Lab Manual

Course: Operating System

Week 1: Demonstration of system calls

Note: System calls used for this exercise are given at the end of this exercise.

I. Write a program to create a child process using system call fork().

II. Write a program to print process Id's of parent and child process i.e. parent should print its own and its child process id while child process should print its own and its parent process id. (use getpid(), getppid())

III. Write a program to create child process which will list all the files present in your system. Make sure that parent process waits until child has not completed its execution. (use wait(), exit()) What will happen if parent process dies before child process? Illustrate it by creating one more child of parent process.

System Calls:

a) **fork() :** Used to create new processes. The new process consists of a copy of the address space of the original process. System call returns zero in child process while it returns child process id in parent process.
b) **getpid() :** Each process is identified by its id value. This function is used to get the id value of a particular process.
c) **getppid() :** Used to get particular process parent's id value.
d) **wait() :** The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.
e) **exit() :** A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Week 2: Demonstration of system calls

I. Write a program to open a directory and list its contents. (use opendir(), readdir(), closedir())

II. Write a program to show working of execlp() system call by executing ls command.

III. Write a program to read a file and store your details in that file. Your program should also create one more file and store your friends details in that file. Once both files are created, print lines which are matching in both files.

System Calls:

a) **opendir() :** open a directory.
b) **readdir() :** reads contents of directory.
c) **closedir() :** close a directory.
d) **open() :** open a file if it is already created otherwise creates a file and then opens it.
e) **close() :** close a file descriptor.
f) **read() :** read from a file descriptor.
g) **write() :** write to a file descriptor.
h) **grep() :** print lines matching a pattern.

i) **execlp() :** Used after the fork() system call by one of the two processes to replace the process's memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.

Week 3: Process Scheduling

Some operating systems allow more than one process to be loaded into the executable memory at a time. These loaded processes must share the CPU in very efficient manner. The act of determining which of these loaded process should be assigned to CPU and removal of currently running process from CPU is known as process scheduling.
Given a list of processes, their CPU burst times and arrival times, print the Gantt chart for the following given scheduling policies. Also find averag waiting time and average turnaround time required for complete execution of these processes.

a) **FCFS –** First Come First Served : process which arrives first will get the CPU first.
b) **SJF NP –** Shortest Job First Non-Preemptive : process which needs CPU for least amount will get the CPU first. Here non-preemptive means currently running process leaves CPU voultarily after completing its execution.
c) **SJF P –** Shortest Job First Preemptive – Here preemptive means operating system decides when to move currently running process.

Input Format:
First input line will take number of processes, n.
Second input line will take n space-separated integers describing burst time of n processes.
Third input line will take n space-separated integers describing arrival time of n processes.

Output Format:
First output line will print Gantt chart.
Second output line will print average waiting time.
Third output line will print average turnaround time.

Sample I/O for FCFS:

Input:	**Output:**
Number of processes : 4 Burst time : 5 3 8 6 Arrival time : 0 1 2 3	Gantt Chart : P0 P1 P2 P3 Average waiting time: 5.75 Average turnaround time : 11.25

Sample I/O for SJF NP:

Input:	**Output:**
Number of processes : 4 Burst time : 5 3 8 6 Arrival time : 0 1 2 3	Gantt Chart : P1 P0 P3 P2 Average waiting time: 5.25 Average turnaround time : 10.75

Sample I/O for SJF P:

Input:	**Output:**
Number of processes : 5 Burst time : Arrival time :	Gantt Chart : P0 P1 P0 P3 P2 Average waiting time: 5.00 Average turnaround time : 10.50

Week 4: Process Scheduling

I. Given a list of processes, their CPU burst times and arrival times, print the Gantt chart for the following given scheduling policies. Also find averag waiting time and average turnaround time required for complete execution of these processes.

   a) **Priority –** process which has highest priority will get CPU first.
   b) **Round Robin –** each process is provided a fix time to execute. Once a process is executed for a given time period, it is preempted and other process executes for the given time period.

   Note: Consider time quantum = 2 units.

   Input Format:
   First input line will take number of processes, n.
   Second input line will take n space-separated integers describing burst time of n processes.
   Third input line will take n space-separated integers describing arrival time of n processes.

   Output Format:
   First output line will print Gantt chart.
   Second output line will print average waiting time.
   Third output line will print average turnaround time.

   Sample I/O for Priority:

Input:	**Output:**
Number of processes : 4 Burst time : 5 3 8 6 Arrival time : 0 1 2 3	Gantt Chart : P0 P1 P3 P1 P0 P2 Average waiting time: 6.75 Average turnaround time : 12.25

   Sample I/O for Round Robin:

Input:	**Output:**
Number of processes : 4 Burst time : 5 3 8 6 Arrival time : 0 1 2 3	Gantt Chart : P0 P1 P2 P0 P3 P1 P2 P0 P3 P2 P3 P2 Average waiting time: 9.75 Average turnaround time : 15.25

II. Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario: all the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

   Input Format:
   First input line will take number of processes, n.
   Second input line will take n space-separated integers describing burst time of n processes.
   Third input line will take n space-separated integers describing arrival time of n processes.
   Last input line will take process ids of system processes.

