WEEK 1 & 2

Linux Basic commands:

ls - list the contents of a directory

cd - change the current working directory

pwd - print the current working directory

mkdir - make a new directory

rmdir - remove an empty directory

touch - create a new file or update the modification time of an existing file

cp - copy a file or directory

mv - move a file or directory

rm - remove a file or directory

echo - display a message or the value of a variable

cat - concatenate and display files

less - view a file one screen at a time

head - display the first few lines of a file

tail - display the last few lines of a file

grep - search for a pattern in a file

find - search for files based on criteria such as name, size, or time modified

chmod - change the permissions of a file or directory

chown - change the owner of a file or directory

ps - display information about the running processes

top - display real-time information about the system and running processes

These commands are just a few of the many available in Linux. Practical exposure can be gained by using them in a terminal emulator, and by solving problems through the command line.

1.“ls" is a Linux command used to list the contents of a directory. By default, it displays the files and directories in the current working directory. You can use it to list the contents of other directories as well by specifying the path to the directory as an argument:

ls <directory>

Some common options you can use with ls are:

-l: display the output in a long format, showing details such as permissions, owner, group, size, and modification time

-**a**: show hidden files and directories

-h: display file sizes in a human-readable format

-r: display the contents in reverse order

-t: sort the contents by modification time, with the newest files appearing first

Here is an example usage:

ls -lh

This will list the contents of the current directory in a long format, showing file sizes in a human-readable format.

2."cd" is a Linux command used to change the current working directory. The current working directory is the directory that you are currently in, and all relative file and directory paths are based on this directory.

To change to a different directory, simply type cd followed by the path to the desired directory:

cd <directory>

Here are some special directories you can use with cd:

.: refers to the current directory

..: refers to the parent directory

~: refers to your home directory

For example:

cd ~/Desktop

This will change the current working directory to your Desktop directory inside your home directory.

You can also use cd without any arguments to change back to your home directory:

cd

3. "pwd" is a Linux command that stands for "print working directory". It displays the full path of the current working directory on the terminal.

When you run pwd in the terminal, it will show you the absolute path of the directory that you are currently in. This can be useful when you are not sure where you are in the file system and need to know your current location.

Here is an example usage:

pwd

This will display the absolute path of the current working directory in the terminal.

4."mkdir" is a Linux command used to create a new directory. To create a directory, simply type mkdir followed by the name of the directory:

mkdir <directory_name>

For example:

mkdir test_directory

This will create a new directory with the name test_directory in the current working directory.

You can also create multiple directories at once by specifying multiple names:

mkdir dir1 dir2 dir3

This will create three directories, dir1, dir2, and dir3, in the current working directory.

It is also possible to create directories with subdirectories using the -p option:

mkdir -p parent/child/grandchild

This will create a directory parent, with a subdirectory child, which contains a subdirectory grandchild.

5."rmdir" is a Linux command used to remove an empty directory. To remove a directory, simply type rmdir followed by the name of the directory:

rmdir <directory_name>

For example:

rmdir test_directory

This will remove the empty directory test_directory from the file system.

It is important to note that rmdir can only remove empty directories. If a directory contains files or subdirectories, you will need to remove them first before using rmdir.

You can use the rm command to remove non-empty directories and their contents:

rm -r <directory_name>

This will remove the directory and all of its contents, including subdirectories and files. Use this command with caution, as it permanently deletes the data.

6."touch" is a Linux command used to create a new, empty file or to update the modification time of an existing file. To create a new file, simply type touch followed by the name of the file:

touch <file_name>

For example:

touch test_file.txt

This will create a new, empty file named test_file.txt in the current working directory.

If the file already exists, touch will update its modification time to the current time, which can be useful for various purposes, such as marking a file as recently accessed.

You can also create multiple files at once by specifying multiple names:

touch file1.txt file2.txt file3.txt

This will create three empty files, file1.txt, file2.txt, and file3.txt, in the current working directory.

7."cp" is a Linux command used to copy files and directories. To copy a file or directory, simply type cp followed by the source file or directory and the destination:

cp <source> <destination>

For example:

cp file1.txt file2.txt

This will copy the file file1.txt to a new file named file2.txt in the current working directory.

You can also copy a directory and its contents by using the -r (or --recursive) option:

cp -r <source_directory> <destination_directory>

For example:

cp -r src_dir dest_dir

This will copy the directory src_dir and all of its contents, including subdirectories and files, to a new directory named dest_dir in the current working directory.

It is important to note that the cp command only creates a copy of the files, and the original files are not deleted or modified.

8."mv" is a Linux command used to move files and directories or to rename files and directories.

To move a file or directory to a new location, simply type mv followed by the source file or directory and the destination:

mv <source> <destination>

For example:

mv file1.txt ~/Desktop

This will move the file file1.txt from the current working directory to the Desktop directory in the home directory.

