**Index:**

1. **Programs on fork() system call.**
2. **Programs on exec() system call.**
3. **Programs on wait() system call.**
4. **Programs on exit() system call.**

**Note:** Outputs: Left side

**Process Management System calls**

**The fork() System call**

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

* If **fork()** returns a negative value, the creation of a child process was unsuccessful.
* **fork()** returns a zero to the newly created child process.
* **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid\_t** defined in **sys/types.h**.

If the call to **fork()** is executed successfully, Unix will

* Make two identical copies of address spaces, one for the parent and the other for the child.
* Both processes will start their execution at the next statement following the **fork()** call.
* Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces.

When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 27421 and the one in the child receives 0.

* In the parent, since **pid** is non-zero, **Parent Process gets executed**. On the other hand, the child has a zero **pid** and **Child Process gets executed**

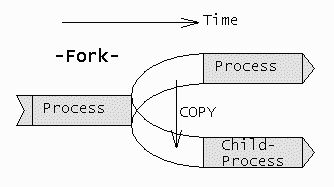
****

Fig llustrating the Fork System Call

##### **Experiment # 4: program to illustrate the usage of fork system call.**

**Aim:** To write a C program to illustrate the usage of fork system call

**Program 1:**

#include <stdio.h> /\* printf, stderr, fprintf \*/

#include<sys/types.h>

int main()

{

int id,childid;

id=getpid();

if(childid=fork()>0)

{

printf("\n i am in the parent process %d",id);

printf("\n i am in the parent process %d",getpid());

printf("\n i am in the parent process %d\n",getppid());

}

else

{

printf("\n i am in child process %d",id);

printf("\n i am in the child process %d",getpid());

printf("\n i am in the child process %d",getppid());

}

}

**Result:** A program in c language to illustrate the usage of fork() system call is executed successfully.

**Program No 2:**

**Aim:** To write a C program to illustrate the usage of fork system call

**Program :**

#include<stdio.h>

#include<unitstd.h>

int main()

{

int p1,p2;

p1=fork();

if(p1==-1)

{

printf(“Error”);

return 0;

}

else

{

printf(“parent is %d\n”,getppid());

printf(“child is %d\n”,getpid());

}

p2=fork();

printf(“parent is %d\n”,getppid());

printf(“child is %d\n”,getpid());

}

**Result:** A program in c language to illustrate the usage of fork() system call is executed successfully.

**Program No 3:**

**Aim:** To write a program in to implement process hierarchy using Fork() System Call

**Program :**

#include<stdio.h>

#include<sys/types.h>

main()

{

int pid,pid1,pid2,pid3,pid4;

pid=fork();

if(pid==0)

{

printf("i am A%d\n",getpid());

printf("i am parentA%d\n",getppid());

pid1=fork();

if(pid1==0)

{

printf(" i am B%d\n",getpid());

printf(" parent of B%d\n",getppid());

pid3=fork();

if(pid3==0)

{

printf("i am c %d\n",getpid());

printf("parent of c %d\n",getppid());

}

}

else

{

pid2=fork();

if(pid2==0)

{

printf("i am D%d\n",getpid());

printf(" parent of D%d\n",getppid());

pid4=fork();

if(pid4==0)

{

printf(" iam E%d\n",getpid());

printf(" parent of E %d\n",getppid());

}

}

}

}

}

**Result:** A program in c language to illustrate the usage of fork() system call is executed successfully.

**1) Define operating system**

**Ans.** It is an interface between user and hardware

**2) What is kernel?**

Kernel is the core of every operating system. It connects applications to the actual processing of data. It also manages all communications between software and hardware components to ensure usability and reliability.

**3) Describe the objective of multiprogramming.**

The main objective of multiprogramming is to have process running at all times. With this design, CPU utilization is said to be maximized.

**4) What is about fork() system call**

It is used to create a child process. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.

