PRACTICAL-11

Write a program that creates a fan of n processes where n is passed as a command line argument.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

{

pid_t childpid = 0;

int i, n;

if (argc != 2) /* check for valid number of command-line arguments */

{

fprintf(stderr, "Usage: %s processes\n", argv[0]);

return 1;

}

n = atoi(argv[1]);

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

if ((childpid = fork()) <= 0)

break;

fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n", i, (long)getpid(), (long)getppid(), (long)childpid);

return 0;
```

Output:

```
muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim fanOfN.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ gcc -g -o fan fanOfN.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ ./fan 5

i:1 process ID:2842 parent ID:2841 child ID:0
i:2 process ID:2843 parent ID:2841 child ID:0
i:5 process ID:2841 parent ID:2514 child ID:2845
i:4 process ID:2845 parent ID:2841 child ID:0
i:3 process ID:2844 parent ID:2841 child ID:0
```

$man fork

NAME

fork - create a child process

SYNOPSIS

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

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately

$man getpid

NAME

getpid, getppid - get process identification

SYNOPSIS

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

pid_t getpid(void);
pid_t getppid(void);
```

DESCRIPTION

getpid() returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented (either init(1) or a "subreaper" process defined via the prctl(2) PR_SET_CHILD_SUBREAPER operation).

PRACTICAL-12

Write a program to show that same opened file can be shared by both parent and child processes

// To check whether a parent process and child process should same opened file or not

```
#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<fcntl.h>

#include<sys/stat.h>

#include <unistd.h>

#include <sys/wait.h>

int main()

{

FILE *fp;

int fd;

char ch;

fp=fopen("test","w");

fprintf(fp,"%s\n","This line is written by PARRENT PROCESS");

fflush(NULL);

fd=fork();

if(fd < 0)
```

```
{
printf("Fork Error");
exit(1);
}
if(fd == 0)
{
fprintf(fp,"%s","This line is written by CHILD PROCESS\n");
fclose(fp);
fp=fopen("test","r");
while(!feof(fp))
printf("%c",getc(fp));
}
if(fd > 0)
{ // man 2 wait
if (fd != wait(NULL)) /* parent code */
{
perror("Parent failed to wait due to signal or error");
return 1;
}
}
fclose(fp);
return 0;
```

}

Output:

muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim FileShare.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ gcc -g -o FileShare FileShare.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ ./FileShare

This line is written by PARRENT PROCESS
This line is written by CHILD PROCESS

$man time

NAME

time - run programs and summarize system resource usage

SYNOPSIS

time [-apqvV] [-f FORMAT] [-o FILE]
[--append] [--verbose] [--quiet] [--portability]
[--format=FORMAT] [--output=FILE] [--version]
[--help] COMMAND [ARGS]

DESCRIPTION

time run the program COMMAND with any given arguments ARG.... When COMMAND finishes, time displays information about resources used by COMMAND (on the standard error output, by default). If COMMAND exits with non-zero status, time displays a warning message and the exit status.

time determines which information to display about the resources used by the COMMAND from the string FORMAT. If no format is specified on the command line, but the TIME environment variable is set, its value is used as the format. Otherwise, a default format built into time is used.

Options to time must appear on the command line before COMMAND. Anything on the command line after COMMAND is passed as arguments to COMMAND.

$man wait

NAME

wait, waitpid, waitid - wait for process to change state

SYNOPSIS

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

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
            /* This is the glibc and POSIX interface; see
              NOTES for information on the raw system call. */
```

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the SA_RESTART flag of sigaction(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed waitable.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

waitid(): returns 0 on success or if WNOHANG was specified and no child(ren) specified by id has yet changed state; on error, -1 is returned.

Each of these calls sets errno to an appropriate value in the case of an error.

PRACTICAL-13

Write a program that creates a child process to run ls – l

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main(void)

{

pid_t childpid;

childpid = fork();

if (childpid == -1)

{

perror("Failed to fork");

return 1;

}

if (childpid == 0)

{ /* child code */

execl("/bin/ls", "ls", "-l", NULL); // man 3 exec

perror("Child failed to exec ls");

return 1;

}
if (childpid != wait(NULL)) /* parent code */
```

```
{

perror("Parent failed to wait due to signal or error");

return 1;

}

return 0;

}
```