To rename a file or directory, simply specify the new name as the destination:

mv <source> <new_name>

For example:

mv file1.txt file2.txt

This will rename the file file1.txt to file2.txt in the current working directory.

It is important to note that the mv command only moves or renames the files and directories, and the original files and directories are deleted or modified.

9."rm" is a Linux command used to remove files and directories. To remove a file, simply type rm followed by the name of the file:

rm <file_name>

For example:

rm test_file.txt

This will remove the file test_file.txt from the file system.

To remove a directory and its contents, including subdirectories and files, use the -r (or --recursive) option:

rm -r <directory_name>

For example:

rm -r test_directory

This will remove the directory test_directory and all of its contents from the file system.

It is important to note that the rm command permanently deletes the data, so use it with caution. You may want to use the -i (or --interactive) option to prompt for confirmation before removing each file or directory.

10."echo" is a Linux command used to display text or the contents of a variable. To display a message, simply type echo followed by the message in quotes:

echo "Hello, World!"

This will display the message "Hello, World!" in the terminal.

You can also use echo to display the value of a variable by using a $ sign followed by the variable name:

echo $<variable_name>

For example:

message="Hello, World!"

echo $message

This will display the value of the variable message, which is "Hello, World!".

echo is a very versatile command, and it has many options and usage scenarios, such as writing to a file, displaying the value of an environment variable, and more.

11."cat" is a Linux command used to display the contents of a file. To display the contents of a file, simply type cat followed by the name of the file:

cat <file_name>

For example:

cat test_file.txt

This will display the contents of the file test_file.txt in the terminal.

You can also concatenate (combine) the contents of multiple files into a single file by specifying multiple file names:

cat <file1_name> <file2_name> > <new_file_name>

For example:

cat file1.txt file2.txt > combined_file.txt

This will combine the contents of file1.txt and file2.txt into a new file named combined_file.txt.

cat is a very simple but powerful command, and it is often used in combination with other commands and pipelines to manipulate text data.

12."less" is a Linux command used to display the contents of a file one screen at a time. Unlike the cat command, which displays the entire contents of a file, less allows you to navigate through the contents of a large file more easily.

To display the contents of a file using less, simply type less followed by the name of the file:

less <file_name>

For example:

less test_file.txt

This will display the contents of the file test_file.txt in the terminal, one screen at a time. You can use the up and down arrow keys to navigate through the file, or type q to quit.

less has many other options and features, such as searching, highlighting, and jumping to specific lines. It is an essential tool for viewing large or complex text files in the terminal.

13."grep" is a Linux command used to search for patterns in text data. It is a powerful tool for searching through large amounts of data, such as log files, and finding specific lines or patterns that match a given search criteria.

The basic syntax for using grep is as follows:

grep "<pattern>" <file_name>

For example:

grep "error" log_file.txt

This will search the file log_file.txt for lines that contain the word "error" and display only those lines in the terminal.

grep has many options and usage scenarios, such as searching recursively through directories, ignoring case, displaying the line number, and more. It is an essential tool for anyone working with text data in the Linux environment.

14.The "find" command in Linux is used to search for files and directories based on different criteria such as name, type, size, date, etc. It is a versatile command that is commonly used in shell scripts and other automated tasks. Here is a basic syntax for using the "find" command:

find [path] [expression]

[path] is the location where you want to start the search, and it can be a directory or a file.

[expression] is a set of options and tests that define the search criteria.

Here is an example that searches for files with the .txt extension in the current directory and all its subdirectories:

find . -name "*.txt"

You can find more information on how to use the "find" command and its options by typing **“man find”** in the terminal.

15.The “chmod” command in Linux is used to change the permissions on files and directories. Permissions determine who can read, write, or execute a file or directory. The chmod command operates on either symbolic or numeric (octal) representations of permissions.

Here is the basic syntax for using chmod:

chmod [OPTION]... MODE[,MODE]... FILE...

[OPTION]: Options to modify the behavior of chmod.

MODE: The permissions to be set. Can be specified either as a symbolic representation (e.g. u+rwx) or as an octal number (e.g. 755).

FILE: The file or directory whose permissions are to be modified.

Here is an example that changes the permissions of a file named file.txt to allow the owner to read, write, and execute the file, while allowing everyone else to only read the file:

chmod 744 file.txt

In symbolic representation, the same change could be made as follows:

chmod u+rwx,g+r,o+r file.txt

You can find more information on how to use the chmod command and its options by typing man chmod in the terminal.

16.The "chown" command in Linux is used to change the owner and/or group ownership of files and directories. The owner and group ownership determine which user and group have access to the file or directory and what level of access they have.

Here is the basic syntax for using chown:

chown [OPTION]... OWNER[:[GROUP]] FILE...

[OPTION]: Options to modify the behavior of chown.

