement locks.

It is a means of suspending active processes which are later to be reactivated at such time conditions are right for it to continue. A binary semaphore is a pointer which when held by a process grants them exclusive use to their critical section. It is a (sort of) integer variable which can take the values 0 or 1 and be operated upon only by two commands termed in English wait and signal.

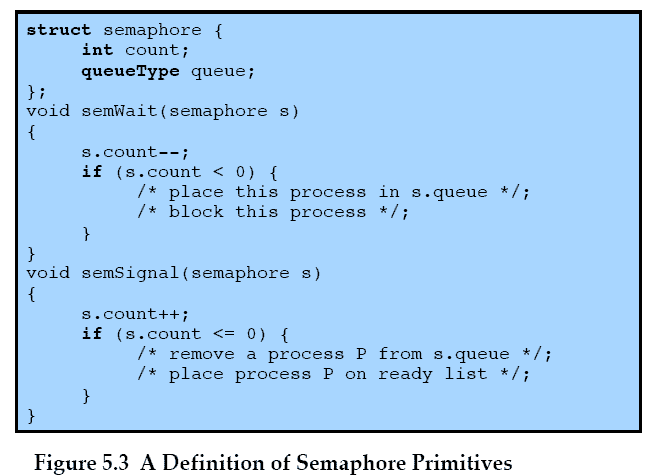
**Counting semaphore**

Semaphores which allow an arbitrary resource count are called counting semaphores.A counting semaphore comprises:

An integer variable, initialized to a value K (K>=0). During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

**A counting semaphore can be implemented as follows:**

* **Initialize** – initialize to non-negative integer
* **Decrement (semWait)** –
* Process executes this to receive a signal via semaphore.
* If signal is not transmitted, process is suspended.
* Decrements semaphore value if value becomes negative, process is blocked Otherwise it continues execution.
* **Increment (semSignal)**
* Process executes it to transmit a signal via semaphore.
* Increments semaphore value
* If value is less than or equal to zero, process blocked by semWait is unblocked



**Value of semaphore**

* **Positive**

Indicates number of processes that can issue wait & immediately continue to execute.

* **Zero**

By initialization or because number of processes equal to initial semaphore value have issued a wait Next process to issue a wait is blocked.

* **Negative**

Indicates number of processes waiting to be unblocked

Each signal unblocks one waiting process.

1. **The Producer/Consumer Problem**

There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer. To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

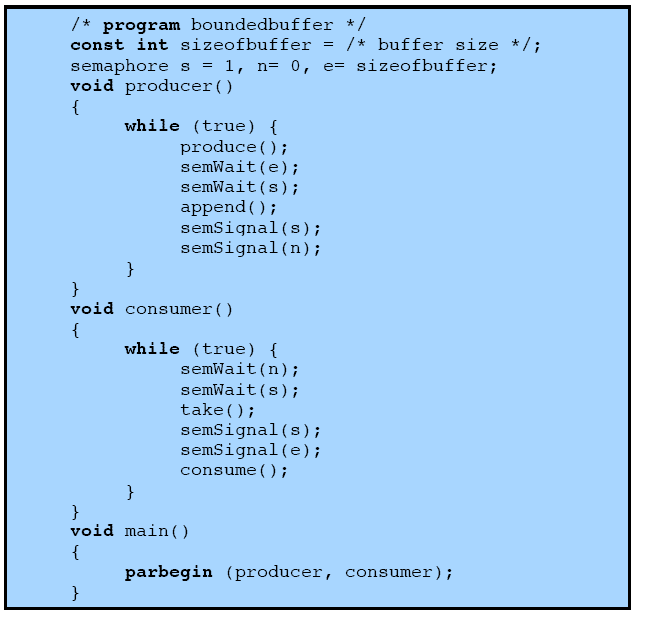


Figure illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the



**Figure: Infinite buffer for producer/consumer problem**

**Solution for bounded buffer using counting semaphore**

****

**POSIX Semaphores**

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem\_post(3)); and decrement the semaphore value by one (sem\_wait(3)). If the value of a semaphore is currently zero, then a sem\_wait(3) operation will block until the value becomes greater than zero.

