1. Unix Process Control.

Linux tools and the proc file system

Goal: The goal of this lab is to get familiar with Linux tools and files used for system and process behaviour information and monitoring.

Tools: Following are some basic Linux tools. The first step of this lab is to get familiar with the usage and capabilities of these tools.

To know more about them use: man <command> . Start with man man !

● **top**

The top program provides a dynamic real-time view of a running system. It can display system summary information, as well as a list of processes or threads currently being managed by the kernel. The types of system summary information shown and the types, order and size of information displayed for tasks are all user-configurable.

● **ps**

The ps command is used to view the processes running on a system. It provides a snapshot of the current processes along with detailed information like user id, cpu usage, memory usage, command name etc. It does not display data in real time like top or htop commands, but even though being simpler in features and output it is still an essential process management/monitoring tool that every linux newbie should know about and learn well.

● **iostat**

iostat is a command used for monitoring input/output device usage by observing the time the devices are active in relation to their average transfer rates. iostat creates reports that can be used to change system configuration for better balance the input/output between physical disks.

● **strace**

strace is a diagnostic, debugging and instructional userspace utility for Linux. It is used to monitor interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state.

● **lsof**

lsof is a tool used to list open files based on user, process, commands, network services, etc.

● **lsblk**

lsblk is a tool used to list information about all available block devices such as hard disk, flash drives, CD-ROM…

● Also look up the following commands: pstree , lshw, lspci, lscpu, dig,

netstat, df, du. (note that some of these may need root privileges)

The /proc file system

The / proc file system is a mechanism provided by Linux, so that kernel can send information to processes and also report information about the system and processes to users. This is a file interface provided to the user, to interact with the kernel and get the required information about processes running on the system. The /proc file system is nicely documented in the proc man page. You can access this document by running the command man proc on a Linux system. Understand systemwide proc files such as meminfo , cpuinfo , etc and process related files such as status , stat , limits , maps etc.

Exercises

1. Collect the following basic information about your machine using the /proc file system and answer the following questions:

a. How many CPU and cores does the machine have?

b. What is the frequency of each CPU?

c. How much memory does your system have?

d. How much of it is free and available? What is the difference between them?

e. What is total number of user-level processes in the system?

f. How many context switches has the system performed since bootup?

2. Run all programs in the subdirectory memory and identify memory usage of each program. What is the meaning of the parameters VmSize and VmRSS ? Compare the memory usage of these programs in terms of VmSize & VmRSS and justify your results based on code.

3. Run the executable subprocesses provided in the subdirectory subprocess and provide your roll number as input argument. Find the number of sub processes created by this program. Describe how you obtained the answer.

4. Run strace along with the binary program of empty.c given in subdirectory strace . What do you think the output of strace indicates in this case? How many different system call functions do you see?

Next, use strace along with another binary program of hello.c (which is in the same directory).Compare the two strace outputs

a. Which part of the strace output is common, and which part has to do with the specific program?

b. List all unique system calls along with input and output parameters and overall functionality of each system call?

5. Run the executable openfiles in subdirectory files and list the files which are opened by this program. Describe how you obtained the answer.

6. Find all the block devices on your system, their mount points and file systems present on them. Describe how you obtained the answer.

<u>Home Task</u>

7. Create 5000 files starting from foo1.pdf to foo5000.pdf using script.sh given in the subdirectory disk.

Run the programs disk1 and disk2 and identify the average disk utilization. Justify your answer with the help of code.

OBSERVATION SPACE

2. Shell Programming

If you are using any major operating system you are indirectly interacting to shell. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal. In this article we will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies –

Kernel

Shell

Terminal

What is Kernel?

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

File management

Process management

I/O management

Memory management

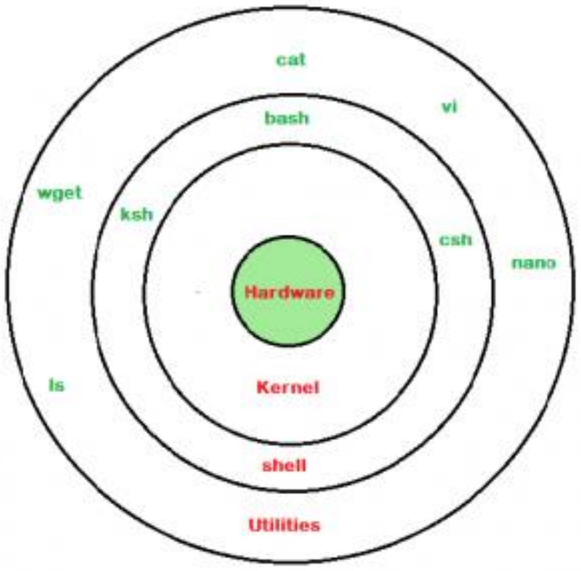
Device management etc.

It is often mistaken that Linus Torvalds has developed Linux OS, but actually he is only responsible for development of Linux kernel.

Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

What is Shell?

A shell is special user program which provide an interface to user to use operating system services. Shell accepts human readable commands from user and converts them into something which kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.



linux shell

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute.

Graphical Shells

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.A typical GUI in Ubuntu system –

There are several shells are available for Linux systems like –

- BASH (Bourne Again SHell) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- CSH (C SHell) – The C shell’s syntax and usage are very similar to the C programming language.
- KSH (Korn SHell) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built in functions.

Shell Scripting

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

A shell script has syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.

Why do we need shell scripts

There are many reasons to write shell scripts –

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful

- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc

1) Write a program to find the factorial of a number using shell programming.

OBSERVATION SPACE

3. Implementation of System Calls

The fork() System Call

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following

the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

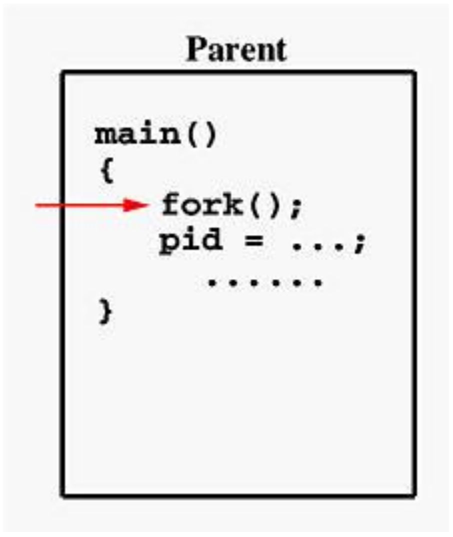
Let us take an example to make the above points clear. This example does not distinguish parent

and the child processes.

