MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A constituent institution of Manipal University

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. ……...……

Reg. No.: …..…… Section: ……..…… Roll No.: ……...

has satisfactorily completed the lab exercises prescribed for OPERATING SYSTEMS LAB [CSE 3163] of Third Year B. Tech. Degree at MIT, Manipal, in the academic year 2023-2024.

Date: …….....

Signature

Faculty in Charge

CONTENTS

Course Objectives

- Illustrate and explore system calls related to Linux operating system.
- Learn process management and thread programming concepts which include scheduling algorithms and inter process communication.
- Understand the working of memory management schemes, disk scheduling algorithms, and page replacement algorithms through simulation.

Course Outcomes

At the end of this course, students will be able to:

- Execute Linux commands, shell scripting using appropriate Linux system calls.
- Design thread programming, simulate process management and inter process communication techniques.
- Implement the memory management, disk scheduling and page replacement algorithms.

Evaluation Plan

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (for each experiment): 3.5 x 10 = 35 marks
  - ✓ Mid Term Examination 15M
  - ✓ Mini Project 10M
  - ✓ Total marks of the 12 experiments will sum up to 60
- End semester assessment of 2-hours duration: 40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre-Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session.
2. Be in time and follow the Instructions from Lab Instructors.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In-Lab Session Instructions

- Follow the instructions on the allotted exercises given in Lab Manual.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs are properly indented and comments should be given whenever it is required.
  - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
  - Solved exercise

- Lab exercises - to be completed during lab hours
- Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.

- In case a student misses a lab class, he/she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given later in the manual as a model for observation.

THE STUDENTS SHOULD NOT

- Carry mobile phones while working with computer.
- Go out of the lab without permission.

LAB NO.: 1 **Date:**

WORKING WITH REGULAR FILES

Objectives:

In this lab, the student will be able to:

- Learn to create files, open them, read, write and close them
- Learn the need to make system calls and a whole range of library functions to efficiently handle files

Each running program, called a process, has several file descriptors associated with it. When a program starts, it usually has three of these
descriptors already opened. These are:

0: Standard input
1: Standard output
2: Standard error

SYSTEM CALLS:

Write System Call:

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes. It returns the number of bytes written. This may be less than nbytes if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the errno global variable.
Here's the syntax:

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
if ((write(1, "Here is some data\n", 18)) != 18)
write(2, "A write error has occurred on file descriptor 1\n",46);
exit(0);
}
```

This program simply prints a message to the standard output. When a program exits, all open file descriptors are automatically closed, so you don't need to close them explicitly This won't be the case, however, when you're dealing with buffered output.

$ ./simple_write
Here is some data
$

A point worth noting again is that write might report that it wrote fewer bytes than you asked it to. This is not necessarily an error. In your programs, you will need to check errno to detect errors and call write to write any remaining data.

Read System Call:

The read system call 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 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.

#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
This program, simple_read.c, copies the first 128 bytes of the standard input to the standard output. It
copies all of the input if there are fewer than 128 bytes.

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
char buffer[128];
int nread;
nread = read(0, buffer, 128);
if (nread == -1)
write(2, "A read error has occurred\n", 26);
if ((write(1,buffer,nread)) != nread)
write(2, "A write error has occurred\n",27); exit(0);
}
```

If you run the program, you should see the following:

```
$ echo hello there | ./simple_read
hello there
$ ./simple_read < draft1.txt
```

In the first execution, you create some input for the program using echo, which is piped to your program. In the second execution, you redirect input from a file. In this case, you see the first part of the file draft1.txt appearing on the standard output.

Open System Call:

To create a new file descriptor, you need to use the open system call.
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
In simple terms, open establishes an access path to a file or device. If successful, it returns a file descriptor that can be used in read, write, and other system calls. The file descriptor is unique and isn't shared by any other processes that may be running. If two programs have a file open at the same time, they maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn't interleaved, but one will overwrite the other. Each keeps its idea of how far into the file (the offset) it has read or written. You can prevent unwanted clashes of this sort by using file locking. The name of the file or device to be opened is passed as a parameter, path; the oflags parameter is used to specify actions to be taken on opening the file. The oflags are specified as a combination of a mandatory file access mode and other optional modes. The open call must specify one of the file access modes shown in the following table:

Mode	Description
O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

The call may also include a combination (using a bitwise OR) of the following optional modes in the
oflags parameter:
O_APPEND: Place written data at the end of the file.
O_TRUNC: Set the length of the file to zero, discarding existing contents.
O_CREAT: Creates the file, if necessary, with permissions given in mode.
O_EXCL: Used with O_CREAT, ensures that the caller creates the file. The open is atomic; that is, it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Other possible values for oflags are documented in the open manual page, which you can find in section 2 of the manual pages (use man 2 open).
open returns the new file descriptor (always a nonnegative integer) if successful, or –1 if it fails, at time open also sets the global variable errno to indicate the reason for the failure. We look at errno more closely in a later section. The new file descriptor is always the lowest-numbered unused descriptor, a feature that can be quite useful in some circumstances. For example, if a program closes its standard output and then calls open again, the file descriptor 1 will be reused and the standard output will have been effectively redirected to a different file or device.

Initial Permission

When you create a file using the O_CREAT flag with open, you must use the three-parameter form. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h. These are:
S_IRUSR: Read permission, owner
S_IWUSR: Write permission, owner
S_IXUSR: Execute permission, owner
S_IRGRP: Read permission, group
S_IWGRP: Write permission, group
S_IXGRP: Execute permission, group
S_IROTH: Read permission, others
S_IWOTH: Write permission, others
S_IXOTH: Execute permission, others
For example,
open (“myfile”, O_CREAT, S_IRUSR|S_IXOTH);

Unmask system call
The umask is a system variable that encodes a mask for file permissions to be used when a file is created. You can change the variable by executing the umask command to supply a new value. The value is a three-digit octal value. Each digit is the result of ORing values from 1, 2, or 4; the meanings are shown in the following table. The separate digits refer to “user,” “group,” and “other” permissions, respectively.

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
	4	User read permission is disallowed.
	2	User write permission is disallowed.
	1	User execute permission is disallowed.
2	0	No group permissions are to be disallowed.
	4	Group read permission is disallowed.
	2	Group write permission is disallowed.

	1	Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
	4	Other read permission is disallowed.
	2	Other write permission is disallowed.
	1	Other execute permission is disallowed

Close System Call

You use close to terminate the association between a file descriptor, fildes, and its file. The file descriptor becomes available for reuse. It returns 0 if successful and –1 on error.

```
#include <unistd.h>
int close(int fildes);
```

Note that it can be important to check the return result from close. Some file systems, particularly networked ones, may not report an error writing to a file until the file is closed, because data may not have been confirmed as written when writes are performed.

A File Copy Program

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
char c;
int in, out;
in = open("file.in", O_RDONLY);
out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
while(read(in,&c,1) == 1)
write(out,&c,1);
exit(0);
}
```

WITH fcntl.h how the system calls can manipulate the files.

I/O SYSTEM CALLS

I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library.

There are seven fundamental file-I/O system calls:

creat() Create a file for reading or writing.
open() Open a file for reading or writing.
close() Close a file after reading or writing.
unlink() Delete a file.
write() Write bytes to file.
read() Read bytes from file.

The creat() System Call

The "creat()" system call creates a file. It has the syntax: int fp; /* fp is the file descriptor variable */
fp = creat(<filename>, <protection bits>);
Ex: fp=creat(“students.dat”,RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file.

The "permission bits" give the "access rights" to the file. A file has three "permissions" associated with it:

Write permission - Allows data to be written to the file.

Read permission - Allows data to be read from the file.

Execute permission - Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level: Permissions apply to individual user.

Group level: Permissions apply to members of user's defined "group".

System level: Permissions apply to everyone on the system

The open() System Call

The "open()" system call opens an existing file for reading or writing. It has the syntax:

<file descriptor variable> = open(<filename>, <access mode>);

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes (defined in the "fcntl.h" header file):

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
int fd;

fd = open( "students.dat", O_WRONLY );
```

A few additional comments before proceeding:

A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() System Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it. The "close()" system call has the syntax:
close(<file descriptor>);

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The write() System Call

The "write()" system call writes data to an open file. It has the syntax:
write(<file descriptor>, <buffer>, <buffer length>);

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

```
float array[10];
write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from an open file. Its syntax is the same as that of the "write()" call:

read(<file descriptor>, <buffer>, <buffer length>);

The "read()" function returns the number of bytes it returns. At the end of the file, it returns 0 or returns -1 on error.

lseek: The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, we can use it to set wherein the file the next read or write will occur. We can set the pointer to an absolute location in the file or a position relative to the current position or the end of file.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

SEEK_SET: offset is an absolute position

SEEK_CUR: offset is relative to the current position

SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to or -1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent type defined in sys/types.h.

Errors*:*

EACCES Permission denied.

EMFILE Too many file descriptors in use by the process.

ENFILE Too many files are currently open in the system.

ENOENT Directory does not exist, or the name is an empty string.

ENOMEM Insufficient memory to complete the operation.

Lab Exercises:

1. Write a program to print the lines of a file that contain a word given as the program argument (a simple version of grep UNIX utility).
2. Write a program to list the files given as arguments, stopping every 20 lines until a key is hit. (a simple version of more UNIX utility)
3. Demonstrate the use of different conversion specifiers and resulting output to allow the items to be printed.
4. Write a program to copy character-by character copy is accomplished using calls to the functions referenced in stdio.h

Additional Exercises:

1. Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.
2. Use lseek() to copy different parts (initial, middle and last) of the file to others. (For lseek() refer to man pages)

LAB NO.: 2 **Date:**

WORKING WITH DIRECTORY STRUCTURES

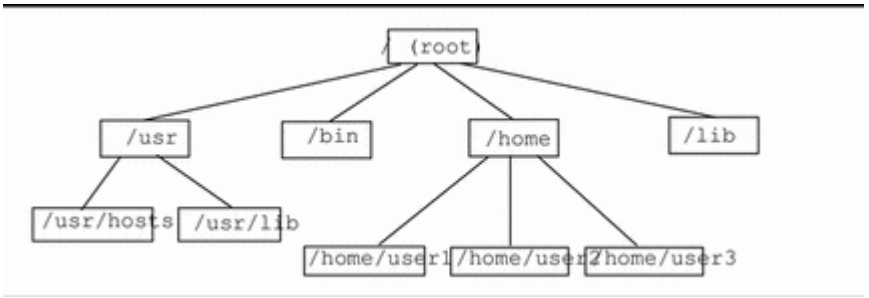
Objectives:
In this lab, the student will be able to:

- Understand how programs can manipulate directories
- Work with system calls to create, scan and work with directories

All subdirectory and file names within a directory must be unique. However, names within different directories can be the same. For example, the directory /usr contains the subdirectory /usr/lib. There is no conflict between /usr/lib and /lib because the path names are different.

Pathnames for files work exactly like path names for directories. The pathname of a file describes that file's place within the file system hierarchy. For example, if the /home/user2 directory contains a file called report5, the pathname for this file is /home/user2/report5. This shows that the file report5 is within the directory user2, which is within the directory home, which is within the root (/) directory.

Directories can contain only subdirectories, only files, or both.



Print Working Directory (pwd)

The pwd command will give the present working directories

$ **pwd**
/home/user1

Your Home Directory

home
Every user has a **home** directory. When you first open the Command Tool or Shell Tool window in the OpenWindows environment, your initial location (working directory) is your home directory.

A program can determine its current working directory by calling the getcwd function.
#include <unistd.h>
char *getcwd(char *buf, size_t size);
The getcwd function writes the name of the current directory into the given buffer, buf . It returns NULL
if the directory name would exceed the size of the buffer (an ERANGE error), given as the parameter
size .
*It returns buf on success.
*getcwd may also return NULL if the directory is removed (EINVAL) or permissions changed (EACCESS) while the program is running.

mkdir and rmdir
You can create and remove directories using the mkdir and rmdir system calls.
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);

The mkdir system call is used for creating directories and is the equivalent of the mkdir program. **mkdir** makes a new directory with a path as its name. The directory permissions are passed in the parameter mode and are given as in the O_CREAT option of the open system call and, again, subject to umask .

#include <unistd.h>
int rmdir(const char *path);
The rmdir system call removes directories, but only if they are empty. The rmdir program uses this
system call to do its job.

Scanning Directories

A common problem on Linux systems is scanning directories, that is, determining the files that reside in a particular directory. In shell programs, it's easy — just let the shell expand a wildcard expression. In the past, different UNIX variants have allowed programmatic access to the low-level file system structure. You can still open a directory as a regular file and directly read the directory entries, but different file system structures and implementations have made this approach nonportable. A standard suite of library functions has now been developed that makes directory scanning much simpler. The directory functions are declared in a header file dirent.h. They use a structure, DIR , as a basis for directory manipulation. A pointer to this structure, called a directory stream (a DIR *), acts in much the same way as a file stream (FILE *) does for regular file manipulation. Directory entries themselves are returned in dirent structures, also declared in dirent.h, because one should never alter the fields in the DIR structure directly.
We'll review these functions:
opendir
closedir
readdir
telldir
seekdir
closedir

opendir
The opendir function opens a directory and establishes a directory stream. If successful, it returns a pointer to a DIR structure to be used for reading directory entries.
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
opendir returns a null pointer on failure. Note that a directory stream uses a low-level file descriptor to access the directory itself, so opendir could fail with too many open files.

readdir
The readdir function returns a pointer to a structure detailing the next directory entry in the directory stream dirp. Successive calls to readdir return further directory entries. On error, and at the end of the directory, readdir returns NULL . POSIX-compliant systems leave errno unchanged when returning NULL at end of directory and set it when an error occurs.

#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
Note that readdir scanning isn't guaranteed to list all the files (and subdirectories) in a directory if other processes are creating and deleting files in the directory at the same time.
The dirent structure containing directory entry details includes the following entries:

- ino_t d_ino : The inode of the file
- char

d_name[] : The name of the file

telldir

The telldir function returns a value that records the current position in a directory stream. You can use
this in subsequent calls to seekdir to reset a directory scan to the current position.
#include <sys/types.h>
#include <dirent.h>
long int telldir(DIR *dirp);

seekdir

The seekdir function sets the directory entry pointer in the directory stream given by dirp . The value of loc , used to set the position, should have been obtained from a prior call to telldir .
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long int loc);

closedir

The closedir function closes a directory stream and frees up the resources associated with it. It returns 0 on success and –1 if there is an error.
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);

Sample Program

A Directory-Scanning Program
1.Start with the appropriate headers and then a function, printdir , which prints out the current directory. It will recurse for subdirectories using the depth parameter for indentation.

