TERM WORK-8

☆ Title Of Experiment:-

Write a C/C++ program to avoid zombie process by forking twice.

☆ Aim:-

1) To understand concept of zombie process.

2) To understand how forking twice prevents creation of a zombie.

☆ Introduction:-

In UNIX, a zombie process is a process that has completed execution but still has an entry in the process table. A process table contains all information that must be saved when CPU switches from process to another, in a multitasking system. This information allows suspended processes to be restarted at a later time. Zombie processes simply fill up entries in process table & as a result, if the table is filled completely, we cannot run any new processes because no entry can be made in the process table. We can use ps system call to check whether a process is zombie or not.

Few techniques to avoid creation of zombie process are:

i) using wait() or waitpid().
ii) The parent process can register a SIGCHLD handler with signal().
iii) By forking twice.

Now, how forking twice prevents creation of zombie process is explained as follows:

i) Process A creates a child B by forking once & assigns it some work. Now, A waits for completion of B.

ii) Now, process B creates a child process C, assigns the work given to it by A to C & then B terminates.

iii) Upon termination of B, 2 things happen:
   a) C becomes an orphan process & hence is adopted by init process.
   b) A resumes it execution & entry of B is erased from the process table.

iv) Now, when C terminates, its entry is cleared by init process & hence C doesn't become a zombie process.

☆ Source Code:-

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

int main()
{
    pid_t pid = fork();

    if (pid < 0)
    {
        printf("fork error");
        exit(0);
    }
```

```
{ if (pid==0) // first child

        pid = fork();
        if (pid<0)
        {
            printf("fork error");
            exit (0);
        }
        if(pid ==0)  //second child
        {
            sleep(1);
            printf("Second child, porent id = %d", getppid());
        }
    }
    else
    {
        //porent process
        wait (NULL);
        sleep(2);
        system("ps -o pid, ppid, state, tty, command");
    }
    return 0;
}
```

✯ Expected Output:-

second child, parent id = 1

PID	PPID	S	TT	COMMAND
731	728	S	pts/0	bash
771	731	S	pts/0	./a.out
774	771	R	pts/0	ps -o pid,ppid,state,tty,command

✯ Conclusion:-

We could understand what is a zombie process, how forking twice avoids its creation & also implement a C program to demonstrate the same.

✯ References:-

Terrence Chan, UNIX System Programming Using C++, Prentice Hall India, 1999 & onwards.

TermWork-9

☆ Title Of Experiment :-

Write a C or C++ program to implement 'system' function.

☆ Aim Of Experiment :-

1) To understand the use of system function.

2) To implement the system function in C.

☆ Introduction :-

The system function is a part of C/C++ standard library. Its used to pass the commands that can be executed in the command processor or the terminal of the OS & finally returns after the command has been completed. Its prototype is:

```
#include <stdlib.h>
int system(const char *command);
```

Example, system("date"); gives Thu Dec 31 09:31:58 UTC 2020 as output. Now, system executes a command by calling /bin/sh -c command.

To implement system function, we use the execl() function. This function replaces the current process image with a new process image specified by path. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

Prototype of execl() is:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ....., const char *argn,
          NULL);
```

where path is the path of file to execute, arg0, ..., argn are NULL terminated C-strings which constitute the argument list available to the new process image. The list must be terminated by NULL. The arg0 argument must point to a filename that's associated with the process being started & cannot be NULL.

★ Source Code:-

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

void mySystem(const char *command)
{
    pid_t pid;

    if ((pid = fork()) == 0)
        execl("/bin/bash", "bash", "-c", command, NULL);
    else
        waitpid(pid, NULL, 0);
}
```

```
int main(int argc, char *argv[])
{
    for (int i=1; i<argc; ++i)
    {
        mySystem(argv[i]);
        printf("\n");
    }
    mySystem("ps -o pid,ppid,state,tty,command");

    exit(0);
}
```

★ Expected output:-

```
$ ./a.out who date ls
sagar   :0       2020-12-31 17:48 (:0)
sagar   pts/3    2020-12-31 18:13 (:0)