OWNER: The new owner of the file or directory. Can be specified as a username or numeric user ID.

GROUP: The new group of the file or directory. Can be specified as a group name or numeric group ID.

FILE: The file or directory whose owner and/or group is to be changed.

Here is an example that changes the owner of a file named file.txt to the user johndoe:

chown satya file.txt

Here is an example that changes both the owner and the group of a file named file.txt to johndoe and users, respectively:

chown satya:users file.txt

You can find more information on how to use the chown command and its options by typing man chown in the terminal.

17.The "ps" command in Linux is used to display information about the processes currently running on the system. The ps command provides a snapshot of the process status and is typically used to identify which processes are running, who owns them, and what resources they are consuming.

Here is a basic syntax for using ps:

ps [OPTION]...

[OPTION]: Options to modify the display format and information provided by ps.

Here is an example that displays information about all processes running on the system:

ps aux

The aux options display information about all processes for all users in a format that includes the following columns:

USER: The username of the process owner.

PID: The process ID.

%CPU: The percentage of CPU utilization.

%MEM: The percentage of memory utilization.

VSZ: The virtual memory size of the process.

RSS: The resident set size of the process.

TTY: The terminal associated with the process.

STAT: The status of the process.

START: The start time of the process.

TIME: The cumulative execution time of the process.

COMMAND: The command that started the process.

You can find more information on how to use the ps command and its options by typing man ps in the terminal.

18.The "top" command in Linux is a real-time process monitoring tool that provides information about the system's processes and their resource utilization. The top command continuously updates the display and provides a dynamic view of the system's process status, including the resource utilization of CPU, memory, and swap space.

Here is a basic syntax for using top:

top [OPTION]...

[OPTION]: Options to modify the behavior of top.

When you run the top command, it will display a list of the processes currently running on the system, sorted by the amount of CPU utilization by default. The display includes the following information:

PID: The process ID.

USER: The username of the process owner.

PR: The priority of the process.

NI: The nice value of the process, which determines its priority.

VIRT: The virtual memory size of the process.

RES: The resident set size of the process.

SHR: The shared memory size of the process.

S: The status of the process.

%CPU: The percentage of CPU utilization by the process.

%MEM: The percentage of memory utilization by the process.

TIME+: The cumulative CPU time of the process.

COMMAND: The command that started the process.

The display is updated continuously until you press q to quit the top command. You can also sort the display by various criteria, such as memory utilization, by pressing different keys.

You can find more information on how to use the top command and its options by typing man top in the termina

WEEK 2

The **/proc** file system in Linux provides access to information about the system and its processes. The /proc file system is a virtual file system that provides a view of the current state of the system, including information about hardware, system resources, and running processes.

To find the /proc file system in Linux, you can use the ls command to list the contents of the root directory. The /proc file system should be listed as one of the directories, along with other common directories like /bin, /usr, and /etc.

For example, you can use the following command to list the contents of the root directory:

ls /

Once you have located the /proc file system, you can use the cd command to change to that directory and then use the ls command to list its contents. The contents of the /proc file system are organized into subdirectories, each of which corresponds to a running process or system information.

For example, to list the contents of the /proc directory, you can use the following command:

cd /proc

ls

You can then use the cat command to display the contents of individual files in the /proc file system, as described in my previous answer.

Here are some of the important files in the /proc file system that can be used to gather basic information about your machine:

cat/proc/version: This file contains information about the Linux kernel version, including the release number, the build date, and the compiler used to build the kernel.

cat/proc/cpuinfo: This file provides information about the processor architecture, the number of cores, the clock speed, and other details about the processor.

cat/proc/meminfo: This file provides information about the system's memory, including the total amount of memory, the amount of free memory, the amount of cached memory, and the amount of memory used by buffers.

cat/proc/loadavg: This file provides information about the system load average, which is a measure of the amount of system resources being consumed by processes.

cat/proc/uptime: This file provides information about the system uptime, including the amount of time the system has been running and the amount of time the processor has spent in idle mode.

cat /proc/mounts: Displays information about the mounted file systems and the options used when mounting them.

cat /proc/partitions: Displays information about the disk partitions on the system.

cat /proc/swaps: Displays information about the swap space on the system, including the size of the swap space and the amount of space used.

cat /proc/devices: Displays information about the devices present on the system, including the type and major and minor device numbers.

cat /proc/interrupts: Displays information about the interrupts received by the system, including the number of interrupts received by each processor.

cat /proc/stat: Displays information about system-wide performance statistics, including the number of context switches, the number of processes created, and the amount of CPU time spent in user mode, system mode, and idle mode.

You can access the information in these files by using the cat command in the terminal. For example, to display the information in the /proc/version file, you can run the following command:

cat /proc/version

Note that the information provided by the /proc file system is read-only and cannot be modified.