   Output Format:
   First output line will print Gantt chart.
   Second output line will print average waiting time.
   Third output line will print average turnaround time.

Sample I/O:

Input:	**Output:**
Number of processes : 6 Burst time : 3 1 4 2 6 3 Arrival time : 0 6 3 4 2 5 System Processes : 1 2 5	Gantt Chart : P0 P2 P5 P1 P4 P3 Average waiting time: 4.66 Average turnaround time : 7.83

Week 5: Deadlock

I. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Input Format:
First input line will take number of processes p.
Second input line will take number of resources r.
Second line is followed by p input lines where each line i contains r space-separated integers describing each row of maximum requirement matrix i.e. process i requires how must instances of each resource in r.
Next p input lines describe allocated matrix where each line i contains r space-separated integers describing how much instances of each resource in r is allocated to process i.
Last line of input will take r space-separated integers describing resource vector i.e. total number of instances provided for each resource type (before allotment of resources to processes).

Output Format:
Output will print '**Request can be fulfilled**' if the request of all the processes can be fulfilled and also find the safe sequence, otherwise it will print '**Request cannot be fulfilled**'.

Sample:

Input:	**Output:**
Enter number of processes : 5 Enter number of resources : 3 Enter maximum requirement : 7 5 3 3 2 2 9 0 2 2 2 2 4 3 3 Enter allocated matrix : 0 1 0 2 0 0 3 0 2 2 1 1 0 0 2 Resource Vector : 10 5 7	Request can be fulfilled Safe Sequence : P1 P3 P0 P2 P4

II. Write a program to implement deadlock detection algorithm.

Input Format:
First input line will take number of processes p.
Second input line will take number of resources r.
Second line is followed by p input lines where each line i contains r space-separated integers describing each row of maximum requirement matrix i.e. process i requires how must instances of each resource in r.

Next p input lines describe allocated matrix where each line i contains r space-separated integers describing how much instances of each resource in r is allocated to process i.
Last line of input will take r space-separated integers describing resource vector i.e. total number of instances provided for each resource type (before allotment of resources to processes).

Output Format:
Output will print '**No deadlock detected**', if need of all the resources can be fulfilled, otherwise it will print '**Deadlock detected**'.

Sample:

Input:	**Output:**
Enter number of processes : 3 Enter number of resources : 3 Enter maximum requirement : 3 2 2 9 0 2 2 2 2 Enter allocated matrix : 2 0 0 3 0 2 2 1 1 Resource Vector : 7 4 5	Deadlock detected

Week 6: Process Synchronization

There may be situation arise where more than one process wants to share resourcesor data. This sharing must be done in such a mannar that data remains consistent. To maintain data consistency synchronized execution of these cooperating processes is required.

I. Write a program to communicate parent and child process with each other in such a way that whenever child writes something, parent process can read it. Conside mode of communication is through-
   a) pipe
   b) message passing
   c) shared memory

II. Write a program to implement the concept of Producer-Consumer problem using semaphores.
III. Write a program to implement the concept of Dining-Philosopher problem.

Week 7: Page Replacement Algorithms

Whenever a new page is referred and not present in memory, page fault occurs. To keep this new referenced page in memory, you need to move some other page out of the memory so as to make room for this referenced page. Page replacement algorithms are needed to decide which page needed to be replaced when this new page comes in. There are number of algorithms for page replacement. You have to write programs to implement following page replacement algorithms:

I. **FIFO –** First In First Out : page which came first (i.e. oldest page) need to be moved out.
II. **LRU –** Least Recently Used : page which is has not been used for longest time need to be moved out.

Input Format:
First input line will take number of frames available.
Second line will take number of requests n.
Third line will take n space-separated requested page numbers.

Output Format:
Output will print total number of page faults occured.

Sample I/O for FIFO:

Input:	**Output:**
Enter number of frames available : 3 Enter number of requests : 12 2 3 2 1 5 2 4 5 3 2 5 2	Total number of page faults : 9

Sample I/O for LRU:

Input:	**Output:**
Enter number of frames available : 3 Enter number of requests : 12 2 3 2 1 5 2 4 5 3 2 5 2	Total number of page faults : 7

Week 8: Memory Allocation Techniques

Memory allocation is the process of assigning blocks of memory on request. Operating system provides small number of large blocks. These blocks must be divided in such a manner that request for small blocks can be fulfilled and also these small blocks when become empty can be reused again.

I. **Best Fit –** block which is closes to the size of request is allocated i.e. the smallest hole that is big enough to allocate to the requesting program.
II. **First Fit –** start searching the list from beginning, take the first block whose size is greater than or equal to the requesting program size and allocate it to program.
III. **Worst Fit –** block which is largest among all is allocated for the program.

Input Format:
First input line will take number of blocks b present in memory.
Second line contains b space-separated integers describing size of each block.
Third input line will take number of processes p which are need to be give memory block.
Fourth line contains p space-separated integers describing how much memory each process requires.

Output Format:
Output will have p lines, where each line i describes process i has assigned which block number. If no free block of appropriate size is available to allocate, then print message '**no free block available**'.