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100
```

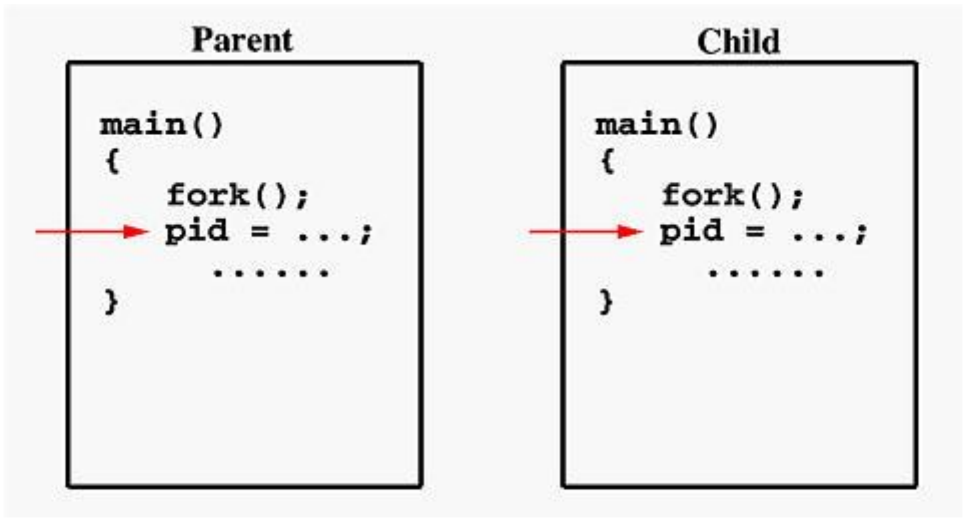
```
void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces. Since every process has its own address space, any

modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

If you run this program, you might see the following on the screen:

....

```
This line is from pid 3456, value 13
This line is from pid 3456, value 14
    ................
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
    ................
This line is from pid 3456, value 21
This line is from pid 3456, value 22
    ................
```

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

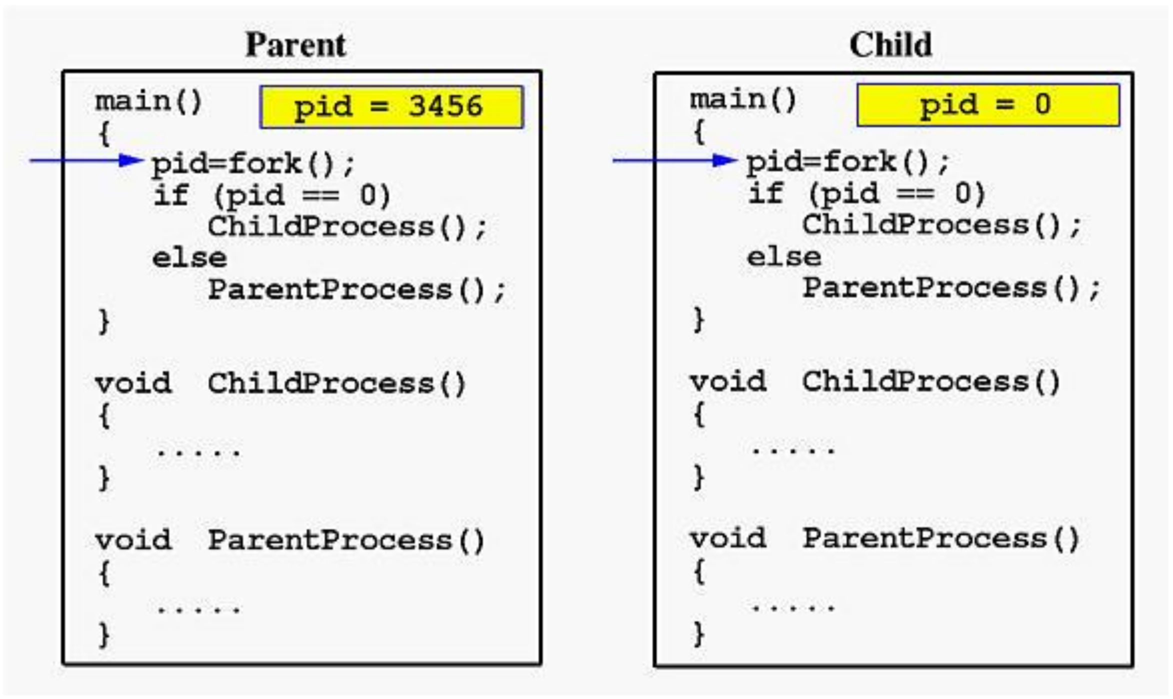
Consider one more simple example, which distinguishes the parent from the child.

```
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}
```

```
void  ParentProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

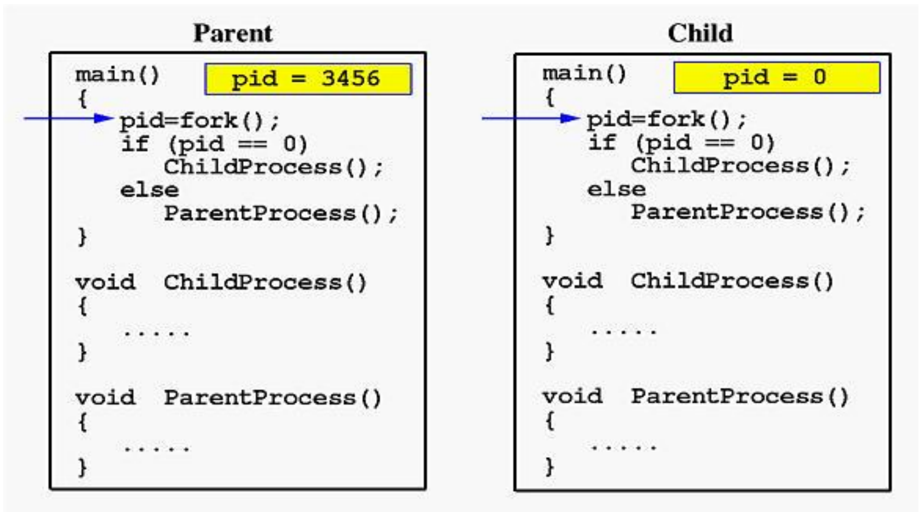
In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable **i**. For simplicity, **printf()** is used.

When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



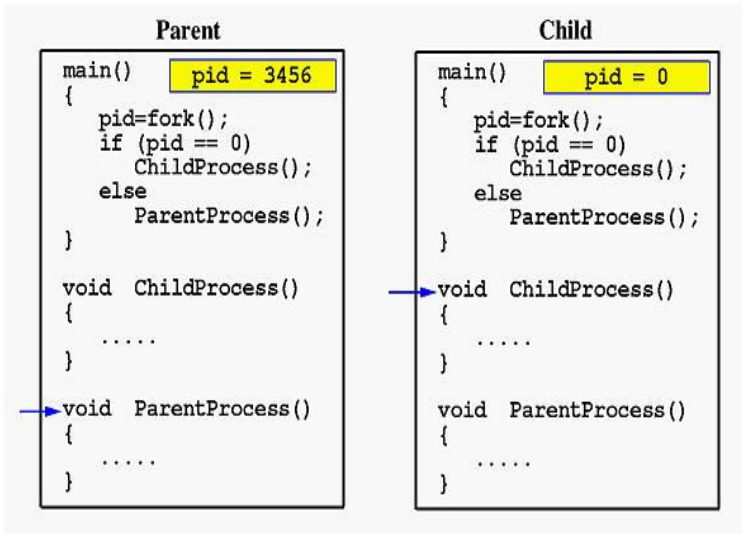
```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

2. exec() system call:

The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways. Within the exec family there are functions that vary slightly in their capabilities.

exec family:

execl() and execlp():

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL. e.g.

execl("/bin/ls", "ls", "-l", NULL);

execlp(): It does same job except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The function

execlp() can also take the fully qualified name as it also resolves explicitly.

e.g.

execlp("ls", "ls", "-l", NULL);

2. execv() and execvp():

execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string. e.g.

char *argv[] = ("ls", "-l", NULL);

execv("/bin/ls", argv);

execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.

e.g.

execvp("ls", argv);

3. execve( ):

int execve(const char *filename, char *const argv[ ], char *const envp[ ]);

It executes the program pointed to by filename. Filename must be either a binary executable, or a script starting with a line of the form:

argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

int main(int argc, char *argv[ ] , char *envp[ ])]

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Zombie Process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX nohup command is one means to accomplish this.

