**Experiment:3**

**Aim: Study system calls related to process & process control**

1. **Study the system call fork(), exec(), wait() & exit() using 'man' command. Write down the use, return value, required header files & available options of 4 system calls.**

**Ans.**

* + - **Fork():**
      * **fork - create a child process**
      * **SYNOPSIS**

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

**#include <unistd.h>**

**pid\_t fork(void);**

* + - * **DESCRIPTION**
        + fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.
        + Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent’s page tables, and to create a unique task structure for the child.
        + **RETURN VALUE:** On success, the PID of the child process is returned in the parent’s thread of execution, and a 0 is returned in the child’s thread of execution. On failure, a -1 will be returned in the parent’s context, no child process will be created, and errno will be set appropriately.
    - **Exec():**
      * **exec [-cl] [-a name] [command [arguments]]**
      * If command is specified, it replaces the shell. No new process is created. The arguments become the arguments to command. If the -l option is supplied, the shell places a dash at the beginning of the zeroth arg passed to command. This is what login(1) does.
      * The -c option causes command to be executed with an empty environment. If -a is supplied, the shell passes name as the zeroth argument to the executed command.
      * If command cannot be executed for some reason, a non-interactive shell exits,unless the shell option execfail is enabled, in which case it returns failure.
      * An interactive shell returns failure if the file can not be executed. If command is not specified, any redirections take effect in the current shell, and the return status is 0.If there is a redirection error, the return status is 1.
* **Wait():**

**USE:** wait, waitpid, waitid - wait for process to change state.

**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.

**wait() and waitpid():**

The wait() system call suspends execution of the calling thread

until one of its children terminates. The call wait(&wstatus) is

equivalent to:

waitpid(-1, &wstatus, 0);

The waitpid() system call suspends execution of the calling

thread until a child specified by pid argument has changed state.

By default, waitpid() waits only for terminated children, but

this behavior is modifiable via the options argument, as

described below.

The value of pid can be:

< -1 meaning wait for any child process whose process group ID

is equal to the absolute value of pid.

-1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID

is equal to that of the calling process at the time of the

call to waitpid().

> 0 meaning wait for the child whose process ID is equal to

the value of pid.

The value of options is an OR of zero or more of the following

constants:

**WNOHANG**

return immediately if no child has exited.

**WUNTRACED**

also return if a child has stopped (but not traced via

ptrace(2)). Status for traced children which have stopped

is provided even if this option is not specified.

**WCONTINUED (since Linux 2.6.10)**

also return if a stopped child has been resumed by

delivery of SIGCONT.

**4. exit():**

**EXIT (3)**

exit - cause normal process termination

**SYNOPSIS**

#include <stdlib.h>

void exit(int status);

**DESCRIPTION**

The exit() function causes normal process termination and the value of status & 0xFF is returned to the parent.

All functions registered with atexit(3) and on\_exit(3) are called, in the reverse order of their registration.

(It is possible for one of these functions to use atexit(3) or on\_exit(3) to register an additional function to be executed during exit processing; the new registration is added to the front of the list of functions that remain to be called.) If one of these functions does not return (e.g., it calls \_exit(2), or kills it self with a signal), then none of the remaining functions is called, and further exit processing (in particular, flushing of stdio(3) streams) is abandoned. If a function has been registered multiple times using atexit(3) or on\_exit(3), then it is called as many times as it was registered.

All open stdio(3) streams are flushed and closed. Files created by tmpfile(3) are removed.

The C standard specifies two constants, EXIT\_SUCCESS and EXIT\_FAILURE, that may be passed to exit() to indicate successful or unsuccessful termination, respectively.

**RETURN VALUE**

The exit() function does not return.

1. **Print Process ID, Parent process ID, Read user ID, Real group ID, Effective user ID, effective group ID for current process.**

**File: printid.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<unistd.h>

int main()

{

pid\_t a;

pid\_t pid=getpid();

pid\_t ppid=getppid();

uid\_t uid=getuid();

uid\_t ueid=geteuid();

gid\_t gid=getgid();

printf("Current process id = %d\n",pid);

printf("Parent process id = %d\n",ppid);

printf("User id = %d\n",uid);

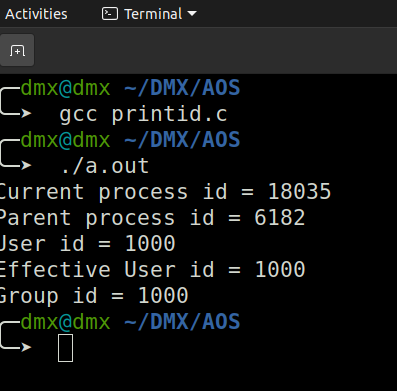
printf("Effective User id = %d\n",ueid);

printf("Group id = %d\n",gid);

return 0;

