**i 100**

|  |  |  |
| --- | --- | --- |
| **Exp.No. 1** | Implement a C program for the following system calls:  i) getpid() ii) getppid() iii) opendir( ) iv) readdir( ) v) closedir() vi) fork() | **Date:** |

**1. Aim of the Experiment**

The aim of this experimen it is to familiarize students with system calls in the Unix/Linux environment by implementing and verifying several fundamental system calls using C programming language. Specifically, the experiment aims to implement getpid(), getppid(), opendir(), readdir(), closedir(), and fork() system calls, and validate their functionality through appropriate program execution and output analysis.

**2. Requirements of the Experiment**

* A Unix/Linux system environment (like Ubuntu or CentOS).

C programming environment (GCC compiler)..

* Basic understanding of C programming concepts.
* Access to terminal for compiling and running C programs.
* Text editor for writing C code.

**3. Theoretical Background of the Experiment**

System calls are interfaces provided by the operating system that allow applications to request services such as file operations, process management, and directory operations. getpid() returns the process ID of the current process, getppid() returns the process ID of the parent process, opendir() opens a directory stream, readdir() reads entries from the directory stream, and closedir() closes the directory stream. fork() creates a new process (child process) that runs concurrently with the parent process. Understanding and implementing these system calls is crucial for understanding the interaction between user-level applications and the operating system kernel.

* + - 1. **Procedure/Step-by-Step Instructions**

getpid()

1. Include the necessary header files.
2. Define a function get\_process\_id that calls the getpid() system call.
3. Use the get\_process\_id function in the main function to print the process ID.

getppid()

1. Define a function get\_parent\_process\_id that calls the getppid() system call.
2. Use the get\_parent\_process\_id function in the main function to print the parent process ID

opendir()

1. Include the necessary header files.
2. Define a function open\_directory that calls the opendir() function.
3. Use the open\_directory function in the main function to open a directory and handle errors.
4. If the directory is successfully opened, close it using closedir()

readdir()

1. Include the necessary header files.
2. Define a function read\_directory that calls the readdir() function to read directory entries.
3. Use the opendir function to open a directory, and then use the read\_directory function to read and print the entries.
4. Close the directory stream using closedir().

closedir()

1. Include the necessary header files.
2. Define a function close\_directory that calls the closedir() function.
3. Use the opendir() function to open a directory, then use the closedir() function to close it.
4. Handle errors appropriately if the directory cannot be opened or closed.

fork()

1. Include the necessary header files.
2. Define a function create\_process that calls the fork() function.
3. In the create\_process function, handle the return value of fork() to differentiate between the parent and child processes.
4. In the main function, call create\_process and handle the behavior for both parent and child processes.
5. Properly handle errors if fork() fails.

**5. Sample Output/Result of the Experiment**

getpid()

The process ID is: 12345

getppid()

The parent process ID is: 6789

opendir()

This will execute the program and print the names of all files and directories in the current directory. Adjust directory\_name to open other directories as needed.

4. readdir()

This will execute the program and print the names of all files and directories in the current directory. Adjust directory\_name to open other directories as needed.

5. closedir()

This will execute the program, which opens and reads the names of files and directories in the current directory, and then closes the directory.

1. fork()

Hello from Parent Process! Child PID: 12345

Hello from Child Process! PID: 12345

**6.Inferences Obtained from the Experiment**

* Understanding of how system calls interface with the operating system kernel.
* Recognition of the differences between various system calls in terms of their functionality and usage.
* Insight into the process creation mechanism and parent-child relationships in Unix/Linux.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **What is the purpose of system calls in an operating system?**
   * Answer: System calls provide an interface between user-level programs and the operating system kernel, allowing programs to request services like I/O operations, process management, and file system access.
2. **Explain the difference between getpid() and getppid() system calls.**
   * Answer: getpid() returns the process ID of the current process, whereas getppid() returns the process ID of the parent process.
3. **How does fork() system call work?**
   * Answer: fork() creates a new process (child process) that is a copy of the calling process. The child process runs concurrently with the parent process and typically continues executing from the point where fork() was called.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 2** | Implement a C-program for the following system calls:  i)open () ii) read () iii) write () iv) close () | **Date** |

**1. Aim of the Experiment**

The aim of this experiment is to implement and validate the functionality of essential file handling system calls in Unix/Linux operating systems using C programming. Specifically, the experiment focuses on implementing open(), read(), write(), and close() system calls to understand their usage in file operations.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Access to a terminal for compiling and executing programs.
* Basic understanding of C programming concepts, including file operations and system calls.

**3. Theoretical Background of the Experiment**

System calls like open(), read(), write(), and close() are fundamental interfaces provided by the operating system kernel to manage files.

* **open()** is used to open or create a file and returns a file descriptor.
* **read()** reads data from an open file descriptor into a buffer.
* **write()** writes data from a buffer to an open file descriptor.
* **close()** closes a file descriptor, releasing associated resources.

These system calls are crucial for performing file operations in Unix/Linux environments and provide efficient mechanisms for handling file input and output operations in C programs.

1. **Procedure/Step-by-Step Instructions**
   1. **open()**
   2. Include the necessary headers for open() and related constant.
   3. Define variables to store file descriptors and other parameters.
   4. Use the open() function to open or create the file.
   5. Perform operations on the file using the file descriptor returned by open().
   6. After finishing operations, close the file using close().
   7. **read()**
2. Include the necessary headers for read() and related functions.
3. Define variables to store the file descriptor, buffer for data, and other parameters.
4. Use the open() function to open the file for reading.
5. Use the read() function to read data from the file into the buffer.
6. After finishing reading, close the file using close().
   1. **write()**
      1. Include the necessary headers for write() and related functions.
      2. Define variables to store the file descriptor, buffer containing data, and other parameters.
      3. Use the open() function to open or create the file for writing.
      4. Use the write() function to write data from the buffer to the file.
      5. After finishing writing, close the file using close().
   * Close()
   1. Include the necessary headers for close() and related functions.
   2. Define variables to store the file descriptor and any other necessary parameters.
   3. Use the open() function to open the file.
   4. Perform any necessary operations using the file descriptor (fd), such as reading or writing data.
   5. After finishing operations, close the file using close().

**5. Sample Output/Result of the Experiment**

It will create (or truncate) the file example.txt,

write "Hello, World!\n" to it, read it back, and then

print "Hello, World!\n" as the output.

