Operating Systems Lab

Lab # 08

System Calls

Instructor: Engr. Muhammad Usman
Email: usman.rafiq@nu.edu.pk

Course Code: CL2006

Department of Computer Science,
National University of Computer and Emerging Sciences FAST
Peshawar Campus

Contents

Operating Systems Processes

Termination States

A process can terminate in one of the following ways:

1. Main function calls return
2. Exit function called
3. Aborted by higher priority process (e.g. parent or kernel)
4. Terminated by a signal

Regardless of any of these reasons, the same kernel code is invoked which performs the following:

1. Close open files
2. Notifies Parent and Children
3. Release memory resources

Exit() System Call

void exit (int status);

exit() ---> terminates the process normally.

status: Status value returned to the parent process. Generally, a status value of 0 or **EXIT_SUCCESS** indicates success, and any other value or the constant **EXIT_FAILURE** is used to indicate an error.

Return Value

This function does not return any value.

exit() performs following operations:

* Flushes unwritten buffered data.
* Closes all open files.
* Removes temporary files.
* Returns an integer exit status to the operating system.

exit1.c

```
/* exit example */
#include <stdio.h>
#include <stdlib.h>
int main ()
{
FILE  * pFile;    //  pointer of type file
pFile = fopen ("myfile.txt", "r");        // opening the file in read-only mode
if (pFile == NULL)
{
        printf ("Error opening file");
        exit (1);
}
else
{
        /* file operations here */
        printf("File is Opening here...");
        //exit(0);
}
return 0;
}
```

1. **Exit Success: Exit Success** is indicated by **exit(0)** statement which means successful termination of the program, i.e. program has been executed without any error or interrupt.

Note: Create a file called ‘myfile.txt’ and run the above code again in your local device to see the output.

2. **Exit Failure: Exit Failure** is indicated by **exit(1)** which means the abnormal termination of the program, i.e. some error or interrupt has occurred. We can use different integer other than 1 to indicate different types of errors.

The exit() system call causes a normal program to terminate and return status to the parent process. Study the behavior of the following program. You will see that there may be a number of exit points from a program. Can you identify which exit() call is being used each time a program exits??

exit4.c

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int num;
   void anotherExit(void);  // function prototype
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num>25)
   {
         printf("Exit 1\n");
         exit(1);
   }
   else
         anotherExit();
}
void anotherExit()    //  function definition
{
         printf("Exit 2\n");
         exit(1);
}
```

what is the difference between exit() and return() in c?

exit() is a system call which terminates current process. exit() is not an instruction of C language.

Whereas, return() is a C language instruction/statement and it returns from the current function (i.e. provides exit status to calling function and provides control back to the calling function).

return returns from the current function; it's a language keyword like **for** or **break**.

exit() terminates the whole program, wherever you call it from. (After flushing stdio buffers and so on).

Example with return:

return.c

```
#include <stdio.h>

void f(){
   printf("Executing f\n");
   return;
}

int main(){
   f();
   printf("Back from f\n");
}
```

If you execute this program it prints:

```
Executing f
Back from f
```

Another example for exit():

exit.c

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

void f(){
   printf("Executing f\n");
   exit(0);
}
```

```
int main(){
   f();
   printf("Back from f\n");
}
```

If you execute this program it prints:

Executing f

You never get "Back from f". Also notice the `#include <stdlib.h>` necessary to call the library function `exit()`.

Atexit() System Call

Which is used to register a function which has to be called in a before terminating the program.

Declaration

Following is the declaration for atexit() function.

```
int atexit(void (*func)(void))
```

Atexit() causes the function pointed by this (***func**) to be called upon normal program termination. So for every normal program termination the function registered through this atexit() function will be called.

The C library function **int atexit(void (*func)(void))** causes the specified function **func** to be called when the program terminates. You can register your termination function anywhere you like, but it will be called at the time of the program termination.

The function pointed by ***atexit()*** is automatically called without arguments when the program terminates normally. In case more than one function has been specified by different calls to the ***atexit()*** function, all are executed in the order of a **stack** (i.e. the last function specified is the first to be executed at exit). A single function can be registered to be executed at exit more than once.

Parameters: The function accepts a single mandatory parameter **func** which specifies the pointer to the function to be called on normal program termination (Function to be called).

Return Value: The function returns following values:

- Zero, if the function registration is successful
- Non zero, if the function registration failed

Example

The following example shows the usage of atexit() function.

atexit1.c

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

void functionA () {
   printf("This is functionA\n");
}

int main () {
   /* register the termination function */
   atexit(functionA);

   printf("Starting  main program...\n");

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

   return 0;
}
```

Let us compile and run the above program that will produce the following result −

```
Starting main program...
Exiting main program...
This is functionA
```

The code below provides two functions; atexit() and exit(). Note the structure of code and the behavior of execution and then attempt the questions in the end.

atexit2.c

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

int main(void)
{
```

```
    //functions prototypes
    void f1(void);
    void f2(void);
    void f3(void);

    atexit(f1);
    atexit(f2);
    atexit(f3);

    printf("Getting ready to exit\n");
    exit(0);
}

//  function definitions
void f1(void)
{
    printf("In f1\n");
}

void f2(void)
{
    printf("In f2\n");
}

void f3(void)
{
    printf("In f3\n");
//    abort();
}
```

Output:

```
Getting ready to exit
In f3
In f2
In f1
```

Q1: What is the difference between exit() and atexit()? What do they do? (Check man atexit and man 3 exit).

Q2: What does the **0** provided in the exit() call mean? What will happen if we change it to 1? (Check manual page for exit).

Q3: If we add an exit call to function f1, f2, or f3. What will happen to execution of our program?