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

Example:

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
```

```
int global; /* In BSS segement, will automatically be assigned '0'*/

int main()
{
   pid_t child_pid;
   int status;
   int local = 0;
   /* now create new process */
   child_pid = fork();

   if (child_pid >= 0) /* fork succeeded */
   {
      if (child_pid == 0) /* fork() returns 0 for the child process */
      {
         printf("child process!\n");

         // Increment the local and global variables
         local++;
         global++;

         printf("child PID =  %d, parent pid = %d\n", getpid(), getppid());
         printf("\n child's local = %d, child's global = %d\n",local,global);

         char *cmd[] = {"whoami",(char*)0};
         return execv("/usr/bin/",cmd); // call whoami command

       }
       else /* parent process */
       {
          printf("parent process!\n");
          printf("parent PID =  %d, child pid = %d\n", getpid(), child_pid);
          wait(&status); /* wait for child to exit, and store child's exit status */
          printf("Child exit code: %d\n", WEXITSTATUS(status));

          //The change in local and global variable in child process should not reflect here in
parent process.
          printf("\n Parent'z local = %d, parent's  global = %d\n",local,global);
```

```
            printf("Parent says bye!\n");
            exit(0);  /* parent exits */
        }
    }
    else /* failure */
    {
        perror("fork");
        exit(0);
    }
}
```

Now, when the above program is executed, it produces the following output :

```
$ ./fork
parent process!
parent PID =  3184, child pid = 3185
child process!
child PID =  3185, parent pid = 3184

child's local = 1, child's global = 1
himanshu
Child exit code: 0

Parent'z local = 0, parent's  global = 0
Parent says bye!
```

1. **Write a Unix C program using the fork() system call that generates the factorial and gives a sequence of series like 1, 2, 6, 24, 120… in the child process. The number of the sequence is provided in the command line.**
2. **Implement the C program in which main program accepts an integer array. Main program uses the fork system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve system call. The child process uses execve system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.**

OBSERVATION SPACE

4.Interprocess Communication using PIPE.

We use the term pipe to mean connecting a data flow from one process to another. Generally you attach, or pipe, the output of one process to the input of another. .Most Linux users will already be familiar with the idea of a pipeline, linking shell commands together so that the output of one process is fed straight to the input of another. For shell commands; this is done using the pipe character to join the commands, such as

cmd1| cmd2

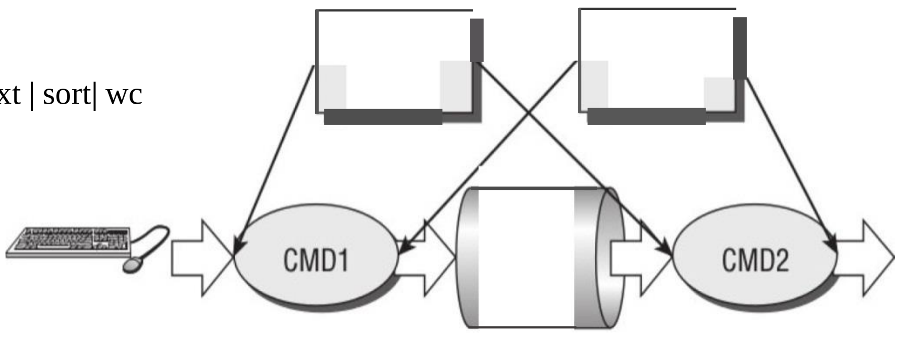
The output of first command is given as input to the second command.

Examples:

– ls|wc

– who|sort

– cat file.txt | sort| wc



The pipe call

The lower-level pipe function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives you more control over the reading and writing of data.

•The pipe function has the following prototype:

```
#include <unistd.h>

int pipe(int file_descriptor[2]);
```

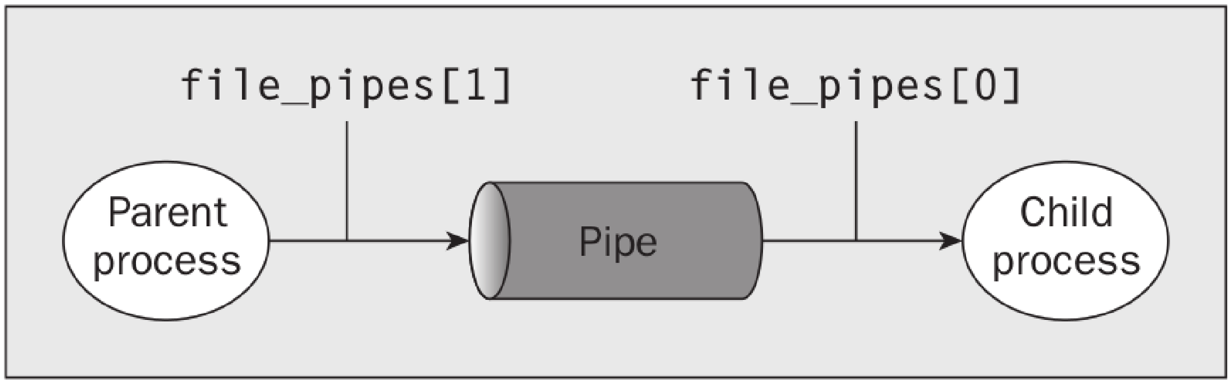
•pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure.

File Descriptor

The two file descriptors returned are connected in a special way.

•Any data written to file_descriptor[1] can be read back from file_descriptor[0] . The data is processed in a first in, first out basis, usually abbreviated to FIFO.

•This means that if you write the bytes 1 , 2 , 3 to file_descriptor[1] , reading from file_descriptor[0] will produce 1 , 2 , 3 .This is different from a stack, which operates on a last in, first out basis, usually abbreviated to LIFO.



The Write system Call

```
#include<unistd.h>

size_t write(int fildes,const void*buf, size_t nbytes);
```

•It arranges for the first nbytes bytes from buf to bewritten to the file associated with the file descriptor fildes.

•It returns the number of bytes actually written. This may be less than nbytes if there has been an error in the file descriptor. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call.

The Read System Call

#include<unistd.h>

size_t read(int fildes, void*buf,size_t nbytes);

•It reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf.

•It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

Example:

```
#include<stdio.h>
#include<string.h>
int main()
{
  int file_pipes[2], data_pro;
 const char data[] = "Hello SMIT";
 char buffer[20];
 if (pipe(file_pipes) == 0)
{
 data_pro = write(file_pipes[1], data, strlen(data));
 printf("Wrote %d bytes\n", data_pro); data_pro = read(file_pipes[0], buffer, 20);
 printf("Read %d bytes: %s\n", data_pro, buffer);
}
return 0;
}
```

Output:

```
Wrote 10 bytes

Read 10 bytes: Hello SMIT
```

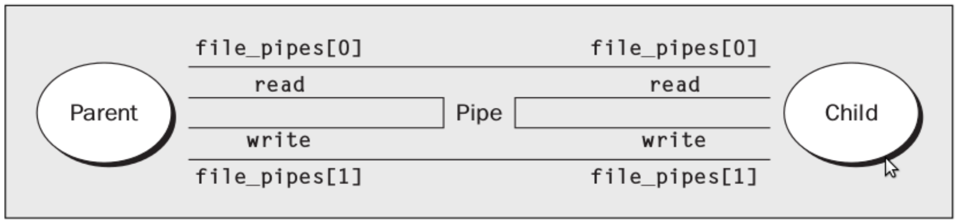
Example 2:(Add Fork to pipe)

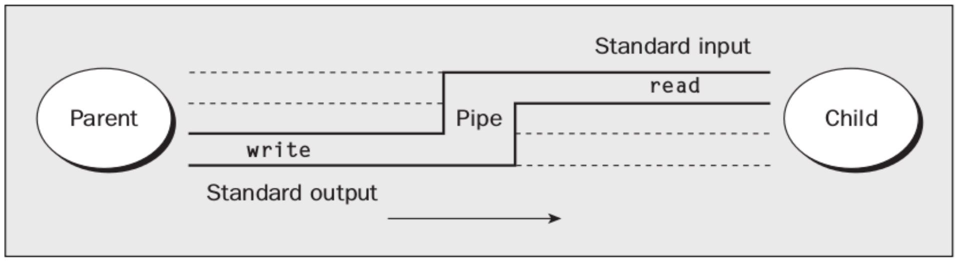
```
#include<stdio.h>
#include<string.h>
int main()
{
int file_pipes[2], data_pro, pid;
const char data[] = "Hello SMIT", buffer[20];
pipe(file_pipes);
pid = fork();
if (pid == 0)
{
data_pro = write(file_pipes[1], data, strlen(data));
printf("Wrote %d bytes\n", data_pro);
}
else
{
data_pro = read(file_pipes[0], buffer, 20);
printf("Read %d bytes: %s\n", data_pro, buffer);
}
return 0;

}
```

Output:

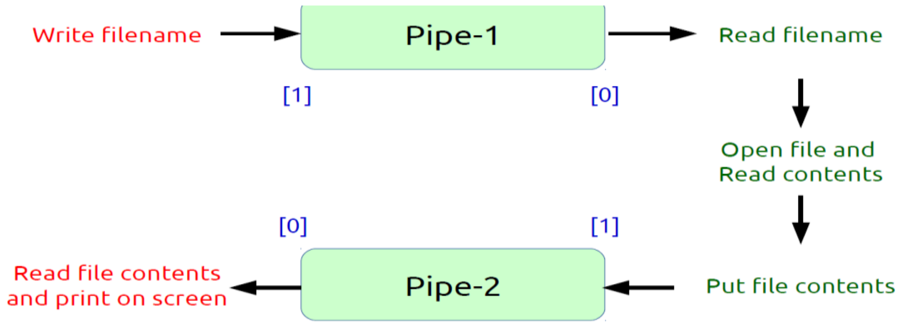
```
Read 10 bytes: Hello SMIT
Write 10 bytes.
```

Use of Pipes:





1. **Implement using Pipe: Full duplex communication between parent and child processes. Parent process writes a pathname of a file (the contents of the file are desired) on one pipe to be read by child process and child process writes the contents of the file on second pipe to be read by parent process and displays on standard output.**
   How to do it?



OBSERVATION SPACE

5. Interprocess Communication using SIGNALS.

Signals

The common communication channel between user space program and kernel is given by the system calls. But there is a different channel, that of the signals, used both between user processes and from kernel to user process.

Sending signals

A program can signal a different program using the kill() system call with prototype

int kill(pid_t pid, int sig);

This will send the signal with number sig to the process with process ID pid. Signal numbers are small positive integers.A user can send a signal from the command line using the kill command. Common uses are kill -9 N to kill the process with pid N, or kill -1 N to force process N (maybe init or inetd) to reread its configuration file.

Certain user actions will make the kernel send a signal to a process or group of processes: typing the interrupt character (probably Ctrl-C) causes SIGINT to be sent, typing the quit character (probably Ctrl-\) sends SIGQUIT, hanging up the phone (modem) sends SIGHUP, typing the stop character (probably Ctrl-Z) sends SIGSTOP.

Certain program actions will make the kernel send a signal to that process: for an illegal instruction one gets SIGILL, for accessing non-existing memory one gets SIGSEGV, for writing to a pipe while nobody is listening anymore on the other side one gets SIGPIPE, for reading from the terminal while in the background one gets SIGTTIN, etc.

More interesting communication from the kernel is also possible. One can ask the kernel to be notified when something happens on a given file descriptor.

A whole group of signals is reserved for real-time use.

Receiving signals

When a process receives a signal, a default action happens, unless the process has arranged to handle the signal for the list of signals and the corresponding default actions. For example, by default SIGHUP, SIGINT, SIGKILL will kill the process; SIGQUIT will kill the process and force a core dump; SIGSTOP, SIGTTIN will stop the process; SIGCONT will continue a stopped process; SIGCHLD will be ignored.

Traditionally, one sets up a handler for the signal using the signal system call with prototype

typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t handler);

This sets up the routine handler() as handler for signals with number sig. The return value is (the address of) the old handler. The special values SIG_DFL and SIG_IGN denote the default action and ignoring, respectively.

When a signal arrives, the process is interrupted, the current registers are saved, and the signal handler is invoked. When the signal handler returns, the interrupted activity is continued.

It is difficult to do interesting things in a signal handler, because the process can be interrupted in an arbitrary place, data structures can be in arbitrary state, etc. The three most common things to do in a signal handler are (i) set a flag variable and return immediately, and (ii) (messy) throw away all the program was doing, and restart at some convenient point, perhaps the main command loop or so, and (iii) clean up and exit.

Setting up a handler for a signal is called “catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or blocked or ignored.

Semantics

The traditional semantics was: reset signal behavior to SIG_DFL upon invocation of the signal handler. Possibly this was done to avoid recursive invocations. The signal handler would do its job and at the end call signal() to establish itself again as handler.

This is really unfortunate. When two signals arrive shortly after each other, the second one will be lost if it arrives before the signal handler is called - there is no counter. And if it arrives after the signal handler is called, the default action will happen - this may very well kill the process. Even if the handler calls signal() again as the very first thing it does, that may be too late.

Various Unix flavors played a bit with the semantics to improve on this situation. Some block signals as long as the process has not returned from the handler. The BSD solution was to invent a new system call, sigaction() where one can precisely specify the desired behavior. Today signal() must be regarded as deprecated - not to be used in serious applications.

Blocking signals

Each process has a list (bitmask) of currently blocked signals. When a signal is blocked, it is not delivered (that is, no signal handling routine is called), but remains pending.

The sigprocmask() system call serves to change the list of blocked signals.

The sigpending() system call reveals what signals are (blocked and) pending.

The sigsuspend() system call suspends the calling process until a specified signal is received. When a signal is blocked, it remains pending, even when otherwise the process would ignore it.

wait and SIGCHLD

When a process forks off a child to perform some task, it is probably interested in how things went. Upon exit, the child leaves an exit status that should be returned to the parent. So, when the child finishes it becomes a zombie - a process that is dead already but does not disappear yet because it has not yet reported its exit status.

Whenever something interesting happens to the child (it exits, crashes, traps, stops, continues), and in particular when it dies, the parent is sent a SIGCHLD signal.

The parent can use the system call wait() or waitpid() or so, there are a few variations, to learn about the status of its stopped or deceased children. In the case of a deceased child, as soon as a status has been reported, the zombie vanishes.

If the parent is not interested it can say so explicitly (before the fork) using signal(SIGCHLD, SIG_IGN);

or

struct sigaction act; act.sa_handler = something; act.sa_flags = SA_NOCLDWAIT; sigaction (SIGCHLD, &act, NULL);

and as a result it will not hear about deceased children, and children will not be transformed into zombies. Note that the default action for SIGCHLD is to ignore this signal; nevertheless signal(SIGCHLD, SIG_IGN) has effect, namely that of preventing the transformation of children into zombies. In this situation, if the parent does a wait(), this call will return only when all children have exited, and then returns -1 with errno set to ECHILD.

It depends on the UNIX flavor whether SIGCHLD is sent when SA_NOCLDWAIT was set. After act.sa_flags = SA_NOCLDSTOP no SIGCHLD is sent when children stop or stopped children continue.If the parent exits before the child, then the child is reparented to init, process 1, and this process will reap its status.

Returning from a signal handler

When the program was interrupted by a signal, its status (including all integers and floating point registers) was saved, to be restored just before execution continues at the point of interruption.

This means that the return from the signal handler is more complicated than an arbitrary procedure return - the saved state must be restored.

To this end, the kernel arranges that the return from the signal handler causes a jump to a short code sequence (sometimes called trampoline) that executes a sigreturn() system call. This system call takes care of everything.

Signal Concepts

Signals are defined in <signal.h>

• man 7 signal for complete list of signals and their numeric values.

• kill –l for full list of signals on a system.

• 64 signals. The first 32 are traditional signals, the rest are for real time applications

Signal Function

Programs can handle signals using the signal library function.

void (*signal(int signo, void (*func)(int)))(int);

• signo is the signal number to handle

• func defines how to handle the signal

– SIG_IGN

– SIG_DFL

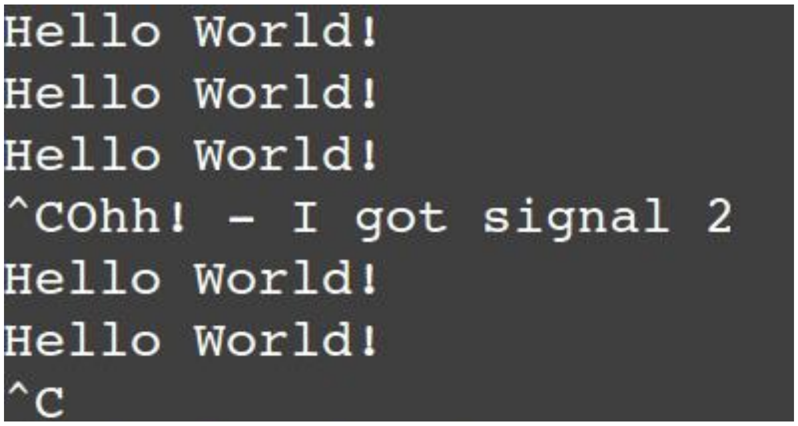
– Function pointer of a custom handler.

• Returns previous disposition if ok, or SIG_ERR on error.

Example:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ohh(int sig)
{
printf("Ohh! I
got signal %d\n", sig);
(void) signal(SIGINT, SIG_DFL);
}
int main()
{
(void) signal(SIGINT, ohh);
while(1)
{
printf("Hello World!\n");
sleep(1);
}
return 0;
}
```