Output

```
muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim childRunLs.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ gcc -g -o childRunLs childRunLs.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ ./childRunLs
total 72

-rwxr-xr-x 1 muditjain788 muditjain788 11240 Apr 18 13:49 childRunLs
-rw-r--r-- 1 muditjain788 muditjain788   457 Apr 18 13:48 childRunLs.c
-rwxr-xr-x 1 muditjain788 muditjain788 11360 Apr 18 13:44 fan
-rw-r--r-- 1 muditjain788 muditjain788   475 Apr 18 13:44 fanOfN.c
-rwxr-xr-x 1 muditjain788 muditjain788 11648 Apr 18 13:47 FileShare
-rw-r--r-- 1 muditjain788 muditjain788   648 Apr 18 13:46 FileShare.c
-rw-r--r-- 1 muditjain788 muditjain788 15786 Apr 18 13:46 Os_practical_file11-15docx.odt
-rw-r--r-- 1 muditjain788 muditjain788    17 Apr 16 13:56 README.md
-rw-r--r-- 1 muditjain788 muditjain788    78 Apr 18 13:47 test
```

$man execl

NAME

execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...
        /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
        /* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
        /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
        char *const envp[]);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

execvpe(): _GNU_SOURCE

DESCRIPTION

The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for execve(2). (See the manual page for execve(2) for further details about the replacement of the current process image.)

RETURN VALUE

The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

PRACTICAL-14

Write a program to create a zombie child and find its status using system (ps) command
//DEFUNCT MEANS ZOMBIE
// Zombie process is a process that has terminated, but whose parent has not yet waited for it. So parent's parent will become parent of this process
// also remember if parent process dies before child process then init process (process id = 1) becomes the parent of the executing child process.

```
#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<fcntl.h>

#include<sys/stat.h>

#include <unistd.h>

#include <sys/wait.h>

#include <signal.h>

main()

{

int fd;

if((fd=fork())<0)

{

printf("error in creating child");

exit(1);

}

if(fd==0)
```

```
kill(getpid(),SIGKILL);

else

sleep(2);

system("ps -f");

}
```

Output

```
muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim zombieChild.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ gcc -g -o zombieChild zombieChild.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ ./zombieChild

UID        PID  PPID  C STIME TTY          TIME CMD
muditja+  2514  2501  0 13:40 pts/0    00:00:00 bash
muditja+  3241  2514  0 13:52 pts/0    00:00:00 ./zombieChild
muditja+  3242  3241  0 13:52 pts/0    00:00:00 [zombieChild] <defunct>
muditja+  3245  3241  0 13:52 pts/0    00:00:00 sh -c ps -f
muditja+  3246  3245  0 13:52 pts/0    00:00:00 ps -f
```

$man kill

NAME

kill - send a signal to a process

SYNOPSIS

kill [options] <pid> [...]

DESCRIPTION

The default signal for kill is TERM. Use -l or -L to list available signals. Particularly useful signals include HUP, INT, KILL, STOP, CONT, and 0. Alternate signals may be specified in three ways: -9, -SIGKILL or -KILL. Negative PID values may be used to choose whole process groups; see the PGID column in ps command out put. A PID of -1 is special; it indicates all processes except the kill process itself and init.

EXAMPLES

kill -9 -1

Kill all processes you can kill.

kill -l 11

Translate number 11 into a signal name.

kill -L

List the available signal choices in a nice table.

kill 123 543 2341 3453

Send the default signal, SIGTERM, to all those processes.

