**OPERATING SYSTEMS LABORATORY SYLLABUS:**

**(19A05403P)**

***Course Objective:***

* Explain basic UNIX commands
* Understand the architecture of OS
* Distinguish the CPU Scheduling algorithms
* Interpret process management and memory management
* Use appropriate APIs’ available in modern operating systems (such as threads, system calls, semaphores, etc…) for software development.
* Explain models for deadlock handling mechanisms

***Learning Outcome:***

*After completion of the course the students will be able.*

* *Ensure the development of applied skills in operating systems related areas.*
* *Able to write software routines modules or implementing various concepts of*

*operating system..*

**List of Experiments: (All the Experiments are to be conducted)**

1. Practicing of Basic UNIX Commands.

2. Write programs using following UNIX operating system calls

Fork, exec, getpid, exit, wait, close, stst, opendir and readdir

3. Simulate UNIX commands like cp, ls, grep, etc.,

4. Simulate the following CPU scheduling algorithms

a) Round Robin b) SJF c) FCFS d) Priority

5. Implement dynamic priority scheduling algorithm.

6. Assume that there are five jobs with different weights ranging from 1 to 5. Implement

round robin algorithm with time slice equivalent to weight.

7. Implement priority scheduling algorithm. While executing, no process should wait for

more than 10 seconds. If waiting time is more than 10 seconds, that process has to be

executed for atleast 1 second before waiting again.

8. Control the number of ports opened by the operating system with

a) Semaphore b) Monitors.

9. Simulate how parent and child processes use shared memory and address space.

10. Simulate sleeping barber problem.

11. Simulate dining philosopher’s problem.

12. Simulate producer and consumer problem using threads.

13. Implement the following memory allocation methods for fixed partition

a) First fit b) Worst fit c) Best fit

14. Simulate the following page replacement algorithms

a) FIFO b) LRU c) LFU etc.,

15. Simulate Paging Technique of memory management

16. Simulate Bankers Algorithm for Dead Lock avoidance and prevention

69 Page

17. Simulate following file allocation strategies

a) Sequential b) Indexed c) Linked

18. Simulate all File Organization Techniques

a) Single level directory b) Two level c) Hierarchical d) DAG

**Recommended Systems/Software Requirements:**

* Intel based desktop PC with minimum of 166 MHZ or faster processor with at least 64 MB RAM and 100 MB free disk space.
* Turbo C or TC3 complier in Windows XP or Linux Operating System.

**INDEX**

|  |  |
| --- | --- |
| *Name of the Student :* | *Roll No:* |
| *Class & Branch:***III** B.Tech. **I** - sem | *Lab :* **OS** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.No.** |  | **Name of the experiment** | **Page No.** | **Date** | **Signature** |
| 1 |  | Practicing of Basic UNIX Commands |  |  |  |
| 2 |  | Write programs using following UNIX operating system calls  Fork, exec, getpid, exit, wait, close, stst, opendir and readdir |  |  |  |
| 3 |  | Simulate UNIX commands like cp, ls, grep, etc., |  |  |  |
| 4 |  | Simulate the following CPU scheduling algorithms |  |  |  |
|  | a) | Round Robin scheduling |  |  |  |
|  | b) | **Shortest Patch First (SJF)** |  |  |  |
|  | c) | First Come First Served (FCFS) |  |  |  |
|  | d) | **Priority Scheduling** |  |  |  |
| 5 |  | Implement dynamic priority scheduling algorithm. |  |  |  |
| 6 |  | Assume that there are five jobs with different weights ranging from 1 to 5. Implement  round robin algorithm with time slice equivalent to weight. |  |  |  |
| 7 |  | Implement priority scheduling algorithm. While executing, no process should wait for  more than 10 seconds. If waiting time is more than 10 seconds, that process has to be  executed for atleast 1 second before waiting again |  |  |  |
| 8 | a) | Control the number of ports opened by the operating system with  . |  |  |  |
|  | b) | Semaphore |  |  |  |
| 9 |  | Simulate how parent and child processes use shared memory and address space. |  |  |  |
| 10 |  | Simulate sleeping barber problem. |  |  |  |
| 11 |  | Simulate dining philosopher’s problem. |  |  |  |
| 12 |  | Simulate producer and consumer problem using threads. |  |  |  |
| 13 |  | Implement the following memory allocation methods for fixed partition |  |  |  |
|  | a) | First fit |  |  |  |
|  | b) | Worst fit |  |  |  |
|  | c) | Best fit |  |  |  |
| 14 |  | Simulate the following page replacement algorithms |  |  |  |
|  | a) | FIFO |  |  |  |
|  | b) | LRU |  |  |  |
|  | c) | LFU |  |  |  |
| 15 |  | Simulate Paging Technique of memory management |  |  |  |
| 16 |  | Simulate Bankers Algorithm for Dead Lock avoidance and prevention |  |  |  |
| 17 |  | Simulate following file allocation strategies |  |  |  |
|  | a) | Sequential |  |  |  |
|  | b) | Indexed |  |  |  |
|  | c) | Linked |  |  |  |
| 18 |  | Simulate all File Organization Techniques |  |  |  |
|  | a) | Single level directory |  |  |  |
|  | b) | Two level |  |  |  |
|  | c) | Hierarchical |  |  |  |
|  | d) | DAG |  |  |  |