Q4: Why do you think we are getting reverse order of execution of atexit calls?

atexit3.c

```
// C program to illustrate atexit() function
#include <stdio.h>
#include <stdlib.h>

// Returns no value, and takes nothing as a parameter
void done()
{
	printf("Exiting Successfully\n"); // Executed second
}
// Driver Code
int main()
{
	int value;
	value = atexit(done);

	if (value != 0) {
		printf("atexit () function registration failed \n");
		exit(1);
	}
	printf("Registration successful \n");// Executed First
	return 0;
}
```

Output:

```
Registration successful
Exiting Successfully
```

If *atexit* function is called more than once, then all the specified functions will be executed in a reverse manner, same as of the functioning of the stack.

atexit4.c

```
// C program to illustrate more than one atexit function
#include <stdio.h>
#include<stdlib.h>

// Executed last, in a Reverse manner
```

```
void first()
{
	printf("Exit first\n");
}


// Executed third
void second()
{
	printf("Exit Second\n");
}

// Executed Second
void third()
{
	printf("Exit Third\n");
}

// Executed first
void fourth()
{
	printf("Exit Fourth\n");
}
// Driver Code
int main()
{
	int value1, value2, value3, value4;
	value1 = atexit(first);
	value2 = atexit(second);
	value3 = atexit(third);
	value4 = atexit(fourth);
	if ((value1 != 0) || (value2 != 0) ||
		(value3 != 0) || (value4 != 0))
	{
		printf("atexit() function registration Failed\n");
		exit(1);
	}
	// Executed at the starting
	printf("Registration successful\n");
	return 0;
}
```

Output:

```
Registration successful

Exit Fourth

Exit Third
```

```
Exit Second
Exit first
```

atexit5.c

```
// C program to illustrate
//atexit() function when it throws an exception.
#include <stdio.h>
#include<stdlib.h>
void shows_Exception()
{
        int y = 50, z = 0;
        // Program will terminate here
        int x = y / z;
        // Cannot get printed as the program has terminated
        printf("Divided by zero\n");
}
// Driver Code
int main()
{
        int value;
        value = atexit(shows_Exception);
        if (value != 0) {
                printf("atexit() function registration failed\n");
                exit(1);
        }

        // Executed at the starting
        printf("Registration successful\n");
        return 0;
}
```

Note: If a registered function throws an exception which cannot be handled, then the terminate() function is called.

atexit6.c

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

 void onexit(void);    //functions prototypes
int main(void)
{

   int counter =1;
   int value=atexit(onexit);

   if(value!=0)
   {
            printf("Failed to register onexit as the termination function\n");
   }

   while(1)
   {
            printf("%d \n",counter);
            if(counter==10)
            {
              exit(0);
            }
            counter++;
   }

 return 0;
}

//  function definitions
void onexit(void)
{
   printf("Hi I am called before termination\n");
}
```

Output:

```
1
2
3
4
5
6
7
```

8
9
10
Hi I am called before termination

Abort System Call

'abort()' function terminates execution of a program abnormally.

It's defined in <stdlib.h> header and is prototyped below:

```
void abort(void);
```

Abort cause abnormal process termination. The abort() function never returns.

Type, run and execute the code below. Then answer the questions in the end.

abort.c

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
   abort();
   exit(0);
}
```

Q1: Check the man pages for abort. How does the abort call terminate the program? What is the name of the particular signal?

Q2: Execute your program. What is the output of our program?

Q3: Include the abort call in function f3 in our code provided for Atexit() call. How does our program terminate using this?

Kill Call

Full description of this call will appear in Lab manual 11. But as a demonstration, you may run the following code to see how the kill() call works.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
```

```
printf("Hello");
kill(getpid(), 9);
printf("Goodbye");
}
```

You are already familiar with getpid() system call. To find out what 9 is, first look at the output of the command:

kill -l

Find out the word mentioned next to 9. Search on the internet for this.

Death of Parent or Child

We covered the process termination using the exit() system call and that a program may have more than one exit points. In a relationship where a process has spawned children, what would be the effect if either the parent dies or the child dies on the other processes linked to it? This section will look and study such a development.

Parent Dies Before Child

If a parent process dies before it's children, the children will be orphaned. They will be assigned to another process in the system and as such the children should be informed about who their new parent is going to be. This new parent process is the parent of all processes in the system, i.e., the "**init**" process.

parentdies.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(void)
{
   int i, pid;
   pid= fork();
   if(pid> 0)       // Parent
   {
         sleep(2);
         exit(0);
   }
```

```
    else if(pid ==0)     // child
    {
        for(i=0; i<5; i++)
        {
            printf("My parent is %d\n", getppid());
            sleep(3);
        }
    }
}
```

Output:

```
My parent is 2893
My parent is 1316
My parent is 1316
My parent is 1316
My parent is 1316
```

Run the code.

1. What are the PPID values you are receiving from the for loop?

2. What has happened when the numbers of the PPID change?

3. What is now PID of the init process?

Child Dies Before Parent

A process that is waiting for its parent to accept its return code is called a zombie process.

Following code is complete opposite of what we have seen so far.

childdies.c

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

int main(void)
{
    int i, pid;
    printf("I am Parent with PID: %d\n", getpid());
```

```
    pid= fork();

    if(pid> 0)          //Parent
    {
          sleep(60);
    }
    else if(pid ==0)    //child
    {
          exit(0);
    }
}
```

Run the code.

In another window (terminal), check the status of your parent and child processes using the “ps au” command. You should be able to see a Z, or <defunct> next to the child process which has been created. Such a process is usually termed a “**Zombie**” process.

Wait for 60 seconds, then check the status of your process again. You will note that both the parent, as well as the Zombie process are now gone.