**Semaphore functions:**

1. **sem\_init()**

It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

intsem\_init(sem\_t \*sem, intpshared, unsigned int value);

1. **sem\_wait()**

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

intsem\_wait(sem\_t \*sem);

1. **sem\_post()**

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem\_wait(3) call will be woken up and proceed to lock the semaphore.

intsem\_post(sem\_t \*sem);

1. **sem\_unlink()**

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

intsem\_unlink(const char \*name)

All the above functions returns

0 : Success

-1 : Error

**Mutex:**

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

**pthread\_mutex\_init()**

The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

intpthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, cons pthread\_mutexattr\_t \*restrict attr);

**pthread\_mutex\_lock()**

The mutex object referenced by mutex shall be locked by calling pthread\_mutex\_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

intpthread\_mutex\_lock(pthread\_mutex\_t \* mutex);

**pthread\_mutex\_unlock()**

The function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread\_mutex\_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

intpthread\_mutex\_unlock(pthread\_mutex\_t \* mutex);

**pthread\_mutex\_destroy()**

The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using pthread\_mutex\_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

intpthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

**(B)The Reader/Writer Problem.6 READERS/WRITERS PROBLEM**

In dealing with the design of synchronization and concurrency mechanisms, it isuseful to be able to relate the problem at hand to known problems and to be ableto test any solution in terms of its ability to solve these known problems. The readers/writers problem is defined as follows: There is a data area sharedamong a number of processes. The data area could be a file, a block of main memory,or even a bank of processor registers. There are a number of processes thatonly read the data area (readers) and a number that only write to the data area(writers).

The conditions that must be satisfied are as follows:

**1.** Any number of readers may simultaneously read the file.

**2.** Only one writer at a time may write to the file.

**3.** If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another andwriters are processes that are required to exclude all other processes, readers andwriters alike.

In the readers/writers problem readers do not also write to the data area, nor do writersread the data area while writing.

For example, suppose that the shared area is a library catalog.Ordinary users of the library read the catalog to locate a book. One or more librariansare able to update the catalog. In the general solution, every access to thecatalog would be treated as a critical section, and users would be forced to readthe catalog one at a time. This would clearly impose intolerable delays. At thesame time, it is important to prevent writers from interfering with each other andit is also required to prevent reading while writing is in progress to prevent the access

of inconsistent information.

**Readers Have Priority**

Figure is a solution using semaphores, showing one instance each of a readerand a writer; the solution does not change for multiple readers and writers.The writer process is simple. The semaphore wsem is used to enforce mutualexclusion. As long as one writer is accessing the shared data area, no other writersand no readers may access it. The reader process also makes use of wsem toenforce mutual exclusion. However, to allow multiple readers, we require that,when there are no readers reading, the first reader that attempts to read shouldwait on wsem. When there is already at least one reader reading, subsequentreaders need not wait before entering. The global variable readcount is used tokeep track of the number of readers, and the semaphore x is used to assure thatreadcount is updated properly.



**POSIX Semaphores**

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem\_post(3)); and decrement the semaphore value by one (sem\_wait(3)). If the value of a semaphore is currently zero, then a sem\_wait(3) operation will block until the value becomes greater than zero.

**Semaphore functions:**

1. **sem\_init()**

It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

intsem\_init(sem\_t \*sem, intpshared, unsigned int value);

1. **sem\_wait()**

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

intsem\_wait(sem\_t \*sem);

1. **sem\_post()**

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem\_wait(3) call will be woken up and proceed to lock the semaphore.

intsem\_post(sem\_t \*sem);

1. **sem\_unlink**

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

intsem\_unlink(const char \*name)

All the above functions return

0: Success

-1: Error

**Mutex**

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

**pthread\_mutex\_init()**

The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

intpthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, cons pthread\_mutexattr\_t \*restrict attr);

**pthread\_mutex\_lock()**

The mutex object referenced by mutex shall be locked by calling pthread\_mutex\_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

intpthread\_mutex\_lock(pthread\_mutex\_t \* mutex);

**pthread\_mutex\_unlock()**

The function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread\_mutex\_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

intpthread\_mutex\_unlock(pthread\_mutex\_t \* mutex);

**pthread\_mutex\_destroy()**

The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using pthread\_mutex\_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

intpthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

**Platform**: - Linux

**Language**: - C/C++-Programming Language

**Conclusion**: - Thus, we implemented **Producer/Consumer problem** and **Reader/Writer** Problem using Multithreading using POSIX (Portable Operating Systems Interface) standard --pthread functionalities and Semaphores and Mutex.

