## 

## OS Lab Record Work

### 

### Exercise 1:

1) The working of fork():

Aim:

To develop a C program to understand the working of fork().

Algorithm:

1. Start
2. Get a process id by forking.
3. If the process id is 0, then proceed to the child process. This will print a chosen unique
4. message a certain number of times.
5. If not, then proceed to the parent process. This will print a different identifying message a required number of times.
6. Stop

2) Reading a file:

Aim:

To develop a C program using system calls to open a file, read the contents of the same,

display it and close the file.

Algorithm:

1. Start
2. Check if the number of command line arguments is 2. If it is greater than 2, print that there are too many arguments. If it is lesser, then print that an argument is expected.
3. Find the file descriptor for the source file given as the argument. Include an error message for the case when the source file is not found.
4. Save the contents of the source file in a string.
5. Display the contents of the file by printing the string.
6. Close the source file.
7. Stop

### Exercise 2:

Aim:

1. To develop a C program to simulate the cp command with the -i option using system calls.
2. To develop a C program to simulate the ls command with the -l option using system calls.
3. To develop a C program to simulate the grep command with the -c, -n and -v options using system calls.

Algorithm:

1. my\_cp (-i)
2. Start
3. Check if an option (-i) is included in the command.
4. If the difference between the number of arguments and the number of options is less than three, then print that two arguments are expected. If it is more than three, communicate to the user that there are too many arguments. If it is exactly three, then proceed.
5. If an option is included, then set the 2nd argument as the source file and the 3rd argument as the destination file.
6. If an option is not included, then set the 1st argument as the source file and the 2nd argument as the destination file.
7. Open the source file and locate its file descriptor. If it already exists, proceed. If not, then report that the source file was not found.
8. Copy the contents of the source file to a string.
9. Open the destination file and do one of the following:
   1. If it does not exist, then create a new file and write the contents of the string to it.
   2. If it already exists and there is no (-i) option, then overwrite the contents of the string to the file.
   3. If it already exists and there is a (-i) option, then prompt the user to confirm before overwriting the file.
      * If the user says “yes”, then overwrite the file.
      * If the user says “no” (or does not say yes), then exit the program without making any changes.
10. Stop
11. my\_ls (-l)
12. Start
13. Check if an option (-l) is included in the command.
14. Find the difference between the number of arguments and the number of options.
    1. If it is 0, set the current directory to be the directory under consideration.
    2. If it is 1 or more, set each of the directories entered as an argument on the command line to be the set of directories under consideration.
15. For each directory under consideration, get the entries in the directory structure.
16. If option (-l) is included, then list the directory entries along with their file properties in long format. If not, then simply list the directory entries.
17. Stop
18. my\_grep (-c,-n,-v)
19. Start
20. Check if there is at least one argument. If not, exit.
21. Check for options that are included in the command and validate them.
22. If the difference between the number of arguments and the number of options is less than three, report that the number of arguments is less. If not, proceed.
23. Set the argument immediately succeeding the command name (and options, if any) to be the pattern that is to be searched for.
24. Set the rest of the arguments to be the files that are to be searched.
25. Locate the file descriptor of each file. If a file is not found, report an error.
26. Read the contents of each file in a string.
27. Search the string for substrings that match the given pattern.
28. Do one of the following:
    1. If no option is present, display the lines containing matches of the pattern.
    2. If option (-c) is given, display the frequency of pattern matches in the string.
    3. If option (-n) is given, display the lines containing matches of the pattern along with their line numbers.
    4. If option (-v) is given, display the lines which do not contain matches of the pattern.
    5. If a combination of these options is given, display the corresponding combined output as required.
29. Stop

### Exercise 3:

Aim:

To develop a menu-driven C program to implement the CPU Scheduling Algorithms FCFS, SJF and SRTF.