```
#include<unistd.h>
#include<stdio.h>
#include<dirent.h>
#include<string.h>
#include<sys/stat.h>
#include<stdlib.h>

void printdir(char *dir, int depth)
{
DIR *dp;
struct dirent *entry;
struct stat statbuf;
if((dp = opendir(dir)) == NULL) {
fprintf(stderr,"cannot open directory: %s\n", dir);
return;
}
chdir(dir);
while((entry = readdir(dp)) != NULL) {
lstat(entry->d_name,&statbuf);
if(S_ISDIR(statbuf.st_mode)) {
/* Found a directory, but ignore . and .. */
if(strcmp(".",entry->d_name) == 0 ||
strcmp("..",entry->d_name) == 0)
continue;
printf("%*s%s/\n",depth,"",entry->d_name);
/* Recurse at a new indent level */
printdir(entry->d_name,depth+4);
}
else printf("%*s%s\n",depth,"",entry->d_name);
}
chdir("..");
closedir(dp);
}
```

Lab Exercises:

1. Write a C program to emulate the ls -l UNIX command that prints all files in a current directory and lists access privileges, etc. DO NOT simply exec ls -l from the program.

2. Write a program that will list all files in a current directory and all files in subsequent subdirectories.
3. How do you list all installed programs in Linux?
4. How do you find out what RPM packages are installed on Linux?

Additional Exercises:

1. Write a program that will only list subdirectories in alphabetical order.
2. Write a program that allows the user to remove any or all of the files in a current working directory. The name of the file should appear followed by a prompt as to whether it should be removed.

LAB NO.: 3 **Date:**

PROCESSES AND SIGNALS

Objectives:

In this lab, the student will be able to

- Learn how the operating system manages processes
- Learn how the system handles processes, processes send, and receive messages
- Understand the concept of orphan and zombie processes

System Calls Related to Processes

getpid()

This function returns the process identifiers of the calling process.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);   // this function returns the process identifier (PID)
pid_t getppid(void); // this function returns the parent process identifier (PPID)
```

fork()

A new process is created by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the **exec** functions, the **fork** is all we need to create new processes.

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

The return value of fork() is pid_t (defined in the header file sys/types.h). As seen in Fig. 4.1, the call to the fork in the parent process returns the PID of the new child process. The new process continues to execute just like the parent process, with the exception that in the child process, the PID returned is 0. The parent and child process can be determined by using the PID returned from the fork() function. To the parent, the fork() returns the PID of the child, whereas to the child the PID returned is zero. This is shown in the following Fig. 3.1.

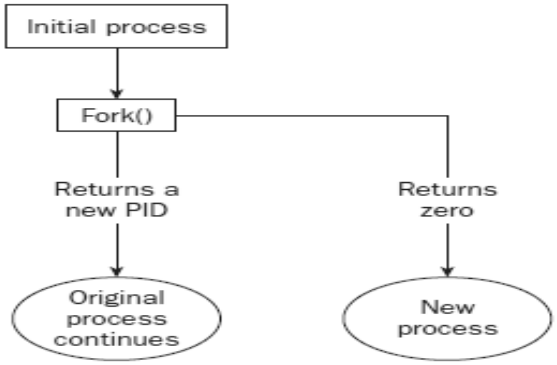


Figure 3.1: Fork system call

In Linux in case of any error observed in calling the system functions, then a special variable called errno will contain the error number. To use errno a header file named errno.h has to be included in the program. If the fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code snippet using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Sample Program on fork1.c

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

```
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        break;
                default:
                        message = "This is the parent";
                        n = 3;
                        break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
        exit(0);
}
```

This program runs as two processes. A child process is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the parent
This is the child
```

This is the parent

This is the child

This is the parent

This is the child

This is the child

This is the child

When fork is called, this program divides into two separate processes. The parent process is identified by a nonzero return from fork and is used to set several messages to print, each separated by one second.

The wait() System Call

A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions. The wait() system call allows the parent process to suspend its activities until one of these actions has occurred. The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t. If the calling process does not have any child associated with it, the wait will return immediately with a value of -1. If any child processes are still active, the calling process will suspend its activity until a child process terminates.

Example of wait():

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

void main()
{
        int status;
        pid_t  pid;
        pid = fork();
        if(pid = -1)
          printf("\nERROR child not created");
         else if (pid == 0) /* child process */
        {
                printf("\n I'm the child!");
                 exit(0);
        }
```

```
        else /* parent process */
        {
                wait(&status);
                printf("\n I'm the parent!")
                printf("\n Child returned: %d\n", status)
        }
}
```

A few notes on this program:

wait(&status) causes the parent to sleep until the child process has finished execution. The exit status of the child is returned to the parent.

The exit() System Call

This system call is used to terminate the currently running process. A value of zero is passed to indicate that the execution of the process was successful. A non-zero value is passed if the execution of the process was unsuccessful. All shell commands are written in C including grep. grep will return 0 through exit if the command is successfully run (grep could find a pattern in the file). If grep fails to find a pattern in a file, then it will call exit() with a non-zero value. This applies to all commands.

The exec() System Call

The exec function will execute a specified program passed as an argument to it, in the same process (Fig. 3.2). The exec() will not create a new process. As a new process is not created, the process ID (PID) does not change across an execution, but the data and code of the calling process are replaced by those of the new process.

fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec().

The versions of exec are:

- execl
- execv
- execle

- execve
- execlp
- execvp

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null-terminated arguments)
- 'v' indicates the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

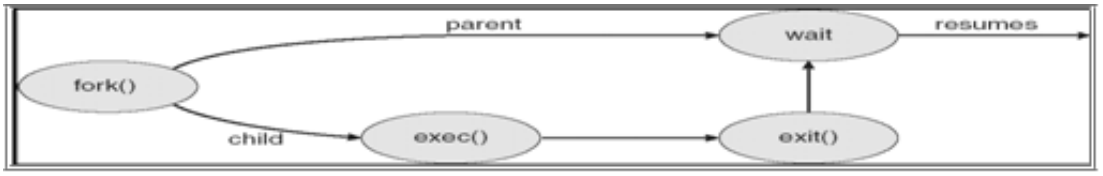


Figure 3.2: exec() system call

The parent process can either continue execution or wait for the child process to complete. If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. If a parent does not wait for the child, and the child terminates before the parent, then the child is called a **zombie** process. If a parent terminates before the child process then the child is attached to a process called init (whose PID is 1). In this case, whenever the child does not have a parent then the child is called the **orphan** process.

Sample Program:

C program forking a separate process.

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
```

```
{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
        /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

execl: is used with a list comprising the command name and its arguments:

int execl(const char *path, const char *arg0, …../*, (char *) 0 */);

This is used when the number of arguments is known in advance. The first argument is the pathname which could be absolute or a relative pathname, The arguments to the command to run are represented as separate arguments beginning with the name of the command (*arg0). The ellipsis representation in the syntax (…/*) points to the varying number of arguments.

Example: How to use execl to run the wc –l command with the filename foo as argument:

execl (“/bin/wc”, “wc”, “-l”, “foo”, (char *) 0);
execl doesn’t use PATH to locate wc so pathname is specified as the first argument.

execv: needs an array to work with.

int execv(const char *path, char *const argv[ ]);

Here path represents the pathname of the command to run. The second argument represents an array of pointers to char. The array is populated by addresses that point to strings representing the command name and its arguments, in the form they are passes to the main function of the program to be executed. In this case, also the last element of the argv[ ] must be a null pointer.

Here the following program uses execv program to run grep command with two options to look up the author's name in /etc/passwd. The array *cmdargs[ ] are populated with the strings comprising the command line to be executed by execv. The first argument is the pathname of the command:

```
#include<stdio.h>
int main(int argc, char **argv){
char *cmdargs[ ] = {"grep", "-I", "-n", "SUMIT", "/etc/passed", NULL};
execv("/bin/grep", cmdargs);
printf ("execv error\n");
}
```

Drawbacks:

Need to know the location of the command file since neither execl nor execv will use PATH to locate it. The command name is specified twice - as the first two arguments. These calls can't be used to run a shell script but only binary executable. The program has to be invoked every time there is a need to run a command.

execlp and execvp: requires the pathname of the command to be located. They behave exactly like their other counterparts but overcomes two of the four limitations discussed above. First the first argument need not be a pathname it can be a command name. Second these functions can also run a shell script.

```
int execlp(const char *file, const char *arg0, …./*, (char *) 0 */);
int execvp(const char *file, char *const argv[ ]);
```

```
execlp ("wc", "wc", "-l", "foo", (char *) 0);
```

execle and execve: All of the previous four exec calls silently pass the environment of the current process to the executed process by making available the environ[ ] variable to the overlaid process. Sometime there may be a need to provide a different environment to the new program - a restricted shell for instance. In that case, these

functions are used.

int execle(const char *path, const char *arg0, … /*, (char *) 0, char * const envp[ ] */);
int execve(const char *path, char * const argv[ ], char *const envp[ ]);

These functions unlike the others use an additional argument to pass a pointer to an array of environment strings of the form variable = value to the program. It's only this environment that is available in the executed process, not the one stored in envp[ ].