***Signature of the Staff member***:

|  |  |
| --- | --- |
| Exp.No. 1 | **Practicing of Basic UNIX Commands** |
| Dt. |

**AIM:**  **Practicing of Basic UNIX Commands.**

**Description:** A command is an instruction to the computer, which it interprets to perform a specific task. Most commonly a command is a directive to the command-line interface such as Shell.

**UNIX Commands**

Login Unix When you first connect to a Unix system, you usually see a prompt such as the following:

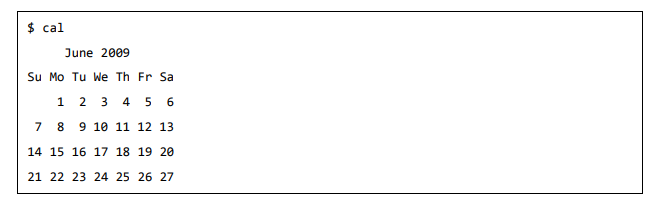


**To log in**

* Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
* Type your userid at the login prompt, then press ENTER. Your userid is casesensitive, so be sure you type it exactly as your system administrator has instructed.
* Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.
* If you provide the correct userid and password, then you will be allowed to enter into the system. Read the information and messages that comes up on the screen, which is as follows.

You will be provided with a command prompt (sometime called the $ prompt ) where you type all your commands.

For example, to check calendar, you need to type the cal command as follows –



Change Password

All Unix systems require passwords to help ensure that your files and data remain your own

and that the system itself is secure from hackers and crackers. Following are the steps to

change your password –

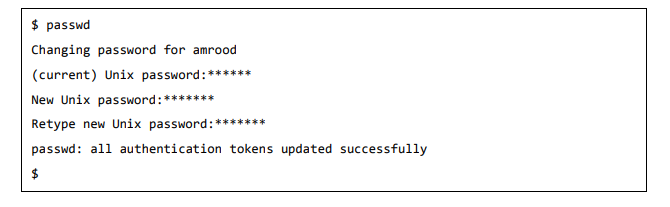
Step 1: To start, type password at the command prompt as shown below.

Step 2: Enter your old password, the one you're currently using.

Step 3: Type in your new password. Always keep your password complex enough so that

nobody can guess it. But make sure, you remember it.

Step 4: You must verify the password by typing it again.



Listing Directories and Files

All data in Unix is organized into files. All files are organized into directories. These directories

are organized into a tree-like structure called the filesystem.

You can use the ls command to list out all the files or directories available in a directory.

Following is the example of using ls command with -l option.



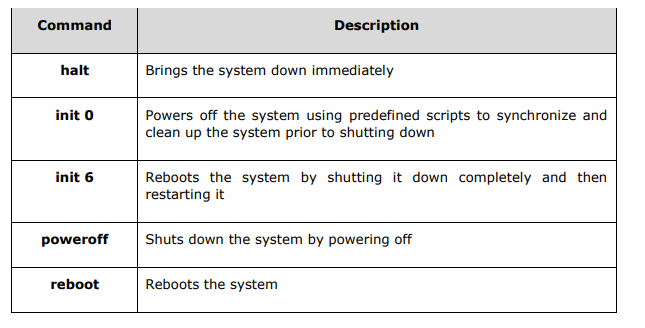
To log out

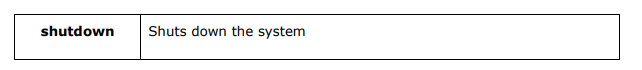
* Just type the logout command at the command prompt, and the system will clean up everything and break the connection.

**System Shutdown**

The most consistent way to shut down a Unix system properly via the command line is to use

one of the following commands −





|  |  |
| --- | --- |
| Exp.No. 2 | Write programs using following UNIX operating system calls  Fork, exec, getpid, exit, wait, close, stst, opendir and readdir |
| Dt. |

**AIM:** Write programs using following UNIX operating system calls

Fork, exec, getpid, exit, wait, close, stst, opendir and readdir

**Description:**

**Fork** system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

**Proram: for fork() system call**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// make two process which run same

// program after this instruction

fork();

printf("Hello world!\n");

return 0;

}

**Output:**

Hello world!

Hello world!