Algorithm:

1. Start
2. Declare a structure with elements such as the process name, arrival time, burst time, waiting time and turnaround time.
3. Create a menu with options for the following:
   1. FCFS
   2. SJF
   3. SRTF
4. FCFS:
   1. Get the details of the processes as input from the user. This includes the arrival time and the burst time of each process.
   2. As the first process does not need to wait, assign the waiting time for the first process as zero.
   3. For every subsequent, set the current waiting time as the sum of the burst time and the waiting time of the previous process.
   4. For the wording target, set the turnaround time as the sum of the waiting time and the burst time.
   5. Compute the average of the total waiting time and the turnaround time for all the processes.
5. SJF:
   1. Get the details of the processes as input from the user. This includes the arrival time and the burst time of each process.
   2. Sort the processes in the increasing order of their burst times.
   3. As the first process does not need to wait, assign the waiting time for the first process as zero.
   4. For every subsequent project, set the current waiting time as the sum of the burst time and the waiting time of the previous process.
   5. For every process, set the turnaround time as the sum of the waiting time and the burst time.
   6. Compute the average of the total waiting time and the turnaround time for all the processes.
6. SRTF:
7. Repeat the following steps until all the processes have been completed:
8. Find the process with the minimum remaining time at every single time lap.
9. Reduce its time by 1.
10. Check if its remaining time becomes 0.
11. Increment the counter for process completion.
12. Calculate the waiting time and turnaround time for each completed process.
13. Increment time lapsed by 1.

||. Compute the average of the total waiting time and the turnaround time.

4. Stop

### Exercise 4:

Aim:

To develop a menu driven C program to implement the CPU Scheduling Algorithms Priority (Non-Preemptive and Preemptive) and Round Robin.

Algorithm:

1. Start
2. Declare a storage structure
3. Create a menu with options for the following
   1. Round Robin

* Get the arrival time and the burst time of each process as input from the user.
* Set the waiting time for all the processes to be 0.
* Repeat the following steps until all the processes have been used. Start with the first process on the list.
* Reduce its remaining type by 1.
* Check if its remaining time becomes 0. If yes, increment the counter for process completion and move on to the next process.
* Check if the allotted quantum for the process has expired. If yes, save the remaining time for the process and move on to the next one.
* If the last process has exhausted its time, lol.
  1. Priority
* Get the arrival time, the burst time and priority of each process as input from the user.
* Set the waiting time for all the processes to be 0.
* Repeat the following steps until all the processes have been used. Start with the first process on the list.
  + Find the process with the maximum priority.
  + Reduce its remaining type by 1.
  + Check if its remaining time becomes 0. If yes, increment the counter for process completion and move on to the next process.
  + Check if the allotted quantum for the process has expired. If yes, save the remaining time for the process and move on to the next one.
  + Increase time lapsed by 1.
* Calculate the turnaround time for each process. It is given by the sum of the respective burst and waiting times.
* Compute the average waiting time and the average turnaround time.

1. Stop

### Exercise 5:

Aim:

To develop the following applications that use interprocess communication concepts using shared memory:

1. An application for getting a name in parent and converting it into uppercase in child.
2. A client/server application for file transfer.
3. A client/server chat application.

Algorithm:

1) Application for converting to uppercase in child:

1. Start
2. Get the shared memory identifier.
3. Get the process id by forking.
4. If the process id is positive, then do the following:
   1. Attach a variable name to the shared memory.
   2. Get the name as input from the user.
   3. Detach the variable from the shared memory.
5. If the process id is 0, then do the following:
   1. Attach a variable to the shared memory.
   2. Convert the given name to uppercase.
   3. Print the name in uppercase.
   4. Detach the variable from the shared memory.
6. Remove the shared identifier and destroy the segment.
7. Stop

2) Application for file transfer:

Client side:

1. Start
2. Get the key for the shared memory by forking.
3. Get the shared memory identifier.
4. Attach a variable to the shared memory.
5. Get file name from the user.
6. Print the contents of the file onto the server.
7. Stop.

Server side:

1. Start
2. Get the key for the shared memory by forking.
3. Get the shared memory identifier.
4. Read the file name from the shared memory.
5. Read the contents of the file using a pointer.
6. Print the contents of the file.
7. Close the file.
8. Detach from the shared memory.
9. Destroy the shared memory space.
10. Stop

3) Chat application:

Client side:

1. Start
2. Get the process id and generate a shared memory identifier.
3. Set the key to some value.
4. Attach the structure to the shared memory.
5. Assign the second process id to the first and make the status ‘not ready’.
6. Signal the handler to receive a message.
7. Get a message from the user.
8. Set the status to ‘ready’.
9. Send the message by using the kill command to interrupt the process.
10. Wait until the status is ‘not ready’ and continue.
11. Detach the pointer from the shared memory.
12. Stop

Server side:

1. Start
2. Get the process id and assign the common key value to be the key.
3. Generate the shared memory identifier.
4. Attach the structure pointer to the shared memory.
5. Set the process id with the first process and the status to ‘not ready’.
6. Signal the handler function to receive a message.
7. Wait until the status is either ‘filled’ or ‘not ready’.
8. Get a message and set the status to ‘filled’.
9. Send the message by using the kill command to interrupt the process.
10. Detach the pointer from the shared memory.
11. Destroy the shared memory space.
12. Stop

### Exercise 6:

Aim:

i) To write a C program to create a parent/child process to implement the producer/consumer problem using semaphores in the pthread library.

ii) To modify the program as separate client/server process programs to generate ‘N’ random numbers in producer, write them into the shared memory and execute a consumer process to read them from the shared memory and display them on the terminal.