PRACTICAL-15

Write a program to copy a file.

```
#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

//#include "restart.h"

#define READ_FLAGS O_RDONLY

#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)

#define WRITE_PERMS (S_IRUSR | S_IWUSR)

#include <errno.h>

#include <unistd.h>

#define BLKSIZE 1024

int copyfile(int fromfd, int tofd)

{

char *bp;

char buf[BLKSIZE];

int bytesread;

int byteswritten = 0;

int totalbytes = 0;

for ( ; ; )

{
```

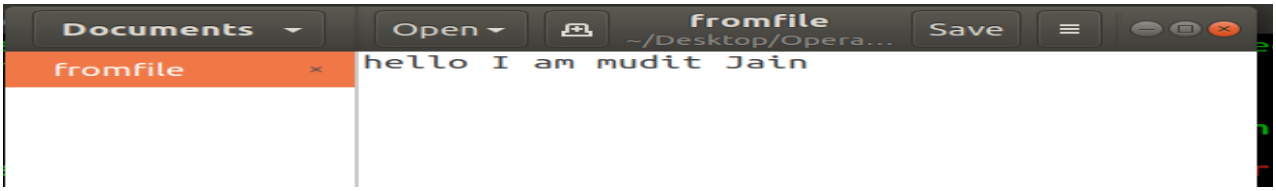
```
while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&

(errno == EINTR)) ; /* handle interruption by signal */

if (bytesread <= 0) /* real error or end-of-file on fromfd */

break;

bp = buf;

while (bytesread > 0)

{

while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) &&
(errno == EINTR)) ; /* handle interruption by signal */

if (byteswritten < 0) /* real error on tofd */

break;

totalbytes += byteswritten;

bytesread -= byteswritten;

bp += byteswritten;

}

if (byteswritten == -1) /* real error on tofd */

break;

}

return totalbytes;

}
```

```
//the main program to copy a file

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

{

int bytes;

int fromfd, tofd;

if (argc != 3)

{

fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);

return 1;

}

if ((fromfd = open(argv[1], READ_FLAGS)) == -1)

{

perror("Failed to open input file");

return 1;

}

if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1)

{

perror("Failed to create output file");

return 1;

}

bytes = copyfile(fromfd, tofd);
```

printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);

return 0; /* the return closes the files */

}



Output

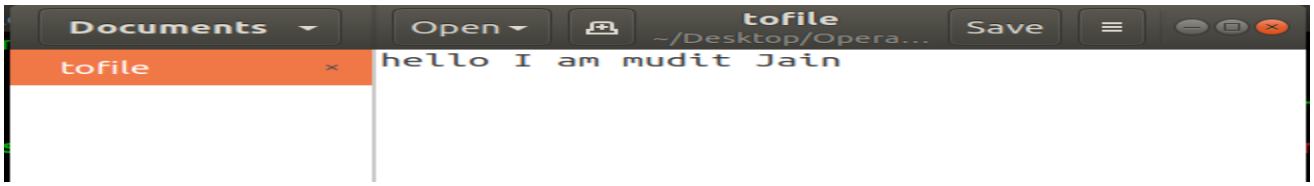
muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim copyFile.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim fromfile

muditjain788@muditjain788:~/Desktop/OperatingSystem$ gcc -g -o copyFile copyFile.c

muditjain788@muditjain788:~/Desktop/OperatingSystem$ ./copyFile fromfile tofile
22 bytes copied from fromfile to tofile

muditjain788@muditjain788:~/Desktop/OperatingSystem$ vim tofile



$man read

NAME

read - read from a file descriptor

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effects.

According to POSIX.1, if count is greater than SSIZE_MAX, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. See also NOTES.

On error, -1 is returned, and errno is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

$man write

NAME

write — send a message to another user

SYNOPSIS

write user [tty]

DESCRIPTION

The write utility allows you to communicate with other users, by copying lines from your terminal to theirs.

When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines you enter will be copied to the specified user's terminal. If the other user wants to reply,they must run write as well.

When you are done, type an end-of-file or interrupt character. The other user will see the message ‘EOF’ indicating that the conversation is over.

You can prevent people (other than the super-user) from writing to you with the mesg(1) command.
If the user you want to write to is logged in on more than one terminal, you can specify which terminal to write by specifying the terminal name as the second operand to the write command. Alternatively, you can let write select one of the terminals - it will pick the one with the shortest idle time. This is so that if the user is logged in at work and also dialed up from home, the message will go to the right place.

The traditional protocol for writing to someone is that the string ‘-o’, either at the end of a line or on a line by itself, means that it is the other person's turn to talk. The string ‘oo’ means that the person believes the conversation to be over.