**Description:**

**exec** is used when the user wants to launch a new file or program in the same process.

### Inner Working of exec

Consider the following points to understand the working of exec:

1. Current process image is overwritten with a new process image.
2. New process image is the one you passed as exec argument
3. The currently running process is ended
4. New process image has same process ID, same environment, and same file descriptor (because process is not replaced process image is replaced)
5. The CPU stat and virtual memory is affected. Virtual memory mapping of the current process image is replaced by virtual memory of new process image.

**Proram:**

### Example 1: Using exec system call in C program

Consider the following example in which we have used exec system call in C programming in Linux, Ubuntu: We have two c files here example.c and hello.c:

#### ****example.c****

**CODE:**

*#include <stdio.h>*  
*#include <unistd.h>*  
*#include <stdlib.h>*  
int main(int argc, char \*argv[])  
{  
    [printf](https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of example.c = %d\n", getpid());  
    char \*args[] = {"Hello", "C", "Programming", NULL};  
    execv("./hello", args);  
    [printf](https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Back to example.c");  
    return 0;  
}

#### ****hello.c****

**CODE:**

*#include <stdio.h>*  
*#include <unistd.h>*  
*#include <stdlib.h>*  
int main(int argc, char \*argv[])  
{  
    [printf](https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("We are in Hello.c\n");  
    [printf](https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of hello.c = %d\n", getpid());  
    return 0;  
}

**Output:**

**OUTPUT:**

PID of example.c = 4733  
We are in Hello.c  
PID of hello.c = 4733

**Description:**

**getppid() :** returns the process ID of the parent of the calling process. If the calling process was created by the [fork()](https://www.geeksforgeeks.org/fork-system-call/) function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for **init** process.  
**Syntax:**

**pid\_t getppid(void);**

**Return type:** getppid() returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

**Proram:**

#include <iostream>

#include <unistd.h>

using namespace std;

// Driver Code

int main()

{

int pid;

pid = fork();

if (pid == 0)

{

cout << "\nParent Process id : "

<< getpid() << endl;

cout << "\nChild Process with parent id : "

<< getppid() << endl;

}

return 0;

}

**Output:**

Parent Process id of current process : 3849

Child Process with parent id : 3851

**Description:**

The function \_exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process’s parent is sent a SIGCHLD signal.

The value *status* is returned to the parent process as the process’s exit status, and can be collected using one of the wait() family of calls.

The function \_Exit() is equivalent to \_exit().

**Proram:**

void exit(int status);

**Output:**

**status**

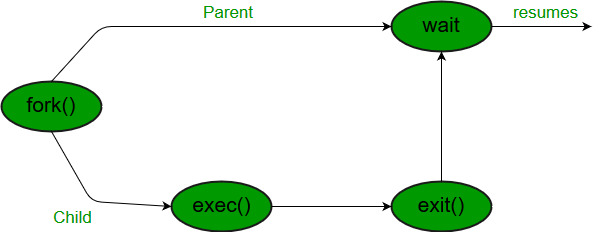
Indicates whether the program terminated normally. It can be one of the following:

| Value | Description |
| --- | --- |
| EXIT\_SUCCESS | Successful termination |
| 0 | Successful termination |
| EXIT\_FAILURE | Unsuccessful termination |

**Description:**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.   
Child process may terminate due to any of these:

* It calls exit();
* It returns (an int) from main
* It receives a signal (from the OS or another process) whose default action is to terminate.



**Proram:**

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>

int main()

{

    pid\_t cpid;

    if (fork()== 0)

        exit(0);           /\* terminate child \*/

    else

        cpid = wait(NULL); /\* reaping parent \*/

    printf("Parent pid = %d\n", getpid());

    printf("Child pid = %d\n", cpid);

    return 0;

}

**Output:**

Parent pid = 12345678

Child pid = 89546848

**Description:**

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see fcntl(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using unlink(2) the file is deleted.

**close**() returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

**Description:**

Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by Stat() function. An inode contains the metadata of the file. An inode contains: the type of the file, the size of the file, when the file was accessed (modified, deleted) that is time stamps, and the path of the file, the user ID and the group ID, links of the file, and physical address of file content.