**FAQ**

1. Explain the concept of semaphore?
2. Explain wait and signal functions associated with semaphores.
3. What is meant by binary and counting semaphores?

**Title: Banker’s Algorithm for Deadlock Avoidance**

**Assignment No. 5:**

Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

**Aim:**

To implement Bankers Algorithm for deadlock avoidance using

(a) Safety algorithm

(b) resource request algorithm

**Objectives:**

To understand the concept of deadlock.

How to avoid deadlock by implementing safety algorithm.

**Theory:**

In a multiprogramming environment, several processes may compete for a finite set of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called as a **deadlock**. The resources are partitioned into several types, each of which consists of some number of identical instances. A process must request a resource before using it and release it after using it. The number of resources requested must not exceed the number of resource available in the system.

**A Deadlock situation can arise if the following four conditions hold simultaneously in a system.**

**NECESSORY CONDITIONS FOR DEADLOCK EXISTANCE**

1. Mutual Exclusion: At least one resource must be held in a non-shareable mode, that is only one process at a time can use the resource.

2. Hold and Wait: A process must hold one resource and waiting to acquire additional resources that are currently being held by other processes

3. No Preemption : Resources cannot be pre-empted. The resources can be released by the process only when it completes the task.

1. Circular Wait : A set of (P0,P1,P2,….Pn} of waiting processes must exist such that process P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, and so on and lastly Pn is waiting for a resource held by P0.

**METHODS OF HANDLING DEADLOCKS**:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system

will never enter a deadlock state.

2. We can allow the system to enter a deadlock state, detect and recover it.

3. We can ignore the problem altogether and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX

**DEADLOCK PREVENTION:**

It is a set of methods ensuring that at least one of the necessary conditions do not hold. These methods prevent deadlocks by constraining how requests for resources can be made.

**DEADLOCK AVOIDANCE:**

It requires that the Operating system be given in advance additional information concerning which resource a process will request and use during its lifetime. With this additional knowledge we can decide for each request whether a process should wait or not.

**SAFE AND UNSAFE STATE:**

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence, represented as: < P1, P2, P3, .... Pn >. A sequence of processes < P1 , P2 , P3 , ..... Pn > is a safe sequence for the current allocation state if, for each process Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. If no such sequence exists, the system is in unsafe state, or in a deadlock.

**BANKERS ALGORITHM:**

This is a deadlock avoidance algorithm. It is mainly used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all customers, and hence the name. Whenever a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number should not exceed the total number of resources in the system. When user requests resources, the system determines whether the allocation of these resources will leave the system in a safe state. If system remains in safe state, request is granted, otherwise it enters a wait state.

**Data Structures:**

Let n be the number of processes and m the number of resource types . Then the following data structures are maintained:

1. Available[]: A vector of length m indicates the number of available resource of each type.

If Available[j] = k, there are k instances of resource type Rj are available.

2. Max [][] : An n\*m matrix defines maximum demand of each process.

If Max[i,j] = k, then process Pi may request at most k instances of

resource Type Rj.

3. Allocation[][] : An n\*m matrix that defines the number of resources of each type currently allocated to each process.

If Allocation[i,j] = k, then process Pi is currently owns k instances of resource type Rj.

. 4. Need [][] : An n\*m matrix indicating the remaining resource need of each process.

If Need[i,j] = k, then process Pi may need k more instances of resource type Rj.

**(A)** **SAFETY ALGORITHM :**

The algorithm for finding whether the system is in a safe state or not.

Step.1. : Let Work and Finish be vectors of length m and n respectively .

Initialize Work = Available and Finish[i] = false for i=0 ,1, 2, 3 , ...... , n-1 .

Step.2. : Find an 'i' such that both

(a) Finish[i] = false

(b) Need <= work for ith process for every resource type.

If no such 'i' exists go to step.4. .

step.3. : Do for the 'i'th process ;

Work = Work + Allocation, for all resource types

Finish[i] = true.

Goto step.2. .

step.4. : If Finish[i]==true for all 'i' , system is in safe state .

else system is in unsafe state .

Time complexity of this algorithm is m\* n \* n.