The following program (assume fork2.c) is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
        case -1:
                perror("fork failed");
                exit(1);

        case 0:
                message = "This is the child";
                n = 3;
                break;
        default:
                message = "This is the parent";
                n = 5;
                break;
}
```

When the preceding program is run with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, a line such as this. (Some systems may say <zombie> rather than <defunct>) is seen.

Lab Exercises:

1. Write a C program to block a parent process until the child completes using a wait

system call.

2. Write a C program to load the binary executable of the previous program in a child process using the exec system call.
3. Write a program to create a child process. Display the process IDs of the process, parent and child (if any) in both the parent and child processes.
4. Create a zombie (defunct) child process (a child with exit() call, but no corresponding wait() in the sleeping parent) and allow the init process to adopt it (after parent terminates). Run the process as a background process and run the "ps" command.

Additional Exercises

1. Create an orphan process (parent dies before child – adopted by "init" process) and display the PID of the parent of the child before and after it becomes orphan. Use sleep(n) in the child to delay the termination.
2. Modify the program in the previous question to include wait (&status) in the parent and to display the exit return code (leftmost byte of status) of the child.

LAB NO.: 4 **Date:**

FILE SYSTEM

Objectives:

In this lab, the student will be able to:

- Understand the file system and their metadata associated with them
- Operations that could be performed on the files in the file system

Each file is referenced by an inode, which is addressed by a filesystem-unique numerical value known as an inode number. An inode is both a physical object located on the disk of a Unix-style filesystem and a conceptual entity represented by a data structure in the Linux kernel. The inode stores the metadata associated with a file, such as a file's access permissions, last access timestamp, owner, group, and size, as well as the location of the file's data.

The file name is not available in an inode is the file's name and it is stored in the directory entry.

Obtain the inode number for a file using the -i flag to the ls command:

$ ls -i

To obtain information of files in the current directory in detail, use -il:

$ ls -il

To display the filesystem inode space information, the df command could be used:

$ df -i

Stat functions

Linux provides a family of functions for obtaining the metadata of a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Each of these functions returns information about a file.
stat() returns information about the file denoted by the path, while fstat() returns information about the file represented by the file descriptor fd.
lstat() is identical to stat(), except that in the case of a symbolic link, lstat() returns information about the link itself and not the target file.

Each of these functions stores information in a stat structure, which is provided by the user. The stat structure is defined in <bits/stat.h>, which is included from <sys/stat.h>:

```
struct stat {
dev_t st_dev; /* ID of device containing file */
ino_t st_ino; /* inode number */
mode_t st_mode; /* permissions */
nlink_t st_nlink; /* number of hard links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner */
dev_t st_rdev; /* device ID (if special file) */
off_t st_size; /* total size in bytes */
blksize_t st_blksize; /* blocksize for filesystem I/O */
blkcnt_t st_blocks; /* number of blocks allocated */
time_t st_atime; /* last access time */
time_t st_mtime; /* last modification time */
time_t st_ctime; /* last status change time */
};
```

A file is mapped to its inode by its parent directory.
The inode number of / (topmost) directory is fixed, and is always 2.

```
$ stat /
 File: '/'
 Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: 806h/2054d Inode: 2 Links: 27
Access: (0755/drwxr-xr-x) Uid: ( 0/ root) Gid: ( 0/ root)
```

Access: 2019-12-07 01:40:01.565097799 +0100
Modify: 2019-12-07 01:27:33.651924301 +0100
Change: 2019-12-07 01:27:33.651924301 +0100
Birth:

The fields are as follows:

• The st_dev field describes the device node on which the file resides. If the file is not backed by a device—for example, if it resides on an NFS volume—this value is 0.
• The st_ino field provides the file's inode number.
• The st_mode field provides the file's mode bytes, which describe the file type (such as a regular file or a directory) and the access permissions.
• The st_nlink field provides the number of hard links pointing at the file. Every file on a filesystem has at least one hard link.
The st_uid field provides the user ID of the user who owns the file.
• The st_gid field provides the group ID of the group who owns the file.
• If the file is a device node, the st_rdev field describes the device that this file represents.
• The st_size field provides the size of the file, in bytes.
• The st_blksize field describes the preferred block size for efficient file I/O. This value is the optimal block size for user-buffered I/O.
• The st_blocks field provides the number of filesystem blocks allocated to the file. This value multiplied by the block size will be smaller than the value provided by st_size if the file has holes.
• The st_atime field contains the last file access time. This is the most recent time at which the file was accessed.
• The st_mtime field contains the last file modification time—that is, the last time the file was written to.
• The st_ctime field contains the last file change time. The field contains the last time that the file's metadata (for example, its owner or permissions) was changed.

Return Values
On success, all three calls return 0 and store the file's metadata in the provided stat structure. On error, they return −1 and set errno to one of the following:

EACCES
The invoking process lacks search permission for one of the directory components of path (stat() and lstat() only).

EBADF
fd is invalid (fstat() only).

EFAULT
path or buf is an invalid pointer.

ELOOP
path contains too many symbolic links (stat() and lstat() only).

ENAMETOOLONG
path is too long (stat() and lstat() only).

ENOENT
A component in path does not exist (stat() and lstat() only).

ENOMEM
There is insufficient memory available to complete the request.

ENOTDIR
A component in path is not a directory (stat() and lstat() only).

Program that uses stat() to retrieve the size of a file provided on the command line:

```
/* Filename: stat.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
```

```
}
ret = stat (argv[1], &sb);
if (ret) {
perror ("stat");
return 1;
}
printf ("%s is %ld bytes\n", argv[1], sb.st_size);
return 0;
}
```

Here is the result of running the program on its own source file:

```
$ ./stat stat.c
stat.c is 392 bytes
```

Program to find the file type (such as symbolic link or block device node) of the file given by the first argument to the program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
}
ret = stat (argv[1], &sb);
if (ret) {
perror ("stat");
return 1;
}
printf ("File type: ");
switch (sb.st_mode & S_IFMT) {
case S_IFBLK:
```

```
printf("block device node\n");
break;
case S_IFCHR:
printf("character device node\n");
break;
case S_IFDIR:
printf("directory\n");
break;
case S_IFIFO:
printf("FIFO\n");
break;
case S_IFLNK:
printf("symbolic link\n");
break;
case S_IFREG:
printf("regular file\n");
break;
case S_IFSOCK:
printf("socket\n");
break;
default:
printf("unknown\n");
break;
}
return 0;
}
```

Links

Each name-to-inode mapping in a directory is called a link. A link is essentially just a name in a list (a directory) that points at an inode. A single inode (and thus a single file) could be referenced from, say, both /etc/customs and /var/run/ledger. However, because links map to inodes and inode numbers are specific to a particular filesystem, /etc/customs and /var/run/ledger must both reside on the same filesystem. Within a single filesystem, there can be a large number of links to any given file. The only limit is in the size of the integer data type used to hold the number of links. Among various

links, no one link is the “original” or the “primary” link. All of the links get the same status, pointing at the same file and called as hard links. Files can have 0, 1, or many links. Most files have a link count of 1 i.e. they are pointed at by a single directory entry, but some files have 2 or even more links. Files with a link count of 0 have no corresponding directory entries on the filesystem. When a file’s link count reaches 0, the file is marked as free, and its disk blocks are made available for reuse.5 Such a file, however, remains on the filesystem if a process has the file open. Once no process has the file open, the file is removed.

The link() system call standardized by POSIX, creates a new link for an existing file:

```
#include <unistd.h>
int link (const char *oldpath, const char *newpath);
```

A successful call to link() creates a new link under the path new path for the existing file oldpath and then returns 0. Upon completion, both the oldpath and newpath refer to the same file. There is no way to tell which was the “original” link.

On failure, the call returns −1 and sets errno to one of the following:

EACCES
The invoking process lacks search permission for a component in oldpath, or the invoking process does not have write permission for the directory containing newpath.

EEXIST
newpath already exists—link() will not overwrite an existing directory entry.

EFAULT
oldpath or newpath is an invalid pointer.

EIO
An internal I/O error occurred (this is bad!).

ELOOP
Too many symbolic links were encountered in resolving oldpath or newpath.

EMLINK
The inode pointed at by oldpath already has the maximum number of links pointing at it.

ENAMETOOLONG
oldpath or newpath is too long.

ENOENT
A component in oldpath or newpath does not exist.

ENOMEM
There is insufficient memory available to complete the request.

ENOSPC
The device containing newpath has no room for the new directory entry.

ENOTDIR
A component in oldpath or newpath is not a directory.

EPERM
The filesystem containing newpath does not allow the creation of new hard links, or oldpath is a directory.

EROFS
newpath resides on a read-only filesystem.

EXDEV
newpath and oldpath are not on the same mounted filesystem. (Linux allows a single filesystem to be mounted in multiple places, but even in this case, hard links cannot be created across the mount points.)

This example creates a new directory entry, pirate, that maps to the same inode (and thus the same file) as the existing file privateer, both of which are in /home/nayan:

```
int ret;
/* create a new directory entry, '/home/nayan/somecontents', that points at the same inode as '/home/nayan/contents' */

ret = link ("/home/ nayan/somecontents", /home/nayan/contents");
if (ret)
perror ("link");
```

Symbolic Links

Symbolic links, also known as symlinks or soft links, are similar to hard links in that both point at files in the filesystem. The symbolic link differs, however, in that it is not merely an additional directory entry, but a special type of file altogether. This special file contains the pathname for a different file, called the symbolic link's target. At runtime, on the fly, the kernel substitutes this pathname for the symbolic link's pathname. Thus, whereas one hard link is indistinguishable from another hard link to the same file, it is easy to tell the difference between a symbolic link and its target file.

A symbolic link may be relative or absolute. It may also contain the special dot directory discussed earlier, referring to the directory in which it is located, or the dot-dot directory, referring to the parent of this directory. Soft links, unlike hard links, can span filesystems. Symbolic links can point at files that exist or at nonexistent files. The latter type of link is called a dangling symlink. Sometimes, dangling symlinks are unwanted—such as when the target of the link was deleted, but not the symlink—but at other times, they are intentional. Symbolic links can even point at other symbolic links. This can create loops. System calls that deal with symbolic links check for loops by maintaining a maximum traversal depth. If that depth is surpassed, they return ELOOP.

The system call for creating a symbolic link is very similar as seen above:

```
#include <unistd.h>
int symlink (const char *oldpath, const char *newpath);
```

A successful call to symlink() creates the symbolic link newpath pointing at the target oldpath, and then returns 0. On error, symlink() returns −1 and sets errno to one of the following:

EACCES
The invoking process lacks search permission for a component in oldpath, or the invoking process does not have write permission for the directory containing new path.

EEXIST
newpath already exists—symink() will not overwrite an existing directory entry.

EFAULT
oldpath or newpath is an invalid pointer.

EIO
An internal I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving oldpath or newpath.

EMLINK
The inode pointed at by oldpath already has the maximum number of links pointing at it.

ENAMETOOLONG
oldpath or newpath is too long.

ENOENT
A component in oldpath or newpath does not exist.

ENOMEM
There is insufficient memory available to complete the request.

ENOSPC
The device containing newpath has no room for the new directory entry.

ENOTDIR
A component in oldpath or newpath is not a directory.

EPERM
The filesystem containing newpath does not allow the creation of new symbolic links.

EROFS
newpath resides on a read-only filesystem.

This snippet is the same as our previous example, but it creates /home/nayan/somecontents as a symbolic link to /home/nayan/contents:

```
int ret;
/* create a symbolic link, '/home/nayan/somecontents ', that points at
'/home/nayan/contents' */

ret = symlink ("/home/nayan/somecontents", "/home/nayan/contents");
if (ret)
perror ("symlink");
```

Unlinking

The converse to linking is unlinking, the removal of pathnames from the filesystem. A single system call, unlink(), handles this task:

```
#include <unistd.h>
int unlink (const char *pathname);
```

A successful call to unlink() deletes pathname from the filesystem and returns 0. If that name was the last reference to the file, the file is deleted from the filesystem. If, however, a process has the file open, the kernel will not delete the file from the filesystem until that process closes the file. Once no process has the file open, it is deleted.

If pathname refers to a symbolic link, the link, not the target, is destroyed. If pathname refers to another type of special file, such as a device, FIFO, or socket, the special file is removed from the filesystem, but processes that have the file open may continue to utilize it.

On error, unlink() returns −1 and sets errno to one of the following error codes:

EACCES
EFAULT
EIO
EISDIR
ELOOP
ENAMETOOLONG
ENOENT
ENOMEM
ENOTDIR
EPERM
EROFS

unlink() does not remove directories. For that, applications should use rmdir.

To ease the want on destruction of any type of file, the remove() function is provided:

```
#include <stdio.h>
int remove (const char *path);
```

A successful call to remove() deletes path from the filesystem and returns 0. If path is a file, remove() invokes unlink(); if path is a directory, remove() calls rmdir().

On error, remove() returns −1 and sets errno to any of the valid error codes set by unlink() and rmdir(), as applicable.

Lab Exercises:

1. Write a program to find the inode number of an existing file in a directory. Take the input as a filename and print the inode number of the file.
2. Write a program to print out the complete stat structure of a file.
3. Write a program to create a new hard link to an existing file and unlink the same. Accept the old path as input and print the newpath.
4. Write a program to create a new soft link to an existing file and unlink the same. Accept the old path as input and print the newpath.