**6.Inferences Obtained from the Experiment**

* Understanding of file handling system calls and their respective functionalities.
* Insight into the role of file descriptors in file management.
* Practical experience in implementing file operations using system calls in C.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the purpose of file descriptors in Unix/Linux operating systems.**
   * Answer: File descriptors are integer identifiers used by the operating system to uniquely identify open files or other I/O resources. They facilitate communication between processes and the kernel for file operations.
2. **What are the differences between open () and fopen() functions in C?**
   * Answer: open () is a system call that directly interacts with the operating system to open or create files, returning a file descriptor. fopen() is a standard library function that provides buffered I/O and returns a FILE pointer.
3. **How does the read () system call handle file input operations?**
   * Answer: read () reads data from an open file descriptor into a buffer specified by the caller. It returns the number of bytes read or -1 on error and advances the file pointer.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 3** | Implement a C-program for the following scheduling algorithms:   * + 1. First Come First Serve (FCFS)     2. Round Robin (RR)     3. Shortest Job First (SJF)w | **Date:** |

**1. Aim of the Experiment-**

The aim of this experiment is to design and implement three different CPU scheduling algorithms—First Come First Serve (FCFS), Round Robin (RR), and Shortest Job First (SJF)—in a C program.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (like Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of process scheduling concepts such as arrival time, burst time, waiting time, turnaround time, and context switching.

**3. Theoretical Background of the Experiment**

CPU scheduling is a key component of operating systems responsible for deciding which process to execute next on the CPU.

* **FCFS** schedules processes in the order they arrive, without considering burst times.
* **RR** allocates a fixed time slice (quantum) to each process in a cyclic manner.
* **SJF** selects the process with the shortest burst time first, minimizing average waiting time.

These algorithms differ in their approach to prioritizing processes, impacting overall system performance and efficiency. Implementing and comparing these algorithms provides insights into their strengths and weaknesses under different workload scenarios.

**4. Procedure/Step-by-Step Instructions**

**Implementing Algorithms:**

* + Implement functions for FCFS, RR, and SJF scheduling algorithms.
  + Define structures or arrays to represent processes with attributes like arrival time, burst time, waiting time, and turnaround time.
  + **FCFS:**

1- Input the processes along with their burst time (bt).

2- Find waiting time (wt) for all processes.

3- As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

4- Find waiting time for all other processes i.e. for

process i ->

wt[i] = bt[i-1] + wt[i-1] -(at[i]-at[i-1]);

21

5- Find turnaround time =waiting\_time + burst\_time for all processes.

6- Find average waiting time = total\_waiting\_time / no\_of\_processes.

7- Similarly, find average turnaround time = total\_turn\_around\_time / no\_of\_processes.

* + **sjf Algorithm:**

1. Sort all the processes according to the arrival time.

2. Then select that process which has minimum arrival time and minimum Burst time.

3. After completion of process make a pool of processes which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

a. Completion Time: Time at which process completes its execution.

rb. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time –Arrival Time

c. Waiting Time (W.T): Time Difference between turnaround time and burst time.

Waiting Time = Turnaround Time – Burst Time

* + **ROUND ROBIN Algorithm:**

\* The CPU scheduler picks the process from the circular/ready queue, set a timer to interrupt it after 1 time slice / quantum and dispatches it.

\* If process has burst time less than 1 time slice/quantum

> Process will leave the CPU after the completion

> CPU will proceed with the next proc

ess in the ready queue / circular queue.

else If process has burst time longer than 1 time

slice/quantum

> Timer will be stopped. It causes interruption to the OS.

p > Executed process is then placed at the tail of the circular / ready queue by applying the

context switch

> CPU scheduler then proceeds by selecting the next process in the ready queue.

1. Completion Time: Time at which process completes its execution.

2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time =

Completion Time – Arrival Time

3. Waiting Time (W.T): Time Difference between turnaround time and burst time.

Waiting Time = Turnaround Time – Burst Time

1. **Sample Output/Result of the Experiment\***

Upon execution, the program should output:

* Average waiting time and turnaround time for each scheduling algorithm (FCFS, RR, SJF).
* Demonstration of scheduling order and metrics calculation for a set of simulated processes.

Processes Burst time Waiting time

1 10 0 10

2 5 10 15

3 8 15 23

Average waiting time = 8.33333

Average turnaround time = 16

1. **Inferences Obtained from the Experiment**

* Comparison of average waiting time and turnaround time for FCFS, RR, and SJF algorithms.
* Understanding of how scheduling policies impact CPU utilization and efficiency.
* Insight into the suitability of each algorithm based on process characteristics like burst times.

1. **Viva Questions for the Experiment**
2. **Explain the FCFS scheduling algorithm and its limitations.**
   * Answer: FCFS schedules processes in the order of their arrival. It is simple to implement but may lead to poor average waiting times, especially if long processes arrive first (convoy effect).
3. **How does the Round Robin (RR) scheduling algorithm work? Discuss its parameters and impact.**
   * Answer: RR allocates a fixed time slice (quantum) to each process in a cyclic manner. It ensures fairness among processes but may result in higher context switching overhead and longer response times for short processes.
4. **What is the difference between preemptive and non-preemptive SJF scheduling?**
   * Answer: Non-preemptive SJF selects the next process based on its burst time without preemption, while preemptive SJF may preempt a running process if a shorter job arrives, optimizing for shorter average waiting times

|  |  |  |
| --- | --- | --- |
| **Exp.No. 4** | Implement a C-program for Priority scheduling algorithm. | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to design and implement the Priority Scheduling algorithm in a C program. Priority scheduling is a CPU scheduling algorithm where each process is assigned a priority. The objective is to simulate and understand how processes are scheduled based on their priority values, ensuring that higher priority processes are executed before lower priority ones.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of proces`s scheduling concepts including priority scheduling and context switching.

**3. Theoretical Background of the Experiment**

Priority Scheduling is a non-preemptive or preemptive scheduling algorithm where each process is assigned a priority value. Processes with higher priority values are executed before those with lower priority values. In the context of operating systems, priorities can be static or dynamic, and the scheduler must ensure fairness and prevent starvation of lower priority processes. The implementation involves assigning priorities to processes and scheduling them accordingly, ensuring optimal system performance and responsiveness based on priority levels.

