**Practical - 7**

**Aim: -Write a Program to implement Wait() System call.**

The system call **wait()** is easy. This function blocks the calling process until one of its *child* processes exits or a signal is received. For our purpose, we shall ignore signals. **wait()** takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of **wait()** is to wait for completion of child processes.

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

1. If there are at least one child processes running when the call to **wait()** is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to **wait()** is made, then this **wait()** has no effect at all. That is, it is as if no **wait()** is there.

Consider the following program.

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#define MAX\_COUNT 200

#define BUF\_SIZE 100

void ChildProcess(char [], char []); /\* child process prototype \*/

void main(void)

{

pid\_t pid1, pid2, pid;

int status;

int i;

char buf[BUF\_SIZE];

printf("\*\*\* Parent is about to fork process 1 \*\*\*\n");

if ((pid1 = fork()) < 0) {

printf("Failed to fork process 1\n");

exit(1);

}

else if (pid1 == 0)

ChildProcess("First", " ");

printf("\*\*\* Parent is about to fork process 2 \*\*\*\n");

if ((pid2 = fork()) < 0) {

printf("Failed to fork process 2\n");

exit(1);

}

else if (pid2 == 0)

ChildProcess("Second", " ");

sprintf(buf, "\*\*\* Parent enters waiting status .....\n");

write(1, buf, strlen(buf));

pid = wait(&status);

sprintf(buf, "\*\*\* Parent detects process %d was done \*\*\*\n", pid);

write(1, buf, strlen(buf));

pid = wait(&status);

printf("\*\*\* Parent detects process %d is done \*\*\*\n", pid);

printf("\*\*\* Parent exits \*\*\*\n");

exit(0);

}

void ChildProcess(char \*number, char \*space)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

pid = getpid();

sprintf(buf, "%s%s child process starts (pid = %d)\n",

space, number, pid);

write(1, buf, strlen(buf));

for (i = 1; i <= MAX\_COUNT; i++) {

sprintf(buf, "%s%s child's output, value = %d\n", space, number, i);

write(1, buf, strlen(buf));

}

sprintf(buf, "%s%s child (pid = %d) is about to exit\n",

space, number, pid);

write(1, buf, strlen(buf));

exit(0);

}

This program shows some typical process programming techniques. The main program creates two child processes to execute the same printing loop and display a message before exit. For the parent process (*i.e.*, the main program), after creating two child processes, it enters the wait state by executing the system call **wait()**. Once a child exits, the parent starts execution and the ID of the terminated child process is returned in **pid** so that it can be printed. There are two child processes and thus two **wait()**s, one for each child process. In this example, we do not use the returned information in variable **status**.