Thu Dec 31 18:14:47 IST 2020

a.out  SystemCall.c  SystemCall.c~

  PID  PPID S TT       COMMAND
 4105  4098 S pts/3    bash
 4176  4105 S pts/3    ./a.out who date ls
 4180  4176 R pts/3    ps -o pid,ppid,state,tty,command
```

✯ Conclusion:-

We understood the use of system() function & could write a C program to implement the system function.

✯ References:-

Terrence Chan: UNIX System Programming Using C++, Prentice Hall India, 1999 & onwards.

TermWork-10

☆ Title Of Experiment:-

Write a C/C++ program to set up a real time clock interval timer using the alarm API.

☆ Aim Of Experiment:-

1) To understand the use of alarm API.

2) To implement real time interval timer using alarm API.

☆ Introduction:-

An interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to limit the time allowed for the execution of some tasks.

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a specified number of real clock seconds. Prototype is:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

It returns 0 or number of CPU seconds left in the process timer, as set by a previous alarm() call.

For signal handling, we use sigaction API & it allows us to examine or modify the action associated with a particular signal. Its prototype is:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict_act,
              struct sigaction *restrict_oact);
```

It returns 0 if ok & 1 on error.

The struct sigaction data type is defined in <signal.h> header as:

```
struct sigaction
{
   void     (*sa_handler)(int);
   sigset_t  sa_mask;
   int       sa_flag;
};
```

☆ Source Code:-

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#define INTERVAL 5

void callme(int sig_no)
{
      alarm(INTERVAL);
      printf("Hello!!\n");
}
```

```
int main()
{
    struct sigaction action;
    action.sa_handler = (void (*)(int)) callme;
    sigaction(SIGALRM, &action, 0);
    alarm(2);
    sleep(5);

    return 0;
}
```

✡ Expected Output:-

Hello!!

✡ Conclusion:-

We could understand the use of alarm API & use it to implement real time interval timer.

✡ References:-

Terrence Chan: UNIX System Programming Using C++, Prentice Hall India, 1999 & onwards.

Course Activity

☆ Title Of Experiment :-

Write a C or C++ program to implement 'system' function.

☆ Aim Of Experiment :-

1) To understand the use of system function.

2) To implement the system function in C.

☆ Introduction :-

The system function is a part of C/C++ standard library. Its used to pass the commands that can be executed in the command processor or the terminal of the OS & finally returns after the command has been completed. Its prototype is:

```
#include <stdlib.h>
int system(const char *command);
```

Example, system("date"); gives Thu Dec 31 09:31:58 UTC 2020 as output. Now, system executes a command by calling /bin/sh -c command.

To implement system function, we use the execl() function. This function replaces the current process image with a new process image specified by path. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

Prototype of execl() is:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ....., const char *argn,
          NULL);
```

where path is the path of file to execute, arg0, ..., argn are NULL terminated C-strings which constitute the argument list available to the new process image. The list must be terminated by NULL. The arg0 argument must point to a filename that's associated with the process being started & cannot be NULL.

★ Source Code:-

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

void mySystem(const char *command)
{
    pid_t pid;

    if ((pid = fork()) == 0)
        execl("/bin/bash", "bash", "-c", command, NULL);
    else
        waitpid(pid, NULL, 0);
}
```

```
int main(int argc, char *argv[])
{
    for (int i=1; i<argc; ++i)
    {
        mySystem(argv[i]);
        printf("\n");
    }
    mySystem("ps -o pid,ppid,state,tty,command");

    exit(0);
}
```

☆ Expected output:-

```
$ ./a.out who date ls
sagar   :0       2020-12-31 17:48 (:0)
sagar   pts/3    2020-12-31 18:13 (:0)

Thu Dec 31 18:14:47 IST 2020

a.out  SystemCall.c  SystemCall.c~

  PID  PPID S TT       COMMAND
 4105  4098 S pts/3    bash
 4176  4105 S pts/3    ./a.out who date ls
 4180  4176 R pts/3    ps -o pid,ppid,state,tty,command
```

✯ Conclusion:-

We understood the use of system() function & could write a C program to implement the system function.

✯ References:-

Terrence Chan: UNIX System Programming Using C++, Prentice Hall India, 1999 & onwards.