Additional Exercises:

1. Write a program to find the inode number of all files in a directory. Take the input as a directory name and print the inode numbers of all the files in it.
2. Write a program to print the full stat structure of a directory.

LAB NO.: 5 **Date:**

IPC-1 : PIPE, FIFO

In this lab, the student will be able to:

- Gain knowledge as to how Interprocess Communication (IPC) happens between two processes
- Execute programs for IPC using the different methods of pipe and fifo

Inter-Process Communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange of data. IPC in Linux can be implemented by using a pipe, shared memory and message queue.

Pipe

- Pipes are unidirectional byte streams that connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors.

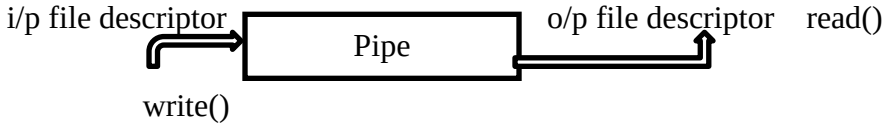


Figure 5.1. Working of pipe

- As shown in Figure 5.1 and 5.2 call to the pipe () function which returns an array of file descriptors fd[0] and fd [1]. fd [1] connects to the write end of the pipe, and fd[0] connects to the read end of the pipe. Anything can be written to the pipe, and read from the other end in the order it came in.
- A pipe is one-directional providing one-way flow of data and it is created by the pipe() system call.

  int pipe (int *filedes) ;

- An array of two file descriptors are returned- fd[0] which is open for reading, and fd[1] which is open for writing. It can be used only between parent and child processes.

PROTOTYPE: int pipe(int fd[2]);
RETURNS: 0 on success
-1 on error: errno = EMFILE (no free descriptors)
EMFILE (system file table is full)
EFAULT (fd array is not valid)

fd[0] is set up for reading, fd[1] is set up for writing. i.e., the first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

```
#include <stdlib.h>
#include <stdio.h>      /* for printf */
#include <string.h>     /* for strlen */

int main(int argc, char **argv)
{
        int n;
        int fd[2];
        char buf[1025];
        char *data = "hello... this is sample data";
        pipe(fd);
        write(fd[1], data, strlen(data));
        if ((n = read(fd[0], buf, 1024)) >= 0) {
                buf[n] = 0;       /* terminate the string */
                printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
        }
        else
                perror("read");
        exit(0);
}
```

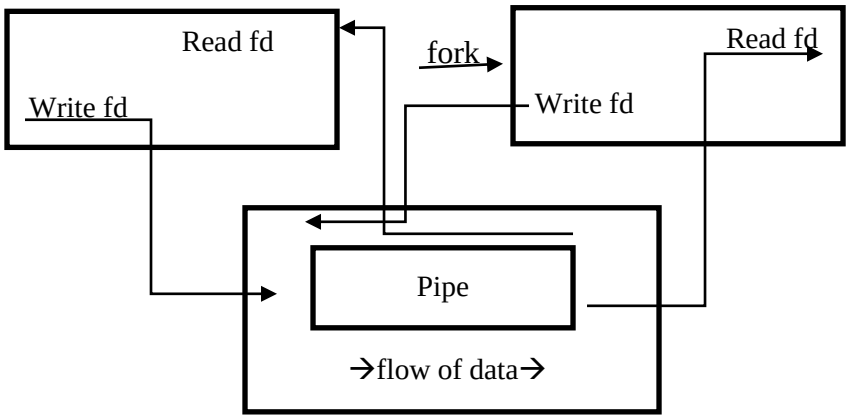


Fig. 5.2: Working of pipe in a single process which is immediately after fork()

- First, a process creates a pipe and then forks to create a copy of itself.
- The parent process closes the read end of the pipe.
- The child process closes the write end of the pipe.
- The fork system call creates a copy of the process that was executing.
- The process which executes the fork is called the parent process and the new process is which is created is called the child process.

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int pfd[2];
   pid_t cpid;
   char buf;
   assert(argc == 2);
   if (pipe(pfd) == -1) { perror("pipe");
      exit(EXIT_FAILURE); }
```

```
    cpid = fork();
    if (cpid == -1) { perror("fork");
    exit(EXIT_FAILURE); }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pfd[1]);          /* Close unused write end */
        while (read(pfd[0], &buf, 1) > 0)
         write(STDOUT_FILENO, &buf, 1);
         write(STDOUT_FILENO, "\n", 1);
        close(pfd[0]);
        exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pfd[0]);          /* Close unused read end */
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);          /* Reader will see EOF */
        wait(NULL);             /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Named Pipes: FIFOs

Pipes can share data between related processes, i.e. processes that have been started from a common ancestor process. We can use named pipe or FIFOs to overcome this. A named pipe is a special type of file that exists as a name in the file system but behaves like the unnamed pipes we have discussed already. We can create named pipes from the command line using

$ mkfifo filename

From inside a program, we can use

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

```
int mkfifo(const char *filename, mode_t mode);

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
    {
        int res = mkfifo("/tmp/my_fifo", 0777);
        if (res == 0) printf("FIFO created\n");
        exit(EXIT_SUCCESS);
    }
```

We can look for the pipe with

$ **ls -lF /tmp/my_fifo**

prwxr-xr-x 1 rick users 0 July 10 14:55 /tmp/my_fifo|

Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the ls command's -F option and also indicates a pipe. We can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system call.

Producer-Consumer Problem (PCP):

- The producer process produces information that is consumed by a consumer process. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. Two types of buffers can be used.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is fixed buffer size.
- For bounded-buffer PCP basic synchronization requirement is:
  - Producer should not write into a full buffer (i.e. producer must wait if the buffer is full)
  - Consumer should not read from an empty buffer (i.e. consumer must wait if the buffer is empty)
  - All data written by the producer must be read exactly once by the consumer

Following is a program for Producer-Consumer problem using named pipes.

//producer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_WRONLY;
        int bytes_sent = 0;
        char buffer[BUFFER_SIZE + 1];
        if (access(FIFO_NAME, F_OK) == -1) {
                res = mkfifo(FIFO_NAME, 0777);
                if (res != 0) {
                        fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
                        exit(EXIT_FAILURE);
                }
        }
        printf("Process %d opening FIFO O_WRONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                while(bytes_sent < TEN_MEG) {
                        res = write(pipe_fd, buffer, BUFFER_SIZE);
                        if (res == -1) {
                                fprintf(stderr, "Write error on pipe\n");
                                exit(EXIT_FAILURE);
                        }
                        bytes_sent += res;
```

```
                }
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
        }
        printf("Process %d finished\n", getpid());
        exit(EXIT_SUCCESS);
}

//consumer.c

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_RDONLY;
        char buffer[BUFFER_SIZE + 1];
        int bytes_read = 0;
        memset(buffer, '\0', sizeof(buffer));
        printf("Process %d opening FIFO O_RDONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                do {
                        res = read(pipe_fd, buffer, BUFFER_SIZE);
                        bytes_read += res;
```

```
            } while (res > 0);
            (void)close(pipe_fd);
        }
        else {
            exit(EXIT_FAILURE);
        }
        printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
        exit(EXIT_SUCCESS);
}
```

The Readers-Writers Problem:

- Concurrent processes share a file, record, or other resources
- Some may read-only (readers), some may both read and write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - concurrent reads and writes
  - multiple writes

Two Variations

- First Readers-Writers problem: No reader be kept waiting unless a writer has already obtained exclusive write permissions (Readers have high priority)
- Second Readers-Writers problem: If a writer is waiting/ready, no new readers may start reading (Writers have high priority)

Lab Exercises:

1. Write a producer and consumer program in C using the FIFO queue. The producer should write a set of 4 integers into the FIFO queue and the consumer should display the 4 integers.
2. Demonstrate creation, writing to, and reading from a pipe.
3. Write a C program to implement one side of FIFO.
4. Write two programs, producer.c implementing a producer and consumer.c implementing a consumer, that do the following:

   Your product will sit on a shelf: that is an integer - a count of the items "on the shelf". This integer may never drop below 0 or rise above 5.

   Your producer sets the value of the count to 5. It is the producer program's

responsibility to stock product on the shelf, but not overstocked. The producer may add one item to the shelf at a time, and must report to STDOUT every time another item is added as well as the current shelf count.

Your consumer will remove one item from the shelf at a time, provided the item count has not dropped below zero. The consumer will decrement the counter and report the new value to STDOUT. Have your consumer report each trip to the shelf, in which there are no items.

Additional Exercises:

1. Demonstrate creation of a process that writes through a pipe while the parent process reads from it.
2. Demonstrate second readers-writers problem using FIFO

LAB NO.: 6 **Date:**

IPC – 2 : MESSAGE QUEUE, SHARED MEMORY

Objectives:

In this lab, the student will be able to:

- Gain knowledge as to how IPC (Interprocess Communication) happens between two processes
- Execute programs for IPC using the different methods of message queues and shared memory

Message Queues

- It is an IPC facility. Message queues are similar to named pipes without the opening and closing of pipe. It provides an easy and efficient way of passing information or data between two unrelated processes.
- The advantages of message queues over named pipes are, it removes a few difficulties that exist during the synchronization, the opening, and closing of named pipes.
- A message queue is a linked list of messages stored within the kernel. A message queue is identified by a unique identifier. Every message has a positive long integer type field, a non-negative length, and the actual data bytes. The messages need not be fetched on FCFS basis. It could be based on the type field.

Creating a Message Queue

- To use a message queue, it has to be created first. The msgget() system call is used for that. This system call accepts two parameters - a queue key and flags.
- IPC_PRIVATE - use to create a private message queue. A positive integer - used to create or access a publicly accessible message queue.

The message queue function definitions are
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

msgget

We create and access a message queue using the msgget function:

int msgget(key_t key, int msgflg);

The program must provide a key-value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. The second parameter, msgflg, consists of nine permission flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd

The msgsnd function allows us to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function. When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */
}
```

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type. The fourth parameter, msgflg, controls what happens if either the current message queue is full or the system-wide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1. If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue. On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv

The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described above in the msgsnd function. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type. The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved. This sounds more complicated than it is in practice. If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n. The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive. On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl
The final message queue function is msgctl.

int msgctl(int msqid, int command, struct msqid_ds *buf);

The msqid_ds structure has at least the following members:

struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}

The first parameter, msqid, is the identifier returned from msgget. The second parameter, command, is the action to take. It can take three values:

Command Description

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.

0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in an msgsnd or msgrcv function, the send or receive function will fail.

Receiver program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
struct my_msg_st {
        long int my_msg_type;
        char some_text[BUFSIZ];
};

int main()
{
        int running = 1;
        int msgid;
        struct my_msg_st some_data;
        long int msg_to_receive = 0;
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, “msgget failed with error: %d\n”, errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
                        msg_to_receive, 0) == -1) {
                        fprintf(stderr, “msgrcv failed with error: %d\n”, errno);
                        exit(EXIT_FAILURE);
                }
                printf(“You wrote: %s”, some_data.some_text);
                if (strncmp(some_data.some_text, “end”, 3) == 0) {
                        running = 0;
                }
        }
        if (msgctl(msgid, IPC_RMID, 0) == -1) {
                fprintf(stderr, “msgctl(IPC_RMID) failed\n”);
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

Sender Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
        long int my_msg_type;
        char some_text[MAX_TEXT];
};

int main()
{
        int running = 1;
        struct my_msg_st some_data;
        int msgid;
        char buffer[BUFSIZ];
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                printf("Enter some text:");
                fgets(buffer, BUFSIZ, stdin);
                some_data.my_msg_type = 1;
                strcpy(some_data.some_text, buffer);
                if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
```

```
                fprintf(stderr, “msgsnd failed\n”);
                exit(EXIT_FAILURE);
            }
            if (strncmp(buffer, “end”, 3) == 0) {
                running = 0;
            }
        }
        exit(EXIT_SUCCESS);
}
```

Shared memory

Shared memory allows two or more processes to access the same logical memory. Shared memory is efficient in transferring data between two running processes. Shared memory is a special range of addresses that is created by one process and the Shared memory appears in the address space of that process. Other processes then attach the same shared memory segment into their own address space. All processes can then access the memory location as if the memory had been allocated just like malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

The functions for shared memory are,

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);

The include files sys/types.h and sys/ipc.h are normally also required before shm.h is included.

shmget

We create shared memory using the shmget function:

int shmget(key_t key, size_t size, int shmflg);

The argument key names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequently shared memory functions. There’s a special key-value, IPC_PRIVATE, that creates shared memory private to the process. The second parameter, size, specifies the amount of memory required in bytes.

The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory but only read by processes that other users have created. We can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.

If the shared memory is successfully created, shmget returns a non-negative integer, the shared memory identifier. On failure, it returns –1.

shmat

When we first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, we must attach it to the address space of a process. We do this with the shmat function:

void *shmat(int shm_id, const void *shm_addr, int shmflg);

The first parameter, shm_id, is the shared memory identifier returned from shmget. The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears. The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, as doing otherwise will make the application highly hardware-dependent. If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files. An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

shmdt

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

shmctl

int shmctl(int shm_id, int command, struct shmid_ds *buf);

The first parameter, shm_id, is the identifier returned from shmget. The second parameter, command, is the action to take. It can take three values:

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The shmid_ds structure has the following members:

```
struct shmid_ds {
        uid_t shm_perm.uid;
        uid_t shm_perm.gid;
        mode_t shm_perm.mode;
}
```

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory. On success, it returns 0, on failure returns –1.

We will write a pair of programs shm1.c and shm2.c. The first will create a shared memory segment and display any data that is written into it. The second will attach into an existing shared memory segment and enters data into the shared memory segment.

First, we create a common header file to describe the shared memory we wish to pass around. We call this shm_com.h.