Output:



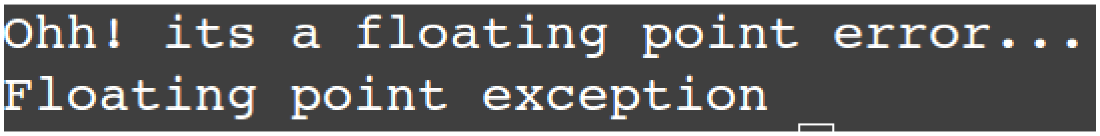
```
Hello World!
Hello World!
Hello World!
^COhh! - I got signal 2
Hello World!
Hello World!
^C
```

Example 2:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void error(int sig)
{
printf("Ohh! its a floating point error...\n");
(void) signal(SIGFPE, SIG_DFL);
}
int main()
{
(void) signal(SIGFPE, error);
int a = 12, b = 0, result;
result = a / b;
printf("Result is : %d\n",result);
```

return 0;
}

Output:



```
Ohh! its a floating point error...
Floating point exception _
```

Sigaction

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);

• The sigaction structure, used to define the actions to be taken on receipt of the signal specified by sig, is defined in signal.h and has at least the following members:

void (*) (int) sa_handler function, SIG_DFL or SIG_IGN sigset_t sa_mask signals to block in sa_handler int sa_flags signal action modifiers

• The sigaction function sets the action associated with the signal sig.

If oact is not null, sigaction writes the previous signal action to the location it refers to. If act is null, this is all sigaction does. If act isn't null, the action for the specified signal is set.

As with signal , sigaction returns 0 if successful and -1 if not. The error variable errno will be set to EINVAL if the specified signal is invalid or if an attempt is made to catch or ignore a signal that can't be caught or ignored.

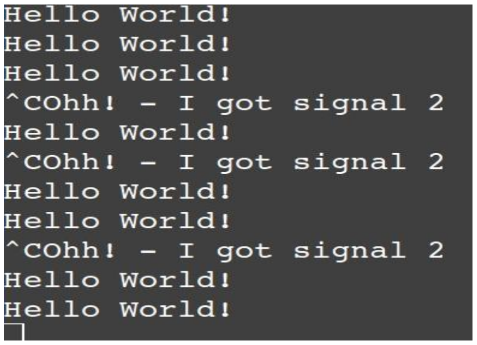
• Within the sigaction structure pointed to by the argument act, sa_handler is a pointer to a function called when signal sig is received. This is much like the function func you saw earlier passed to signal .

• You can use the special values SIG_IGN and SIG_DFL in the sa_handler field to indicate that the signal is to be ignored or the action is to be restored to its default, respectively.

Example:

```
void ohh(int sig)
{
printf("Ohh! I
got signal %d\n", sig);
}
int main()
{
struct sigaction act;
act.sa_handler = ohh;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, 0);
while(1)
{
printf("Hello World!\n");
sleep(1);
}
}
```

Output:



```
Hello World!
Hello World!
Hello World!
^COhh! - I got signal 2
Hello World!
^COhh! - I got signal 2
Hello World!
Hello World!
^COhh! - I got signal 2
Hello World!
Hello World!
```

1. **Implement the C program to demonstrate the use of SIGCHLD signal. A parent process Creates multiple child process (minimum three child processes). Parent process should be Sleeping until it creates the number of child processes. Child processes send SIGCHLD signal to parent process to interrupt from the sleep and force the parent to call wait for the Collection of status of terminated child processes.**

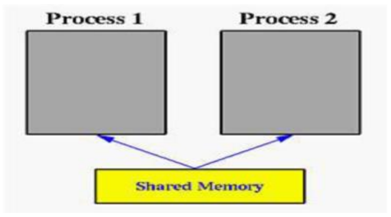
OBSERVATION SPACE

6. Interprocess Communication using shared memory.

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. In this lab the following issues related to share memory utilization are discussed:

- Creating a Shared Memory Segment
- Controlling a Shared Memory Segment
- Attaching and Detaching a Shared Memory Segment

In the discussion of the fork ( ) system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.