**4. Procedure/Step-by-Step Instructions**

1. **Implementing Priority Scheduling Algorithm:**
2. First input the processes with their arrival time, burst time and priority.
3. First process will schedule, which have the lowest arrival time, if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.
4. Now further processes will be schedule according to the arrival time and priority of the process. (Here we are assuming that lower the priority number having higher priority). If two process priority are same, then sort according to process number.
5. Note: In the question, they will clearly mention, which number will have higher priority and which number will have lower priority.
6. Once all the processes have been arrived, we can schedule them based on their priority.

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* The order of process execution based on their assigned priority values.
* Metrics such as average turnaround time and average waiting time for the simulated processes.

**Input :**

process no-> 1 2 3 4 5

arrival time-> 0 1 3 2 4

burst time-> 3 6 1 2 4

priority-> 3 4 9 7 8

**Output :**

Process\_no arrival\_time Burst\_time Complete\_time Turn\_Around\_Time Waiting\_Time

1 0 3 3 3 0

2 1 6 9 8 2

3 3 1 16 13 12

4 2 2 11 9 7

5 4 4 5g 15 11 7

Average Waiting Time is : 5.6

Average Turn Around time is : 8.8

1. **Inferences Obtained from the Experiment**

* Comparison of turnaround times and waiting times for processes under different priority levels.
* Understanding of how priority scheduling affects system responsiveness and throughput.
* Insight into the trade-offs between preemptive and non-preemptive priority scheduling approaches.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **What is priority scheduling, and how does it differ from other CPU scheduling algorithms?**
   * Answer: Priority scheduling assigns priorities to each process and schedules them based on these priorities. It differs from algorithms like FCFS and RR by prioritizing higher priority processes for execution before lower priority ones.
2. **Explain the difference between preemptive and non-preemptive priority scheduling.**
   * Answer: In preemptive priority scheduling, a higher priority process can preempt a lower priority process currently running. In non-preemptive priority scheduling, once a process starts executing, it continues until it finishes or voluntarily yields the CPU.
3. **How does priority inversion occur, and how can it be mitigated in priority scheduling?**
   * Answer: Priority inversion happens when a lower priority process holds a resource needed by a higher priority process, causing the higher priority process to wait unnecessarily. Techniques like priority inheritance can mitigate priority inversion by temporarily boosting the priority of processes involved in resource conflicts.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 5** | Implement a C-program for handling producer-consumer problem using semaphores | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to implement a C program that simulates the classic producer-consumer problem using semaphores. The objective is to demonstrate how semaphores can be utilized to synchronize access to a shared buffer between producer and consumer processes. This experiment aims to illustrate the importance of synchronization mechanisms in concurrent programming and how semaphores can prevent issues such as race conditions and buffer overflows.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of semaphores and concurrent programming concepts.

**3. Theoretical Background of the Experiment**

The producer-consumer problem involves two processes, a producer and a consumer, sharing a common fixed-size buffer. The producer generates data items and places them into the buffer, while the consumer removes items from the buffer and processes them. To avoid race conditions where the consumer tries to consume from an empty buffer or the producer tries to produce into a full buffer, synchronization mechanisms like semaphores are used. Semaphores allow mutual exclusion (mutex) and synchronization among processes by controlling access to shared resources. In this experiment, semaphores will be employed to ensure that producers and consumers access the buffer in a mutually exclusive and coordinated manner, thereby preventing conflicts and maintaining data integrity.

**4. Procedure/Step-by-Step Instructions**

1. **Algorithm/Description**

Step 1: Start the program.

Step 2: Declare and initialize the necessary variables.

Step 3: Create a Producer.+

Step 4: Producer (Child Process) performs a down operation and writes a message.

Step 5: Producer performs an up operation for the consumer to consume.

Step 6: Consumer (Parent Process) performs a down operation and reads or consumes the data

(message)

Step 7: Consumer then performs an up operation.

Step 8: Stop the program.

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Sequential production and consumption of items in the buffer by producers and consumers.
* Correct synchronization ensuring that producers do not produce into a full buffer and consumers do not consume from an empty buffer.

Producer produced-0

Producer produced-1

Consumer consumed-0

Consumer consumed-1

Producer produced-2

**6.Inferences Obtained from the Experiment**

* Understanding of how semaphores facilitate synchronization and mutual exclusion in concurrent programs.
* Insight into handling the producer-consumer problem to prevent race conditions and ensure data integrity.
* Practical experience in implementing synchronization mechanisms using semaphores in C.

1. **Viva Questions for the Experiment**
2. **What is the producer-consumer problem, and why is synchronization necessary to solve it?**
   * Answer: The producer-consumer problem involves two processes sharing a common buffer. Synchronization is necessary to prevent issues such as race conditions where the consumer might access an empty buffer, or the producer might overwrite existing data in a full buffer.
3. **Explain the role of semaphores in the producer-consumer problem.**
   * Answer: Semaphores are used to enforce mutual exclusion and synchronization between the producer and consumer processes accessing a shared buffer. They ensure that only one process can access the buffer at a time, preventing conflicts and maintaining data integrity.
4. **What are the differences between binary semaphores and counting semaphores?**
   * Answer: Binary semaphores (mutex) have two states (0 and 1) and are used for mutual exclusion, typically to protect shared resources. Counting semaphores can have a value greater than 1 and are used for tasks like resource management or limiting the number of concurrent accesses.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 6** | Develop a C program to provide synchronization among the 5 philosophers in Dining Philosophers problem using semaphore. | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to develop a C program that solves the Dining Philosophers problem using semaphores for synchronization. The objective is to illustrate how semaphores can be utilized to prevent deadlock and ensure that each philosopher can access the shared dining table (resource) without conflict.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of semaphores, mutual exclusion, and deadlock avoidance.

**3. Theoretical Background of the Experiment**

The dining philosopher’s problem used to demonstrate the concept of deadlock that is - another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

The Dining Philosopher Problem –The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

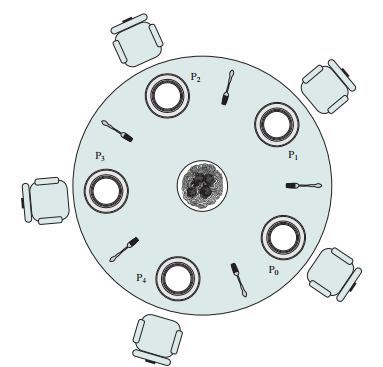


Fig: The Dining Philosopher Problem

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Algorithm/Description:**

Five philosophers are sitting at a rounded dining table.