**Proram:**

The stat structure which is defined in <sys/stat.h> header file contains the following fields:

struct stat  
{  
  mode\_t         st\_mode;  
  ino\_t          st\_ino;  
  dev\_t          st\_dev;  
  dev\_t          st\_rdev;  
  nlink\_t        st\_nlink;  
  uid\_t          st\_uid;  
  gid\_t          st\_gid;  
  off\_t          st\_size;  
  struct timspec st\_atim;  
  struct timspec st\_mtim;  
  struct timspec st\_ctim;  
  blksize\_t      st\_blksize;  
  blkcnt\_t       st\_blocks;  
};

1. st\_dev: It is the ID of device in which we have our file residing currently.
2. st\_rdev: This field describes that a particular file represents a particular device.
3. st\_ino: It is the inode number or the serial number of the file. As it is an index number so it should be unique for all files
4. st\_size: st\_size is the size of the file in bytes.
5. st\_atime: It is the last time or the recent time at which the file was accessed.
6. st\_ctime: It is the recent time at which the status or the permissions of the file was changed.
7. st\_mtime: It is the recent time at which the file was modified.
8. st\_blksize: This field gives the preferred block size for I/O file system which may vary from file to file.
9. st\_blocks: This field tells the total number of blocks in multiples of 512 bytes.
10. st\_nlink: This field tells the total number of hard links.
11. st\_uid: This field indicates the user ID.
12. st\_gid: This field indicates the group ID**.**

#include<stdio.h>  
#include<sys/stat.h>  
int main()  
{  
  //pointer to stat struct  
  struct stat sfile;  
  
  //stat system call  
  stat("stat.c", &sfile);  
  
  //accessing st\_mode (data member of stat struct)    
  [printf](https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("st\_mode = %o", sfile.st\_mode);  
  return 0;  
}

**Description:**

opendir - open a directory

**Proram:**

#include <[dirent.h](https://pubs.opengroup.org/onlinepubs/009695399/basedefs/dirent.h.html)>  
  
DIR \*opendir(const char \**dirname*);

The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN\_MAX} files and directories.

**Output:**

Upon successful completion, *opendir*() shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

**Description:**

**readdir**() reads one *dirent* structure from the directory pointed at by *fd* into the memory area pointed to by *dirp*. The parameter *count* is ignored; at most one dirent structure is read.

The *dirent* structure is declared as follows:

|  |
| --- |
| struct dirent  {  long d\_ino; /\* inode number \*/  off\_t d\_off; /\* offset to this *dirent* \*/  unsigned short d\_reclen; /\* length of this *d\_name* \*/  char d\_name [NAME\_MAX+1]; /\* filename (null-terminated) \*/  } |

*d\_ino* is an inode number. *d\_off* is the distance from the start of the directory to this *dirent*. *d\_reclen* is the size of *d\_name*, not counting the null terminator. *d\_name* is a null-terminated filename.

**Proram:**

On success, 1 is returned. On end of directory, 0 is returned. On error, -1 is returned, and *errno* is set appropriately.

|  |  |
| --- | --- |
| Exp.No. 3 | Simulate UNIX commands like cp, ls, grep, etc., |
| Dt. |

**AIM:**  **Simulate UNIX commands like cp, ls, grep, etc.,**

**Description:**

**cp**- copy- it is used to copying contents from one file to another file.

**Syntax:**

**cp Source Destination**

**ls**-list- its used to list all file names in the directory.

**Syntax:**

**ls**

**grep-**The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern.

**Syntax:**

**grep [options] pattern [files]**

**Options Description**

**-c** : This prints only a count of the lines that match a pattern

**-h :** Display the matched lines, but do not display the filenames.

**-i :** Ignores, case for matching

**-l :** Displays list of a filenames only.

**-n :** Display the matched lines and their line numbers.

Below code is included code of  both cp & ls command.

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<dirent.h>

#define DATA\_SIZE 1000

void createf()

{ char data[DATA\_SIZE];

char n[100];

FILE \* fPtr;

int i;

printf("create 2 files \nfile1: with data \nfile2: without data for copying\n");

for ( i=0;i<2;i++){

printf("enter a file name:");

gets(n);

fPtr = fopen(n,"w");

if(fPtr == NULL)

{ printf("Unable to create file.\n");

exit(EXIT\_FAILURE);

}

printf("Enter contents to store in file : \n");

fgets(data, DATA\_SIZE, stdin);

fputs(data, fPtr);

fclose(fPtr);

printf("File created and saved successfully. ?? \n");

}

}

void copyfun(){

char ch, source\_file[20], target\_file[20];

FILE \*source, \*target;

printf("Enter name of file to copy\n");

gets(source\_file);

source = fopen(source\_file, "r");

if (source == NULL)

{

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

printf("Enter name of target file\n");

gets(target\_file);

target = fopen(target\_file, "w");

if (target == NULL)

{

fclose(source);

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

while ((ch = fgetc(source)) != EOF)

fputc(ch, target);

printf("File copied successfully.\n");

fclose(source);

fclose(target);

}

void lsandgrep(){

char fn[10],pat[10],temp[200];

FILE \*fp;

char dirname[10];

DIR\*p;

struct dirent \*d;

printf("Enter directory name\n");

scanf("%s",dirname);

p=opendir(dirname);

if(p==NULL)

{

perror("Cannot find directory");

exit(0);

}

while(d=readdir(p))

printf("%s\n",d->d\_name);