```
/*
                    SHARING MEMORY BETWEEN PROCESSES

In this example, we show how two processes can share a common portion of the memory. Recall that when a
process forks, the new child process has an identical copy of the variables of the parent process. After fork the
parent and child can update their own copies of the variables in their own way, since they dont actually share
the variable. Here we show how they can share memory, so that when one updates it, the other can see the
change.

*/

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h> /*  This file is necessary for using shared
                            memory constructs
                        */

main()
{
        int shmid. status;
        int *a, *b;
        int i;

        /*
          The operating system keeps track of the set of shared memory
          segments. In order to acquire shared memory, we must first
          request the shared memory from the OS using the shmget()
          system call. The second parameter specifies the number of
          bytes of memory requested. shmget() returns a shared memory
          identifier (SHMID) which is an integer. Refer to the online
          man pages for details on the other two parameters of shmget()
        */
        shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);
                        /* We request an array of two integers */

        /*
          After forking, the parent and child must "attach" the shared
          memory to its local data segment. This is done by the shmat()
          system call. shmat() takes the SHMID of the shared memory
          segment as input parameter and returns the address at which
          the segment has been attached. Thus shmat() returns a char
          pointer.
```

```
*/
if (fork() == 0) {

        /* Child Process */

        /*  shmat() returns a char pointer which is typecast here
            to int and the address is stored in the int pointer b. */
        b = (int *) shmat(shmid, 0, 0);

        for( i=0; i< 10; i++) {
                sleep(1);
                printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);
        }
        /* each process should "detach" itself from the
           shared memory after it is used */

        shmdt(b);

}
else {

        /* Parent Process */

        /*  shmat() returns a char pointer which is typecast here
            to int and the address is stored in the int pointer a.
            Thus the memory locations a[0] and a[1] of the parent
            are the same as the memory locations b[0] and b[1] of
            the parent, since the memory is shared.
        */
        a = (int *) shmat(shmid, 0, 0);

        a[0] = 0; a[1] = 1;
        for( i=0; i< 10; i++) {
                sleep(1);
                a[0] = a[0] + a[1];
                a[1] = a[0] + a[1];
                printf("Parent writes: %d,%d\n",a[0],a[1]);
        }
        wait(&status);
```

```
                    /* each process should "detach" itself from the
                       shared memory after it is used */

                    shmdt(a);

                    /* Child has exited, so parent process should delete   the cretaed shared memory. Unlike
attach and detach,
                       which is to be done for each process separately,
                       deleting the shared memory has to be done by only
                       one process after making sure that noone else
                       will be using it
                     */