Step 1: There is one chopstick between each pair of adjacent philosophers.

Step 2: Philosophers are either thinking or eating.

Step 3: Whenever a philosopher wishes to eat, she first needs to find two chopsticks.

Step 4: If the hungry philosopher does not have two chopsticks (i.e. one or two of her neighbours already picked up the chopstick) she will have to wait until both chopsticks are available.,

Step 5: When a philosopher finishes eating, she puts down both chopsticks to their original places, and resumes thinking.

Step 6: There is an infinite amount of food on their plate, so they only need to worry about the chopsticks.

There are a few conditions:

Step 7: Philosophers are either thinking or eating. They do not talk to each other.

Step 8: Philosophers can only fetch chopsticks placed between them and their neighbours.

Step 9: Philosophers cannot take their neighbours’ chopsticks away while they are eating.

Step 10: Hopefully no philosophers should starve to death (i.e. wait over a certain amount of time before she acquires both chopsticks).

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Sequential actions of philosophers, including thinking, acquiring forks, eating, and releasing forks.
* Output indicating the state transitions of each philosopher and the successful prevention of deadlock through semaphore synchronization.
  + Fork 1 taken by Philosopher 1
  + Fork 2 taken by Philosopher 2
  + Fork 3 taken by Philosopher 3
  + Philosopher 4 is waiting for fork 3
  + Till now num of philosophers completed dinner are 0
  + Fork 4 taken by Philosopher 1
  + Philosopher 2 is waiting for Fork 1
  + Philosopher 3 is waiting for Fork 2
  + Philosopher 4 is waiting for fork 3

**6Inferences Obtained from the Experiment**

* Understanding of how semaphores can be used to solve synchronization problems like the Dining Philosophers problem.
* Insight into the challenges of concurrent programming, such as resource contention and deadlock.
* Practical experience in implementing synchronization mechanisms using semaphores in C.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the Dining Philosophers problem and its significance in concurrent programming.**
   * Answer: The Dining Philosophers problem involves multiple philosophers sitting around a dining table, alternating between thinking and eating. It illustrates challenges in resource allocation and synchronization among concurrent processes.
2. **How do semaphores solve the Dining Philosophers problem?**
   * Answer: Semaphores are used to represent the forks (resources) in the Dining Philosophers problem. They ensure mutual exclusion so that only one philosopher can pick up a fork (resource) at a time, preventing deadlock and ensuring fair access to resources.
3. **What are the potential issues that can arise in the Dining Philosophers problem if synchronization is not properly managed?**
   * Answer: Without proper synchronization, issues such as deadlock can occur where philosophers are unable to acquire the necessary resources (forks) to proceed, leading to a system halt. Semaphore-based synchronization ensures that such issues are mitigated by controlling access to shared resources.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 7** | Implement a C-program to handle deadlock using Bankers Algorithm | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to simulate the Banker's Algorithm for deadlock avoidance and prevention in a C program. The objective is to demonstrate how the Banker's Algorithm can be used by an operating system to allocate resources to processes in a way that avoids deadlock or ensures that the system remains in a safe state to prevent deadlock from occurring.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of processes, resources, and deadlock concepts in operating systems.

**3. Theoretical Background of the Experiment**

The Banker's Algorithm is a deadlock avoidance and prevention algorithm used in operating systems to allocate finite resources to multiple processes in a safe manner. It works by dynamically evaluating the state of the system before allocating resources to ensure that the allocation will not lead to a deadlock. The algorithm operates in two phases: request phase and release phase. During the request phase, a process requests resources and the system checks if granting these resources will keep the system in a safe state (i.e., avoid deadlock). If so, the resources are allocated; otherwise, the process must wait. During the release phase, a process releases resources, and the system checks if other processes waiting for these resources can now proceed without causing deadlock. The Banker's Algorithm employs data structures like available, allocation, max, and need matrices to track and manage resource allocation and ensure safety.

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Implementing Banker's Algorithm Functions:**

Following **Data structures** are used to implement the Banker’s Algorithm:

Let **‘n’** be the number of processes in the system and **‘m’** be the number of resources types.

**Available :**

● It is a 1-d array of size ‘m’ indicating the number of available resources of each type.

● Available[ j ] = k means there are ‘k’ instances of resource type Rj

**Max :**

● It is a 2-d array of size ‘n\*m’ that defines the maximum demand of each process in a system.

● Max[ i, j ] = k means process Pi may request at most ‘k’ instances of resource type Rj.

**Allocation :**

● It is a 2-d array of size ‘n\*m’ that defines the number of resources of each type currently allocated to each process.

● Allocation[i,j] = k means process Pi is currently allocated ‘k’ instances of resource type Rj

**Need :**

● It is a 2-d array of size ‘n\*m’ that indicates the remaining resource need of each process.

● Need [ i, j ] = k means process Pi currently need ‘k’ instances of resource type Rj for its execution.

● Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

Banker’s algorithm consists of Safety algorithm and Resource request algorithm

**Safety Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) Needi <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

**Resource-Request Algorithm**

Let Requesti be the request array for process Pi. Requesti [j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1) If Requesti <= Needi

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If Requesti <= Available

Goto step (3); otherwise, Pi must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as

follows:

Available = Available – Requesti

Allocationi = Allocationi + Requesti

Needi = Needi– Requesti

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Successful allocation and deallocation of resources by processes using Banker's Algorithm.
* Output indicating the state transitions of processes, including resource requests, grants, and releases.
* Verification that the system remains in a safe state throughout the execution, preventing deadlock situations.

Enter the number of resources: 4

Enter the number of processes: 5

Enter Claim Vector: 8 5 9 7

Enter Allocated Resource Table: 2 0 1 1 0 1 2 1 4 0 0 3 0 2 1 0 1 0 3 0

Enter Maximum Claim table: 3 2 1 4 0 2 5 2 5 1 0 5 1 5 3 0 3 0 3 3

The Claim Vector is: 8 5 9 7

The Allocated Resource Table:

2 0 1 1

0 1 2 1

4 0 0 3

0 2 1 0

1 0 3 0

The Maximum Claim Table:

3 2 1 4

0 2 5 2

5 1 0 5

1 5 3 0

3 0 3 3

Allocated resources: 7 3 7 5

Available resources: 1 2 2 2

Process3 is executing.

The process is in safe state.