**(B)RESOURCE-REQUEST ALLOCATION ALGORITHM :**

This algorithm is used to grant a request. Let Request be the request vector for process Pi. If Request[j] = k, for ith process, then process Pi wants k instances of resource type Rj. When a request for resources is made by the process Pi, following actions are taken:

step.1. : If Request <= Need , for ith process for all resource types, goto step

step.2, else raise an error condition, since process has exceeded its

maximum claim .

step.2. : If Request <= Available, for ith process for all resource types goto

step.3. else process must enter a wait state since resources are

unavailable .

step.3. : Have the system pretend to have allocated the requested resources to

process Pi by modifying the state as follows:

**Available = Available – Request,** for ith process for all resource types

**Allocation = Allocation + Request,** for ith process for all resource types

**Need = Need- Request,** for ith process for all resource types

step.4. : If the resulting resource allocation state is a safe state , the transaction

is completed and the process Pi is allocated its resources. However, if the

new state is unsafe , then process Pi must wait for request and the old

resource allocation state is restored .

**Sample Input/Output:**

**Input:**

Enter No. of processes in system : 5

Enter No. of resources in system : 3

Enter Total instances of each resource type :

R0 = 10

R1 = 5

R2 = 7

Enter Maximum instances required of each resource type for each process

Max

R0 R1 R2

P0 7 5 3

P1 3 2 2

P2 9 0 2

P3 2 2 2

P4 4 3 3

Enter number of instances of each resource type allocated to each process

Allocation

R0 R1 R2

P0 0 1 0

P1 2 0 0

P2 3 0 2

P3 2 1 1

P4 0 0 2

**Output : New State is**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ALLOCATION | | | MAX | | | NEED | | | AVAILABLE | | |
|  | R0 | R1 | R2 | R0 | R1 | R2 | R1 | R0 | R1 | R0 | R1 | R2 |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 3 | 3 | 2 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 | 1 | 2 | 2 |  |  |  |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 |  |  |  |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 |  |  |  |

**SYSTEM IS IN SAFE STATE** .... SAFE SEQUENCE IS < P1 , P3 , P4 , P2 , P0 >

**Input** :

Do you want to request additional resources (y/n) ? y

ENTER PROCESS ID :0

ENTER NO. OF INSTANCES FOR :

R0 = 1

R1 = 0

R2 = 2

**Output : New state is**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ALLOCATION | | | MAX | | | NEED | | | AVAILABLE | | |
|  | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 2 | 3 | 0 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 |  |  |  |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 |  |  |  |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 |  |  |  |

**SYSTEM IN SAFE STATE** . . . REQUEST GRANTED .

Any more (y/n) ? y

ENTER PROCESS ID :4

ENTER NO. OF INSTANCES FOR :

R0 = 3

R1 = 3

R2 = 0

INSUFFICIENT RESOURCES .... REQUEST CANNOT BE GRANTED .

Any more (y/n) ? n

**Output :**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ALLOCATION | | | MAX | | | NEED | | | AVAILABLE | | |
|  | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 2 | 3 | 0 |
| P2 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 |  |  |  |
| P3 | 3 | 0 | 2 | 9 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P4 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 |  |  |  |
| P5 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 |  |  |  |

**Test Cases**

1. Display error message” Process cannot demand instances which are more than available” , when you are trying to input Max matrix , for ith process which exceeds maximum resources available.
2. Same as above, but for allocation matrix.
3. Test your program for safe as well as unsafe state.

**Platform :-** Linux

**Language**:- C/C++-Programming Language

**Conclusion** :-

The Bankers algorithm was studied and successfully implemented in c and all concepts understood.

**FAQ'S :-**

(**1)What is meant by deadlock ? What are the necessary conditions for**

**a deadlock situation ?**

**(2)What is the difference between deadlock and starvation ?**

**(3) Explain the difference between deadlock avoidance, prevention and detection?**

**(4)What is safety algorithm ?**

**Title: Demand Paging – Page Replacement Algorithms**

Write program to implement following page replacement algorithms

1) First in First Out (FIFO)

2) Least Recently Used (LRU)

**Objectives:**

To understand various concepts such as

* + - Memory management
    - Paging
    - Segmentation
    - Virtual memory
    - Page Replacement Strategies

**Theory:**

One of the most important and complex tasks of an operating system is memory management. Memory management involves treating main memory as a resource to be allocated to and shared among a number of active processes. To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free users from size restrictions in program development.

Relocation, Protection, Sharing, Logical Organization and physical Organization are the requirements that memory management is intended to satisfy.

