

NAME: Shawn Louis

ROLL NO: 31

SE COMPS

Experiment no 4

Explore commands like wait and waitpid before termination of process.

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

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

If any process has more than one child processes, then after calling wait(), the parent process has to be in wait state if no child terminates.

If only one child process is terminated, then return a wait() returns process ID of the terminated child process.

If more than one child processes are terminated then wait() reap any *arbitrary child* and return a process ID of that child process. When

wait() returns they also define exit status (which tells us, a process why terminated) via pointer, If status is not NULL.

If any process has no child process then wait() returns immediately “-1”.

// C program to demonstrate working of wait()

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

```
int main() { pid_t cpid; if (fork()== 0)
exit(0); /* terminate child */ else cpid
= wait(NULL); /* reaping parent */
printf("Parent pid = %d\n", getpid());
printf("Child pid = %d\n", cpid);

return 0; }
```

// C program to demonstrate working of wait()

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

```
int main() { if (fork()== 0)
printf("HC: hello from child\n"); else

{ printf("HP: hello from parent\n");
wait(NULL); printf("CT: child has
terminated\n"); } printf("Bye\n");
return 0; }
```

Child status information:

Status information about the child reported by wait is more than just the

exit status of the child, it also includes

- normal/abnormal termination
- termination cause
- exit status

For find information about status, we use

WIF....macros

1. WIFEXITED(status): child exited normally

- WEXITSTATUS(status): return code when child exits

2. WIFSIGNALED(status): child exited because a signal wasnot caught

- WTERMSIG(status): gives the number of the terminating signal

3. WIFSTOPPED(status): child is stopped

- WSTOPSIG(status): gives the number of the stop signal

if more than one child processes are terminated, then wait() reaps any arbitrarily child process but if we want to reap any specific child process,

we use waitpid() function.

Options Parameter

- If 0 means no option parent has to wait for terminates child.
- If WNOHANG means parent does not wait if child does not terminate just check and return waitpid().(not block parentprocess)
- If child_pid is -1 then means any arbitrarily child, herewaitpid() work same as wait() work.

Return value of waitpid()

- pid of child, if child has exited
- 0, if using WNOHANG and child hasn't exited

// C program to demonstrate waitpid()

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

```
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
void waitexample()
{
int i, stat;
pid_t pid[5];
for (i=0; i<5; i++)
{
if ((pid[i] = fork()) == 0)
{
sleep(1);
exit(100 + i);
}
}
// Using waitpid() and printing exit status
// of children.
for (i=0; i<5; i++)
{
pid_t cpid = waitpid(pid[i], &stat, 0);
if (WIFEXITED(stat))
printf("Child %d terminated with status: %d\n",
cpid, WEXITSTATUS(stat));
}
}
// Driver code
int main()
```

```
{  
waitexample();  
return 0;  
}
```

b) Explore commands like `getuid()` , `geteuid()`, `getgid()`, `getegid`

Syntax:

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

```
#include <unistd.h>
```

```
uid_t
```

```
uid_t
```

```
gid_t
```

```
gid_t
```

```
getuid(void);
```

```
geteuid(void);
```

```
getgid(void);
```

```
getegid(void);
```

Description

The `getuid()` function returns the real user ID of the calling process.

The real

user ID identifies the person who is logged in.

The `geteuid()` function returns the effective user ID of the calling process. The

effective user ID gives the process various permissions during execution of

“set-user-ID” mode processes which use `getuid()` to determine the real user ID

of the process that invoked

them.

The `getgid()` function returns the real group ID of the calling process.

Demonstration of the `getuid()`, `geteuid()`, `getgid()`, `getegid()`

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
printf("Get a real user ID:%\n", getuid());
printf("Get the effective user ID:%\n", geteuid());
printf("Get the real group ID:%\n", getgid());
printf("Get the effective group ID:%\n", getegid());
```

- File and Directory related System calls

Basically there are total 5 types of I/O system calls:

`create ()` System Call : Used to Create a new empty file.

Syntax in C language:

```
int creat(char *filename, mode_t mode)
```

Parameter :

filename : name of the file which you want to create

mode : indicates permissions of new file.

Returns : return first unused file descriptor (generally 3 when first `creat`

use in process because 0, 1, 2 fd are reserved) return -1 when error

How it works in OS

Create new empty file on disk

Create file table entry

Set first unused file descriptor to point to file table entry

Return file descriptor used, -1 upon failure

open () System Call :

Used to Open the file for reading, writing or both.

Syntax in C language

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

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