Available vector: 5 2 2 5

Process1 is executing.

The process is in safe state.

Available vector: 7 2 3 6

Process2 is executing.

The process is in safe state.

Available vector: 7 3 5 7

Process4 is executing.

The process is in safe state.

Available vector: 7 5 6 7

Process5 is executing.

The process is in safe state.

Available vector: 8 5 9 7

**6.Inferences Obtained from the Experiment**

* Understanding of how the Banker's Algorithm prevents deadlock by ensuring that resource allocation does not lead to an unsafe state.
* Insight into the importance of resource management and allocation strategies in preventing system-wide deadlock scenarios.
* Practical experience in implementing and simulating resource allocation using Banker's Algorithm in a controlled environment.

**7. Viva Questions for the Experiment**

1. **What is deadlock, and how does the Banker's Algorithm help in preventing it?**
   * Answer: Deadlock is a situation where two or more processes are unable to proceed because each is waiting for resources held by the other(s). The Banker's Algorithm prevents deadlock by ensuring that resources are allocated in a manner that guarantees the system remains in a safe state, where processes can always proceed and complete their execution.
2. **Explain the concept of a "safe state" in the context of the Banker's Algorithm.**
   * Answer: A safe state is a state where the system can allocate resources to processes in such a way that it avoids deadlock. In a safe state, even if each process requests its maximum resources and other processes continue to request resources, the system can still allocate resources in a way that all processes eventually complete.
3. **What are the limitations or assumptions of the Banker's Algorithm?**
   * Answer: The Banker's Algorithm assumes that the maximum resource requirements of each process are known in advance, which may not always be practical. Additionally, it assumes that processes do not hold resources indefinitely and eventually release them after use, which may not always hold true in real-world scenarios.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 8** | Implement a C-program to demonstrate the first fit and best fit algorithm | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to implement two memory allocation algorithms, First Fit and Best Fit, in a C program. The objective is to compare these algorithms in terms of efficiency and memory utilization when allocating memory blocks to processes.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of memory management concepts including fragmentation, allocation algorithms, and system performance metrics.

**3. Theoretical Background of the Experiment**

Memory management in operating systems involves allocating and deallocating memory dynamically to processes. Two common strategies for allocating memory blocks are First Fit and Best Fit. In First Fit, the operating system allocates the first available memory block that is large enough to accommodate a process. In Best Fit, the system searches the entire list of available memory blocks and allocates the smallest block that is large enough to fit the process, minimizing wastage of memory. Both algorithms aim to reduce fragmentation and optimize memory usage, but they differ in terms of implementation complexity and efficiency. Implementing these algorithms in a C program allows us to observe how they perform under different scenarios of memory allocation and deallocation.

Need to have knowledge including

* The location of process control information,
* The **execution** stack, and the code entry.
* After loading of the **program** into main **memory**, the processor and the operating system must be able to translate logical addresses into physical addresses

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Algorithm/Description:**

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block

ALGORITHM: **BEST-FIT**

Step 1: Include the necessary header files required.

Step 2: Declare the variables needed.

Step 3: Read the number of blocks and the size of each block.

Step 4: Read the number of process and the size of each process.

Step 5: Arrange both the process and block size in an order.

Step 6: Check if the process size is less than or equal to block size.

Step 7: If yes, assign the corresponding block to the current process.

Step 8: Else print the current process is not allocated.

ALGORITHM: **FIRST-FIT**

Step 1: Include the necessary header files required.

Step 2: Declare the variables needed.

Step 3: Read the number of blocks and the size of each block.

Step 4: Read the number of process and the size of each process.

Step 5: Check if the process size is less than or equal to block size.

Step 6: If yes, assign the corresponding block to the current process.

Step 7: Else print the current process is not allocated.

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Allocation and deallocation of memory blocks using First Fit and Best Fit algorithms.
* Output showing how each algorithm handles memory requests and manages fragmentation.
* Metrics such as average memory utilization and fragmentation levels for comparison between the two algorithms.

**MEMORY MANAGEMENT SCHEME - BEST FIT**

Enter No. of Blocks: 5

Enter the 0st block size: 500

Enter the 1st block size: 100

Enter the 2st block size: 250

Enter the 3st block size: 650

Enter the 4st block size: 850

Enter No. of Process: 5

Enter the size of 0st process: 450

Enter the size of 1st process: 605

Enter the size of 2st process: 820

Enter the size of 3st process: 110

Enter the size of 4st process: 230

Process Block Size

820 850

605 650

450 500

230 250

110 100

OUTPUT:

**MEMORY MANAGEMENT SCHEME - FIRST FIT**

Enter No. of Blocks: 5

Enter the 0st block size: 120

Enter the 1st block size: 230

Enter the 2st block size: 340

Enter the 3st block size: 450

Enter the 4st block size: 560

Enter No. of Process: 5

Enter the size of 0st process: 530

Enter the size of 1st process: 430

Enter the size of 2st process: 630

Enter the size of 3st process: 203

Enter the size of 4st process: 130

Process Block Size

530 120

430 230

630 340

203 450

130 560

The process 3 [size 203] allocated to block 230

The process 4 [size 130] allocated to block 340

The process 1 [size 430] allocated to block 450

The process 0 [size 530] allocated to block 560

The process 2 is not allocated.

**6.Inferences Obtained from the Experiment**

* Comparison of memory utilization efficiency between First Fit and Best Fit algorithms.
* Evaluation of fragmentation levels and their impact on overall system performance.
* Insight into the advantages and disadvantages of each algorithm in terms of implementation complexity and runtime efficiency.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the First Fit and Best Fit memory allocation algorithms. How do they differ in terms of approach and efficiency?**
   * Answer: First Fit allocates the first available memory block that is large enough for a process, while Best Fit searches for the smallest available block that can accommodate the process. First Fit is simpler but may lead to more fragmentation, whereas Best Fit minimizes wastage but requires more computational effort.
2. **What are the advantages and disadvantages of First Fit and Best Fit algorithms in memory management?**
   * Answer: First Fit is straightforward to implement and executes quickly but can lead to larger fragments of unused memory. Best Fit reduces wasted memory by finding the smallest suitable block but may take longer to find an appropriate block due to the need to search the entire list of free memory blocks.
3. **How does fragmentation affect system performance, and how do First Fit and Best Fit algorithms address this issue?**
   * Answer: Fragmentation occurs when memory is allocated and deallocated over time, leaving small unused gaps that cannot be utilized. Both algorithms attempt to minimize fragmentation, with First Fit accepting the first available block and Best Fit searching for the smallest suitable block to reduce wasted memory, thereby optimizing overall system performance.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 9** | Implement a C-program for illustrating Page replacement algorithms   1. First in First Out (FIFO) 2. Least Recently Used (LRU) 3. Optimal | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to implement and compare three-page replacement algorithms — First in First Out (FIFO), Least Recently Used (LRU), and Optimal — in a C program. The objective is to simulate how these algorithms manage the swapping of pages between main memory and secondary storage to optimize memory usage and minimize page faults.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of virtual memory, page replacement algorithms, and system performance metrics.