                    shmctl(shmid, IPC_RMID, 0);
          }
}
```

POINTS TO NOTE:

In this case we find that the child reads all the values written by the parent. Also the child does not print the same values again.

1. Modify the sleep in the child process to sleep(2). What happens now?

2. Restore the sleep in the child process to sleep(1) and modify the sleep in the parent process to
sleep(2). What happens now?

Thus we see that when the writer is faster than the reader, then the reader may miss some of the values written into the shared memory. Similarly, when the reader is faster than the writer, then the reader may read the same values more than once. Perfect inter-process communication requires synchronization between thereader and the writer. You can use semaphores to do this. Further note that "sleep" is not a synchronization construct. We use "sleep" to model some amount of computation which may exist in the process in a real world application. Also, we have called the different shared memory related functions such as shmget, shmat, shmdt, and shmctl, assuming that they always succeed and never fail.

This is done to keep this program simple. In practice, you should always check for the return values from this function and exit if there is an error.

1. **Write a program that creates a shared memory segment and waits until two other separate processes writes something into that shared memory segment after which it prints what is written in shared memory. For the communication between the processes to take place assume that the process 1 writes 1 in first position of shared memory and waits; process 2 writes 2 in first position of shared memory and goes on to write 'hello' and then process 3 writes 3 in first position of shared memory and goes on to write 'memory' and finally the process 1 prints what is in shared memory written by two other processes.**

2. **Write a C program to implement the following game. The parent program P first creates two pipes, and then spawns two child processes C and D. One of the two pipes is meant for communications between P and C, and the other for communications between P and D. Now, a loop runs as follows. In each iteration (also called round), P first randomly chooses one of the two flags: MIN and MAX (the choice randomly varies from one iteration to another). Each of the two child processes C and D generates a random positive integer and sends that to P via its pipe. P reads the two integers; let these be c and d. If P has chosen MIN, then the child who sent the smaller of c and d gets one point. If P has chosen MAX, then the sender of the larger of c and d gets one point. If c = d, then this round is ignored. The child process who first obtains ten points wins the game. When the game ends, P sends a user-defined signal to both C and D, and the child processes exit after handling the signal (in order to know who was the winner). After C and D exit, the parent process P exits. During each iteration of the game, P should print appropriate messages (like P's choice of the flag, the integers received from C and D, which child gets the point, the current scores of C and D) in order to let the user know how the game is going on.**

   Name your program childsgame.c.

OBSERVATION SPACE

7. Implementation of CPU Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

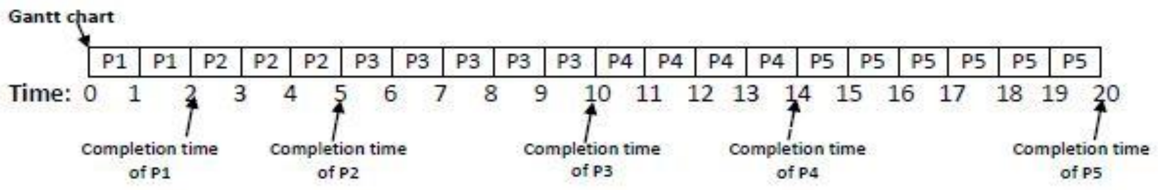
First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Q. Consider the following processes with burst time (CPU Execution time). Calculate the average waiting time and average turnaround time?

Process id	Arrival time	Burst time/CPU execution time
P1	0	2
P2	1	3
P3	2	5
P4	3	4
P5	4	6

Sol.


Gantt chart
P1 P1 P2 P2 P2 P3 P3 P3 P3 P3 P4 P4 P4 P4 P5 P5 P5 P5 P5 P5
Time: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Completion time of P1
Completion time of P2
Completion time of P3
Completion time of P4
Completion time of P5

Turnaround time= Completion time – Arrival time

Waiting time= Turnaround time – Burst time

Process id	Arrival time	Burst time	Completion time	Turnaround time	Waiting time
P1	0	2	2	2-0=2	2-2=0
P2	1	3	5	5-1=4	4-3=1
P3	2	5	10	10-2=8	8-5=3
P4	3	4	14	14-3=11	11-4=7
P5	4	6	20	20-4=16	16-6=10

Average turnaround time= $\sum_{i=0}^{n}$ Turnaround time(i)/n where, n= no. of process

Average waiting time= $\sum_{i=0}^{n}$ Wating time(i)/n where, n= no. of process

Average turnaround time= 2+4+8+11+16/5 =41/5 =8.2

Average waiting time= 0+1+3+7+10/5 = 21/5 =4.2

Shortest Job First (SJF) (Non-preemptive)

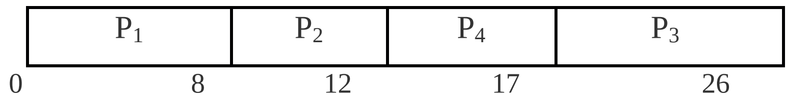
Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

- Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

  Example:

PROCESS	**ARRIVAL TIME**	**BURST TIME**
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart



PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	8 – 0 = 8
P2	8 – 1 = 7	12 – 1 = 11
P3	17 – 2 = 15	26 – 2 = 24
P4	12 – 3 = 9	17 – 3 = 14

Total Wait Time
0 + 7 + 15 + 9 = 31 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)
31/4 = 7.75 ms

Total Turn Around Time
8 + 11 + 24 + 14 = 57 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)
57/4 = 14.25 ms

Shortest Job First (SJF) (preemptive)

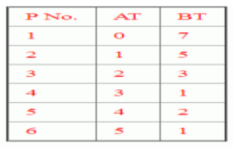
According to the SJF algorithm, the jobs in the queue are compared with each other and the one with shortest burst time gets executed first. The remaining processes are also executed in the order of their burst times. However, there may be scenarios where one or more processes have same execution time. In such cases, the jobs are based on **first come first serve** basis or in other words, FIFO approach is used.

This is a **preemptive algorithm** which means that the CPU can leave a process while under execution, and can move to the next process in the queue.
Meanwhile, the current state of the process is saved by context switch and another job can be processed in the meantime.

Once the CPU scheduler comes back to the previous job which was incomplete, resumes it from where it was stopped.

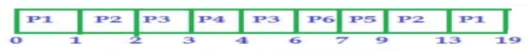
The shortest job first algorithm (preemptive) is also popularly known as Shortest Remaining Time First algorithm.

Example:

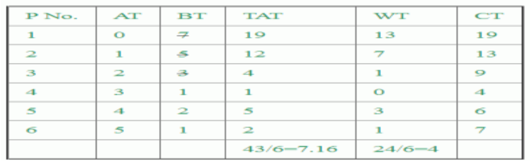


P No.	AT	BT
1	0	7
2	1	5
3	2	3
4	3	1
5	4	2
6	5	1

Solution-



Gantt Chart



P No.	AT	BT	TAT	WT	CT
1	0	~~7~~	19	13	19
2	1	~~5~~	12	7	13
3	2	~~3~~	4	1	9
4	3	1	1	0	4
5	4	2	5	3	6
6	5	1	2	1	7
			43/6—7.16	24/6—4	

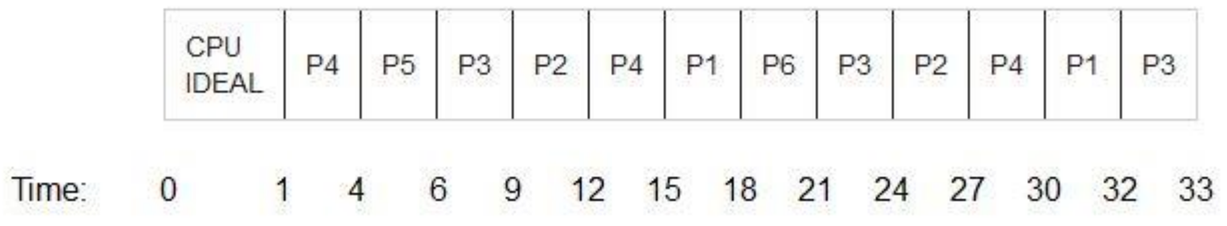
Round Robin Scheduling

Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum***. When a process is given the CPU, a timer is set for whatever value has been set for a time quantum. If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm. If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue. The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on. RR scheduling can give the effect of all processors sharing the CPU equally, although the average
wait time can be longer than with other scheduling algorithms.

Example: Consider the following processes with arrival time and burst time. Calculate average turnaround time, average waiting time and average response time using round robin with time quantum 3?

Process id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

Queue: P4,P5,P3,P2,P4,P1,P6,P3,P2,P4,P1,P3



Process id	Arrival time	Burst time	Completion time	Turnaround time	Waiting time	Response time
P1	5	5	32	27	22	10
P2	4	6	27	23	17	5
P3	3	7	33	30	23	3
P4	1	9	30	29	20	0
P5	2	2	6	4	2	2
P6	6	3	21	15	12	12

Average turnaround time=(27+23+30+29+4+15)6=21.33

Average waiting time=(22+17+23+20+2+12)6=16

Average response time=(10+5+3+0+2+12)6=5.33

1. **WAP to implement the First Come First Serve Scheduling algorithm.**
2. **WAP to implement shortest job first (Premptive) scheduling algorithm.**
3. **WAP to implement round robin scheduling algorithm.**

OBSERVATION SPACE

8. Multithreading using pthread.

What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.

Unlike Java, multithreading is not supported by the language standard. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

Example 1:

Two threads displaying two strings “Hello” and “How are you?” independent of each other.

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

void * thread1()
{
	while(1){
		printf("Hello!!\n");
	}
}

void * thread2()
{
	while(1){
		printf("How are you?\n");
	}
}

int main()
{
	int status;
	pthread_t tid1,tid2;

	pthread_create(&tid1,NULL,thread1,NULL);
	pthread_create(&tid2,NULL,thread2,NULL);
	pthread_join(tid1,NULL);
	pthread_join(tid2,NULL);
	return 0;
}
```

Now compile this program (Note the -l option is to load the pthread library)

```
$gcc thread.c -lpthread
```

On running, you can see many interleaved “Hello!!” and “How are you?” messages

Example 2

This example involves a reader and a writer thread. The reader thread reads a string from the user and writer thread displays it. This program uses semaphore so as to achieve synchronization

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

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

char n[1024];
sem_t len;

void * read1()
{
	while(1){
		printf("Enter a string");
		scanf("%s",n);
		sem_post(&len);
	}
}

void * write1()
{
	while(1){
		sem_wait(&len);
		printf("The string entered is :");
		printf("==== %s\n",n);
	}

}

int main()
{
	int status;
	pthread_t tr, tw;

	pthread_create(&tr,NULL,read1,NULL);
	pthread_create(&tw,NULL,write1,NULL);
```

```
        pthread_join(tr,NULL);
        pthread_join(tw,NULL);
        return 0;
}
```

On running, in most cases we may be able to achieve a serial read and write(Thread1reads a string and Thread2 displays the same string). But suppose we insert a sleep function() in write1 like

