

## Experiment – 2

---

### **INDEX**

#### 2. Multiprogramming-Memory management

Implementation of fork(),wait(),exec() and exit() system calls.

**AIM:** Design a c program to implement the multiprogramming memory management

Implementation of Fork() exit(),exec(),wait(),by using System call.

## A)Fork

20 October 2020 17:24

### Fork() in C

Fork system call use for creates a new process, which is called child process, which runs concurrently with process (which process called system call fork) and this process is called parent process. After a new child process created, both processes will execute the next instruction following

the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

**Negative Value:** creation of a child process was unsuccessful.

**Zero:** Returned to the newly created child process.

**Positive value:** Returned to parent or caller. The value contains process ID of newly created child Process.

### Example 1:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
// make two process which run same
// program after this instruction
fork();
printf("Hello world!\n");
return 0;
}
```

### Output:

```
nag-1211@hp-laptop:~/Documents/3-1/OS Lab/Exp 2$ cc fork1.c
nag-1211@hp-laptop:~/Documents/3-1/OS Lab/Exp 2$ ./a.out
Hello world!
```

```
Hello world! oh run same
```

### Example 2:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

### Output:

```
nag-1211@hp-laptop:~/Documents/3-1/OS Lab/Exp 2$ cc fork2.c
fork2.c: In function 'main':
fork2.c:5:1: warning: implicit declaration of function 'fork' [-Wimplicit-function-declaration]
    fork();
    ^~~~
nag-1211@hp-laptop:~/Documents/3-1/OS Lab/Exp 2$ ./a.out
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

### Example 3:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");
    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
```

```
int main()
{
    forkexample();
    return 0;
}
```

### OUTPUT:

```
nag-1211@hp-laptop:~/Documents/3-1/0S Lab/Exp 2$ cc forkexample3.c
nag-1211@hp-laptop:~/Documents/3-1/0S Lab/Exp 2$ ./a.out
Hello from Parent!
Hello from Child!
```

### Example 4:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    int x = 1;
    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

### OUTPUT:

```
nag-1211@hp-laptop:~/Documents/3-1/0S Lab/Exp 2$ cc forkexample4.c
nag-1211@hp-laptop:~/Documents/3-1/0S Lab/Exp 2$ ./a.out
Parent has x = 0
Child has x = 2
```

### fork() vs exec()

The fork system call creates a new process. The new process created by fork() is copy of the current process except the returned value. The exec system call replaces the current process with a new program.

**Questions:****1. A process executes the following code.**

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

fork(); The total number of child processes created is: (GATE CS 2008)

(A)  $n$

(B)  $2^n - 1$

(C)  $2^n$

(D)  $2^{(n+1)} - 1$ ;

**Answer: C****2. Consider the following code fragment:**

```
if (fork() == 0) {  
    a = a + 5;  
    printf("%d, %d\n", a, &a);  
}  
else {  
    a = a - 5;  
    printf("%d, %d\n", a, &a);  
}
```

Let  $u$ ,  $v$  be the values printed by the parent process, and  $x$ ,  $y$  be the values printed by the child

process. Which one of the following is TRUE? (GATE-CS-2005)

(A)  $u = x + 10$  and  $v = y$

(B)  $u = x + 10$  and  $v \neq y$

(C)  $u + 10 = x$  and  $v = y$

(D)  $u + 10 = x$  and  $v \neq y$

**ANSWER -****3. Predict output of below program.**

```
#include <stdio.h>  
#include <unistd.h>  
int main()  
{  
    fork();  
    fork() && fork() || fork();  
    fork();
```

```
printf("forked\n");  
return 0;  
}
```

**OUTPUT:**

## B)Exec family

20 October 2020 17:21

### Execv() and execvp()

#### exec family of functions in C

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file **unistd.h**. There are many members in the exec family which are shown below with examples.

- **execvp** : Using this command, the created child process does not have to run the same program as the parent process does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script . **Syntax:**

```
int execvp (const char *file, char *const argv[]);
```

**file:** points to the file name associated with the file being executed.

**argv:** is a null terminated array of character pointers.

#### **execvp():-**

Let us see a small example to show how to use execvp() function in C. We will have two .C files , **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling execvp() function in execDemo.c .

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

```
int main()
{
    int i;
    printf("I am EXEC.c called by execvp() ");
    printf("\n");
    return 0;
}
```

Now,create an executable file of EXEC.c using command