```
#define TEXT_SZ 2048

struct shared_use_st {
        int written_by_you;
```

```
        char some_text[TEXT_SZ];
};

//shm1.c – Consumer process
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"

int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        int shmid;
        srand((unsigned int)getpid());
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, "shmget failed\n");
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, "shmat failed\n");
                exit(EXIT_FAILURE);
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        shared_stuff->written_by_you = 0;
        while(running) {
                if (shared_stuff->written_by_you) {
                printf("You wrote: %s", shared_stuff->some_text);
                sleep( rand() % 4 ); /* make the other process wait for us ! */
                shared_stuff->written_by_you = 0;
                if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                        running = 0;
                }
```

```
        }
        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, "shmdt failed\n");
                exit(EXIT_FAILURE);
        }
        if (shmctl(shmid, IPC_RMID, 0) == -1) {
                fprintf(stderr, "shmctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}

//shm2.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"
int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        char buffer[BUFSIZ];
        int shmid;
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, "shmget failed\n");
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, "shmat failed\n");
                exit(EXIT_FAILURE);
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        while(running) {
```

```
            while(shared_stuff->written_by_you == 1) {
                sleep(1);
                printf("waiting for client...\n");
            }
            printf("Enter some text:");
            fgets(buffer, BUFSIZ, stdin);
            strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
            shared_stuff->written_by_you = 1;
            if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
            }
        }
        if (shmdt(shared_memory) == -1) {
            fprintf(stderr, "shmdt failed\n");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

Lab Exercises:

1. Process A wants to send a number to Process B. Once received, Process B has to check whether the number is palindrome or not. Write a C program to implement this interprocess communication using a message queue.
2. Implement a parent process, which sends an English alphabet to a child process using shared memory. The child process responds with the next English alphabet to the parent. The parent displays the reply from the Child.
3. Write two programs named Interface and Display for the following problem.
   Interface program
   This program, when run, gives a prompt to the user as "Enter your message:"When the user enters his/her message string and presses Enter, the program writes the message into the shared memory, tells the Display to start processing,and then prompts the user again for another message.
   Display program
   The process waits until a new message becomes available on the sharedmemory. Then it reads the contents of the memory and prints it on the screen. It also clears the contents of the shared memory when it has read the message.
4. Write a two-player 3x3 tic-tac-toe console game using shared memory.

Additional Exercises:

1. Write a producer-consumer program in C in which producer writes a set of words into shared memory and then consumer reads the set of words from the shared memory. The shared memory need to be detached and deleted after use.
2. Write a program which creates a message queue and writes message into queue which contains number of users working on the machine along with observed time in hours and minutes. This is repeated for every 10 minutes. Write another program which reads this information form the queue and calculates on average in each hour how many users are working.

LAB NO.: 7 **Date:**

IPC – 3: DEADLOCK, LOCKING, SYNCHRONIZATION

In this lab, students will be able to:

- Synchronize various processes with the use of semaphore
- Understand how communication between two processes can take place with the help of a named pipe

For a multithreaded application spanning a single process or multiple processes to do useful work, some kind of common state must be shared between the threads. The degree of sharing that is necessary depends on the task. At one extreme, the only sharing necessary may be a single number that indicates the task to be performed. For example, a thread in a web server might be told only the port number to respond to. At the other extreme, a pool of threads might be passing information constantly among themselves to indicate what tasks are complete and what work is still to be completed.

Data Races

A data race occurs when multiple threads spanning a single process or multiple processes use the same data item and one or more of those threads are updating it.

Suppose there is a function update, which takes an integer pointer and updates the value of the content pointer by 4. If multiple threads call the function, then there is a possibility of a data race. If the current value of *a is 10, then when two threads simultaneously call update function, the final value of *a might be 14, instead of 18. To visualize this, we need to write the corresponding assembly language code for this function.

```
void update(int * a)
{
*a = *a + 4;
}
```

Another situation might be when one thread is running, but the other thread has been context switched off of the processor. Imagine that the first thread has loaded the value of the variable **a** and then gets context switched off the processor. When it eventually

runs again, the value of the variable a will have changed, and the final store of the restored thread will cause the value of the variable a to regress to an old value. The following code has a data race.

```
//race.c
#include <pthread.h>
int counter = 0;
void * func(void * params)
{
        counter++;
}
void main()
{
        pthread_t thread1, thread2;
        pthread_create(&thread1, 0, func, 0);
        pthread_create(&thread2, 0, func, 0);
        pthread_join(thread1, 0 );
        pthread_join(thread2, 0 );
}
```

Using tools to detect data races

We can compile the above code using gcc, and then use Helgrind tool which is part of Valgrind suite to identify the data race.

$ gcc –g race.c -lpthread

$ valgrind –tool=helgrind ./a.out

Avoiding Data Races

Although it is hard to identify data races, avoiding them can be very simple. The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.

Synchronization Primitives:

Mutex Locks:

A mutex lock is a mechanism that can be acquired by only one thread at a time. Other threads that attempt to acquire the same mutex must wait until it is released by the thread that currently has it.

Mutex locks need to be initialized to the appropriate state by a call to pthread_mutex_init() or for statically defined mutexes by assignment with the PTHREAD_MUTEX_INITIALIZER. The call to pthread_mutex_init() takes an optional parameter that points to attributes describing the type of mutex required. Initialization through static assignment uses default parameters, as does passing in a null pointer in the call to pthread_mutex_init().

Once a mutex is no longer needed, the resources it consumes can be freed with a call to pthread_mutex_destroy().

```
#include <pthread.h>

...

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;
pthread_mutex_init( &m2, 0 );
...
pthread_mutex_destroy( &m1 );
pthread_mutex_destroy( &m2 );
```

A thread can lock a mutex by calling pthread_mutex_lock(). Once it has finished with the mutex, the thread calls pthread_mutex_unlock(). If a thread calls pthread_mutex_lock() while another thread holds the mutex, the calling thread will wait, or *block*, until the other thread releases the mutex, allowing the calling thread to attempt to acquire the released mutex.

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
volatile int counter = 0;
void * count( void * param)
```

```
{
        for ( int i=0; i<100; i++)
        {
                pthread_mutex_lock(&mutex);
                counter++;
                printf("Count = %i\n", counter);
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        pthread_t thread1, thread2;
        pthread_mutex_init( &mutex, 0 );
        pthread_create( &thread1, 0, count, 0 );
        pthread_create( &thread2, 0, count, 0 );
        pthread_join( thread1, 0 );
        pthread_join( thread2, 0 );
        pthread_mutex_destroy( &mutex );
        return 0;
}
```

Semaphores:

A semaphore is a counting and signaling mechanism. One use for it is to allow threads access to a specified number of items. If there is a single item, then a semaphore is essentially the same as a mutex, but it is more commonly used in a situation where there are multiple items to be managed.

A semaphore is initialized with a call to sem_init(). This function takes three parameters. The first parameter is a pointer to the semaphore. The next is an integer to indicate whether the semaphore is shared between multiple processes or private to a single process. The final parameter is the value with which to initialize the semaphore. A semaphore created by a call to sem_init() is destroyed with a call to sem_destroy().

The code below initializes a semaphore with a count of 10. The middle parameter of the call to sem_init() is zero, and this makes the semaphore private to the process; passing the value one rather than zero would enable the semaphore to be shared between multiple processes.

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

```
int main()
{
        sem_t semaphore;
        sem_init( &semaphore, 0, 10 );
        ...
        sem_destroy( &semaphore );
}
```

The semaphore is used through a combination of two methods. The function sem_wait() will attempt to decrement the semaphore. If the semaphore is already zero, the calling thread will wait until the semaphore becomes nonzero and then return, having decremented the semaphore. The call to sem_post() will increment the semaphore. One more call, sem_getvalue(), will write the current value of the semaphore into an integer variable.

In the following program, an order is maintained in displaying Thread 1 and Thread 2. Try removing the semaphore and observe the output.

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

sem_t semaphore;

void *func1( void * param )
{
        printf( "Thread 1\n" );
        sem_post( &semaphore );
}
void *func2( void * param )
{
        sem_wait( &semaphore );
        printf( "Thread 2\n" );
}
int main()
{
        pthread_t threads[2];
```

```
        sem_init( &semaphore, 0, 1 );
        pthread_create( &threads[0], 0, func1, 0 );
        pthread_create( &threads[1], 0, func2, 0 );
        pthread_join( threads[0], 0 );
        pthread_join( threads[1], 0 );
        sem_destroy( &semaphore );
}
```

Solution to Producer-Consumer problem

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
int buf[5],f,r;
sem_t mutex,full,empty;
void *produce(void *arg)
{
   int i;
   for(i=0;i<10;i++)
   {
      sem_wait(&empty);
      sem_wait(&mutex);
      printf("produced item is %d\n",i);
      buf[(++r)%5]=i;
      sleep(1);
      sem_post(&mutex);
      sem_post(&full);
      printf("full %u\n",full);
   }
}
void *consume(void *arg)
{
      int item,i;
      for(i=0;i<10;i++)
      {
            sem_wait(&full);
```

```
            printf("full %u\n",full);
            sem_wait(&mutex);
            item=buf[(++f)%5];
            printf("consumed item is %d\n",item);
            sleep(1);
            sem_post(&mutex);
            sem_post(&empty);
        }
}
main()
{
   pthread_t tid1,tid2;
   sem_init(&mutex,0,1);
   sem_init(&full,0,1);
   sem_init(&empty,0,5);
   pthread_create(&tid1,NULL,produce,NULL);
   pthread_create(&tid2,NULL,consume,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}
```

Solution to First Readers-Writers Problem using semaphores:

- The reader processes share the following data structures:
  semaphore mutex , wrt;
  int readcount;
- The binary semaphores mutex and wrt are initialized to 1; readcount is initialized to 0;
- Semaphore wrt is common to both reader and writer process
  - wrt functions as a mutual exclusion for the writers
  - It is also used by the first or last reader that enters or exits the critical section
  - It is not used by readers who enter or exit while other readers are in their critical section
- The readcount variable keeps track of how many processes are currently reading the object
- The mutex semaphore is used to ensure mutual exclusion when readcount is updated

The structure of a writer process

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

The structure of a reader process

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

The Dining Philosophers Problem:

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - five chopsticks total
- A philosopher needs two chopsticks to eat
  - philosophers must share chopsticks to eat
- No interaction occurs while thinking

The situation of the dining philosophers is shown in Fig. 7.1

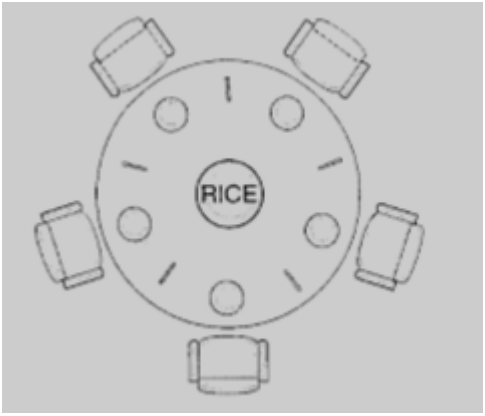


Figure 7.1 Dining Philosophers Problem

Lab Exercises:

1. Modify the above Producer-Consumer program so that, a producer can produce at the most 10 items more than what the consumer has consumed.
2. Write a C program for the first readers-writers problem using semaphores.
3. Write a Code to access a shared resource which causes deadlock using improper use of semaphore.
4. Write a program using semaphore to demonstrate the working of sleeping barber problem.

Additional Exercises:

1. Write a C program for Dining-Philosophers problem using monitors.
2. Demonstrate the working of counting semaphore.

LAB NO.: 8 **Date:**

PROGRAMS ON THREADS

Objectives:

In this lab, the student will be able to:

- Understand the concepts of multithreading
- Grasp the execution of the different processes concerning multithreading

A process will start with a single thread which is called the main thread or master thread. Calling pthread_create() creates a new thread. It takes the following parameters.

- A pointer to a pthread_t structure. The call will return the handle to the thread in this structure.
- A pointer to a pthread attributes structure, which can be a null pointer if the default attributes are to be used. The details of this structure will be discussed later.
- The address of the routine to be executed.
- A value or pointer to be passed into the new thread as a parameter.

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

void* thread_code( void * param )
{
        printf( "In thread code\n" );
}
int main()
{
        pthread_t thread;
        pthread_create(&thread, 0, &thread_code, 0 );
        printf("In main thread\n" );
}
```

