Exp# 2a **`fork` system call**

Date:

Aim

To create a new child process using fork system call.

`fork()`

- The fork system call is used to create a new process called *child* process.
  - The return value is 0 for a child process.
  - The return value is negative if process creation is unsuccessful.
  - For the parent process, return value is positive
- The child process is an exact copy of the parent process.
- Both the child and parent continue to execute the instructions following fork call.
- The child can start execution before the parent or vice-versa.

`getpid()` and `getppid()`

- The getpid system call returns process ID of the calling process
- The getppid system call returns parent process ID of the calling process

Algorithm

1. Declare a variable *x* to be shared by both child and parent.
2. Create a child process using fork system call.
3. If return value is -1 then
   Print "Process creation unsuccessfull"
   Terminate using exit system call.
4. If return value is 0 then
   Print "Child process"
   Print process id of the child using getpid system call
   Print value of *x*
   Print process id of the parent using getppid system call
5. Otherwise
   Print "Parent process"
   Print process id of the parent using getpid system call
   Print value of *x*
   Print process id of the shell using getppid system call.
6. Stop

Program

```
/* Process creation - fork.c */

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

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

main()
{
   pid_t  pid;
   int x = 5;
   pid = fork();
   x++;

   if (pid < 0)
   {
      printf("Process creation error");
      exit(-1);
   }
   else if (pid == 0)
   {
      printf("Child process:");
      printf("\nProcess id is %d", getpid());
      printf("\nValue of x is %d", x);
      printf("\nProcess id of parent is %d\n", getppid());
   }
   else
   {
      printf("\nParent process:");
      printf("\nProcess id is %d", getpid());
      printf("\nValue of x is %d", x);
      printf("\nProcess id of shell is %d\n", getppid());
   }
}
```

Output

```
$ gcc fork.c

$ ./a.out
Child process:
Process id is 19499
Value of x is 6
Process id of parent is 19498

Parent process:
Process id is 19498
Value of x is 6
Process id of shell is 3266
```

Result

Thus a child process is created with copy of its parent's address space.

Exp# 2b **`wait` system call**

Date:

Aim

To block a parent process until child completes using wait system call.

`wait()`

- The wait system call causes the parent process to be blocked until a child terminates.
- When a process terminates, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Without wait, the parent may finish first leaving a *zombie* child, to be adopted by init process

Algorithm

1. Create a child process using fork system call.
2. If return value is -1 then
   a. Print "Process creation unsuccessfull"
3. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Parent starts"
   c. Print even numbers from 0–10
   d. Print "Parent ends"
5. If return value is 0 then
   a. Print "Child starts"
   b. Print odd numbers from 0–10
   c. Print "Child ends"
6. Stop

Program

```
/* Wait for child termination - wait.c */

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

main()
{
   int i, status;
   pid_t pid;

   pid = fork();
```

```
   if (pid < 0)
   {
      printf("\nProcess creation failure\n");
      exit(-1);
   }
   else if(pid > 0)
   {
      wait(NULL);
      printf ("\nParent starts\nEven Nos: ");
      for (i=2;i<=10;i+=2)
         printf ("%3d",i);
      printf ("\nParent ends\n");
   }
   else if (pid == 0)
   {
      printf ("Child starts\nOdd Nos: ");
      for (i=1;i<10;i+=2)
         printf ("%3d",i);
      printf ("\nChild ends\n");
   }
}
```

Output

```
$ gcc wait.c

$ ./a.out
Child starts
Odd Nos:   1  3  5  7  9
Child ends

Parent starts
Even Nos:   2  4  6  8 10
Parent ends
```

Result

Thus using wait system call zombie child processes were avoided.

Exp# 2c **exec system call**

Date:

Aim

To load an executable program in a child processes exec system call.

execl()

- The exec family of function (execl, execv, execle, execve, execlp, execvp) is used by the child process to load a program and execute.
- execl system call requires path, program name and null pointer

Algorithm

1. Create a child process using fork system call.
2. If return value is -1 then
   a. Print "Process creation unsuccessfull"
3. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Child Terminated".
   c. Terminate the parent process.
5. If return value is 0 then
   a. Print "Child starts"
   b. Load date program into child process using exec system call.
   c. Terminate the child process.
6. Stop

Program

```
/* Load a program in child process - exec.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
   pid_t pid;

   switch(pid = fork())
   {
```

```
      case -1:
         perror("Fork failed");
         exit(-1);

      case 0:
         printf("Child process\n");
         execl("/bin/date", "date", 0);
         exit(0);

      default:
         wait(NULL);
         printf("Child Terminated\n");
         exit(0);
   }
}
```

Output

```
$ gcc exec.c

$ ./a.out
Child process
Sat Feb 23 17:46:59 IST 2013
Child Terminated
```

Result

Thus the child process loads a binary executable file into its address space.