1. **Theoretical Background of the Experiment**

In virtual memory management, page replacement algorithms determine which page to evict from memory when a new page needs to be loaded and memory is full. Three commonly used algorithms are FIFO, LRU, and Optimal:

* **FIFO** (First in First Out): This algorithm replaces the oldest page in memory, regardless of how frequently or infrequently it is used.
* **LRU** (Least Recently Used): This algorithm replaces the page that has not been used for the longest period of time.
* **Optimal**: This theoretical algorithm replaces the page that will not be used for the longest period in the future. It serves as a benchmark for comparison against practical algorithms.

These algorithms aim to minimize page faults (instances where a page required by a process is not available in memory) and improve overall system performance by optimizing the use of available memory resources.

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Algorithm/Description**

**A) FIRST IN FIRST OUT (FIFO) PAGE REPLACEMENT ALGORITHM:**

Step1: Start

Step2: Read no of frames and reference

Step3: Read the frame list

Step4: Copy the reference list into stack

Step5: Insert the frame number into frame by FIFO

Step6: Display the frame stack S

Step7: Stop

**B) LEAST RECENTLY USED(LRU) PAGE REPLACEMENT ALGORITHM:**

Step1: Start

Step2: Read no of frames and reference and reference list values

Step3: Insert the element into the frame by least recently used

Step4: While inserting an element i, the frame contents also having the same element i occur then print no

page fault occurs

Step5: Otherwise print page fault occurs and continue from step3 until reference list number becomes zero

Step6: Stop

**C) OPTIMAL PAGE REPLACEMENT ALGORITHM:**

Step1: Start

Step2: Read the number of frames, reference and reference list

Step3: Replace the page fault that will not be used for longer period of time

Step4: Count and print the no. of page faults occur

Step5: Display the reference list stack

Step6: Stop

1. **Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Number of page faults and page replacements incurred by each algorithm.
* Comparison of efficiency metrics such as average memory utilization and execution time.
* Output indicating which algorithm performs best under specific conditions of page referencing patterns and memory sizes.

Enter no of pages:18

Enter reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7

Enter no of frames:3

7

7 0

7 0 1

2 0 1

2 0 1

2 0 3

2 0 3

2 4 3

2 4 3

2 4 3

2 0 3

2 0 3

2 0 3

2 0 1

2 0 1

2 0 1

2 0 1

2 0 7

total no of page faults=9

1. **Inferences Obtained from the Experiment**

* Comparison of page fault rates and efficiency among FIFO, LRU, and Optimal algorithms.
* Understanding of how different page replacement strategies impact overall system performance and memory utilization.
* Insight into the strengths and weaknesses of each algorithm under various scenarios of page referencing patterns.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the concept of page replacement algorithms in virtual memory management. How do FIFO, LRU, and Optimal algorithms differ in their approach?**
   * Answer: Page replacement algorithms decide which page to evict from memory when new pages need to be loaded. FIFO replaces the oldest page, LRU replaces the least recently used page, and Optimal replaces the page that will not be used for the longest period in the future. Each algorithm aims to minimize page faults and improve memory utilization.
2. **What are the factors that influence the choice of a page replacement algorithm in an operating system?**
   * Answer: Factors include the frequency and pattern of page references, the size of memory available, and the computational overhead of each algorithm. Some algorithms perform better with certain patterns of page references than others.
3. **Discuss the advantages and disadvantages of FIFO, LRU, and Optimal algorithms in memory management.**
   * Answer: FIFO is simple to implement but may not always result in optimal memory usage. LRU is effective in reducing the number of page faults but requires additional overhead to track page usage. Optimal provides the best possible performance benchmark but is impractical to implement in real systems due to its requirement of future page reference knowledge.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 10** | Implement a C-program for Contiguous file allocation method. | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to simulate the Contiguous file allocation method in a C program. The objective is to understand how files are allocated and managed in a contiguous manner on disk storage.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of file systems, disk allocation methods, and data structures.

**3. Theoretical Background of the Experiment**

Contiguous file allocation is a method where each file occupies a contiguous set of blocks on disk. In this method, the starting block and the length of the file are stored in the file allocation table (FAT). The main advantage of contiguous allocation is simplicity and fast access to files, as the entire file is stored in one continuous block. However, it suffers from issues such as fragmentation, where free space is fragmented into small blocks that are too small to accommodate larger files, leading to inefficient use of disk space. Implementing this method in C involves managing file allocation and deallocation, handling fragmentation, and ensuring efficient storage management.

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

· Address of starting block

· Length of the allocated portion.

The file ‘mail’ in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies 19, 20, 21, 22, 23, 24 blocks.

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Implementing Contiguous File Allocation:**

STEP 1: Start the program.

STEP 2: Gather information about the number of files.

STEP 3: Gather the memory requirement of each file.

STEP 4: Allocate the memory to the file in a sequential manner.

STEP 5: Select any random location from the available location.

STEP 6: Check if the location that is selected is free or not.

STEP 7: If the location is allocated set the flag = 1.

STEP 8: Print the file number, length, and the block allocated.

STEP 9: Gather information if more files must be stored.

STEP 10: If yes, then go to STEP 2.

1. STEP 11: If no, Stop the program. **Simulating File Operations:**
   * Create a simulation where files are created, read, updated, and deleted using the contiguous allocation method.
   * Track file allocations, disk space utilization, and fragmentation issues.

**5 . Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Allocation of contiguous blocks for new files.
* Output indicating file allocation status, including allocated blocks and free space.
* Evaluation of fragmentation levels and their impact on storage efficiency and file access performance.