**Description of Memory management Techniques**

**Fixed Partitioning**

**Description:** Main memory is divided into a no .of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.

**Strengths:** Simple to implement, little operating system overhead

**Weakness**: Inefficient use of memory due to internal fragmentation, Maximum number of active processes is fixed.

**Dynamic Partitioning:**

**Description:** Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process**.**

**Strengths: No** external Fragmentation, more efficient use of main memory

**Weaknesses:** Inefficient use of processor due to the need for compaction to counter external fragmentation

**Simple Paging**

**Description:** Main memory is divided into a no. of equal size frames. Each process is divided into a no. of equal size pages of the same length of frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames

**Strengths:** No external fragmentation

**Weaknesses:** A small amount of internal fragmentation

**Simple Segmentation:**

**Description:** Each process is divided into no. of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.

**Strengths:** No internal fragmentation

**Weaknesses:** Improved memory utilization and reduced overhead compared to dynamic partitioning.

**Virtual Memory Paging**

**Description:** As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.

**Strengths**: No external fragmentation, higher degree of multiprogramming, large virtual address space.

**Weaknesses:** Overhead of complex memory management

**Virtual Memory Segmentation**

**Description**: As with simple segmentation except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.

**Strengths:** No internal fragmentation, higher degree of multiprogramming, large virtual address space, protection and sharing support

**Weaknesses**: Overhead of complex memory management

**Key Terms**

**Internal Fragmentation**: In case of fixed partition technique, there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition. This is known as internal fragmentation.

**External Fragmentation**: In case of dynamic partitions, method leads to a situation in which there are a lot of small holes in a memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation. This is due to the fact that memory that is external to all partitions becomes increasingly fragmented.

**Compaction:** This is a technique used to overcome external fragmentation. From time to time, the operating system shifts the processes so that they are contiguous so that all of the free memory is together in one block. This is a time consuming procedure and wasteful of processor time. It also implies that need for dynamic relocation capability should exist.

**Swapping**: At intervals determined by the operating system, usually dictated by CPU scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This allows more processes to be run than can be fit into memory at one time.

**Logical Address**: It is a reference to a memory location independent of current assignment of data to memory.

**Relative Address**: It is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually the beginning of the program.

**Physical Address**: It is an actual address in main mempry.

**Frame**:: The main memory is divided into equal fixed size chunks that are relatively small. Chunks of main memory are called as frames.

**Page**: Each process is divided into small size chunks of the same size. Chunks of process are known as pages.

**Segmentation**: A process is divided into a number of segments which need not be of equal size. With simple segmentation a process is brought in means all segments are loaded into available regions of memory and segment table is set up.

**Virtual Memory**: It is a technique that allows the execution of processes that may not be completely in memory. The programs can be larger than physical memory. Virtual memory abstracts main memory into extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory storage limitations. This also allows user processes to easily share files and address spaces, and it provides efficient mechanism for process creation.

**Page fault**: With virtual memory only few pages of the entire process are present in the main memory. If the program branches to an instruction or references a data item on a piece not in the main memory, page fault is triggered. This tells the operating system to bring the desired page in the memory.

**Thrashing**: If all of the main memory is occupied with pages of processes, and if a page fault occurs, then operating system must throw one page out main memory and bring the desired page in. Sometimes it is required to bring in the page which was just thrown out. Thrashing is the process of bringing in the pages just thrown out again and again.

**Principle of locality**: It states that program and data references within a process tend to cluster.

**Demand Paging**: It is a fetch policy in which page is brought into memory only when a reference is made to a location on that page.

**Replacement Policy**: This deals with the selection pf a page in memory to be replaced when a new page is to be brought in.

**Frame Locking:** Some of the frames in main memory may be locked. When a frame is locked, the page currently residing in that frame may not be used for replacement. Key control structures, i/o buffers or time critical areas are examples of locked frames.

**Page Replacement Algorithms**: There are certain basic algorithms that are used for the selection of a page to replace. These are **Optimal**, Least Recently Used(**LRU**), First In First Out(**FIFO**), **Clock**

**Brief description of algorithms.**

* **Optimal Page replacement:** This policy selects for replacement that page for which the time to next reference is the longest. This algorithm results into fewest number of page faults.

Limitations: Impossible to implement, since operating system must have a prior knowledge of future events, that is which page will be or will not be referenced in future.

* **Least Recently Used (LRU)**: This policy replaces the page in memory that has not been referenced for the longest time.