```
#include <fcntl.h>
```

```
int open (const char* Path, int flags [, int mode ]);
```

Parameters

Path : path to file which you want to use absolute path begin with “/”, when you are not work in same directory of file. Use relative path which is

only file name with extension, when you are work in same directory of file.

flags : How you like to use

O_RDONLY: read only,

O_WRONLY: write only,

O_RDWR: read and write,

O_CREAT: create file if it doesn't exist,

O_EXCL: prevent creation if it already exists it works in OS

Find existing file on disk

Create file table entry

Set first unused file descriptor to point to file table entry

Return file descriptor used, -1 upon failure

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

```
#include<fcntl.h>
```

```
#include<errno.h>
```

```

extern int errno;
int main()
{
// if file does not have in directory, then file foo.txt is created.
int fd = open("foo.txt", O_RDONLY | O_CREAT);
printf("fd = %d/n", fd);
if (fd ==-1)
{
// print which type of error have in a code
printf("Error Number % d\n", errno);
// print program detail "Success or failure"
perror("Program");
}
return 0;
}

```

Close () System Call :

Tells the operating system you are done with a file descriptor and
Close the
file which pointed by fd.

Syntax in C language

```
#include <fcntl.h>
```

```
int close(int fd);
```

Parameter

fd :file descriptor

Return

0 on success.

-1 on error.

How it works in the OS

Destroy file table entry referenced by element fd of file descriptor table

– As long as no other process is pointing to it!

Set element fd of file descriptor table to NULL

// C program to illustrate close system Call

```
#include<stdio.h>
#include <fcntl.h>
int main()
{
int fd1 = open("foo.txt", O_RDONLY);
if (fd1 < 0)
{
perror("c1");
exit(1);
}
printf("opened the fd = % d\n", fd1);
// Using close system Call
if (close(fd1) < 0)
{
perror("c1");
exit(1);
}
printf("closed the fd.\n");
}
```

Output:

opened the fd = 3

closed the fd.

// C program to illustrate close system Call

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

```
#include<fcntl.h>
```

```
int main()
```

```
{
```

```
// assume that foo.txt is already created
```

```
int fd1 = open("foo.txt", O_RDONLY, 0);
```

```
close(fd1);
```

```
// assume that baz.txt is already created
```

```
int fd2 = open("baz.txt", O_RDONLY, 0);
```

```
printf("fd2 = % d\n", fd2);
```

```
exit(0);
```

```
}
```

Output:

fd2 = 3

Here, In this code first open() returns 3 because when main process created, then

fd 0, 1, 2 are already taken by stdin, stdout and stderr. So first unused file

descriptor

is 3 in file descriptor table. After that in close() system call is free it this 3

file descriptor

and then after set 3 file descriptor as null. So when we called second open(), then first unused fd is also 3. So, output of this program is 3.

Read () System Call :

From the file indicated by the file descriptor fd, the read() function reads

cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

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

Parameters

fd: file descriptor

buf: buffer to read data from

cnt: length of buffer

Returns: How many bytes were actually read

return Number of bytes read on success

return 0 on reaching end of file

return -1 on error

return -1 on signal interrupt

Important points

buf needs to point to a valid memory location with length not smaller than the

specified size because of overflow.

fd should be a valid file descriptor returned from open() to perform read

operation because if fd is NULL then read should generate error.

cnt is the requested number of bytes read, while the return value is the

actual number of bytes read. Also, some times read system call should read

less bytes than cnt.

```

// C program to illustrate
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
int fd, sz;
char *c = (char *) calloc(100, sizeof(char));
fd = open("foo.txt", O_RDONLY);
if (fd < 0) { perror("r1"); exit(1); }
sz = read(fd, c, 10);
printf("called read(%d, c, 10). returned that"
"%d bytes were read.\n", fd, sz);
c[sz] = '\0';
printf("Those bytes are as follows: %s\n", c);
}

```

Output:

called read(3, c, 10).

returned that 10 bytes were read.

Those bytes are as follows: 0 0 0 foo.

Suppose that foobar.txt consists of the 6 ASCII characters “foobar”.

Then

what is the output of the following program?