**Execv( ) System Call**

The exec family of functions provides a facility for overlaying the process image of the calling process with a new image.

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

The execv function takes exactly two parameters, a pathname for the executable file and an argument array.

It is similar to execl() and also allows the flexibility of passing as arguments

1. A process is created

2. Array of characters are explicitly defined and are passed as arguments to execv()

3. The array ‘arg’ includes object code file, “-l” for long listing of files and character termination

4. These arguments are passed as arg to execv() in addition to path of object code file

##### **Experiment # 5(a): program to illustrate the usage of “Execv()”** **system call.**

**Aim:** To write a C program to illustrate the use of “Execv()” System call

**Program 1:**

#include<stdio.h>

#include<sys/types.h>

main(int argc,char \*argv[])

{

printf("before execv\n");

Char \*argv[]={"/bin/ls","-l",(char\*)0}

execv("/bin/ls",argv);

printf("after execv\n");

}

**Result:** A program in c language to illustrate the usage of execv() system call is executed successfully.

**Program No 2:**

**Aim:** To write a C program to illustrate the use of “Execv()” System call

**Program :**

**First create a program called helloexec.c**

#include<stdio.h>

#include<unistd.h>

int number=77;

int main(int argc, char \* argv[])

{

printf(“helloexec.c: number is %d\n”,number);

printf(“ MY PID %d.\n”, getpid());

printf(“ My Arguments: “);

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

{

printf( “%s\n “, argv[i]);

}

printf(“\n”);

return 0;

}

**Program: execdemo.c**

#include<stdio.h>

#include<unistd.h>

int number= 88;

int main( int argc, char \* argv[])

{

printf(“execdemo: my pid is %d.\n”, getpid());

char \*args[]= { “./helloexec”, “hello”, “world”, NULL};

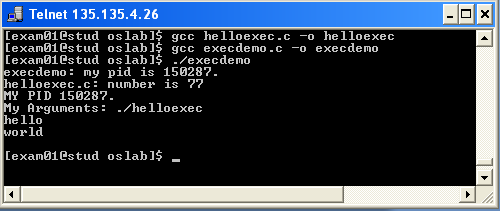
execv(“./helloexec”, args);

printf(“ hello\n”);// we will not see this, because after execv runs, it replaces the currently running image with the new image. Execv does not return until there is an error.so we will not see this being displayed on the command prompt.(memory space, code will be replaced.)

return 0;

**}**

**OUTPUT:**

****

**Result:** A program in c language to illustrate the usage of execv() system call is executed successfully.

**Execl() System call**

It is used to replace the currently running process with a new process image.

This is similar to Execv system call.

**int execl( const char \*path, const char \*arg, ...)**

A processs is created

1. Path of the object file and command line arguments terminated by null value is passed as arguments of the function.
2. Since the command “ls” is used, it displays the file names in that directory

##### **Experiment # 5(b): program to illustrate the usage of “Execl()”** **system call.**

**Program 1:**

**Aim:** To write a C program to illustrate the use of execl() system call

#include<stdio.h>

#include<unistd.h>

int main( int argc, char\* argv[])

{

int ret;

printf(“ calling execl....\n”);

ret= execl(“/bin/ls”, “ls”, “-l”,NULL);

printf(“Failed execl... ret= %d\n”,ret);

return 0;

}

**Result:** A program in c language to illustrate the usage of execl() system call is executed successfully.

**Program 2:**

**Aim:** To write a C program to illustrate the use of execl() system call

**Program: hello.c**

#include<stdio.h>

#include<unistd.h>

int main( int argc, char\* argv[])

{

int i;

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

{

printf(“argv[%d] = %s\n”, i, argv[i]);

}

return 0;

}

**Program:execl.c**

#include<stdio.h>

#include<unistd.h>

int main( int argc, char\* argv[])

{

int ret;

printf(“ calling execl....\n”);