It is difficult to implement. One can have a tag with each page with the time of its last reference and it should be done at each memory reference

* **First In first Out(FIFO):** It treats the page frames allocated to a process as a circular buffer, and pages are removed in round robin manner. A pointer is required that circles through page frames of the process.

This is the simplest algorithm to implement and one is replacing that page which is there in the memory for the longest period. Number of page faults could be more as compared to Optimal and LRU.

**Belady’s Anomaly**: For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases. Giving more memory to a process would not improve the performance. This is known as Belady’s Anomaly.

**Data Structure Used:- Arrays**

**Input-Output along with test cases**

**Test your program with the following input.**

**2 3 2 1 5 2 4 5 3 2 5 2 (No of Frames = 3 )**

**FIFO (9 Page Faults)**

**2 3 2 1 5 2 4 5 3 2 5 2**

**2 2 2 2 5 5 5 5 3 3 3 3**

**3 3 3 3 2 2 2 2 2 5 5**

**1 1 1 4 4 4 4 4 2**

**F F F F F F F F F**

**F indicates page fault**

**LRU(7 Page Faults)**

**2 3 2 1 5 2 4 5 3 2 5 2**

**2 2 2 2 2 2 2 2 3 3 3 3**

**3 3 3 5 5 5 5 5 5 5 5**

**1 1 1 4 4 4 2 2 2**

**F F F F F F F**

**Test your program with frame size equal to 4.**

**Check for Belady’s anomaly and verify the results.**

**Platform: Linux**

**FAQS**

1. **Describe page table, frame table and explain the hardware support required to implement paging**
2. **Explain “TLB”.**
3. **Explain following terms First Fit, Best Fit and Worst Fit.**
4. **Explain Paging and Page Table**
5. **Explain virtual memory**
6. **Explain Page fault**
7. **What is Belady’s Anamoly, explain with example**

**Programming Language**

**Assignment No. 6:**

Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

**Assignment Statement:**

Write a menu driven program to simulate the following page replacement algorithms.

·          First In First Out (FIFO).

·          Least Recently Used (LRU).

**Objectives:**

To understand various concepts such as

* + - Memory management
    - Paging
    - Segmentation
    - Virtual memory
    - Page Replacement Strategies

**Theory:**

One of the most important and complex tasks of an operating system is memory management. Memory management involves treating main memory as a resource to be allocated to and shared among a number of active processes. To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free users from size restrictions in program development.

Relocation, Protection, Sharing, Logical Organization and physical Organization are the requirements that memory management is intended to satisfy.

**Description of Memory management Techniques**

**Fixed Partitioning**

**Description:** Main memory is divided into a no .of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.

**Strengths:** Simple to implement, little operating system overhead

**Weakness**: Inefficient use of memory due to internal fragmentation, Maximum number of active processes is fixed.

**Dynamic Partitioning:**

**Description:** Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process**.**

**Strengths:** No internal Fragmentation, more efficient use of main memory

**Weaknesses:** Inefficient use of processor due to the need for compaction to counter external fragmentation

**Simple Paging**

**Description:** Main memory is divided into a no. of equal size frames. Each process is divided into a no. of equal size pages of the same length of frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames

**Strengths:** No external fragmentation

**Weaknesses:** A small amount of internal fragmentation

**Simple Segmentation:**

**Description:** Each process is divided into no. of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.

**Strengths:** No internal fragmentation

**Weaknesses:** Improved memory utilization and reduced overhead compared to dynamic partitioning.

**Virtual Memory Paging**

**Description:** As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.

**Strengths**: No external fragmentation, higher degree of multiprogramming, large virtual address space.

**Weaknesses:** Overhead of complex memory management

**Virtual Memory Segmentation**

**Description**: As with simple segmentation except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.

**Strengths:** No internal fragmentation, higher degree of multiprogramming, large virtual address space, protection and sharing support

**Weaknesses**: Overhead of complex memory management

**Key Terms**

**Internal Fragmentation**: In case of fixed partition technique, there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition. This is known as internal fragmentation.

**External Fragmentation**: In case of dynamic partitions, method leads to a situation in which there are a lot of small holes in a memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation. This is due to the fact that memory that is external to all partitions becomes increasingly fragmented.

**Compaction:** This is a technique used to overcome external fragmentation. From time to time, the operating system shifts the processes so that they are contiguous so that all of the free memory is together in one block. This is a time consuming procedure and wasteful of processor time. It also implies that need for dynamic relocation capability should exist.