```

// C program to illustrate
// read system Call
#include<stdio.h>
#include<fcntl.h>

```

```

int main()
{
char c;
int fd1 =
int fd2 =
Read(fd1,
Read(fd2,
printf("c
exit(0);
}
Open("foobar.txt", O_RDONLY, 0);
Open("foobar.txt", O_RDONLY, 0);
&c, 1);
&c, 1);
= % c\n", c);

```

Output:

c = f

The descriptors fd1 and fd2 each have their own open file table entry, so each descriptor has its own file position for foobar.txt. Thus, the read from fd2 reads the first byte of foobar.txt, and the output is c = f, not c = o.

write () System Call :

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

Syntax in C language

```
#include <fcntl.h>
```

```
size_t write (int fd, void* buf, size_t cnt);
```

Parameters

fd: file descriptor

buf: buffer to write data to

cnt: length of buffer

Returns: How many bytes were actually written

return Number of bytes written on success

return 0 on reaching end of file

return -1 on error

return -1 on signal interrupt

Important points

The file needs to be opened for write operations

buf needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.

cnt is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when fd have a less number of bytes to write than cnt.

If write() is interrupted by a signal, the effect is one of the following:

-If write() has not written any data yet, it returns -1 and sets errno to EINTR.

-If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

```
// C program to illustrate
```

```
// write system Call
```

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

```

#include <fcntl.h>
main()
{
int sz;
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0)
{
perror("r1");
exit(1);
}
sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));
printf("called write(% d, \"hello geeks\\n\", %d).\"
\" It returned %d\n\", fd, strlen("hello geeks\n"), sz);
close(fd);
}

```

Output:

called write(3, "hello geeks\n", 12).

it returned 11

Here, when you see in the file foo.txt after running the code, you get a “hello geeks“. If foo.txt file already have some content in it then write system call overwrite the content and all previous content are deleted and only “hello geeks” content will have in the file.

AIM:

To implement the following system calls

a] Explore wait and waitpid before termination of process.

b]Explore the following system calls: open, read, write, close, getuid, getgid, getegid, geteuid

```
// write system Call
```

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

```
#include <fcntl.h>
```

```
main()
```

```
{
```

```
int sz;
```

```
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

```
if (fd < 0)
```

```
{
```

```
perror("r1");
```

```
exit(1);
```

```
}
```



```

sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));

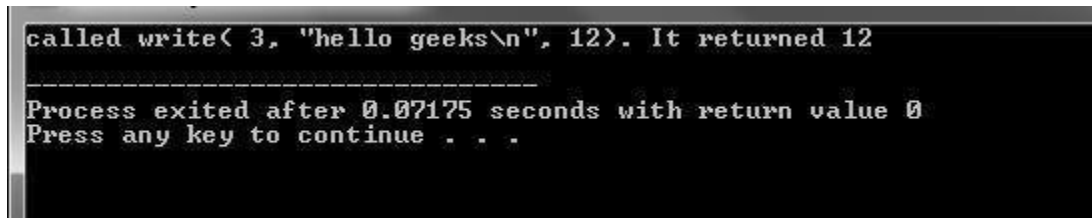
printf("called write(% d, \"hello geeks\\n\\", %d).\"

\" It returned %d\\n\", fd, strlen("hello geeks\n"), sz);

close(fd);

}

```



```

called write( 3, "hello geeks\n", 12). It returned 12
-----
Process exited after 0.07175 seconds with return value 0
Press any key to continue . . .

```

```

#include<stdio.h>

#include <fcntl.h>

int main()

{

int fd, sz;

char *c = (char *) calloc(100, sizeof(char));

fd = open("foo.txt", O_RDONLY);

if (fd < 0) { perror("r1"); exit(1); }

sz = read(fd, c, 10);

```

```
printf("called read(% d, c, 10). returned that"

" %d bytes were read.\n", fd, sz);

c[sz] = '\0';

printf("Those bytes are as follows: % s\n", c);

}
```

A screenshot of a terminal window with a black background and white text. The text shows the output of a C program: "called read(3, c, 10). returned that 10 bytes were read. Those bytes are as follows: hello geek". Below this, a separator line is shown. Then, it says "Process exited after 0.03409 seconds with return value 39" and "Press any key to continue . . . _".

```
called read( 3, c, 10). returned that 10 bytes were read.
Those bytes are as follows: hello geek

-----
Process exited after 0.03409 seconds with return value 39
Press any key to continue . . . _
```

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

```
#include <fcntl.h>
```

```
int main()
```

```
{
```

```
int fd1 = open("foo.txt", O_RDONLY);
```

```
if (fd1 < 0)
```

```
{
```

```
perror("c1");
```

```
exit(1);
```

```
}
```

```
printf("opened the fd = % d\n", fd1);
```

```
// Using close system Call
```

```
if (close(fd1) < 0)
```

```
{
```

```
perror("c1");
```

```
exit(1);
```

```
}
```

```
printf("closed the fd.\n");
```

```
}
```