**INPUT:**

Enter no of files :3

Enter file name 1 :A

Enter starting block of file 1 :85

Enter no of blocks in file 1 :6

Enter file name 2 :B

Enter starting block of file 2 :102

Enter no of blocks in file 2 :4

Enter file name 3 : C

Enter starting block of file 3 : 60

Enter no of blocks in file 3 : 4

Enter the file name to be searched : B

**OUTPUT:**

FILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED

B 102 4 102, 103, 104, 105

**6.Inferences Obtained from the Experiment**

* Evaluation of disk space utilization efficiency under the contiguous file allocation method.
* Comparison of file fragmentation levels and their impact on file access speed and storage management.
* Insight into the advantages (e.g., fast access) and disadvantages (e.g., fragmentation) of using contiguous allocation for file storage.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the concept of contiguous file allocation. How does it differ from other file allocation methods?**
   * Answer: Contiguous file allocation stores each file in a continuous block of disk space, which allows for fast sequential access but suffers from fragmentation issues. It differs from methods like linked allocation or indexed allocation by its simplicity in access and storage management.
2. **What are the advantages and disadvantages of using contiguous file allocation?**
   * Answer: Advantages include fast access to files and simplicity in implementation. Disadvantages include fragmentation, where free space is scattered in small blocks that are inefficient for storing larger files, and the challenge of dynamic space allocation.
3. **How does fragmentation affect the performance of a file system using contiguous allocation?**
   * Answer: Fragmentation leads to wasted disk space and slower file access times. As files are deleted and created, the free space becomes fragmented, making it difficult to find contiguous blocks large enough to store new files efficiently.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 11** | Implement a C-program for demonstrating Disk scheduling algorithms:   * 1. FCFS   2. SCAN   3. C-SCAN | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to implement and compare three disk scheduling algorithms — First-Come, First-Served (FCFS), SCAN, and C-SCAN — in a C program. The objective is to simulate how these algorithms manage the movement of disk arms and optimize disk access time for read and write operations. This experiment aims to illustrate the differences in efficiency, performance, and behavior of these algorithms under varying scenarios of disk request queues.

**2. Requirements of the Experiment**

* Unix/Linux operating system environment (such as Ubuntu or CentOS).
* C programming environment with GCC compiler.
* Text editor for writing C code.
* Terminal for compiling and running programs.
* Basic understanding of disk scheduling algorithms, disk I/O operations, and data structures.

**3. Theoretical Background of the Experiment**

Disk scheduling algorithms determine the order in which disk I/O requests are serviced by the disk arm. Three common algorithms are:

* **FCFS (First-Come, First-Served)**: This algorithm services requests in the order they arrive. It is straightforward but may result in longer seek times if requests are far apart on the disk.
* **SCAN**: Also known as Elevator algorithm, SCAN services requests in one direction until the end of the disk is reached, then reverses direction. It reduces seek times by preventing the disk arm from having to travel across the entire disk repeatedly.
* **C-SCAN**: C-SCAN is a variant of SCAN where the disk arm "scans" the disk in one direction only and when it reaches the end of the disk, it jumps to the beginning of the disk without servicing requests on the return trip. This reduces variance in service time compared to SCAN.

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Algorithm/Description:**

**(i) First Come First Serve (FCFS)**

This algorithm entertains requests in the order they arrive in the disk queue.

**Example:** Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14,124, 65, 67. The head is initially at cylinder number 53.



**Algorithm**

1. Start the program.

2. Mark the ‘head’ as the initial position of disk head.

3. Let request array represent an array storing indexes of track that have been requested in

ascending order of their time of arrival.

4. One by one take the tracks in default order and calculate the absolute distance of the track

from the head.

5. Increment the total seeks count with this distance.

6. New head position is currently serviced track position.

7. Go to step no. 3 until all track in request array have not been serviced.

**(ii) SCAN Algorithm**

This algorithm scans all the cylinders of the disk back and forth.

Head start from one end of the disk and move towards the other end servicing all the request in between. After reaching the other end, head reverses its direction and move towards the starting end servicing all the requests in between. The same process repeats.

**Example:** Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14,124, 65, 67. The head is initially at cylinder number 53.



**Algorithm:**

1. Start the program.

2. Mark the ‘head’ as the initial position of disk head.

3. Let request array represent an array storing indexes of track that have been requested in

ascending order of their time of arrival.

4. Let direction represents whether the head is moving towards left or right.

5. In the direction in which head is moving service all tracks one by one.

6. Calculate the absolute distance of the track from the head.

7. Increments the total seek count with this distance.

8. Currently serviced track position now becomes the new head position.

9. Go to step 5 until we reach at one of the ends of the disk.

10. If reach at the end of the disk reverse the direction and go to step 4 until all tracks in request array have not been serviced.

**(iii) Circular- SCAN (C-SCAN) Algorithm**

Circular-SCAN Algorithm is an improved version of the SCAN.

Head starts from one end of the disk and move towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction. It then returns to the starting end without servicing any request in between. The same process repeats.

**Example:** Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14,124, 65, 67. The head is initially at cylinder number 53.



**Algorithm:**

1. Start the program.

2. Mark the ‘head’ as the initial position of disk head.

3. Let request array represent an array storing indexes of track that have been requested in

ascending order of their time of arrival.

4. The head services only in the right direction from 0 to the size of the disk.

5. While moving in the left directions do not service any of the tracks.

6. When we reach the beginning (left end) reverse the direction.

7. While moving in the right direction it services all tracks one by one.

8. While moving in the right directions calculate the absolute distance of the track from the head.

9. Increment the total seeks count with this distance.

10. Currently serviced track position now becomes the new head position.

11. Go to step 8 until we reach the right end of the disk.

12. If we reach the right end of the disk reverse the direction and go to step 5 until all tracks in the request array have not been serviced.

**5. Sample Output/Result of the Experiment**

Upon execution, the program should demonstrate:

* Total service time and average seek time for each disk scheduling algorithm.
* Output showing the order in which disk requests are serviced and the movement of the disk arm for each algorithm.
* Comparison of performance metrics highlighting the effectiveness of SCAN and C-SCAN in reducing seek times compared to FCFS.