WEEK 3

Implementation of write () and read () system calls.

Write()

The write() system call in Linux is used to write data to a file or other type of file descriptor. The syntax for the write() system call is as follows:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Where:

- **fd** is the file descriptor of the file or device to write to.
- **buf** is a pointer to a buffer containing the data to be written.
- **count** is the number of bytes to write.

The write() system call returns the number of bytes actually written, which may be less than count if an error occurred or the disk is full. On success, the value returned will be equal to count. On error, write() returns -1 and sets the errno global variable to indicate the error.

Here's an example of how you might use the write() system call in C to write data to a file:

```
#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main()

{

   int fd;

   char data[] = "Hello, world!\n";

   fd = open("file.txt", O_WRONLY | O_CREAT, 0644);

   if (fd == -1)
```

```
    {
        perror("open");
        return 1;
    }
    int bytes_written = write(fd, data, sizeof(data));
    if (bytes_written == -1)
    {
        perror("write");
        return 1;
    }
    close(fd);
    return 0;
}
```

read()

The read() system call in Linux is used to read data from a file or other type of file descriptor. The syntax for the read() system call is as follows:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Where:

- fd is the file descriptor of the file or device to read from.
- buf is a pointer to a buffer where the data will be stored.
- count is the number of bytes to read.

The read() system call returns the number of bytes actually read, which may be less than count if an error occurred or the end of the file has been reached. On success, the value returned will be greater than 0. On error or end of file,

read() returns 0. On error, read() returns -1 and sets the errno global variable to indicate the error.

Here's an example of how you might use the read() system call in C to read data from a file:

Copy code

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#define BUFSIZE 1024
int main()
{
   int fd;
   char buf[BUFSIZE];
   fd = open("file.txt", O_RDONLY);
   if (fd == -1)
   {
      perror("open");
      return 1;
   }
   int bytes_read = read(fd, buf, BUFSIZE);
   if (bytes_read == -1)
   {
      perror("read");
      return 1;
```

```
    }
    close(fd);
    write(1, buf, bytes_read);
    return 0;
}
```

Week 4

Implementation of open (), fork () system calls.

Open()

The open() system call in Linux is used to open a file or other type of file descriptor. The syntax for the open() system call is as follows:

#include <fcntl.h>

int open(const char *path, int flags, ... /* mode_t mode */);

Where:

- **path** is the name of the file or device to open.
- **flags** is a combination of values that specify the mode in which the file is opened. Some common values for flags include **O_RDONLY** for read-only access, **O_WRONLY** for write-only access, and **O_RDWR** for read-write access.

mode (optional) is a value that specifies the permissions to be set on the file if it is created. This argument is only used when **O_CREAT** is included in flags.

The open() system call returns a file descriptor that can be used in subsequent system calls such as read() and write(). On error, open() returns -1 and sets the errno global variable to indicate the error.

Here's an example of how you might use the open() system call in C to open a file for writing:

Example 1

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main()

{

```
    int fd;
    fd = open("file.txt", O_RDONLY);
    if (fd == -1)
    {
        perror("open");
        return 1;
    }
    printf("The file was opened successfully!\n");
    close(fd);
    return 0;
}
```

In this example, the open() function is used to open the file file.txt in read-only mode. The second argument to open() is a set of flags that control the behavior of the function. In this case, the O_RDONLY flag is used to indicate that the file should be opened in read-only mode.

Example 2

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int fd;
    fd = open("file.txt", O_WRONLY | O_CREAT, 0644);
    if (fd == -1)
```

```
    {
        perror("open");
        return 1;
    }
    close(fd);
    return 0;
}
```

fork()

The fork() system call in Linux is used to create a new process. The syntax for the fork() system call is as follows:

```
#include <unistd.h>
pid_t fork(void);
```

The fork() system call creates a new process by duplicating the calling process. The new process, called the child process, is an exact copy of the parent process, with the exception of its process ID. The parent process continues executing after the call to fork(), while the child process begins executing from the same point as the parent.

The return value of fork() is used to determine whether the process is the parent or the child. In the parent process, fork() returns the process ID of the child process. In the child process, fork() returns 0. On error, fork() returns -1 and sets the errno global variable to indicate the error.

Here is a simple example of using the fork() system call to create a new process:

```
#include <unistd.h>

#include <stdio.h>


int main()

{

   pid_t pid;

   pid = fork();

   if (pid == -1)

   {

      perror("fork");

      return 1;

   }

   if (pid == 0)

   {

      printf("This is the child process.\n");

   }

   else

   {

      printf("This is the parent process.\n");

   }

   return 0;

}
```

In this example, the fork() function is used to create a new process. The function returns a value of 0 in the child process and the process ID of the child process in the parent process. The parent process can use the process ID to communicate with the child process or wait for it to finish.