In this example, the main thread will create a second thread to execute the routine thread_code(), which will print one message while the main thread prints another. The call to create the thread has a value of zero for the attributes, which gives the thread default attributes. The call also passes the address of a pthread_t variable for the function to store a handle to the thread. The return value from the thread_create() call is zero if the call is successful; otherwise, it returns an error condition.

Thread termination :

Child threads terminate when they complete the routine they were assigned to run. In the above example, child thread will terminate when it completes the routine thread_code().

The value returned by the routine executed by the child thread can be made available to the main thread when the main thread calls the routine pthread_join().

The pthread_join() call takes two parameters. The first parameter is the handle of the thread that is to be waited for. The second parameter is either zero or the address of a pointer to a void, which will hold the value returned by the child thread.

The resources consumed by the thread will be recycled when the main thread calls pthread_join(). If the thread has not yet terminated, this call will wait until the thread terminates and then free the assigned resources.

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

void* thread_code( void * param )
{
        printf( "In thread code\n" );
}
int main()
{
        pthread_t thread;
        pthread_create( &thread, 0, &thread_code, 0 );
        printf( "In main thread\n" );
        pthread_join( thread, 0 );
}
```

Another way a thread can terminate is to call the routine pthread_exit(), which takes a single parameter—either zero or a pointer—to void. This routine does not return and instead terminates the thread. The parameter passed into the pthread_exit() call is returned to the main thread through the pthread_join(). The child threads do not need to explicitly call pthread_exit() because it is implicitly called when the thread exits.

Passing Data to and from Child Threads

In many cases, it is important to pass data into the child thread and have the child thread return status information when it completes. To pass data into a child thread, it should be cast as a pointer to void and then passed as a parameter to pthread_create().

```
for ( int i=0; i<10; i++ )
pthread_create( &thread, 0, &thread_code, (void *)i );
```

Following is a program where the main thread passes a value to the Pthread and the thread returns a value to the main thread.

```
#include <pthread.h>
#include <stdio.h>
void* child_thread( void * param )
{
        int id = (int)param;
        printf( "Start thread %i\n", id );
        return (void *)id;
}

int main()
{
        pthread_t thread[10];
        int return_value[10];
        for ( int i=0; i<10; i++ )
        {
                pthread_create( &thread[i], 0, &child_thread, (void*)i );
        }
        for ( int i=0; i<10; i++ )
        {
                pthread_join( thread[i], (void**)&return_value[i] );
                printf( "End thread %i\n", return_value[i] );
        }
}
```

Setting the Attributes for Pthreads

The attributes for a thread are set when the thread is created. To set the initial thread attributes, first, create a thread attributes structure, and then set the appropriate attributes in that structure, before passing the structure into the pthread_create() call.

```
#include <pthread.h>
...
int main()
{
        pthread_t thread;
        pthread_attr_t attributes;
        pthread_attr_init( &attributes );
        pthread_create( &thread, &attributes, child_routine, 0 );
}
```

Lab Exercises:

1. Write a multithreaded program that generates the Fibonacci series. The program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program then will create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution the parent will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish.
2. Write a multithreaded program that calculates the summation of non-negative integers in a separate thread and passes the result to the main thread.
3. Write a multithreaded program for generating prime numbers from a given starting number to the given ending number.
4. Write a multithreaded program that performs the sum of even numbers and odd numbers in an input array. Create a separate thread to perform the sum of even numbers and odd numbers. The parent thread has to wait until both the threads are done.

Additional Exercises:

1. Write a multithreaded program for matrix multiplication.
2. Write a multithreaded program for finding row sum and column sum

LAB NO.: 9 **Date:**

MEMORY AND DATA MANAGEMENT

Objectives:

In this lab, the student will be able to:

- Understand how to use dynamic memory allocation
- Learn the working of Demand Paged Virtual Memory

Simple Memory Allocation

You allocate memory using the malloc call in the standard C library:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Notice that Linux (following the X/Open specification) differs from some UNIX implementations by not requiring a special malloc.h include file. Note also that the size parameter that specifies the number of bytes to allocate isn't a simple int, although it's usually an unsigned integer type.

You can allocate a great deal of memory on most Linux systems. Let's start with a very simple program, but one that would defeat old MS-DOS-based programs, because they cannot access memory outside the base 640K memory map of PCs.

Sample Program (memory1.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
int main()
{
char *some_memory;
int megabyte = A_MEGABYTE;
int exit_code = EXIT_FAILURE;
some_memory = (char *)malloc(megabyte);
if (some_memory != NULL) {
sprintf(some_memory, "Hello World\n");
printf("%s", some_memory);
exit_code = EXIT_SUCCESS;
}
exit(exit_code);
}
```

When you run this program, it gives the following output:

```
$ ./memory1
```

Hello World

How It Works

This program asks the malloc library to give it a pointer to a megabyte of memory. You check to ensure that malloc was successful and then use some of the memory to show that it exists. When you run the program, you should see Hello World printed out, showing that malloc did indeed return the megabyte of usable memory. We don't check that all of the megabytes are present; we have to put some trust in the malloc code! Notice that because malloc returns a void * pointer, you cast the result to the char * that you need.

The malloc function is guaranteed to return memory that is aligned so that it can be cast to a pointer of any type. The simple reason is that most current Linux systems use 32-bit integers and 32-bit pointers for pointing to memory, which allows you to specify up to 4 gigabytes. This ability to address directly with a 32-bit pointer, without needing segment registers or other tricks, is termed a flat 32-bit memory model.

Allocating Lots of Memory

Now that you've seen Linux exceed the limitations of the MS-DOS memory model, let's give it a more difficult problem. The next program asks to allocate somewhat more memory than is physically present in the machine, so you might expect malloc to start failing somewhere a little short of the actual amount of memory present, because the kernel and all the other running processes are using some memory.

Sample Program (memory2.c) :

Asking for All Physical Memory

With memory2.c , we're going to ask for more than the machine's physical memory. You should adjust the define PHY_MEM_MEGS depending on your physical machine:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
#define PHY_MEM_MEGS
1024 /* Adjust this number as required */
int main()
{
char *some_memory;
size_t size_to_allocate = A_MEGABYTE;
int megs_obtained = 0;
while (megs_obtained < (PHY_MEM_MEGS * 2)) {
some_memory = (char *)malloc(size_to_allocate);
if (some_memory != NULL) {
```

```
megs_obtained++;
sprintf(some_memory, “Hello World”);
printf(“%s - now allocated %d Megabytes\n”, some_memory, megs_obtained);
}
else {
exit(EXIT_FAILURE);
}
}
exit(EXIT_SUCCESS);
}
```

The output, somewhat abbreviated, is as follows:

```
$ ./memory2
Hello World
Hello World
...
Hello World
Hello World
- now allocated 1 Megabytes
- now allocated 2 Megabytes
- now allocated 2047 Megabytes
- now allocated 2048 Megabytes
```

How It Works

The program is very similar to the previous example. It simply loops, asking for more and more memory, until it has allocated twice the amount of memory you said your machine had when you adjusted the define PHY_MEM_MEGS . The surprise is that it works at all because we appear to have created a program that uses every single byte of physical memory on the author’s machine. Notice that we use the size_t type for our call to malloc.

The other interesting feature is that, at least on this machine, it ran the program in the blink of an eye. So not only have we used up all the memory, but we’ve done it very quickly indeed.

Let’s investigate further and see just how much memory we can allocate on this machine with memory3.c . Since it’s now clear that Linux can do some very clever things with memory requests, we’ll allocate memory just 1K at a time and write to each block that we obtain.

Demand Paged Virtual Memory

Sample Program (memory3.c) :
Available Memory

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define ONE_K (1024)
int main()
{
char *some_memory;
int size_to_allocate = ONE_K;
int megs_obtained = 0;
int ks_obtained = 0;
while (1) {
for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++) {
some_memory = (char *)malloc(size_to_allocate);
if (some_memory == NULL) exit(EXIT_FAILURE);
sprintf(some_memory, “Hello World”);
}
megs_obtained++;
printf(“Now allocated %d Megabytes\n”, megs_obtained);
}
exit(EXIT_SUCCESS);
}
```

Output:

```
$ ./memory3
Now allocated 1 Megabytes
...
Now allocated 1535 Megabytes
Now allocated 1536 Megabytes
Out of Memory: Killed process 2365
Killed
```

and then the program ends. It also takes quite a few seconds to run, and slows down significantly around the same number as the physical memory in the machine, and exercises the hard disk quite noticeably. However, the program has allocated, and accessed, more memory than this author physically has in his machine at the time of writing. Finally, the system protects itself from this rather aggressive program and kills it. On some systems, it may simply exit quietly when malloc fails.

How It Works

The application's allocated memory is managed by the Linux kernel. Each time the program asks for memory or tries to read or write to memory that it has allocated, the Linux kernel takes charge and decides how to handle the request. Initially, the kernel was simply able to use free physical memory to satisfy the application's request for memory, but once physical memory was full, it started using what's called swap space. On Linux, this is a separate disk area allocated when the system was installed. If you're familiar with Windows, the Linux swap space acts a little like the hidden Windows swap file. However, unlike Windows, there are no local heap, global heap, or discardable memory segments to worry about in code — the Linux kernel does all the management for you. The kernel moves data and program code between physical memory and the swap space so that each time you read or write memory, the data always appears to have been in physical memory, wherever it was located before you attempted to access it.

- In more technical terms, Linux implements a **demand paged virtual memory** system. All memory seen by user programs is virtual; that is, it doesn't exist at the physical address the program uses. Linux divides all memory into pages, commonly 4,096 bytes per page. When a program tries to access memory, a virtual-to-physical translation is made, although how this is implemented and the time it takes depend on the particular hardware you're using. When the access is to memory that isn't physically resident, a page fault results and control is passed to the kernel. The Linux kernel checks the address being accessed and, if it's a legal address for that program, determines which page of physical memory to make available. It then either allocates it, if it has never been written before or if it has been stored on the disk in the swap space, reads the memory page containing the data into physical memory (possibly moving an existing page out to disk). Then, after mapping the virtual memory address to match the physical address, it allows the user program to continue. Linux applications don't need to worry about this activity because the implementation is all hidden in the kernel. Eventually, when the application exhausts both the physical memory and the swap space, or when the maximum stack size is exceeded, the kernel finally refuses the request for further memory and may pre-emptively terminate the program.

This "killing the process" behavior is different from early versions of Linux and many other flavors of UNIX, where malloc simply fails. It's termed the "out of memory (OOM) killer," and although it may seem rather drastic, it is a good compromise between letting processes allocate memory rapidly and efficiently and having the Linux kernel protect itself from a total lack of resources, which is a serious issue.