```
gcc EXEC.c -o EXEC
```

```
//execDemo.c
```

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

```
int main()
{
    //A null terminated array of character
    //pointers
    char *args[]={ "EXEC", NULL};
    execvp(args[0], args);

    /*All statements are ignored after execvp() call as this whole
    process(execDemo.c) is replaced by another process (EXEC.c)
    */
    printf("Ending-----");

    return 0;
}
```

}

- Now, create an executable file of execDemo.c using command

## OUTPUT SCREENSHOT:

```
nag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./execDemo
I am EXEC.c called by execvp()
```

## execv():-

**execv** : This is very similar to execvp() function in terms of syntax as well. The syntax of **execv()** is as shown below:**Syntax:**

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

**path**: should point to the path of the file being executed.

**argv[]**: is a null terminated array of character pointers.

Let us see a small example to show how to use execv() function in C. This example is similar to the example shown above for execvp() . We will have two .C files , **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling execv() function in execDemo.c .

```
//EXEC.c
```

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

```
int main()
{
    int i;

    printf("I am EXEC.c called by execv() ");
    printf("\n");
    return 0;
}
```

Now,create an executable file of EXEC.c using command

```
gcc EXEC.c -o EXEC
```

```
//execDemo.c
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
```

```
    //A null terminated array of character
    //pointers
    char *args[]={". /EXEC", NULL};
    execv(args[0], args);
```

```
    /*All statements are ignored after execvp() call as this whole
    process(execDemo.c) is replaced by another process (EXEC.c)
    */
    printf("Ending-----");
```

```
    return 0;
```

```
}
```

- Now, create an executable file of execDemo.c using command
- gcc execDemo.c -o execDemo



- After running the executable file of execDemo.c by using command ./excDemo, we get the following output:
- I AM EXEC.c called by execv()

### OUTPUT SCREENSHOT:

```
nag-1211@hp:~/Documents/3-1/05 Lab/Exp 2/others$ ./execDemo
I am EXEC.c called by execv()
```

**execlp and execl** : These two also serve the same purpose but the syntax of them are a bit different which is as shown below:**Syntax:**

- int execlp(const char \*file, const char \*arg,.../\* (char \*) NULL \*/);
- int execl(const char \*path, const char \*arg,.../\* (char \*) NULL \*/);

**file:** file name associated with the file being executed

**const char \*arg and ellipses** : describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.

The same C programs shown above can be executed with execlp() or execl() functions and they will perform the same task i.e. replacing the current process with a new process.

- **execvpe and execl** : These two also serve the same purpose but the syntax of them are a bit different from all the above members of exec family. The syntaxes of both of them are shown below :

#### Syntax:

- int execvpe(const char \*file, char \*const argv[],char \*const envp[]);
- **Syntax:**
- int execl(const char \*path, const char \*arg, .../\*, (char \*) NULL,
- char \* const envp[] \*/);

The syntaxes above shown has one different argument from all the above exec members, i.e.

**char \* const envp[]:** allow the caller to specify the environment of the executed program via the argument envp.

**envp:**This argument is an array of pointers to null-terminated strings and must be terminated by a null pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

# C)Wait

20 October 2020 17:23

## wait()

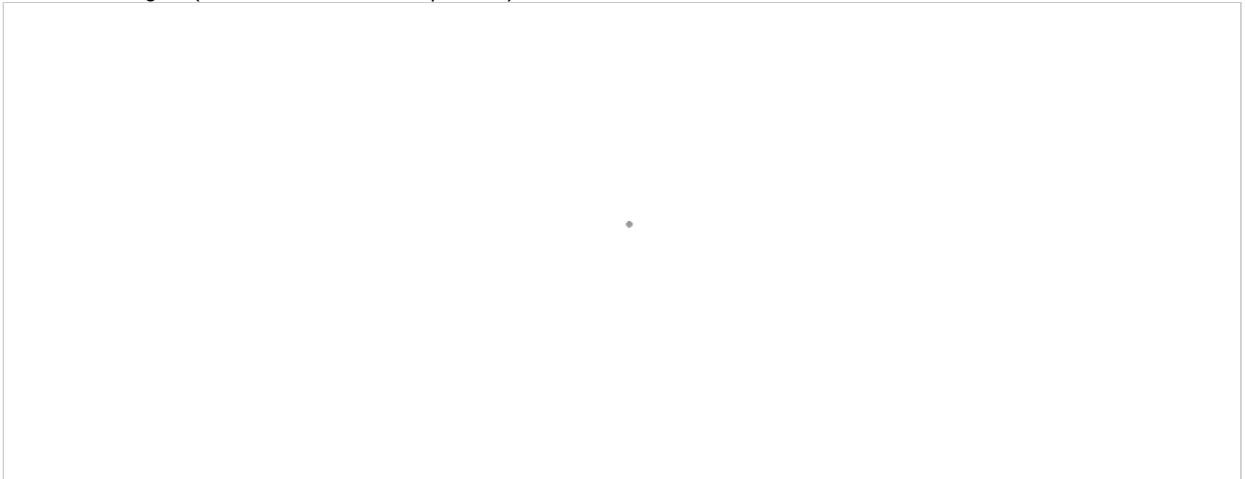
### Wait System Call in C

**Prerequisite :** [Fork System call](#)

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent **continues** its execution after 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.



#### Syntax in c language:

```
#include
```

```
#include
```

```
// take one argument status and returns
```

```
// a process ID of dead children.
```

```
pid_t wait(int *stat_loc);
```

If any process has more than one child processes, then after calling wait(), 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 than wait() reap any **arbitrarily child** and return a process ID of that child process.

When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**.

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

```
#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;
}

```

Output:

Parent pid = 12345678

Child pid = 89546848

```

nag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./wait1
Parent pid = 4167
Child pid = 4168

```

```

#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:** depend on environment

HC: hello from child

HP: hello from parent

CT: child has terminated

(or)

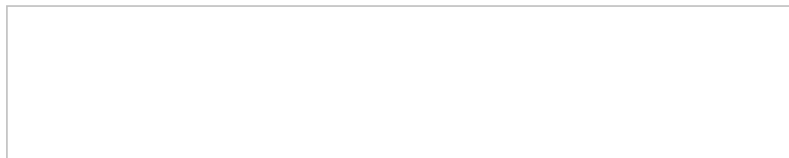
HP: hello from parent

HC: hello from child

CT: child has terminated // this sentence does

// not print before HC

// because of wait.



#### 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 was not 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 we want to prints information about a signal \*/

void psignal(unsigned sig, const char \*s);

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

void waitexample()
{
    int stat;

    // This status 1 is reported by WEXITSTATUS
    if (fork() == 0)
        exit(1);
    else
        wait(&stat);
    if (WIFEXITED(stat))
        printf("Exit status: %d\n", WEXITSTATUS(stat));
    else if (WIFSIGNALED(stat))
        psignal(WTERMSIG(stat), "Exit signal");
}

// Driver code
int main()
{
    waitexample();
    return 0;
}
```

### Output:

Exit status: 1



```
hag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./wait3
Exit status: 1
```

We know 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.

### **Syntax in c language:**

**pid\_t waitpid (child\_pid, &status, options);**

### **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 parent process)
- If child\_pid is -1 then means any **arbitrarily child**, here waitpid() 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

```
#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:**

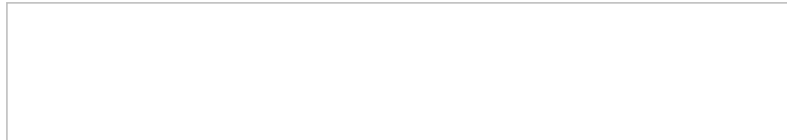
Child 50 terminated with status: 100

Child 51 terminated with status: 101

Child 52 terminated with status: 102

Child 53 terminated with status: 103

Child 54 terminated with status: 104



Here, Children pids depend on the system but in order print all child information.