It is important to note that the open() and fork() system calls are low-level functions that are used to implement higher-level functionality in the C library and other parts of the operating system. In most cases, you should use higher-level functions from the C library or other parts of the system to perform these operations, as they are often easier to use and provide a more abstract interface to the underlying system calls.

Manual for Week 5 ,6,7

5.Implement a program using fork () system call to create a hierarchy of 3 process such that P2 is the child of P1 and P1 is the child of P.

```
#include <stdio.h>

#include <unistd.h>

int main() {

   pid_t pid1, pid2;

// Create P1

   pid1 = fork();

   if (pid1 == 0) {

 // This is P1

      printf("P1 (pid=%d) created.\n", getpid());

// Create P2

      pid2 = fork();

      if (pid2 == 0) {

 // This is P2

         printf("P2 (pid=%d) created.\n", getpid());

         printf("P2's parent is P1 (pid=%d).\n", getppid());

      } else {

  // P1 continues execution

 printf("P1 (pid=%d) waiting for P2 (pid=%d) to complete.\n", getpid(), pid2);

  wait(NULL);

   printf("P2 (pid=%d) completed.\n", pid2);

      }

   } else {

      // This is P

      printf("P (pid=%d) created.\n", getpid());

      printf("P1's parent is P (pid=%d).\n", getpid());
```

```
        // Wait for P1 to complete
        wait(NULL);
        printf("P1 (pid=%d) completed.\n", pid1);
    }
    return 0;
}
```

Explanation:

The program first creates P1 using fork(). If fork() returns 0, it means the current process is the child (P1), and P1 creates P2 using another fork(). If fork() returns a positive value, it means the current process is the parent, and the value returned by fork() is the PID of the child process. The parent (P) waits for P1 to complete using wait().

output :

P (pid=1234) created.

P1's parent is P (pid=1234).

P1 (pid=1235) created.

P1 (pid=1235) waiting for P2 (pid=1236) to complete.

P2 (pid=1236) created.

P2's parent is P1 (pid=1235).

P2 (pid=1236) completed.

P1 (pid=1235) completed.

Note: that the PIDs of the processes will likely be different on your system.

6. Implement the following:

i) Program to create an Orphan process.

ii) Create two child process C1 and C2. Make sure that only C2 becomes an Orphan process.

6.i) Program to create an Orphan process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t pid;
```

```
    // Create child process
    pid = fork();
    if (pid == 0) {
        // This is the child process
        printf("Child process (pid=%d) created.\n", getpid());
        sleep(5); // Wait for 5 seconds
        printf("Child process (pid=%d) exiting.\n", getpid());
    } else if (pid > 0) {
        // This is the parent process
        printf("Parent process (pid=%d) created.\n", getpid());
        printf("Parent process (pid=%d) is sleeping.\n", getpid());
        sleep(2); // Wait for 2 seconds
        printf("Parent process (pid=%d) exiting.\n", getpid());
        exit(0);
    } else {
        // Error creating child process
        printf("Error creating child process.\n");
    }
    return 0;
}
```

Explanation:

The program first creates a child process using fork(). If fork() returns 0, it means the current process is the child, and the child process simply waits for 5 seconds before exiting. If fork() returns a positive value, it means the current process is the parent, and the parent process waits for 2 seconds before exiting. During these 2 seconds, the child process becomes an orphan because its parent process exits.

output :

Parent process (pid=1234) created.

Parent process (pid=1234) is sleeping.

Child process (pid=1235) created.

Parent process (pid=1234) exiting.

After 2 seconds, the parent process exits and the child process becomes an orphan. The operating system takes care of the orphaned child process and assigns it a new parent, such as the **init** process. You can verify this by running **ps** command in another terminal window and looking for the child process PID.

6.ii)Create two child process C1 and C2. Make sure that only C2 becomes an Orphan process.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

   pid_t pid1, pid2;

   // Create C1

   pid1 = fork();

   if (pid1 == 0) {

      // This is C1

      printf("C1 (pid=%d) created with parent P (pid=%d).\n", getpid(), getppid());

      sleep(5); // Wait for 5 seconds

      printf("C1 (pid=%d) exiting.\n", getpid());

   } else if (pid1 > 0) {

      // This is P

      printf("P (pid=%d) created.\n", getpid());

      // Create C2

      pid2 = fork();

      if (pid2 == 0) {

         // This is C2

         printf("C2 (pid=%d) created with parent P (pid=%d).\n", getpid(), getppid());

         printf("C2's parent is P (pid=%d).\n", getppid());

         printf("C2 (pid=%d) is sleeping.\n", getpid());

         sleep(2); // Wait for 2 seconds

         printf("C2 (pid=%d) exiting.\n", getpid());

         exit(0);

      } else if (pid2 > 0) {

         // This is P
```

```
            printf("P (pid=%d) is sleeping.\n", getpid());
            sleep(4); // Wait for 4 seconds
            printf("P (pid=%d) exiting.\n", getpid());
            exit(0);
        } else {
            // Error creating C2
            printf("Error creating C2 process.\n");
        }
    } else {
        // Error creating C1
        printf("Error creating C1 process.\n");
    }
    return 0;
}
```

Explnation:

The program first creates C1 using fork(). If fork() returns 0, it means the current process is C1, and C1 simply waits for 5 seconds before exiting. If fork() returns a positive value, it means the current process is P, and P creates C2 using another fork() call.