So what does this mean to the application programmer? It's all good news. Linux is very good at managing memory and will allow applications to use very large amounts of memory and even very large single blocks of memory. However, you must remember that allocating two blocks of memory won't result in a single continuously addressable block of memory. What you get is what you ask for: two separate blocks of memory.

Does this limitless supply of memory, followed by the preemptive killing of the process, mean that there's no point in checking the return from malloc? No. One of the most common problems in C programs using dynamically allocated memory is writing beyond the end of an allocated block. When this happens, the program may not terminate immediately, but you have probably overwritten some data used internally by the malloc library routines.

Usually, the result is that future calls to malloc may fail, not because there's no memory to allocate, but because the memory structures have been corrupted. These problems can be quite difficult to track down, and in programs the sooner the error is detected, the better the chances of tracking down the cause.

Abusing Memory

Suppose you try to do "bad" things with memory. In this exercise, you allocate some memory and then attempt to write past the end, in memory4.c .

Sample Program (memory4.c) :

```
#include <stdlib.h>
#define ONE_K (1024)
int main()
{
char *some_memory;
char *scan_ptr;
some_memory = (char *)malloc(ONE_K);
if (some_memory == NULL) exit(EXIT_FAILURE);
scan_ptr = some_memory;
while(1) {
*scan_ptr = '\0';
scan_ptr++;
}
exit(EXIT_SUCCESS);
}
```

The output is simply

```
$ /memory4
Segmentation fault
```

How It Works
The Linux memory management system has protected the rest of the system from this abuse of memory. To ensure that one badly behaved program (this one) can't damage any other programs, Linux has terminated it. Each running program on a Linux system sees its memory map, which is different from every other program's. Only the operating system knows how physical memory is arranged, and not only manages it for user programs but also protects user programs from each other.

The Null Pointer
Unlike MS-DOS, but more like newer flavors of Windows, modern Linux systems are very protective about writing or reading from the address referred to by a null pointer, although the actual behavior is implementation-specific.
Sample Program (memory5a.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
char *some_memory = (char *)0;
printf("A read from null %s\n", some_memory);
sprintf(some_memory, "A write to null\n");
exit(EXIT_SUCCESS);
}
```

The output is
```
$ ./memory5a
A read from null (null)
Segmentation fault
```

How It Works
The first printf attempts to print out a string obtained from a null pointer; then the sprintf attempts to write to a null pointer. In this case, Linux (in the guise of the GNU "C" library) has been forgiving about the read and has simply provided a "magic" string containing the characters (n u l l) \0 . It hasn't been so forgiving about the write and has terminated the program. This can sometimes help track down program bugs.

If you try this again but this time don't use the GNU "C" library, you'll discover that reading from location zero is not permitted. Here is memory5b.c :

Sample Program (memory5b.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
char z = *(const char *)0;
printf("I read from location zero\n");
exit(EXIT_SUCCESS);
}
```

The output is

```
$ ./memory5b
Segmentation fault
```

This time you attempt to read directly from location zero. There is no GNU libc library between you and the kernel now, and the program is terminated. Note that some versions of UNIX do permit reading from location zero, but Linux doesn't.

Freeing Memory

Up to now, we've been simply allocating memory and then hoping that when the program ends, the memory we've used hasn't been lost. Fortunately, the Linux memory management system is quite capable of reliably ensuring that memory is returned to the system when a program ends. However, most programs don't simply want to allocate some memory, use it for a short period, and then exit. A much more common use is dynamically using memory as required.

Programs that use memory on a dynamic basis should always release unused memory back to the malloc memory manager using the free call. This enables separate blocks to be remerged and enables the malloc library to look after memory, rather than have the application manage it. If a running program (process) uses and then frees memory, that free memory remains allocated to the process. Behind the scenes, Linux is managing the blocks of memory the programmer is using as a set of physical "pages, " usually 4K bytes each, in memory. However, if a page of memory is not being used, then the Linux memory manager will be able to move it from physical memory to swap space (termed paging), where it has little impact on the use of resources. If the program tries to access data inside the memory page that has been moved to swap space, then Linux will very briefly suspend the program, move the memory page back from swap space into physical memory again, and then allow the program to continue, just as though the data had been in memory all along.

#include <stdlib.h>
void free(void *ptr_to memory);

A call to free should be made only with a pointer to memory allocated by a call to malloc, calloc, or realloc.

Sample Program (memory6.c) :

```
#include <stdlib.h>
#include <stdio.h>
#define ONE_K (1024)
int main()
{
char *some_memory;
int exit_code = EXIT_FAILURE;
some_memory = (char *)malloc(ONE_K);
if (some_memory != NULL) {
free(some_memory);
printf("Memory allocated and freed again\n");
exit_code = EXIT_SUCCESS;
}
exit(exit_code);
}
```

The output is

```
$ ./memory6
Memory allocated and freed again
```

How It Works

This program simply shows how to call free with a pointer to some previously allocated memory.

Remember that once you've called free on a block of memory, it no longer belongs to the process. It's not being managed by the malloc library. **Never try to read or write memory after calling free on it.**

Other Memory Allocation Functions

Two other memory allocation functions are not used as often as malloc and free : calloc and realloc .

The prototypes are

```
#include <stdlib.h>
void *calloc(size_t number_of_elements, size_t element_size);
void *realloc(void *existing_memory, size_t new_size);
```

Although calloc allocates memory that can be freed with free , it has somewhat different parameters from malloc : It allocates memory for an array of structures and requires the number of elements and the size of each element as its parameters. The allocated memory is filled with zeros; and if calloc is successful, a pointer to the first element is returned. Like malloc , subsequent calls are not guaranteed to return contiguous space, so you can't enlarge an array created by calloc by simply calling calloc again and expecting the second call to return memory appended to that returned by the first call.

The realloc function changes the size of a block of memory that has been previously allocated. It's passed a pointer to some memory previously allocated by malloc , calloc , or realloc and resizes it up or down as requested. The realloc function may have to move data around to achieve this, so it's important to ensure that once the memory has been realloced, you always use the new pointer and never try to access the memory using pointers set up before realloc was called.

Another problem to watch out for is that realloc returns a null pointer if it has been unable to resize the memory. This means that in some applications you should avoid writing code like this:

```
my_ptr = malloc(BLOCK_SIZE);
....
my_ptr = realloc(my_ptr, BLOCK_SIZE * 10);
```

If realloc fails, then it returns a null pointer; my_ptr will point to null, and the original memory allocated with malloc can no longer be accessed via my_ptr . It may, therefore, be to your advantage to request the new memory first with malloc and then copy data from the old block to the new block using memcpy before freeing the old block. On error, this would allow the application to retain access to the data stored in the original block of memory, perhaps while arranging a clean termination of the program.

Lab Exercises

1. If you wish to implement Best Fit, First Fit, Next Fit, or Worst Fit memory allocation policy, it is probably best to do this by describing the memory as a structure in a linked list:

```
struct mab {
   int offset;
   int size;
   int allocated;
   struct mab * next;
   struct mab * prev;
};
```

typedef struct mab Mab;
typedef Mab * MabPtr;
Either way, the following set of prototypes give a guide as to the functionality you will need to provide:
MabPtr memChk(MabPtr m, int size); // check if memory available
MabPtr memAlloc(MabPtr m, int size); // allocate a memory block
MabPtr memFree(MabPtr m); // free memory block

MabPtr memMerge(MabPtr m); // merge two memory blocks
MabPtr memSplit(MabPtr m, int size); // split a memory block

2. Write a C program using Malloc for implementing Multilevel feedback queue using three queues with each of them working with different scheduling policies
3. We have five segments numbered 0 through 4. The segments are stored in physical memory as shown in the following Fig 10.3. Write a C program to create segment table. Write methods for converting logical address to physical address. Compute the physical address for the following.
(i) 53 byte of segment 2 (ii) 852 byte of segment 3 (iii) 1222 byte of segment 0

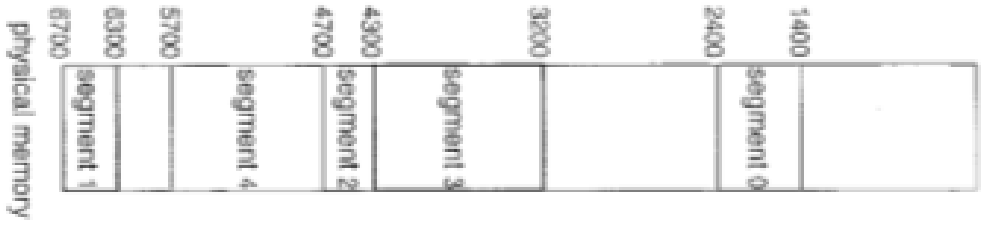


Figure 9.1: Physical memory

2. Write a C program to simulate LRU approximation page replacement using second chance algorithm. Find the total number of page faults and hit ratio for the algorithm.

Additional Exercises:

1. Implement the same concept for the Buddy system.
2. How do you resize and release memory using realloc ?

LAB NO.: 10 **Date:**

DEADLOCK AND DISK MANAGEMENT

Objectives:
In this lab, students will be able to

- Understand the problem of deadlock and ways to manage it
- Understand find the details of underlying operating systems disk space and file system information

DEADLOCK MANAGEMENT

The deadlock problem:
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

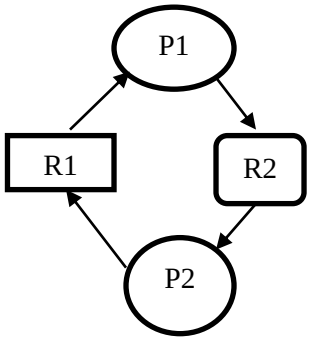


Figure 10.1: Deadlock Situation

Above Fig. 10.1 shows, a situation of deadlock where Process P1 Waiting for resource R2, which is held with process P2 and in the meantime, Process P2 is waiting for resource R1, which is held with process P1. Neither P1 nor P2 can proceed their execution until their needed resources are fulfilled forming a cyclic wait. It is the deadlock situation among processes as both are not progressed. In a single instance of resource type, a cyclic wait is always a deadlock.

Consider Figure 10.2 below, the situation with 4 processes P1, P2, P3 and P4 and 2 resources R1 and R2 both are of two instances. Here, there is no deadlock even though the cycle exists between processes P1 and P3. Once P2 finishes its job, 1

instance of resource will be available which can be accessed by process P1, which turns request edge to assignment edge, thereby removing cyclic-wait. So, in multiple instances of resource type, the cyclic-wait need not be deadlock.

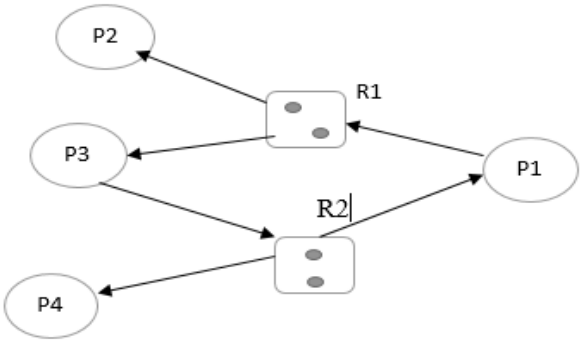


Figure 10.2: Cyclic-wait but no deadlock

Methods for Handling Deadlocks:

(i) Deadlock Avoidance:

The deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State:

System is in safe state if there exists a safe sequence of all processes. **Sequence** of processes $<P_1, P_2, \ldots, P_n>$ is **safe** if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

If a system is in safe state $\Rightarrow$ no deadlocks.

If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

DISK MANAGEMENT

System calls related to disk

df command

The 'df' (Disk Free) command in an inbuilt utility to find the available and the disk usage space on Linux servers/storage.

The following table provides an overview of the options of df command in Linux.