}

int main(){

createf();

copyfun();

lsandgrep();

}

output:

**create 2 files**

**file1: with data**

**file2: without data for copying**

**enter a file name:a.txt**

**Enter contents to store in file :**

**one two three**

**File created and saved successfully. ??**

**enter a file name:b.txt**

**Enter contents to store in file :**

**four**

**File created and saved successfully. ??**

**Enter name of file to copy**

**a.txt**

**Enter name of target file**

**b.txt**

**File copied successfully.**

**Enter directory name**

**PROJECT**

**.**

**..**

**COUNT\_LINE.c**

**COUNT\_LINE.txt**

**fsearch.c**

**fsearch.exe**

**LINE.c**

**LINE.exe**

**LINE.txt**

**tn.txt**

Below code is included code of **Grep**command **n**-  pattern

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include<dirent.h>

#include<conio.h>

#define DATA\_SIZE 1000

#define BUFFER\_SIZE 1000

void countch(){

FILE \*fptr;

char path[100];

char word[50];

int wCount;

printf("Enter file name: ");

scanf("%s", path);

printf("Enter word to search in file: ");

scanf("%s", word);

fptr = fopen(path, "r");

if (fptr == NULL)

{

printf("Unable to open file.\n");

printf("Please check you have read/write previleges.\n");

exit(EXIT\_FAILURE);

}

Search\_in\_File(fptr, word);

printf("%d",wCount);

fclose(fptr);

}

int Search\_in\_File(char \*fname, char \*str)

{

FILE \*fp;

int line\_num = 1;

int find\_result = 0;

char temp[512];

if((fopen(fname, "r")) != NULL) {

return(-1);

}

while(fgets(temp, 512, fp) != NULL) {

if((strstr(temp,str)) != NULL) {

line\_num=line\_num+0;

printf("\n%s\n", temp);

find\_result++;

}

line\_num++;

}

if(find\_result == 0) {

printf("\nSorry, couldn't find a match.\n");

}

if(fp) {

fclose(fp);

}

return(0);

}

int main(){

countch();

getch();

return 0;

}

**Program:**  
Below code is included code of **Grep**command **c**-  pattern

#include<stdio.h>

#define MAX\_FILE\_NAME 100

#include<conio.h>

int main()

{

FILE \*fp;

int count = 0;

char filename[MAX\_FILE\_NAME];

char c;

printf("Enter file name: ");

scanf("%s", filename);

fp = fopen(filename, "r");

if (fp == NULL)

{

printf("Could not open file %s", filename);

return 0;

}

for (c = getc(fp); c != EOF; c = getc(fp))

if (c == '\n')

count = count + 1;

fclose(fp);

printf("The file %s has %d lines\n ", filename, count);

getch();

return 0;}

**Output:**

Enter file name: tn.txt

The file tn.txt has 4 lines

**program:**

Below code is included code of **Grep**command**h**-  pattern.

#include <stdio.h>

#include <stdlib.h>

#include<dirent.h>

#include<conio.h>

#define DATA\_SIZE 1000

#define BUFFER\_SIZE 1000

void countch(){

FILE \*fptr;

char path[100];

char word[50];

int wCount;

printf("Enter file name: ");

scanf("%s", path);

printf("Enter word to search in file: ");

scanf("%s", word);

fptr = fopen(path, "r");

if (fptr == NULL)

{

printf("Unable to open file.\n");

printf("Please check you have read/write previleges.\n");

exit(EXIT\_FAILURE);

}

Search\_in\_File(fptr,word);

printf("%d",wCount);

fclose(fptr);

}

int Search\_in\_File(char \*fname, char \*str)

{

FILE \*fp;

int line\_num = 1;

int find\_result = 0;

char temp[512];

if((fopen(fname, "r")) != NULL) {

return(-1);

}

while(fgets(temp, 512, fp) != NULL) {

if((strstr(temp,str)) != NULL) {

line\_num=line\_num+0;

// printf("\n%s\n", temp);

if(temp!=str){

printf("\n%s\n", temp);

}

find\_result++;

}

line\_num++;

}

if(find\_result == 0) {

printf("\nSorry, couldn't find a match.\n");

}

if(fp) {

fclose(fp);

}

return(0);

}

int main(){

countch();

getch();

return 0;

}

**program:**

Below code is included code of **Grep**command**i**-  pattern.