If fork() returns 0 after the second fork(), it means the current process is C2, and C2 simply waits for 2 seconds before exiting. During these 2 seconds, P sleeps for 4 seconds and then exits.

Output:

P (pid=1234) created.

C1 (pid=1235) created with parent P (pid=1234).

P (pid=1234) is sleeping.

C2 (pid=1236) created with parent P (pid=1234).

C2's parent is P (pid=1234).

C2 (pid=1236) is sleeping.

C1 (pid=1235) exiting.

P (pid=1234) exiting.

After 2 seconds, C2 becomes an orphan because its parent process P exits. The operating system takes care of the orphaned C2 process and assigns it a new parent, such as the init process. You can verify this by running ps command in another terminal window and looking for the C2 process PID.

7. Implement the following:

i) Program to create threads in Linux. Thread prints 0-4 while the main process prints 20-24

ii) Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

7.i) Program to create threads in Linux. Thread prints 0-4 while the main process prints 20-24

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread function
void *thread_func(void *arg) {
    for (int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    int ret;

    // Create thread
    ret = pthread_create(&tid, NULL, thread_func, NULL);

    if (ret) {
        printf("Error creating thread.\n");
        exit(EXIT_FAILURE);
    }
```

```
    // Main process prints 20-24
    for (int i = 20; i < 25; i++) {
        printf("%d\n", i);
    }

    pthread_exit(NULL);
}
```

The program first creates a thread using pthread_create(). The thread function simply loops from 0 to 4 and prints each number.

Meanwhile, the main process also loops from 20 to 24 and prints each number.

20

21

22

23

24

0

1

2

3

4

The order of the output may vary slightly, since the thread and main process are executing concurrently.

7.ii) Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread function
```

```
void *thread_func(void *arg) {
    int n = *(int *)arg;

    for (int i = 0; i <= n; i++) {
        printf("%d\n", i);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    int ret, n = 5;

    // Create thread
    ret = pthread_create(&tid, NULL, thread_func, &n);

    if (ret) {
        printf("Error creating thread.\n");
        exit(EXIT_FAILURE);
    }

    // Wait for thread to finish
    pthread_join(tid, NULL);

    // Main process prints 20-24
    for (int i = 20; i < 25; i++) {
        printf("%d\n", i);
    }
```

```
    pthread_exit(NULL);
}
```

The program first creates a thread using pthread_create(). The main process passes the value of n (which is 5 in this example) to the thread function as an argument.

The thread function loops from 0 to n and prints each number.

After creating the thread, the main process waits for the thread to finish using pthread_join().

Once the thread has finished, the main process then loops from 20 to 24 and prints each number.

```
0
1
2
3
4
20
21
22
23
24
```

The order of the output may vary slightly, since the thread and main process are executing concurrently.

Week 8

8. Implement non-pre-emptive/pre-emptive CPU scheduling algorithms to find turnaround time

and waiting time (minimum 2 from all process scheduling algorithms)

8.1. Non-pre-emptive (FCFS):

```
#include <stdio.h>

int main() {

int n, i, j, at[20], bt[20], wt[20], tat[20], ct[20];

float avwt = 0, avtat = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

// Read arrival time and burst time for each process

printf("\nEnter arrival time and burst time for each process:\n");

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

printf("Process %d: ", i+1);

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

}

// Calculate completion time for each process

ct[0] = at[0] + bt[0];

for (i = 1; i < n; i++) {

if (ct[i-1] < at[i]) {

ct[i] = at[i] + bt[i];

} else {

ct[i] = ct[i-1] + bt[i];

}


 }

// Calculate waiting time and turnaround time for each process

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

```
wt[i] = ct[i] - bt[i] - at[i];

tat[i] = ct[i] - at[i];

avwt += wt[i];

avtat += tat[i];

}

// Calculate average waiting time and turnaround time

avwt /= n;

avtat /= n;

// Print table of process details

printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround

Time\n");

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

printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[i], bt[i], ct[i], wt[i], tat[i]);

}

// Print average waiting time and turnaround time

printf("\nAverage Waiting Time: %.2f\n", avwt);

printf("Average Turnaround Time: %.2f\n", avtat);

return 0;