}

**Output:**

****

1. **Write a program that uses fork() and assign some different task to child process. And make sure that parent exits after child finishes its task (use wait() system call).**

**File: Fork.c**

#include<stdio.h>

#include<stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t a;

a=fork();

int x=5;

int y;

if(a<0)

printf("Child process not created!\n");

else if(a==0){

pid\_t cpid=getpid();

printf("Child process(%d):\n",cpid);

execlp("/bin/ls","ls",NULL);

}

else {

wait();

pid\_t ppid=getpid();

printf("Parent process(%d):\n",ppid);

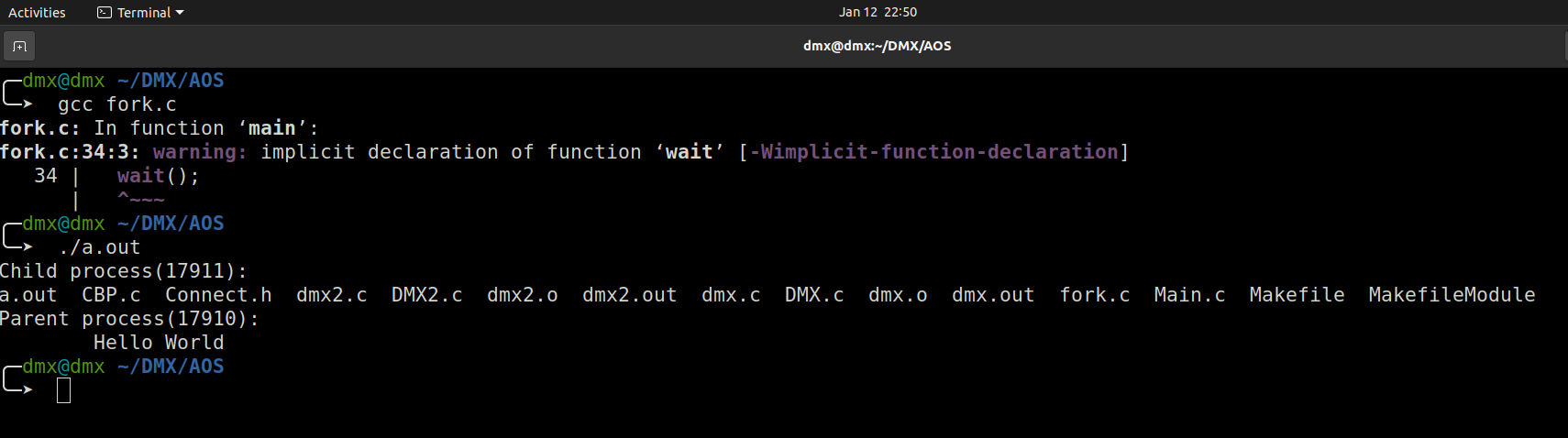
}

printf("\tHello World\n");

return 0;

}

**Output:**

****

1. **Write a program that creates child process and find out/ print values of processID and parent processID for both child and parent processes. Consider following two cases (1) Child exits before parent (2) Parent exits before child.**
2. **Child exits before parent**

**File: CBP.c**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t child\_pid = fork();

// Parent process

if (child\_pid > 0)

{

sleep(5);

printf("Parent Process\n");

}

// Child process

else

{

printf("Child Process\n");

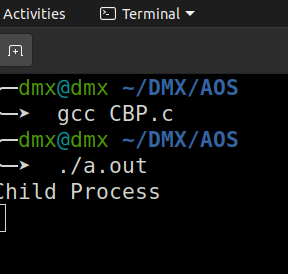
exit(0);

}

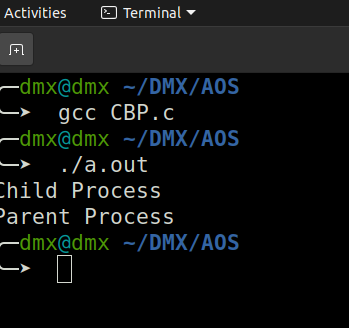
return 0;

}

**Output:**

****

**After sleep of 5 sec.**

****

1. **Parent exits before child.**

**File: PBC.c**

#include<stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// Create a child process

int pid = fork();

if (pid > 0)

printf("in parent process");

// Note that pid is 0 in child process

// and negative if fork() fails

else if (pid == 0)

{

sleep(5);

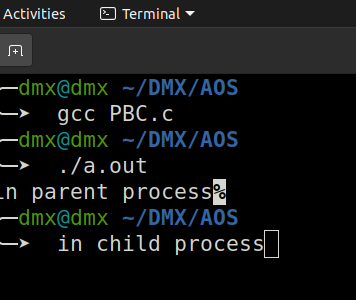
printf("in child process");

}

return 0;

}

**Output:**

****