Algorithm:

i) Producer-Consumer problem using semaphores:

1. Start
2. Create a shared memory for the buffer and semaphores - *empty*, *full* and *mutex*. Get the shared memory identifier. Initialize the semaphores with values ‘buffer-size’, 0 and 1 respectively.
3. Create a parent and a child process with the parent acting as the producer and the child as the consumer.
4. Get a string as input from the user and fork the process.
5. In the producer process, produce an item by taking the first unvisited character from the string entered and placing it in the buffer array. Increment the semaphores *full* and *mutex*, and decrement the semaphore *empty* using the *wait* and *signal* operations suitably.
6. In the consumer process, consume an item by removing the first item in the buffer and displaying it on the terminal. Increment the semaphores *empty* and *mutex*, and decrement the semaphore *full* using the *wait* and *signal* operations appropriately.
7. Detach the pointer from the shared memory.
8. Destroy the shared memory space.
9. Stop

ii) Producer-Consumer problem with client/server process programs and random number generation:

Server side:

1. Start
2. Create a shared memory for the buffer and semaphores - *empty*, *full* and *mutex*. Get the shared memory identifier. Initialize the semaphores with values ‘buffer-size’, 0 and 1 respectively.
3. Get the number of numbers required as input from the user.
4. Randomly generate as many numbers as required and add them to the buffer one by one.
5. After every addition to the buffer, increment *full* and decrement *empty* using the wait and signal operations appropriately.
6. Detach the pointer from the shared memory.
7. Destroy the shared memory space.
8. Stop

Client side:

1. Start
2. Access the shared memory that contains the buffer.
3. Get the numbers stored in the buffer one by one and display them on the terminal.
4. Detach the pointer from the shared memory.
5. Stop

### Exercise 7:

Aim:

i) To develop a C program to implement Banker’s algorithm for deadlock avoidance with multiple instances of resource types.

ii) To develop a C program to implement an algorithm for deadlock detection with multiple instances of resource types and display the processes involved in deadlock.

Algorithm:

i) Banker's Algorithm for Deadlock Avoidance:

1. Start
2. Read the following:
   1. No. of processes, *n.*
   2. No. of resources, *m.*
   3. Total number of instances of each resource.
   4. Maximum requirement of each resource.
   5. Allocated instances of each resource.
   6. Available number of instances of each resource.
3. Calculate the number of resources needed by each process by subtracting the allocated number of resources from the maximum number of resources.
4. Display the given details in a table.
5. To run the safety algorithm and check the system state:
   1. Let the vectors representing the work and finish status be vectors of lengths *m* and *n* respectively. Initialize:
      1. The work vector with the values of the available resources.
      2. The finish vector with 0 for each entry.
   2. Find a process such that its finish value is still 0 (i.e., the process has not been completed) and its need vector is less than the work vector.
   3. If such a process is found, add the resources allocated for the process to the work vector and set the finish states of the process to 1. Add the process to the safety sequence before repeating step 5.b.
   4. If the finish status of every process has been changed to 1, then the system is in a safe state. Print the safety sequence.
   5. If one or more of the processes could not be completed, then display the processes for which the needed resources could not be allocated.
6. To put in a resource request, do the following:
   1. Get the process id that requests the resource and the request vector for the number of resources requested as input from the user.
   2. If the number of resources requested is lesser than the resources needed as well as the number of resources available, then:
      1. Update the available, needed and allocated vector resources for that process.
      2. Run the safety algorithm and print the safety sequence if it exists.
      3. If a safe sequence is obtained, grant the resources to the process by updating the system state. If not, inform the user that the resource request cannot be granted.
7. Stop

ii) Algorithm for Deadlock Detection:

1. Start
2. Read the following:
   1. No. of processes, *n.*
   2. No. of resources, *m.*
   3. Total number of instances of each resource.
   4. Maximum requirement of each resource.
   5. Allocated instances of each resource.
   6. Available number of instances of each resource.
3. Calculate the number of resources needed by each process by subtracting the allocated number of resources from the maximum number of resources.
4. Display the given details in a table.
5. To run the safety algorithm and check the system state:
   1. Let the vectors representing the work and finish status be vectors of lengths *m* and *n* respectively. Initialize:
      1. The work vector with the values of the available resources.
      2. The finish vector with 0 for each entry.
   2. Find a process such that its finish value is still 0 (i.e., the process has not been completed) and its need vector is less than the work vector.
   3. If such a process is found, add the resources allocated for the process to the work vector and set the finish states of the process to 1. Add the process to the safety sequence before repeating step 5.b.
   4. If the finish status of every process has been changed to 1, then the system is in a safe state. Print the safety sequence.
   5. If one or more of the processes could not be completed, then display the incomplete processes that cause a potential deadlock.
6. Stop

### Exercise 8:

Aim:

To implement the memory management techniques:

1. First fit
2. Best fit
3. Worst fit

Algorithm:

1. Start
2. Get the required details on the partitioning of the physical memory such as:
   1. The number of partitions
   2. Starting addresses
   3. Ending addresses
3. Calculate the size of each partition and the free space available in each partition.
4. Display a menu with options for the three memory allocation techniques:
5. First fit
6. Best fit
7. Worst fit
8. For each technique, display a menu with options for allocation, deallocation, displaying the memory and for coalescing of holes.
9. Stop
10. First fit:
11. Start
12. To allocate a process:
    1. Read the process ID and the size of the process as input from the user.
    2. Iterate through the partitions until a partition with enough free space to accommodate the process is found.
    3. Insert the process into the physical memory and update the allocated and free memory accordingly.
    4. If no free space is available, print an error message indicating the same.
13. To deallocate a process:
    1. Read the process ID of the process to be deallocated as input from the user.
    2. If the process is found in the physical memory, remove the process and create a hole in the memory space that the process was occupying.
    3. If the process is not found in the physical memory, print an error message indicating the same.
14. To merge holes:
    1. If a partition contains a hole and the partitions immediately following it also contain holes, then note the starting address of the partition and the ending address of the last partition to contain a hole in this sequence.
    2. Set the newly acquired starting address and ending address as the starting and ending of a single partition that replaces the entire sequence of partitions containing holes.
    3. Repeat this strategy for all the sequences of holes in the physical memory.
15. Stop
16. Best fit:
17. Start
18. To allocate a process:
    1. Read the process ID and the size of the process as input from the user.
    2. Iterate through the partitions until a partition with the minimum free space to accommodate the process is found.
    3. Insert the process into the physical memory and update the allocated and free memory accordingly.
    4. If no free space is available, print an error message indicating the same.
19. To deallocate a process:
    1. Read the process ID of the process to be deallocated as input from the user.
    2. If the process is found in the physical memory, remove the process and create a hole in the memory space that the process was occupying.
    3. If the process is not found in the physical memory, print an error message indicating the same.
20. To merge holes:
    1. If a partition contains a hole and the partitions immediately following it also contain holes, then note the starting address of the partition and the ending address of the last partition to contain a hole in this sequence.
    2. Set the newly acquired starting address and ending address as the starting and ending of a single partition that replaces the entire sequence of partitions containing holes.
    3. Repeat this strategy for all the sequences of holes in the physical memory.
21. Stop
22. Worst fit:
23. Start
24. To allocate a process:
    1. Read the process ID and the size of the process as input from the user.
    2. Iterate through the partitions until a partition with the maximum free space to accommodate the process is found.
    3. Insert the process into the physical memory and update the allocated and free memory accordingly.
    4. If no free space is available, print an error message indicating the same.
25. To deallocate a process:
    1. Read the process ID of the process to be deallocated as input from the user.
    2. If the process is found in the physical memory, remove the process and create a hole in the memory space that the process was occupying.
    3. If the process is not found in the physical memory, print an error message indicating the same.
26. To merge holes:
    1. If a partition contains a hole and the partitions immediately following it also contain holes, then note the starting address of the partition and the ending address of the last partition to contain a hole in this sequence.
    2. Set the newly acquired starting address and ending address as the starting and ending of a single partition that replaces the entire sequence of partitions containing holes.
    3. Repeat this strategy for all the sequences of holes in the physical memory.
27. Stop

### Exercise 9:

Aim:

To develop a C program to implement the paging technique in memory management.