}
```

This program takes the number of processes, arrival time, and burst time as input from the user, and calculates the waiting time and turnaround time for each process based on the arrival time. The program first calculates the completion time for each process based on its arrival time and burst time, then calculates the waiting time and turnaround time for each process based on its completion time and arrival time. Finally, the program prints a table of process details, including arrival time, burst time, completion time, waiting time, and turnaround time, as well as the average waiting time and turnaround time.

Here's an example of the output for a sample input:

Enter the number of processes: 4

Enter arrival time and burst time for each process:

Process 1: 0 5

Process 2: 2 4

Process 3: 4 2

Process 4: 5 1

Process Arrival Time Burst Time Completion Time Waiting Time Turnaround Time

P1 0 5 5 0 5

P2 2 4 9 3 7

P3 4 2 11 5 7

P4 5 1 12 6 7

Average Waiting Time: 3.50

Average Turnaround Time: 6.50

This output shows the table of process details, including arrival time, burst time, completion time, waiting time, and turnaround time for each process, as well as the average waiting time and turnaround time for all the processes. The completion time for each process is calculated based on its arrival time and burst time. The waiting time for each process is calculated as the difference between its completion time, burst time, and arrival time. The turnaround time for each process is calculated as the difference between its completion time and arrival time.

8.2. Pre-emptive (ROUND ROBIN)

```
#include <stdio.h>

int main() {

int n, i, j, quantum, wt[20], tat[20], at[20], bt[20], rt[20], ct[20], t;

float avwt = 0, avtat = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the quantum: ");

scanf("%d", &quantum);

printf("Enter arrival time and burst time for each process:\n");
```

```
for (i = 0; i < n; i++) {
printf("Process %d: ", i+1);
scanf("%d %d", &at[i], &bt[i]);
rt[i] = bt[i];
}
t = 0;
while (1) {
int done = 1;
for (i = 0; i < n; i++) {
if (rt[i] > 0) {
done = 0;
if (rt[i] > quantum) {
t += quantum;
rt[i] -= quantum;
} else {
t += rt[i];
ct[i] = t;

 tat[i] = ct[i] - at[i];
wt[i] = tat[i] - bt[i];
rt[i] = 0;
}
}
}
if (done == 1) {
break;
}
}
printf("\nProcess Arrival Time Burst Time Completion Time Waiting Time Turnaround
```

```
Time\n");
for (i = 0; i < n; i++) {
printf("P%-8d%-16d%-16d%-16d%-16d%-16d\n", i+1, at[i], bt[i], ct[i], wt[i], tat[i]);
avwt += wt[i];
avtat += tat[i];
}
avwt /= n;
avtat /= n;
printf("\nAverage Waiting Time: %.2f", avwt);
printf("\nAverage Turnaround Time: %.2f", avtat);
return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the quantum: 2
Enter arrival time and burst time for each process:

Process 1: 0 7
Process 2: 2 4
Process 3: 4 1
Process 4: 5 4
Process Arrival Time Burst Time Completion Time Waiting Time Turnaround Time
P1 0 7 15 6 15
P2 2 4 11 5 9
P3 4 1 5 0 1
P4 5 4 16 7 11
Average Waiting Time: 4.50
Average Turnaround Time: 9.00
```

9.1 Program to simulate Race Condition

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
 {
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
void *fun1()
{
   int x;
   x=shared;//thread one reads value of shared variable
   printf("Thread1 reads the value of shared variable as %d\n",x);
   x++;  //thread one increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1);  //thread one is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
}
```

```
void *fun2()
{
   int y;
   y=shared;//thread two reads value of shared variable
   printf("Thread2 reads the value as %d\n",y);
   y--;  //thread two increments its value
   printf("Local updation by Thread2: %d\n",y);
   sleep(1); //thread two is preempted by thread 1
   shared=y; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread2 is: %d\n",shared);
}
```

9.2 Program for Process Synchronization using mutex locks

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
 int shared=1; //shared variable
 pthread_mutex_t l; //mutex lock
 int main()
 {
 pthread_mutex_init(&l, NULL); //initializing mutex locks
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
```

```
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
void *fun1()
 {
    int x;
    printf("Thread1 trying to acquire lock\n");
    pthread_mutex_lock(&l); //thread one acquires the lock. Now thread 2 will not be able to acquire the lock //until it is unlocked by thread 1
   printf("Thread1 acquired lock\n");
    x=shared;//thread one reads value of shared variable
    printf("Thread1 reads the value of shared variable as %d\n",x);
    x++;  //thread one increments its value
    printf("Local updation by Thread1: %d\n",x);
    sleep(1);  //thread one is preempted by thread 2
    shared=x; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread1 is: %d\n",shared);
    pthread_mutex_unlock(&l);
    printf("Thread1 released the lock\n");
 }
void *fun2()
 {
    int y;
    printf("Thread2 trying to acquire lock\n");
    pthread_mutex_lock(&l);
    printf("Thread2 acquired lock\n");
    y=shared;//thread two reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
```