**Input Sample**

* Enter Number of Tracks: 10
* Enter Track Position: 50 55 18 40 60 120 67 80 91 22

**6.Inferences Obtained from the Experiment**

* Comparison of average seek time and total service time among FCFS, SCAN, and C-SCAN algorithms.
* Evaluation of efficiency and performance metrics such as throughput and response time for disk I/O operations.
* Insight into the strengths and weaknesses of each algorithm in handling different patterns of disk request queues.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Describe the FCFS disk scheduling algorithm. How does it determine the order in which disk requests are serviced?**
   * Answer: FCFS services disk requests in the order they arrive, without considering the location of the requests on the disk. It handles each request in sequence, starting from the disk's current position.
2. **Compare SCAN and C-SCAN disk scheduling algorithms. How do they differ in their approach to reducing seek times?**
   * Answer: SCAN moves the disk arm in one direction across the disk, servicing requests until the end is reached, and then reverses direction. C-SCAN, however, jumps from one end of the disk to the other without servicing requests on the return trip, which reduces variance in service times.
3. **What are the practical applications of disk scheduling algorithms in operating systems?**
   * Answer: Disk scheduling algorithms optimize disk access times and improve overall system performance by reducing seek times and efficiently managing disk I/O operations. They are crucial in systems where efficient utilization of disk resources is critical, such as in databases, file systems, and multimedia applications.

|  |  |  |
| --- | --- | --- |
| **Exp.No. 12** | Implement a Virtual File System (VFS) interface for your kernel, and a temporary memory-based file system (tmpfs) that mounts as the root file system in C programming language. | **Date:** |

**1. Aim of the Experiment**

The aim of this experiment is to implement a Virtual File System (VFS) interface within a custom kernel and create a memory-based file system (tmpfs) that functions as the root file system. The objective is to understand the design and implementation of a basic file system in an operating system kernel, demonstrating how file operations and system calls interact with the VFS layer.

**2. Requirements of the Experiment**

* Knowledge of C programming and operating system concepts.
* Access to a development environment capable of kernel programming (e.g., Linux kernel source).
* Understanding of file system structures and operations.
* Basic understanding of system calls and kernel interfaces.

**3. Theoretical Background of the Experiment**

In operating systems, a Virtual File System (VFS) provides an abstraction layer between user-space applications and different file systems supported by the operating system. It standardizes file system operations such as file creation, reading, writing, and deletion, allowing multiple file systems to be supported simultaneously. Implementing a VFS interface involves defining structures and functions that facilitate communication between the kernel and file systems.

**tmpfs** is a memory-based file system supported by many Unix-like operating systems, including Linux. It stores files and directories in virtual memory and uses the system's page cache for managing data. As a root file system, tmpfs resides entirely in RAM, offering fast access speeds but volatile storage (data is lost on reboot unless explicitly saved to disk). Implementing tmpfs as the root file system demonstrates the kernel's capability to manage and access files directly from memory.

**4. Procedure/Step-by-Step Instructions**

**Step-by-Step Procedure:**

1. **Setup Development Environment:**
   * Set up a development environment with Linux kernel source code and necessary tools (e.g., GCC compiler, make utility).
2. **Define VFS Structures and Functions:**
   * Define structures such as inode, superblock, file\_operations, and inode\_operations that represent file system entities and operations.
   * Implement functions for file system operations like open, read, write, close, mkdir, rmdir, etc., adhering to the VFS interface.
   * pseudo code, this code doesn’t show crossing of mount point or relative pathname:

int vfs\_lookup(const char\* pathname, struct vnode\*\* target) {

auto vnode\_itr = rootfs->root;

for (component\_name : pathname) {

auto next\_vnode;

auto ret = vnode\_itr->v\_ops->lookup(vnode\_itr, next\_vnode, component\_name);

if(ret != 0) {

return ret;

}

vnode\_itr = next\_vnode;

}

\*target = vnode\_itr;

return 0;

}

1. **Implement tmpfs File System:**
   * Create a tmpfs-specific implementation that uses kernel memory for file and directory storage.
   * Implement functions to manage inode creation, file allocation, directory operations, and memory management within tmpfs.
2. **Integrate tmpfs as Root File System:**
   * Modify the kernel's boot sequence or configuration to mount tmpfs as the root file system.
   * Ensure initialization routines set up tmpfs structures and mount them appropriately during kernel boot.
3. **Test File System Operations:**
   * Write test applications or scripts to perform basic file operations (create files, read/write data, create directories) on the tmpfs root file system.
   * Verify that operations behave as expected and that data persistence (or lack thereof) meets tmpfs characteristics.
4. **Compile and Install the Custom Kernel:**
   * Compile the modified kernel with the integrated tmpfs support.
   * Install the custom kernel and configure the bootloader to boot into the new kernel image with tmpfs as the root file system.
5. **Evaluate Performance and Functionality:**
   * Measure file system performance metrics such as file access speed, memory usage, and CPU utilization under various workloads.
   * Evaluate the reliability and limitations of tmpfs as a root file system, considering its volatile nature and impact on system stability.

**5. Sample Output/Result of the Experiment**

Upon successful execution, the experiment should demonstrate:

* Successful booting of the custom kernel with tmpfs mounted as the root file system.
* Ability to perform file operations such as creating, reading, writing, and deleting files and directories within tmpfs.
* Output showing system resource utilization and performance benchmarks comparing tmpfs to traditional file systems.

**6.Inferences Obtained from the Experiment**

* Comparison of file system performance between tmpfs and traditional disk-based file systems.
* Assessment of memory usage and scalability of tmpfs for handling large numbers of files and directories.
* Insight into the benefits (speed, simplicity) and drawbacks (volatility, size limits) of using a memory-based file system as the root file system.

**7. Viva Questions for the Experiment**

**Sample Viva Questions:**

1. **Explain the concept of a Virtual File System (VFS). What role does it play in operating system design?**
   * Answer: VFS provides an abstraction layer between user applications and different file systems supported by the operating system. It standardizes file operations across different file systems, enabling applications to access files without needing to know specific details of underlying file system implementations.
2. **What are the advantages and disadvantages of using a memory-based file system like tmpfs as the root file system?**
   * Answer: Advantages include fast access times, simplified management, and suitability for temporary or volatile data. Disadvantages include volatility (data loss on power-off), limited storage capacity (dependent on available RAM), and inability to persist data across reboots without additional measures.
3. **How does integrating tmpfs as the root file system impact system performance and reliability?**
   * Answer: Integrating tmpfs can improve performance due to faster access times and reduced I/O overhead. However, its volatile nature may affect system reliability if critical data needs to persist across system reboots. Evaluating trade-offs between performance and data persistence is crucial in such implementations.