Options	Description
-a	-all: includes real files and virtual files like pseudo,pro, sysfs. lxc
-h	It prints the sizes in the human-readable format in power of 1024 (eg: 1K 1M 1G)
-H	--si : likewise '-h' but here in power of 1000
-i	-- inodes: correspondence the inode details
-k	--block-size=1K display the disk space
-l	--local display local file systems only
-m	--megabytes display the disk space
-t	--type=TYPE To filter a particular file system type
-T'	--print-type List the file system types
-x	--exclude-type=TYPE To exclude a particular file system type

1. **How to check the details of disk space used in each file system?**

```
# df
```

```
Output:

Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda2      164962420 142892148  15301196  91% /
udev               10240         0     10240   0% /dev
tmpfs            3291620    329084   2962536  10% /run
tmpfs            8229048         0   8229048   0% /dev/shm
tmpfs               5120         0      5120   0% /run/lock
tmpfs            8229048         0   8229048   0% /sys/fs/cgroup
/dev/sda1          97167     76552     15598  84% /boot
tmpfs            1645812         0   1645812   0% /run/user/301703
tmpfs            1645812         0   1645812   0% /run/user/301677
tmpfs            1645812         0   1645812   0% /run/user/301483
```

Note: Using 'df' command without any option/parameter will display all the partitions and the usage information of the disk space. The result of the above command contains 6 columns which are explained here below:

```
Filesystem      -->    Mount Point Name
1K-blocks       -->    Available total space counted in 1kB (1000 bytes)
Used            -->    Used block size
Available       -->    Free blocks size
Use%            -->    Usage on percentage-wise
Mounted on      -->    Show the path of the mounted point

# df -k
```

Note: Even using the '-k' option also provides the same output as the default 'df' command. Both outputs provide the same data usage of file systems in block size which is measured in 1024 bytes.

2. How to sum up the total of the disk space usage?

```
# df -h --total
```

```
Output:

Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2       158G  137G   15G  91% /
udev             10M     0   10M   0% /dev
tmpfs           3.2G  322M  2.9G  11% /run
tmpfs           7.9G     0  7.9G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           7.9G     0  7.9G   0% /sys/fs/cgroup
/dev/sda1        95M   75M   16M  84% /boot
tmpfs           1.6G     0  1.6G   0% /run/user/301703
tmpfs           1.6G     0  1.6G   0% /run/user/301483
tmpfs           1.6G     0  1.6G   0% /run/user/301613
tmpfs           1.6G     0  1.6G   0% /run/user/301677
total           183G  137G   40G  78% -
```

Note: using '--total' along with '-h' will sum up the total disk usage of all the file systems.

3. How to list the Inodes information of all file systems?

```
# df -i
```

```
 Output:

Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda2      10240000 2491518 7748482   25% /
udev            2054985     305 2054680    1% /dev
tmpfs           2057262     767 2056495    1% /run
tmpfs           2057262       1 2057261    1% /dev/shm
tmpfs           2057262      11 2057251    1% /run/lock
tmpfs           2057262      13 2057249    1% /sys/fs/cgroup
/dev/sda1         25168     336   24832    2% /boot
tmpfs           2057262       4 2057258    1% /run/user/301703
tmpfs           2057262       4 2057258    1% /run/user/301483
tmpfs           2057262       4 2057258    1% /run/user/301613
tmpfs           2057262       4 2057258    1% /run/user/301677
```

Note: Using '-i' will list the information about the Inodes of all the filesystem.

Disk scheduling algorithms for the disk structure as shown in the Figure 10.1 are used by operating systems to determine the order in which read and write operations are performed on a disk. The main goal of these algorithms is to minimize the disk head movements and optimize the overall disk performance.

Here are some common disk scheduling algorithms:

1. First-Come, First-Served (FCFS): In this algorithm, the disk requests are executed in the order they arrive. The disk head moves from its current position to the requested track, serving the requests sequentially. FCFS is simple but can result in poor performance due to the phenomenon called the "elevator effect."

2. Shortest Seek Time First (SSTF): This algorithm selects the request that requires the least disk head movement from the current position. It minimizes the seek time by serving the closest request first. SSTF provides better performance compared to FCFS, but it may cause starvation for requests located farther from the current position.

3. SCAN: Also known as the elevator algorithm, SCAN moves the disk head in one direction, serving requests along the way until it reaches the end of the disk. Then, it changes direction and serves requests in the opposite direction. SCAN provides a fair servicing order for all requests but may cause delays for requests located at the extremes.

4. C-SCAN: Similar to SCAN, C-SCAN moves the disk head in one direction, serving requests until the end of the disk is reached. However, instead of changing direction, it immediately jumps to the opposite end and starts servicing requests from there. This algorithm avoids the delays caused by SCAN for requests located at the extremes.

5. LOOK: LOOK is an improvement over SCAN. Instead of moving to the end of the disk, LOOK changes direction as soon as there are no pending requests in the current direction. This reduces the head movement and improves the average seek time.

6. C-LOOK: C-LOOK combines the advantages of C-SCAN and LOOK. It moves the disk head in one direction, serving requests until the last request in that direction. Then, it jumps to the opposite end and starts servicing requests from there without reversing direction. C-LOOK reduces head movement and provides better performance than both SCAN and LOOK.

The First-Come, First-Served (FCFS) algorithm is one of the simplest disk scheduling algorithms. It operates on the principle that requests are serviced in the order they arrive. In the context of disk scheduling, FCFS refers to the order in which read and write requests are executed on the disk.

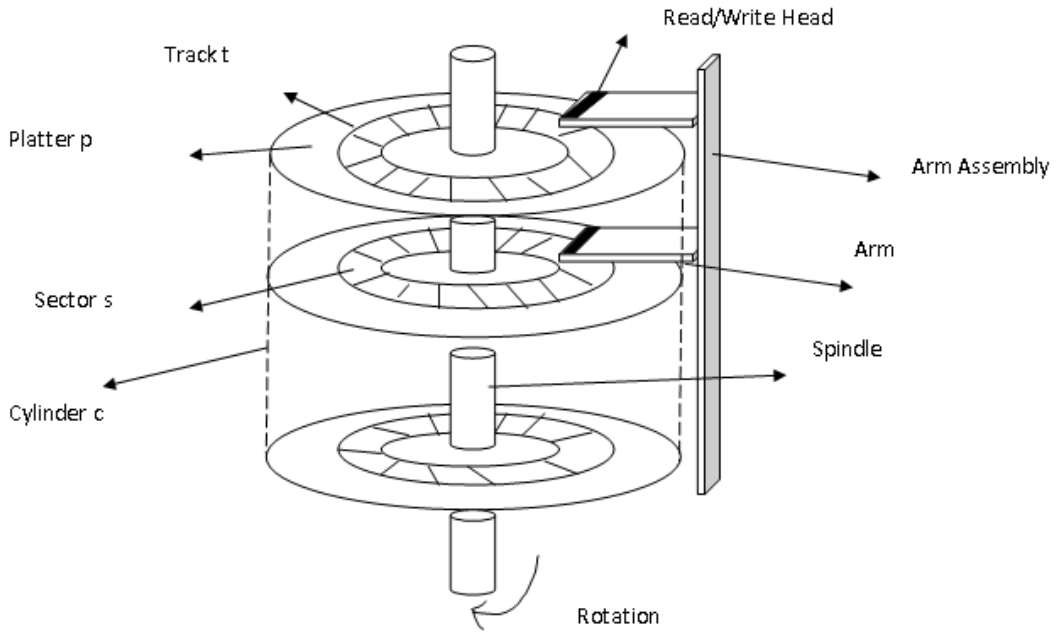


Figure 10.3: Disk structure

Sample Program:

Simulating FCFS Disk Scheduling algorithm

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int queue[100],n,head,i,j,k,seek=0,diff;
        float avg;
        // clrscr();
        printf("*** FCFS Disk Scheduling Algorithm ***\n");
        printf("Enter the size of Queue\t");
        scanf("%d",&n);
        printf("Enter the Queue\t");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&queue[i]);
        }
```

```
        printf("Enter the initial head position\t");
        scanf("%d",&head);
        queue[0]=head;
        printf("\n");
        for(j=0;j<=n-1;j++)
        {
                diff=abs(queue[j+1]-queue[j]);
                seek+=diff;
                printf("Move from %d to %d with Seek
%d\n",queue[j],queue[j+1],diff);
        }
        printf("\nTotal Seek Time is %d\t",seek);
        avg=seek/(float)n;
        printf("\nAverage Seek Time is %f\t",avg);
        getch();
}
```

Output

```
*** FCFS Disk Scheduling Algorithm ***
Enter the size of Queue 8
Enter the Queue 98 183 37 122 14 124 65 67
Enter the initial head position 53

Move from 53 to 98 with Seek 45
Move from 98 to 183 with Seek 85
Move from 183 to 37 with Seek 146
Move from 37 to 122 with Seek 85
Move from 122 to 14 with Seek 108
Move from 14 to 124 with Seek 110
Move from 124 to 65 with Seek 59
Move from 65 to 67 with Seek 2

Total Seek Time is 640
Average Seek Time is 80.000000
```

Lab Exercises:

1. Consider the following snapshot of the system. Write C program to implement Banker's algorithm for deadlock avoidance. The program has to accept all inputs from the user. Assume the total number of instances of A,B and C are 10,5 and 7 respectively.

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

(a) What is the content of the matrix *Need*?
(b) Is the system in a safe state?
(c) If a request from process P1 arrives for (1, 0, 2), can the request be granted immediately? Display the updated Allocation, Need and Available matrices.
(d) If a request from process P4 arrives for (3, 3, 0), can the request be granted immediately?
(e) If a request from process P0 arrives for (0, 2, 0), can the request be granted immediately?

2. Write a multithreaded program that implements the banker's algorithm. Create *n* threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using **pthreads**. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks, which are available in the pthreads libraries.
3. Simulate implementation of Disk Scheduling Algorithms: FCFS, SSTF using a structure DSA. An DSA contains the request ID, arrival timestamp, cylinder, address, and the ID of the process that posted the request.

```
struct DSA {
int request_id;
Int arrival_time_stamp;
Int cylinder;
Int address;
int process_id;
}
```

4. A file system uses contiguous allocation of disk space to files. A few blocks on the disk are reserved as spare blocks. If some disk blocks is found to be bad, the file system allocates a spare disk block to it and notes the address of the bad block and its allocated spare block in a “bad blocks table”. This table is consulted while accessing the disk block. Simulate the same.

Additional Exercises:

1. Consider the following snapshot of the system. Write C program to implement deadlock detection algorithm.
   (a) Is the system in a safe state?
   (b) Suppose that process P2 make one additional request for instance of type C, can the system still be in a safe state?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

2. Display the list of devices connected to your system including the physical names and its instance number.

LAB NO.: 11 and 12 **Date:**

MINI PROJECT

Objectives:
In this lab, the student will be able to:

- To carry out mini-projects in groups

Guidelines: Students need to submit the final report and demonstrate the application developed as a part of the mini-project. Students can choose any topic related to the Operating System.

Final Report Format:

- **Title, Team members**
- **Abstract**
- **Problem Statement and its description**
- **Algorithm**
- **Methodology**
- **Implementation**
- **References**

Evaluation Criteria:

S.N	Topic	Marks
1	Synopsis, Abstract, Problem Statement	2
2	Final Report – Implementation with Results	3
3	Implementation	3
4	Demo and Viva	2

REFERENCES:

1. Operating Systems - Internals and Design Principles", William Stallings, Prentice Hall, 5th Edition, 2004.
2. Maurice Bach, The Design of the Unix Operating System, PHI, 1986.
3. Beginning Linux Programming, 4th Edition, Neil Mathew and Richard Stores
4. Graham Glass and King Abels, Unix for Programmers and Users – *A complete guide*, PHI, 1993.
5. Sumitabha Das, Unix Concepts and Applications, McGraw Hill, 4th Edition, 2015.
6. Neil Matthew and Richard Stones, Beginning Linux Programming, 3rd Edition, Wiley, 1999.
7. A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Wiley, 8th Edition, 2014.
8. Darryl Gove, Multicore Application Programming for Windows, Linux and Oracle Solaris, Addison Wesley, 2011.
9. W. R. Stevens, UNIX Network Programming-Volume II (IPC), PHI, 1998.

REFERENCES

1. Maurice Bach, The Design of the Unix Operating System, PHI, 1986.
2. Beginning Linux Programming, 4th Edition, Neil Mathew and Richard Stores
3. Graham Glass and King Abels, Unix for Programmers and Users – *A complete guide*, PHI, 1993.
4. Sumitabha Das, Unix Concepts and Applications, McGraw Hill, 4th Edition, 2015.
5. Neil Matthew and Richard Stones, Beginning Linux Programming, 3rd Edition, Wiley, 1999.
6. A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Wiley, 8th Edition, 2014.

7. Darryl Gove, Multicore Application Programming for Windows, Linux and Oracle Solaris, Addison Wesley, 2011.
8. W. R. Stevens, UNIX Network Programming-Volume II (IPC), PHI, 1998.

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]

[OBSERVATION SPACE]