Output:

opened the fd = 3

closed the fd.

```
// C program to illustrate close system Call
```

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

```
#include<fcntl.h>
```

```
int main()
```

```
{  
  
// assume that foo.txt is already created  
  
int fd1 = open("foo.txt", O_RDONLY, 0);  
  
close(fd1);  
  
// assume that baz.txt is already created  
  
int fd2 = open("baz.txt", O_RDONLY, 0);  
  
printf("fd2 = % d\n", fd2);  
  
exit(0);  
  
}
```

Output:

fd2 = 3

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

```
#include<fcntl.h>
```

```
#include<errno.h>
```

```
extern int errno;
```

```
int main()
```

```
{
```

```
// if file does not have in directory, then file foo.txt is created.
```

```
int fd = open("foo.txt", O_RDONLY | O_CREAT);
```

```
printf("fd = %d\n", fd);
```

```
if (fd == -1)
```

```
{
```

```
// print which type of error have in a code
```

```
printf("Error Number % d\n", errno);
```

```
// print program detail "Success or failure"
```

```
perror("Program");
```

```
}
```

```
return 0;
```

```
}
```

A terminal window with a black background and white text. The first line shows 'fd = 3\n'. The second line is a separator line consisting of dashes. The third line says 'Process exited after 0.02283 seconds with return value 0'. The fourth line says 'Press any key to continue . . . -'.

```
fd = 3\n-----\nProcess exited after 0.02283 seconds with return value 0\nPress any key to continue . . . -
```

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

```
#include<unistd.h>
```

```
#include<stdlib.h>

int main()

{

printf("Get a real user ID:%d\n", getuid());

printf("Get the effective user ID:%d\n", geteuid());

printf("Get the real group ID:%d\n", getgid());

printf("Get the effective group ID:%d\n", getegid());

}
```

Output:

```
dbit@elab1-06:~$ gcc os.c
```

```
dbit@elab1-06:~$ ./a.out
```

Get a real user ID:1000

Get the effective user ID:1000

Get the real group ID:1000

Get the effective group ID:1000

```
dbit@elab1-06:~$
```

```
// C program to demonstrate waitpid()
```

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

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

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

```
#include<unistd.h>
```

```
void waitexample()
```

```
{
```

```
int i, stat;
```

```
pid_t pid[5];
```

```
for (i=0; i<5; i++)
```

```
{
```

```
if ((pid[i] = fork()) == 0)
```

```
{
```

```
sleep(1);
```

```
exit(100 + i);
```

```
}
```

```
}
```

```
// Using waitpid() and printing exit status
```

```

// of children.

for (i=0; i<5; i++)

{

pid_t cpid = waitpid(pid[i], &stat, 0);

if (WIFEXITED(stat))

printf("Child %d terminated with status: %d\n",

cpid, WEXITSTATUS(stat));

}

}

// Driver code

int main()

{

waitexample();

return 0;

}

```

Output:

```

dbit@elab1-06:~$ gcc os.c
dbit@elab1-06:~$ ./a.out
Child 3202 terminated with status: 100

```



```
Child 3203 terminated with status: 101
Child 3204 terminated with status: 102
Child 3205 terminated with status: 103
Child 3206 terminated with status: 104
dbit@elab1-06:~$
```

```
// C program to demonstrate working of wait()
```

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

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

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
if (fork()== 0)
```

```
printf("HC: hello from child\n");
```

```
else
```

```
{
```

```
printf("HP: hello from parent\n");
```

```
wait(NULL);
```

```
printf("CT: child has terminated\n");
```

```
}
```

```
printf("Bye\n");
```

```
return 0;
```

```
}
```

Output:

```
dbit@elab1-06:~$ gcc os.c
dbit@elab1-06:~$ ./a.out
HP:hello from parent
HC:hello from child
Bye
CT:child has terminated
Bye
dbit@elab1-06:~$
```

ALGORITHM :

Step 1 : Declare the structure elements.

Step 2 : Create a temporary file named temp1.

Step 3 : Open the file named “test” in a write mode.

Step 4 : Enter the strings for the file.

Step 5 : Write those strings in the file named “test”.

Step 6 : Create a temporary file named temp2.

Step 7 : Open the file named “test” in a read mode.

Step 8 : Read those strings present in the file “test” and save it in temp2.

Step 9 : Print the strings which are read.