```
void * write1()
{
        while(1){
                sleep(5);
                sem_wait(&len);
                printf("The string entered is :");
                printf("==== %s\n",n);
        }
}
```

The thread 1 may read one more string and thread2 displays the last read string. That is no serial read and write is achieved.So we may need to use the condition variables to achieve serial read and write.

1) Write a multi-threaded C code with one thread printing all even numbers and the other all odd numbers.

OBSERVATION SPACE

---

9. Producer Consumer problem using semaphore.

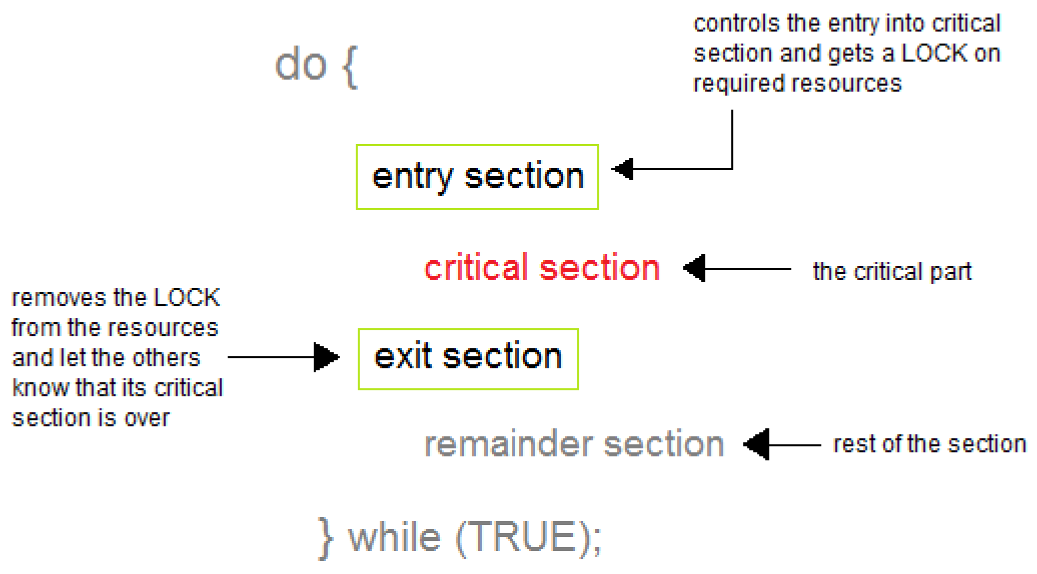
Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Bounded Buffer Problem

- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes. Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

The Bounded-Buffer Problem

This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size. In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

The structure of the producer process

```
While(true){

        //   produce an item

     wait (empty);
     wait (mutex);

        //  add the item to the  buffer

     signal (mutex);
     signal (full);
}
```

The structure of the consumer process

```
while (true) {
     wait (full);
     wait (mutex);

          //  remove an item from  buffer

     signal (mutex);
     signal (empty);

          //  consume the removed item

}
```

1) **Implement producer consumer problem (bounded buffer) using multithreading and semaphore.**

OBSERVATION SPACE

10.Reader Writer Problem using semaphore.

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.

A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write.

Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

Shared Data

- Semaphore mutex initialized to 1.
- Semaphore wrt initialized to 1.
- Integer readcount initialized to 0.

The structure of a writer process

```
 while (true) {
         wait (wrt) ;
             //    writing is performed
         signal (wrt) ;
}
```

The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)  wait (wrt) ;
        signal (mutex)

               // reading is performed
         wait (mutex) ;
         readcount  - - ;
         if (redacount  == 0)  signal (wrt) ;
         signal (mutex) ;
}
```

2) **Implement Reader Writers problem using multithreading and semaphore.**

OBSERVATION SPACE

11.Bankers Algorithm for Deadlock Avoidance.

Deadlock: A process request the resources, the resources are not available at that time, so the process enter into the waiting state. The requesting resources are held by another waiting process,both are in waiting state, this situation is said to be Deadlock. A deadlocked system must satisfied the following 4 conditions. These are:

(i) Mutual Exclusion: Mutual Exclusion means resources are in non-sharable mode only, it means only one process at a time can use a process.

(ii) Hold and Wait: Each and every process is the deadlock state, must hold at least one resource and is waiting for additional resources that are currently being held by another process.

(iii) No Preemption: No Preemption means resources are not released in the middle of the work, they released only after the process has completed its task.

(iv) Circular Wait: If process P1 is waiting for a resource R1, it is held by P2, process P2 is waiting for R2, R2 held by P3, P3 is waiting for R4, R4 is held by P2, P2 waiting for resourceR3, it is held by P1.

$$\begin{tikzcd}
P1 \arrow[r] & R1 \arrow[r] & P2 \arrow[r] & R2 \arrow[r] & P3 \arrow[r] & R4 \arrow[r] & P2 \arrow[r] & R3 \arrow[lllllll, bend left]
\end{tikzcd}$$

Deadlock Avoidance: It is one of the methods of dynamically escaping from the deadlocks. In this scheme, if a process request for resources, the avoidance algorithm checks before the allocation of resources about the state of system. If the state is safe, the system allocate the resources to the requesting process otherwise (unsafe) do not allocate the resources. So taking care before the allocation said to be deadlock avoidance.

Banker’s Algorithm: It is the deadlock avoidance algorithm, the name was chosen because the bank never allocates more than the available cash. Available: A vector of length ‘m’ indicates the number of available resources of each type. If available[j]=k, there are ‘k’ instances of resource types Rj available. Allocation: An nxm matrix defines the number of resources of each type currently allocated to each process. If allocation[i,j]=k, then process Pi is currently allocated ‘k’ instances of resources type Rj.

Max: An nxm matrix defines the maximum demand of each process. If max[i,j]=k, then Pimay request at most ‘k’ instances of resource type Rj.

Need: An nxm matrix indicates the remaining resources need of each process. If need[I,j]=k, then Pi may need ‘k’ more instances of resource type Rj to complete this task. There fore,

Need[i,j]=Max[i,j]-Allocation[I,j]

Safety Algorithm:

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

2. Find an i such that both

 Finish[i] =False

 Need<=Work

If no such I exists go to step 4.

3. work=work+Allocation, Finish[i] =True;

4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.

2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

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

Available=Available-Request I;

Allocation I =Allocation +Request I;

Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

Algorithm for Banker's Deadlock Avoidance:

1. Start the program.

2. Get the values of resources and processes.

3. Get the avail value.

4. After allocation find the need value.

5. Check whether its possible to allocate.

6. If it is possible then the system is in safe state.

7. Else system is not in safety state.

8. If the new request comes then check that the system is in safety.

9. or not if we allow the request.

10. stop the program.

1) Implement Banker's algorithm for deadlock avoidance.

OBSERVATION SPACE

12. Implementation of Page Replacement Algorithm

Description:

FIFO (First in First Out) algorithm: FIFO is the simplest page replacement algorithm, the idea behind this is, “Replace a page that page is oldest page of main memory” or “Replace the page that has been in memory longest”. FIFO focuses on the length of time a page has been in the memory rather than how much the page is being used.

Algorithm for FIFO Page Replacement:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Description:

LRU (Least Recently Used): the criteria of this algorithm are “Replace a page that has been used for the longest period of time”. This strategy is the page replacement algorithm looking backward in time, rather than forward.

Algorithm for LRU Page Replacement:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

1) **Implement FIFO page replacement algorithm**
2) **Implement LRU page replacement algorithm**

OBSERVATION SPACE

REFERENCES

1. IIT Bombay 2017: https://www.cse.iitb.ac.in/~cs347/
2. III Bombay 2016 : https://www.cse.iitb.ac.in/~mythili/teaching/cs347_autumn2016/index.html
3. IIT Kharagpur: http://cse.iitkgp.ac.in/~abhij/course/lab/OSL/Spring14/
4. Prof Tushar B. Kute: https://sites.google.com/site/uopops/pm
5. Kuwait University:
   www.isc.ku.edu.kw/documents/Lab%20Manuals/ISC%20357%20Lab%20Manual