#include <stdio.h>

#include <stdlib.h>

#include<dirent.h>

#include<conio.h>

#define DATA\_SIZE 1000

#define BUFFER\_SIZE 1000

void countch(){

FILE \*fptr;

char path[100];

char word[50];

int wCount;

printf("Enter file name: ");

scanf("%s", path);

printf("Enter word to search in file: ");

scanf("%s", word);

fptr = fopen(path, "r");

if (fptr == NULL)

{

printf("Unable to open file.\n");

printf("Please check you have read/write previleges.\n");

exit(EXIT\_FAILURE);

}

Search\_in\_File(fptr,tolower(word));

printf("%d",wCount);

fclose(fptr);

}

int Search\_in\_File(char \*fname, char \*str)

{

FILE \*fp;

int line\_num = 1;

int find\_result = 0;

char temp[512];

if((fopen(fname, "r")) != NULL) {

return(-1);

}

while(fgets(temp, 512, fp) != NULL) {

if((strstr(temp,str)) != NULL) {

line\_num=line\_num+0;

printf("\n%s\n", temp);

find\_result++;

}

line\_num++;

}

if(find\_result == 0) {

printf("\nSorry, couldn't find a match.\n");

}

if(fp) {

fclose(fp);

}

return(0);

}

int main(){

countch();

getch();

return 0;

}

**program:**

Below code is included code of **Grep**command**l**-  pattern.

#include <stdio.h>

#include <stdlib.h>

#include<dirent.h>

void l(){

char fn[10],pat[10],temp[200];

FILE \*fp;

char dirname[10];

DIR\*p;

struct dirent \*d;

printf("Enter directory name\n");

scanf("%s",dirname);

p=opendir(dirname);

if(p==NULL)

{

perror("Cannot find directory");

exit(0);

}

while(d=readdir(p))

printf("%s\n",d->d\_name);

}

void main(){

l();

}

**Output:**

**nter directory name**

**PROJECT**

**.**

**..**

**COUNT\_LINE.c**

**COUNT\_LINE.txt**

**fsearch.c**

**fsearch.exe**

**LINE.c**

**LINE.exe**

**LINE.txt**

**tn.txt**

|  |  |
| --- | --- |
| Exp.No. 4 (a) | **ROUND ROBIN SCHEDULING** |
| Dt. |

**AIM:**  **Write a C program to implement the various process scheduling mechanisms such as Round Robin Scheduling.**

**Description:**

Scheduling is a fundamental operating system function.

CPU scheduling is the basis of multi programming operating system. CPU scheduling algorithm determines how the CPU will be allocated to the process. These are of two types.

1. Primitive scheduling algorithms
2. Non-Primitive scheduling algorithms
3. **Primitive Scheduling algorithms**: In this, the CPU can release the process even in themiddle of execution. For example: the cpu executes the process p1, in the middle of execution the cpu received a request signal from process p2, then the OS compares the priorities of p1&p2. If the priority p1 is higher than the p2 then the cpu continue the execution of process p1.Otherwise the cpu preempt the process p1 and assigned to process p2.
4. **Non-Primitive Scheduling algorithm**: In this, once the cpu assigned to a process theprocessor do not release until the completion of that process. The cpu will assign to some other job only after the previous job has finished.

**Scheduling methodology:**

**Though put:** It means how many jobs are completed by the CPU with in a time period. **Turn around time:** The time interval between the submission of the process and the time ofthe completion is the turn around time.

**Turn around time=Finished time – arrival time**

**Waiting time:** it is the sum of the periods spent waiting by a process in the ready queue

**Waiting time=Starting time- arrival time**

**Response time**: it is the time duration between the submission and first response

**Response time=First response-arrival time**

**CPU Utilization:** This is the percentage of time that the processor is busy. CPU utilizationmay range from 0 to 100%.

**Round Robin**: It is a primitive scheduling algorithm it is designed especially for time sharingsystems. In this, the CPU switches between the processes. When the time quantum expired, the CPU switches to another job. A small unit of time called a quantum or time slice. A time quantum is generally is a circular queue new processes are added to the tail of the ready queue.

If the process may have a CPU burst of less than one time slice then the process release

the CPU voluntarily. The scheduler will then process to next process ready queue otherwise; the process will be put at the tail of the ready queue**.**

**Algorithm for Round Robin:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

1. Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1 ) + the time difference in getting the CPU from process(n-1)
2. Turn around time for process(n) = waiting time of process(n) + burst time of

process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process.

**/\* Program to Simulate Round Robin CPU Scheduling Algorithm \*/**

#include<stdio.h>

#include<conio.h>

struct process

{

char pn[10]; int bt,ct,time;

}p[10]; void main()

{

int i,full,n,tq,wt[10],tat[10], time1=0; float avgwt=0.0;

clrscr();

printf("Enter number of processes:");

scanf("%d",&n);

printf("Enter process name and burst time of %d process\n", n);

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

{

scanf("%s%d",&p[i].pn,&p[i].bt);

p[i].time=p[i].bt;

}

printf("Enter quantum:"); scanf("%d",&tq); full=n;

while(full)

{

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

{

if(p[i].bt>=tq)