```
    y--;  //thread two increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread two is preempted by thread 1
    shared=y; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
    pthread_mutex_unlock(&l);
    printf("Thread2 released the lock\n");
}
```

9.3 Program for process synchronization using semaphores

```
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
 void *fun2();
 int shared=1; //shared variable
 sem_t s; //semaphore variable
 int main()
 {
 sem_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of processes sharing semaphore, 3rd argument is the initial value of semaphore variable
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
```

```
void *fun1()
{
   int x;
   sem_wait(&s); //executes wait operation on s
   x=shared;//thread1 reads value of shared variable
   printf("Thread1 reads the value as %d\n",x);
   x++;  //thread1 increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1); //thread1 is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
   sem_post(&s);
}
void *fun2()
{
   int y;
   sem_wait(&s);
   y=shared;//thread2 reads value of shared variable
   printf("Thread2 reads the value as %d\n",y);
   y--;  //thread2 increments its value
   printf("Local updation by Thread2: %d\n",y);
   sleep(1); //thread2 is preempted by thread 1
   shared=y; //thread2 updates the value of shared variable
   printf("Value of shared variable updated by Thread2 is: %d\n",shared);
   sem_post(&s);
}
```

9.4 Solution to Dining Philosopher Problem

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
sem_t chopstick[5];
void * philos(void *);
void eat(int);
int main()
{
	int i,n[5];
	pthread_t T[5];
	for(i=0;i<5;i++)
	sem_init(&chopstick[i],0,1);
	for(i=0;i<5;i++){
		n[i]=i;
		pthread_create(&T[i],NULL,philos,(void *)&n[i]);
		}
	for(i=0;i<5;i++)
		pthread_join(T[i],NULL);
}
void * philos(void * n)
{
```

```
        int ph=*(int *)n;
        printf("Philosopher %d wants to eat\n",ph);
        printf("Philosopher %d tries to pick left chopstick\n",ph);
        sem_wait(&chopstick[ph]);
        printf("Philosopher %d picks the left chopstick\n",ph);
        printf("Philosopher %d tries to pick the right chopstick\n",ph);
        sem_wait(&chopstick[(ph+1)%5]);
        printf("Philosopher %d picks the right chopstick\n",ph);
        eat(ph);
        sleep(2);
        printf("Philosopher %d has finished eating\n",ph);
        sem_post(&chopstick[(ph+1)%5]);
        printf("Philosopher %d leaves the right chopstick\n",ph);
        sem_post(&chopstick[ph]);
        printf("Philosopher %d leaves the left chopstick\n",ph);
}
void eat(int ph)
{
        printf("Philosopher %d begins to eat\n",ph);
}
```

FCFS Disk Scheduling Algorithm

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int i,n,req[50],mov=0,cp;
   printf("enter the current position\n");
   scanf("%d",&cp);
   printf("enter the number of requests\n");
   scanf("%d",&n);
   printf("enter the request order\n");
   for(i=0;i<n;i++)
   {
      scanf("%d",&req[i]);
   }
   mov=mov+abs(cp-req[0]); // abs is used to calculate the absolute value
   printf("%d -> %d",cp,req[0]);
   for(i=1;i<n;i++)
   {
      mov=mov+abs(req[i]-req[i-1]);
      printf(" -> %d",req[i]);
   }
   printf("\n");
```

```
    printf("total head movement = %d\n",mov);
}
```

SSTF Algorithm Program in C

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i,n,k,req[50],mov=0,cp,index[50],min,a[50],j=0,mini,cp1;
    printf("enter the current position\n");
    scanf("%d",&cp);
    printf("enter the number of requests\n");
    scanf("%d",&n);
    cp1=cp;
    printf("enter the request order\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&req[i]);
    }
    for(k=0;k<n;k++)
    {
    for(i=0;i<n;i++)
    {
        index[i]=abs(cp-req[i]); // calculate distance of each request from current position
    }
    // to find the nearest request
```

```
    min=index[0];
    mini=0;
    for(i=1;i<n;i++)
    {
       if(min>index[i])
       {
          min=index[i];
          mini=i;
       }
    }
    a[j]=req[mini];
    j++;
    cp=req[mini]; // change the current position value to next request
    req[mini]=999;
    } // the request that is processed its value is changed so that it is not processed again
    printf("Sequence is : ");
    printf("%d",cp1);
    mov=mov+abs(cp1-a[0]);    // head movement
    printf(" -> %d",a[0]);
    for(i=1;i<n;i++)
    {
       mov=mov+abs(a[i]-a[i-1]); ///head movement
       printf(" -> %d",a[i]);
    }
    printf("\n");
    printf("total head movement = %d\n",mov);
}
```