**Swapping**: At intervals determined by the operating system, usually dictated by CPU scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This allows more processes to be run than can be fit into memory at one time.

**Logical Address**: It is a reference to a memory location independent of current assignment of data to memory.

**Relative Address**: It is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually the beginning of the program.

**Physical Address**: It is an actual address in main mempry.

**Frame**:: The main memory is divided into equal fixed size chunks that are relatively small. Chunks of main memory are called as frames.

**Page**: Each process is divided into small size chunks of the same size. Chunks of process are known as pages.

**Segmentation**: A process is divided into a number of segments which need not be of equal size. With simple segmentation a process is brought in means all segments are loaded into available regions of memory and segment table is set up.

**Virtual Memory**: It is a technique that allows the execution of processes that may not be completely in memory. The programs can be larger than physical memory. Virtual memory abstracts main memory into extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory storage limitations. This also allows user processes to easily share files and address spaces, and it provides efficient mechanism for process creation.

**Page fault**: With virtual memory only few pages of the entire process are present in the main memory. If the program branches to an instruction or references a data item on a piece not in the main memory, page fault is triggered. This tells the operating system to bring the desired page in the memory.

**Thrashing**: If all of the main memory is occupied with pages of processes, and if a page fault occurs, then operating system must throw one page out main memory and bring the desired page in. Sometimes it is required to bring in the page which was just thrown out. Thrashing is the process of bringing in the pages just thrown out again and again.

**Principle of locality**: It states that program and data references within a process tend to cluster.

**Demand Paging**: It is a fetch policy in which page is brought into memory only when a reference is made to a location on that page.

**Replacement Policy**: This deals with the selection pf a page in memory to be replaced when a new page is to be brought in.

**Frame Locking:** Some of the frames in main memory may be locked. When a frame is locked, the page currently residing in that frame may not be used for replacement. Key control structures, i/o buffers or time critical areas are examples of locked frames.

**Page Replacement Algorithms**: There are certain basic algorithms that are used for the selection of a page to replace. These are **Optimal**, Least Recently Used(**LRU**), First In First Out(**FIFO**), **Clock**

**Brief description of algorithms.**

* **Optimal Page replacement:** This policy selects for replacement that page for which the time to next reference is the longest. This algorithm results into fewest number of page faults.

Limitations: Impossible to implement, since operating system must have a prior knowledge of future events, that is which page will be or will not be referenced in future.

* **Least Recently Used (LRU)**: This policy replaces the page in memory that has not been referenced for the longest time.

It is difficult to implement. One can have a tag with each page with the time of its last reference and it should be done at each memory reference

* **First In first Out(FIFO):** It treats the page frames allocated to a process as a circular buffer, and pages are removed in round robin manner. A pointer is required that circles through page frames of the process.

This is the simplest algorithm to implement and one is replacing that page which is there in the memory for the longest period. Number of page faults could be more as compared to Optimal and LRU.

**Belady’s Anomaly**: For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases. Giving more memory to a process would not improve the performance. This is known as Belady’s Anomaly.

**Data Structure Used:- Arrays**

**Input-Output along with test cases**

**Test your program with the following input.**

**2 3 2 1 5 2 4 5 3 2 5 2 (No of Frames = 3 )**

**FIFO (9 Page Faults)**

**2 3 2 1 5 2 4 5 3 2 5 2**

**2 2 2 2 5 5 5 5 3 3 3 3**

**3 3 3 3 2 2 2 2 2 5 5**

**1 1 1 4 4 4 4 4 2**

**F F F F F F F F F**

**F indicates page fault**

**LRU(7 Page Faults)**

**2 3 2 1 5 2 4 5 3 2 5 2**

**2 2 2 2 2 2 2 2 3 3 3 3**

**3 3 3 5 5 5 5 5 5 5 5**

**1 1 1 4 4 4 2 2 2**

**F F F F F F F**

**Test your program with frame size equal to 4.**

**Check for Belady’s anomaly and verify the results.**

**Platform: Linux**

**Programming Language** : C/C++

**FAQS**

1. Describe page table, frame table and explain the hardware support required to implement paging
2. Explain “TLB”.
3. Explain following terms First Fit, Best Fit and Worst Fit.
4. Explain Paging and Page Table
5. Explain virtual memory
6. Explain Page fault
7. What is Belady’s Anamoly, explain with example