Sample I/O for Best Fit:

Input:	**Output:**
Enter number of free blocks available : 5 100 500 200 300 600 Enter number of processes : 4 212 417 112 426	212 – 4 417 - 2 112 - 3 426 - 5

Sample I/O for First Fit:

Input:	**Output:**
Enter number of free blocks available : 5 100 500 200 300 600 Enter number of processes : 4 212 417 112 426	212 – 2 417 - 5 112 - 3 426 – no free block allocated

Sample I/O for Worst Fit:

Input:	**Output:**
Enter number of free blocks available : 5 100 500 200 300 600 Enter number of processes : 4 212 417 112 426	212 – 5 417 - 2 112 - 4 426 – no free block allocated

Week 9-10: File Allocation Strategies

A single file can use more than one block storage. File allocation strategies deal with how to allocate space for files on disk storage so that disk space is utilized effectively and files can be accessed quickly.
Write a program to implement following file allocation startegies.

I. **Sequntial/Contiguous –** each file occupied contiguous blocks on the disk. A record of a sequential file can only be accessed by reading all the previous records.
II. **Linked –** each file is linked list of disk blocks. the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.
III. **Indexed –** index is maintained for the list of all blocks used by the file. Each file has its own index block which is an array of disk-block addresses. The ith entry in the inde block points to the ith block of the file. The directory contains the address of the index block.

Input Format:
First input line will take number of files F need to be stored.
For each file i in F, file name, starting block number of file and number of blocks required to store that file are taken as an input.
Last input line will ask the user name of file need to be searched.

Output Format:
Output will be the details of the file which is searched, like file name, file starting block number, number of blocks allocated to file and block numbers of these allocated blocks.

Sample I/O for Contiguous:

Input:	**Output:**
Enter number of files : 3 Enter file 1 name : A Enter starting block of file 1 : 85 Enter no of blocks in file 1 : 6 Enter file 2 name : B Enter starting block of file 2 : 102 Enter no of blocks in file 2 : 4 Enter file 3 name :C Enter starting block of file 3 : 60	File Name Start block No. of blocks Blocks occupied B 102 4 102, 103, 104, 105

Enter no of blocks in file 3 : 4 Enter the file name to be searched : B	

Sample I/O for Linked:

Input:	**Output:**
Enter number of files : 3 Enter file 1 name : A Enter starting block of file 1 : 85 Enter no of blocks in file 1 : 6 Enter blocks for file 1 : 85 74 36 89 45 80 Enter file 2 name : B Enter starting block of file 2 : 102 Enter no of blocks in file 2 : 4 Enter blocks for file 2 : 102 49 75 109 Enter the file name to be searched : B	File Name Start block No. of blocks Blocks occupied B 102 4 102, 49, 75, 109

Sample I/O for Indexed:

Input:	**Output:**
Enter number of files : 3 Enter file 1 name : A Enter starting block of file 1 : 85 Enter no of blocks in file 1 : 6 Enter blocks for file 1 : 85 74 36 89 45 80 Enter file 2 name : B Enter starting block of file 2 : 102 Enter no of blocks in file 2 : 4 Enter blocks for file 2 : 102 49 75 109 Enter the file name to be searched : B	File Name Start block No. of blocks Blocks occupied B 102 4 102, 49, 75, 109

Week 11-12: Disc Scheduling Algorithms

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling.
Note: Consider disk has 200 tracks (0 to 199).

I. **FCFS –** First Dome First Served
II. **SCAN –** The disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.
III. **C-SCAN –** It moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip. Once it is returned, it starts moving to other end, servicing requests along the way. It means C-SCAN service requests only in one direction.

Input Format:
First input line will take total number of disk requests, n.

Second input line will take n space-separated integers describing track numbers.

Output Format:
Output will be total seek movement.

Sample I/O for FCFS:

Input:	**Output:**
Enter number of disk requests : 9 55 58 60 70 18 90 150 160 184	Total seek movement : 233

Sample I/O for SCAN:

Input:	**Output:**
Enter number of disk requests : 9 55 58 60 70 18 90 150 160 184	Total seek movement : 325

Sample I/O for C-SCAN:

Input:	**Output:**
Enter number of disk requests : 9 55 58 60 70 18 90 150 160 184	Total seek movement : 361

Week 13-14: Writing a Shell

Goal and Specifications: The goal of this lab is to write our own, very simple, shell. The task is to program a shell that conforms to the specification given in this section. The shell must be written in C language. The shell should have the following features:

- The shell should print a command prompt and allow the user to enter a command.
- After entering the command, the shell should properly execute the command, this includes supplying the provided arguments to the program.
- The entered command string must be tokenized into an array of strings by removing the space delimiters. Also delimiters consisting of more than one space must be handled correctly.
- The user can exit the shell using the exit command.
- Implement exit (to exit the shell), ls (to list files present in current directory), mv (move file from one location to another), cat (to display contents of file) and cd (to change the current working directory) as built-in commands. The cat should also copies the content from one file to another. The mv and cd should display errors if necessary.
- Display an appropriate error if a requested command cannot be found or is not executable.
- Execute commands and correctly pass the provided arguments to this command.