{

p[i].bt-=tq;

|  |  |
| --- | --- |
| time1=time1+tq; |  |
| } |  |
| else if(p[i].bt!=0) |  |
| { |  |
| time1+=p[i].bt; |  |
| p[i].bt=0; |  |
| } |  |
| else |  |
| continue; |  |
| if(p[i].bt==0) |  |
| { |  |
| full=full-1; |  |
| tat[i]=time1; |  |
| } |  |
| } |  |
| } |  |
| for(i=0;i<n;i++) |  |
| { |  |
| p[i].ct=tat[i]; |  |
| wt[i]=tat[i]-p[i].time; |  |
| } |  |
| printf("---------------------------------- | \n"); |
| printf("PN\tBt\tCt\tTat\tWt\n"); |  |
| printf("---------------------------------- | \n"); |
| for(i=0;i<n;i++) |  |
| { |  |

printf("%2s\t%2d\t%2d\t%2d\t%2d\n",p[i].pn,p[i].time,p[i].ct,tat[i],wt[i]);

avgwt+=wt[i];

}

printf("----------------------------------\n");

avgwt=avgwt/n;

printf(" Average waiting time = %.2f\n",avgwt);

printf("-----------------------------------\n");

getch();

}

**OUTPUT 1:**

Enter number of processes: 5

Enter process name and burst time of 5 process

1. 10
2. 5
3. 15
4. 3
5. 20

Enter quantum: 5

--------------------------------------

PN Bt Ct Tat Wt

--------------------------------------

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 10 | 28 | 28 | 18 |
| 2 | 5 | 10 | 10 | 5 |
| 3 | 15 | 43 | 43 | 28 |
| 4 | 3 | 18 | 18 | 15 |
| 5 | 20 | 53 | 53 | 33 |

--------------------------------------

Average waiting time = 19.79

--------------------------------------

**OUTPUT 2:**

Enter number of processes:3 Enter process name and burst time of 3 process

1. 24
2. 3
3. 3

Enter quantum: 4

--------------------------------------

PN Bt Ct Tat Wt

--------------------------------------

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 24 | 30 | 30 | 6 |
| 2 | 3 | 7 | 7 | 4 |
| 3 | 3 | 10 | 10 | 7 |

--------------------------------------

Average waiting time = 5.67

--------------------------------------

|  |  |
| --- | --- |
| Exp.No. 4 (b) | **SJF SCHEDULING** |
| Dt. |

**Aim: Write a C program to implement the various process scheduling mechanisms such as SJF Scheduling .**

**Description:**

**Shortest Job First:** The criteria of this algorithm are which process having the smallest CPUburst, CPU is assigned to that next process. If two process having the same CPU burst time FCFS is used to break the tie.

**Algorithm for SJF:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to

highest burst time.

Step 5: Set the waiting time of the first process as ‘0’ and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

1. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
2. Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process
3. Step 7: Stop the process

**/\* Program to Simulate Shortest Job First CPU Scheduling Algorithm \*/**

#include<stdio.h>

#include<conio.h>

#include<string.h> void main()

{

int i,j,n,bt[10],compt[10], wt[10],tat[10],temp; float sumwt=0.0,sumtat=0.0,avgwt,avgtat; clrscr();

printf("Enter number of processes: "); scanf("%d",&n);

printf("Enter the burst time of %d process\n", n); for(i=0;i<n;i++)

{

scanf("%d",&bt[i]);

|  |  |
| --- | --- |
| } |  |
| for(i=0;i<n;i++) |  |
| for(j=i+1;j<n;j++) |  |
| if(bt[i]>bt[j]) |  |
| { |  |
| temp=bt[i]; |  |
| bt[i]=bt[j]; |  |
| bt[j]=temp; |  |
| } |  |
| compt[0]=bt[0]; |  |
| for(i=1;i<n;i++) |  |
| compt[i]=bt[i]+compt[i-1]; |  |
| for(i=0;i<n;i++) |  |
| { |  |
| tat[i]=compt[i]; |  |
| wt[i]=tat[i]-bt[i]; |  |
| sumtat+=tat[i]; |  |
| sumwt+=wt[i]; |  |
| } |  |
| avgwt=sumwt/n; |  |
| avgtat=sumtat/n; |  |
| printf("------------------------------ | \n"); |
| printf("Bt\tCt\tTat\tWt\n"); |  |
| printf("------------------------------ | \n"); |
| for(i=0;i<n;i++) |  |
| { |  |

printf("%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);

}

printf("------------------------------\n");

printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);

|  |  |  |  |
| --- | --- | --- | --- |
| printf("------------------------------- | |  | \n"); |
| getch(); | |  |  |
| } |  |  |  |
| **OUTPUT 1:** | |  |  |
| Enter number of processes: 4 | | | |
| Enter the burst time of 4 process | | | |
| 6 8 7 3 |  |  |  |
| ------------------------------------ | | | |
| Bt | Ct | Tat | Wt |
| ------------------------------------ | | | |
| 3 | 3 | 3 | 0 |
| 6 | 9 | 9 | 3 |
| 7 | 16 | 16 | 9 |
| 8 | 24 | 24 | 16 |