Algorithm:

1. Start
2. Define structures for frames and processes. Initialize the frames with a value that indicates allocated but unknown processes, say, -1.
3. Get the total size of the physical memory and the size of each page as input from the user.
4. Divide the physical memory into frames and compute the number of free frames available.
5. Display a menu with operations for allocation and deallocation of processes, displaying the page table and displaying the free frames available.
   1. Allocating a process:
      1. Read the entering process ID and size as input from the user.
      2. Compute the number of pages required for allocating this process.
      3. If the number of free frames is sufficient, iterate through all the frames to identify the free ones and assign pages to them accordingly.
      4. If the number of free frames is not sufficient, print a suitable error message.
   2. Deallocating a process:
      1. Read the required process ID as input from the user.
      2. If all the frames are free or the required process is not found, alert the user to the same.
      3. If not, iterate through all the frames to find the one with the required process.
      4. Append the released frames to the free frame queue and mark them as unallocated.
   3. Displaying the page table:
      1. For each process, iterate through the allocated frames and print all the page-frame pairs for the process.
   4. Displaying the free frames available:
      1. Print the queue containing all the free frames.
6. Stop

### Exercise 10:

Aim:

To develop a C program to implement the page replacement algorithms (FIFO, Optimal, LRU

and LFU) using a linked list.

Algorithm:

1. Start
2. Read the number of frames.
3. Read the number of frames required by the process N.
4. Read the reference string for allocation of page frames.
5. Implement the chosen page replacement algorithm.
6. Stop

Page replacement algorithms:

1. *FIFO replacement:*
   1. Allocate the first N pages into the frames and increment the page faults accordingly.
   2. When the next frame in the reference string is not already available in the allocated list do
      1. Look for the oldest one in the allocated frames and replace it with the next page frame.
      2. Increment the page fault whenever a frame is replaced.
   3. Display the allocated frame list after every replacement.
2. *Optimal replacement:* 
   1. Allocate the first N pages into the frames and increment the page faults accordingly.
   2. When the next frame in the reference string is not already available in the allocated list do
      1. Look for a frame in the reference string that will not be used for the longest period of time.
      2. Increment the page fault whenever a frame is replaced. (Hint: Locate the position of each allocated frame in the reference string; identify a frame for replacement with the largest index position)
   3. Display the allocated frame list after every replacement.
3. *LRU replacement:*
   1. Allocate the first N pages into the frames and increment the page faults accordingly.
   2. When the next frame in the reference string is not already available in the allocated list do
      1. Look for a frame which has not been used recently.
      2. Increment the page fault whenever a frame is replaced.
   3. Display the allocated frame list after every replacement.
4. *LFU replacement:* 
   1. Allocate the first N pages into the frames and increment the page faults accordingly.
   2. When the next frame in the reference string is not already available in the allocated list do
      1. Look for a frame which is least frequently used.
      2. Increment the page fault whenever a frame is replaced.
   3. Display the allocated frame list after every replacement.

### Exercise 11:

Aim:

To create a multithreaded program that calculates various statistical values for a list of numbers passed as command line arguments.

Algorithm:

1. Start
2. Initialize the global variables sum, min, max and avg to zero.
3. Define functions to compute the minimum, maximum and average values of a given list of numbers.
4. In the main function, define the thread identifiers and initialize the thread attributes.
5. Read a list of elements from the command line arguments.
6. Create threads for each function.
7. Wait for the threads to close using the join function.
8. Print the results of the computations performed by each thread.
9. Stop

### Exercise 12:

Aim:

To develop a C program to implement the various file allocation techniques.

Algorithm:

1. Start
2. Get Main memory size and block size as input.
3. Create a Main memory with ‘n’ number of blocks of equal size.
4. Main memory is maintained as Linked List with structure containing block id, Free/Filename, Link to next Memory block , Link to Next File block (only for Linked Allocation), File block table (integer array to hold block numbers only for Indexed Allocation)
5. Get the number of files and their size as input.
6. Calculate the no. of blocks needed for each file.
7. Select the Allocation Algorithm – For every algorithm display Directory information and File information.
8. For Contiguous Allocation - For each file do the following:
   1. Generate a random number between 1 to ‘n’
   2. Check for a continuous number of needed file free blocks starting from that random block no.
   3. If free then allot that file in those continuous blocks and update the directory structure.
   4. Else, repeat step 1.
   5. If no continuous blocks are free then ‘no enough memory error’
   6. The Directory Structure should contain Filename, Starting Block, length (no. of blocks)
9. For Linked Allocation- For each file do the following:
   1. Generate a random number between 1 to ‘n’ blocks.
   2. Check if that block is free or not.
   3. If free then allot it for file. Repeat steps 1 to 3 for the needed number of blocks for the file and create a linked list in Main memory using the field “Link to Next File block”.
   4. Update the Directory entry which contains Filename, Start block number, Ending Block Number.
   5. Display the file blocks starting from start block number in Directory upto ending block number by traversing the Main memory Linked list using the field “Link to Next File block”.
10. For Indexed Allocation - For each file do the following:
    1. Generate a random number between 1 to ‘n’ blocks for index block.
    2. Check if it is free. If not, repeat index block selection.
    3. Generate needed number of free blocks in random order for the file and store those block numbers in index block as array in File block table array.
    4. Display the Directory structure which contains the filename and index block number. Display the File Details by showing the index block number’s File Block table.
11. Stop

### Exercise 13:

Aim:

To develop a C program to implement the following file organization techniques:

a) Single level Directory

b) Hierarchical Structure

Algorithm:

1. Start
2. Let the user choose between single level and hierarchical directory structures.
3. Single Level Directory
   1. Maintain a table containing the filename and the starting address location of that file.
   2. Give options for creating a new file.
   3. Get the name of the file as input from the user. If the file does not already exist, increment the file counter and add the file to the directory.
   4. Update the table accordingly.
4. Tree Structured Directory
   1. Maintain tables for each directory starting from root.
   2. Create a structure for a node in tree which contains an array to hold directories and an array to hold files.
   3. Limit each directory to have a maximum of three sub-directories and files.
   4. For each sub-directory follow the same table structure as described above.
   5. Give options for creating a new directory or a new file.
   6. Get the name and path of the directory or file as input from the user.
   7. Update the table accordingly.
5. Stop

## Learning outcomes:

Exercise 1:

* Several system calls and system commands were studied.
* The system calls and commands were implemented and their functionalities observed.
* The syntax, prerequisites and purpose of each system call and system command was understood.

Exercise 2:

* Linux commands were replicated by programming in C.
* The inner workings of the given commands were understood while implementing them.
* Validation of input and error handling was implemented.

Exercise 3:

* The various CPU Scheduling algorithms were understood.
* Some of the CPU Scheduling algorithms, namely, FCFS, SJF and SRTF, were implemented in C.
* Non-preemptive scheduling was understood.

Exercise 4:

* The various CPU Scheduling algorithms were understood.
* The Round Robin and Priority CPU Scheduling algorithms were implemented in C.
* Preemptive scheduling was understood and implemented in C.

Exercise 5:

* Interprocess communication was understood.
* A simulation of the process of sending and receiving signals between a client and a server was implemented.
* The concept of shared memory was understood and applied.

Exercise 6:

* It was understood that semaphores can be used to solve various synchronization problems and can be implemented efficiently.
* The producer/consumer bounded buffer problem was understood by implementing it using semaphores.
* The method of handling semaphores between a server and a client was understood.

Exercise 7:

* The Banker’s Algorithm for deadlock avoidance was understood and implemented.
* A safety sequence for process synchronization was obtained for a set of given processes and available resources.
* Methods for avoiding and detecting deadlocks were implemented.

Exercise 8:

* The three memory allocation techniques - first fit, best fit and worst fit - were understood and implemented.
* Methods for allocating and deallocating processes were implemented.
* Availability of free space was increased by coalescing holes in partitions in the memory.

Exercise 9:

* Paging in memory management was understood.
* Paging techniques were implemented.
* The concepts of page tables and frames were understood and implemented.

Exercise 10:

* Page replacement techniques were understood and implemented.
* The different page replacement techniques were compared.
* The optimal page replacement technique was found to produce the least number of page faults.

Exercise 11:

* Concurrent execution using threads was understood and implemented.
* Manipulation of threads using the pthread library was understood.

Exercise 12:

* File allocation techniques were understood and implemented.
* Files were allocated in the main memory according to the sequential, linked and indexed allocation techniques.

Exercise 13:

* File organisation techniques were understood and implemented.
* Single level and hierarchical level organisation was understood and implemented.