## C)Wait

20 October 2020 17:23

### wait()

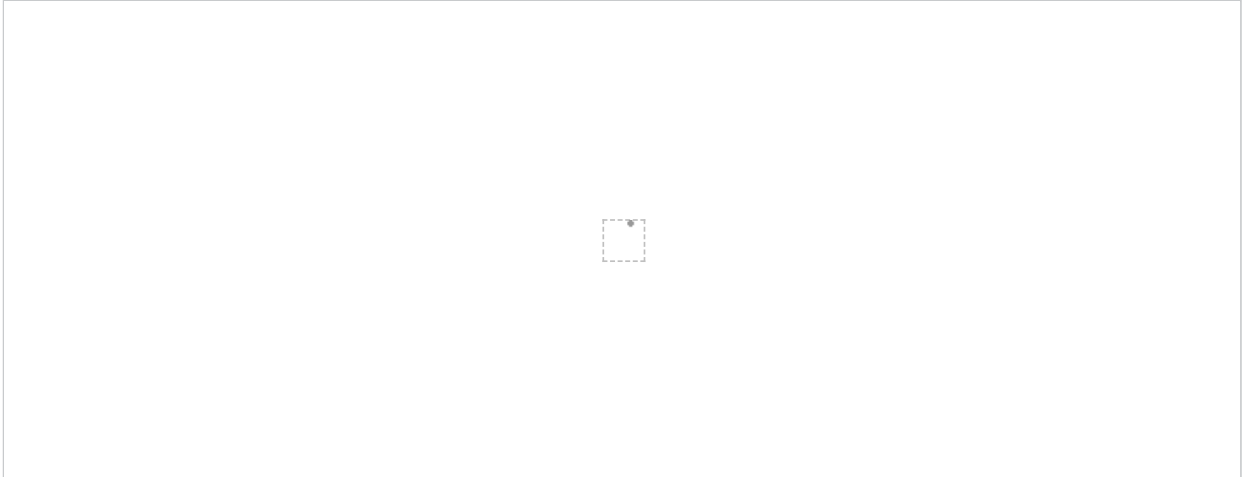
## Wait System Call in C

**Prerequisite :** [Fork System call](#)

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent **continues** its execution after 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.



### Syntax in c language:

```
#include
```

```
#include
```

```
// take one argument status and returns
```

```
// a process ID of dead children.
```

```
pid_t wait(int *stat_loc);
```

If any process has more than one child processes, then after calling wait(), 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 than wait() reap any **arbitrarily child** and return a process ID of that child process.

When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**.

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

```
#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;
}

```

Output:

Parent pid = 12345678

Child pid = 89546848

```

nag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./wait1
Parent pid = 4167
Child pid = 4168

```

```

#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:** depend on environment

HC: hello from child

HP: hello from parent

CT: child has terminated

(or)

HP: hello from parent

HC: hello from child

CT: child has terminated // this sentence does

// not print before HC

// because of wait.

```

nag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./wait2
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye

```

**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 was not 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 we want to prints information about a signal \*/

void psignal(unsigned sig, const char \*s);

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

void waitexample()
{
    int stat;

    // This status 1 is reported by WEXITSTATUS
    if (fork() == 0)
        exit(1);
    else
        wait(&stat);
    if (WIFEXITED(stat))
        printf("Exit status: %d\n", WEXITSTATUS(stat));
    else if (WIFSIGNALED(stat))
        psignal(WTERMSIG(stat), "Exit signal");
}

// Driver code
int main()
{
    waitexample();
    return 0;
}
```

### Output:

Exit status: 1



```
hag-1211@hp:~/Documents/3-1/OS Lab/Exp 2/others$ ./wait3
Exit status: 1
```

We know 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.

### **Syntax in c language:**

**pid\_t waitpid (child\_pid, &status, options);**

### **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 parent process)
- If child\_pid is -1 then means any **arbitrarily child**, here waitpid() 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

```
#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);
    }
}

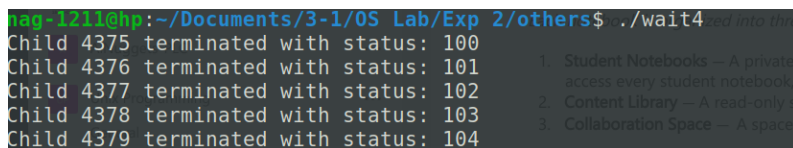
// Using waitpid() and printing exit status
// of children.
for (i=0; i<5; i++)
{
    pid_t cpid = waitpid(pid[i], &stat, 0);
    if (WIFEXITED(stat))
        printf("Child %d terminated with status: %d\n",
            cpid, WEXITSTATUS(stat));
}

// Driver code
int main()
{
    waitexample();
    return 0;
}

```

**Output:**

Child 50 terminated with status: 100  
 Child 51 terminated with status: 101  
 Child 52 terminated with status: 102  
 Child 53 terminated with status: 103  
 Child 54 terminated with status: 104



Here, Children pids depend on the system but in order print all child information.