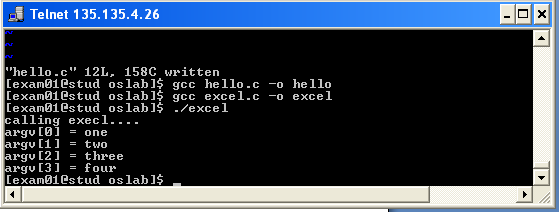
ret= execl(“./hello”, “one”, “two”, “three”,”four”,NULL);

printf(“Failed execl... ret= %d\n”,ret);

return 0;

}

**Output:**

****

**Result:** A program in c language to illustrate the usage of execl() system call is executed successfully.

**Viva Questions:**

**1) What is about exec() system call**

The exec family of functions provides a facility for overlaying the process image of the calling process with a new image.

**2) Difference between execv() and execl() system call**

Both exec family of functions provides a facility for overlaying the process image of the calling process with a new image except types of parameters.

**Wait() System call**

To show synchronization between parent and child process wait system call is used.

This function blocks the calling process until one of its *child* processes exits or a signal is received.

**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.

**pid\_t wait (int \*stat)** – This function causes the caller to suspend execution until a child’s status becomes available or until the caller receives a signal.

1. The parent process creates a child process and waits till the child is executed and terminates. Then parent process is resumed.
2. The exit status of the child process is passed to the parent process through wait system call.

##### **Experiment # 6: program to illustrate the usage of wait system call.**

**Program 1:**

**Aim:** To write a C program to demonstrate working of wait() system call

#include<stdio.h>

#include<sys/types.h>

void main()

{

int i=0,pid;

pid=fork();

if(pid==0)

{

printf("child process \n");

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

printf("\n%d",i);

}

else

{

printf("parent process\n");

for(i=10;i<15;i++)

wait();

printf("process ends\n");

}

}

**Result:** A program in c language to illustrate the usage of wait() system call is executed successfully.

**Program 2:**

**Aim:** To write a C program to demonstrate working of wait() system call

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

main()

{

pid\_t pid;

int rv;

switch(pid=fork())

{

case -1:

perror("fork");

exit(1);

case 0:

printf("\n CHILD: This is the child process!\n");

fflush(stdout);

printf("\n CHILD: My PID is %d\n", getpid());

printf("\n CHILD: My parent's PID is %d\n",getppid());

printf("\n CHILD: Enter my exit status (make it small):\n ");

printf("\n CHILD: I'm outta here!\n");

scanf(" %d", &rv);

exit(rv);

default:

printf("\nPARENT: This is the parent process!\n");

printf("\nPARENT: My PID is %d\n", getpid());

wait(&rv);

printf("\nPARENT: My child's PID is %d\n", pid);

printf("\nPARENT: I'm now waiting for my child to exit()...\n");

printf("\nPARENT: My child's exit status is: %d\n",WEXITSTATUS(rv));

printf("\nPARENT: I'm outta here!\n");

}

}

**Viva Questions:**

**1) What is about wait() system call**

To show synchronization between parent and child process wait system call is used. **wait()** is to wait for completion of child processes.

**Exit() system call**

exit is a system call used to finish a running process from which it is called. The parameter to exit is used to inform the parent process about the status of child process.

#include <std1ib.h>

void exit(int status);

exit(0) or EXIT\_SUCCESS - It terminates the program normally i.e. there is no error in the program then we pass 0. It is fully portable.

exit(1) or EXIT\_FAILURE-It indicates unsuccessful termination and 1 is passed. However, it's usage is non-portable.

##### **Experiment # 7: program to illustrate the usage of exit system call.**

**Aim:** To write a C program to illustrate the use of exit() system call

**Program:**

#include <stdio.h>

#include <stdlib.h>

int main ()

{

printf("Start of the program....\n");

printf("Exiting the program....\n");

exit(0);

printf("End of the program....\n");

return(0);

}

**Result:** A program in c language to illustrate the usage of exit() system call is executed successfully.