--------------------------------------

|  |  |
| --- | --- |
| Avgwt = 7.00 Avgtat = | 13.00 |

|  |  |  |  |
| --- | --- | --- | --- |
| **OUTPUT 2:** | |  |  |
| Enter number of processes: 4 | | | |
| Enter the burst time of 4 process | | | |
| 8 4 9 5 |  |  |  |
| ------------------------------------ | | | |
| Bt | Ct | Tat | Wt |
| ------------------------------------ | | | |
| 4 | 4 | 4 | 0 |
| 5 | 9 | 9 | 4 |
| 8 | 17 | 17 | 9 |
| 9 | 26 | 26 | 17 |
| ------------------------------------ | | | |
| Avgwt = 7.50 | | Avgtat = 14.00 | |
| ------------------------------------ | | | |

|  |  |
| --- | --- |
| Exp.No. 4 (c) | **FCFS SCHEDULING** |
| Dt. |

**Aim: Write a C program to implement the various process scheduling mechanisms such**

**Description:**

**First-come, first-serve scheduling(FCFS):** In this, which process enter the ready queue firstis served first. The OS maintains DS that is ready queue. It is the simplest CPU scheduling algorithm. If a process request the CPU then it is loaded into the ready queue, which process is the head of the ready queue, connect the CPU to that process.

**Algorithm for FCFS scheduling:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

1. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
2. Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 7: Stop the process

**/\* Program to Simulate First Come First Serve CPU Scheduling Algorithm \*/**

#include<stdio.h>

#include<conio.h>

#include<string.h> void main()

{

int i,j,n,bt[10],compt[10],at[10], wt[10],tat[10]; float sumwt=0.0,sumtat=0.0,avgwt,avgtat; clrscr();

printf("Enter number of processes: "); scanf("%d",&n);

printf("Enter the burst time of %d process\n", n); for(i=0;i<n;i++)

{

scanf("%d",&bt[i]);

}

printf("Enter the arrival time of %d process\n", n); for(i=0;i<n;i++)

|  |  |
| --- | --- |
| { |  |
| scanf("%d",&at[i]); |  |
| } |  |
| compt[0]=bt[0]-at[0]; |  |
| for(i=1;i<n;i++) |  |
| compt[i]=bt[i]+compt[i-1]; |  |
| for(i=0;i<n;i++) |  |
| { |  |
| tat[i]=compt[i]-at[i]; |  |
| wt[i]=tat[i]-bt[i]; |  |
| sumtat+=tat[i]; |  |
| sumwt+=wt[i]; |  |
| } |  |
| avgwt=sumwt/n; |  |
| avgtat=sumtat/n; |  |
| printf("---------------------------------- | \n"); |
| printf("PN\tBt\tCt\tTat\tWt\n"); |  |
| printf("---------------------------------- | \n"); |
| for(i=0;i<n;i++) |  |
| { |  |

printf("%d\t%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);

}

printf("----------------------------------\n");

printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);

printf("-----------------------------------\n");

getch();

}

**OUTPUT 1:**

Enter number of processes: 5 Enter the burst time of 5 process 3 6 4 5 2

Enter the arrival time of 5 process 0 2 4 6 8

----------------------------------------

PN Bt Ct Tat Wt

----------------------------------------

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 3 | 3 | 3 | 0 |
| 1 | 6 | 9 | 7 | 1 |
| 2 | 4 | 13 | 9 | 5 |
| 3 | 5 | 18 | 12 | 7 |
| 4 | 2 | 20 | 12 | 10 |

---------------------------------------

Avgwt = 4.60 Avgtat = 8.60

---------------------------------------

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **OUTPUT 2:** | |  |  |  |  |
| Enter number of processes: 5 | | | | |  |
| Enter the burst time of 5 process | | | | |  |
| 5 24 16 10 3 | | |  |  |  |
| Enter the arrival time of 5 process | | | | |  |
| 0 0 0 0 0 | |  |  |  |  |
| -------------------------------------- | | | | |  |
| PN | Bt | Ct | Tat | Wt |  |
| -------------------------------------- | | | | |  |
| 0 | 5 | 5 | 5 | 0 |  |
| 1 | 24 | 29 | 29 | 5 |  |
| 2 | 16 | 45 | 45 | 29 |  |
| 3 | 10 | 55 | 55 | 45 |  |
| 4 | 3 | 58 | 58 | 55 |  |

--------------------------------------

Avgwt = 26.80 Avgtat = 38.40

---------------------------------------