UNIX Programming Guide – Part 4

Content Summary

1. Process Basics
2. Creating Processes – exec() and fork() overview
3. system() Function Call
4. Program Components : Arguments and Environment
5. exec() Family
6. Self Study and Exercises

Process Basics

Program Vs Process

A program is just a file containing instructions and data. These instructions while in execution constitute a process. A running instance of a program is called a process. For example, if you open two windows of internet explorer, the same internet explorer program is executed twice. There is a single program but there are two running instances of the same program, hence two processes.

More formally, a process is an execution environment that consists of instructions, user-data and system-data segments as well as resources acquired at runtime.

Process-ID

Each process in a UNIX system is identified by its unique process id, referred to as *pid*. Process IDs are usually 16-bit numbers that are assigned sequentially by UNIX as new processes are created. Every process also has a parent process (except the special init process and Zombie Processes).Thus, you can think of the processes on a UNIX system as arranged in a tree, with the init process at its root. The parent process ID, or *ppid*, is simply the process ID of the process's parent. When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is defined in<sys/types.h>. A program can obtain the process ID of the process it's running in with the getpid() system call, and it can obtain the process ID of its parent process with the getppid() system call. The program below prints its process ID and its parent's process ID:

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

```
{
printf("\nProcess id: %d",getpid());
printf("\nParent Process id: %d",getppid());
return 0;
}
```

Run the above program in two terminal windows. The output in each window would be different. If new windows are opened, you will find that each time process-id's are printed. This is because each invocation is a new process.

Run the program twice in the same window. This time you with every execution, you will find that only the current process-id changes while the parent process-id (which is the process-id of the terminal shell) remains the same.

Viewing Active Processes

The ps command displays the processes that are running on your system. By default, invoking ps displays the processes controlled by the terminal or terminal window in which ps is invoked. For example:

- `ps`

```
PID TTY TIME CMD
21693 pts/8 00:00:00 bash
21694 pts/8 00:00:00 ps
```

This invocation of ps shows two processes. The first, bash, is the shell running on this terminal. The second is the running instance of the ps program itself. The first column, labeled PID, displays the process ID of each.

Context of a Process

Many processes may be running in a UNIX system. But memory/processor-capacity may be limited and we may not be able to hold all processes in main memory at the same time. Hence the system needs to replace one process from main memory and introduce another process in its place. The replacement of one process with another is called context-switching.

Assume process A executes for some time and is then replaced with process B before it fully completes its execution. After sometime, we may again need to run process A. In that case, process A would be brought back to main memory. When process A comes back to main memory, we need to ensure that process A starts exactly at the same point where it had stopped execution. To be able to do that, some information about process A needs to be stored in the system before we replace it. This information is called the "context" of a process.

The context of a process includes all the information that the OS needs to restart a process after a context switch. Typically, this includes PC, stack, registers, executable code etc

Creating Processes – exec() and fork()

The fork() system call creates a new process which is an exact clone of an existing process. Recall that the context of each process contains the program instructions and data. Hence when we clone a process from an existing process using the fork() call, the new process also would contain the same program instructions and data. Hence the newly created process executes the same program as the old (parent) process.

The exec() system call on the other hand, reinitializes an existing process with some other designated program. exec() does NOT create a new process. It merely flushes the current context of a program and loads a new context (new program).

It is evident that exec() and fork() when used individually have very limited use. When used as a pair, these system calls can be powerful (explained in the next section).

exec() call is the only way to execute programs in UNIX. In fact, the kernel boots itself using the exec() call. And fork() is the only way to create new processes in UNIX.

System() Function call

The System() call creates a new process that will execute a designated program. System() function comes handy when you want to execute a command from a C program. System() function creates a new process. The program corresponding to the command to be executed will be made the context of this new process. The syntax of the call is given below:

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

The following program gives the output of the command “ls –l”

```
#include <stdlib.h>
int main ()
{
int return_value;
```

```
return_value = system (“ls -l /”);
return return_value;
}
```

The System function created a new shell in a separate child process. That shell was made to execute the “ls –l” command by loading the “ls” program. The System function returns the exit status of the shell after the command is executed. If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1.

Understanding the System() Function

System() is a C library call which in turn uses the UNIX system calls to do its job.
Consider a Process , say ‘Y’, that calls the function System(“Z”). Given below are the sequence of steps that the System(“Z”) function would perform:

1. Call fork() that creates a new process, ‘N’, which has the same context as the process ‘Y’
2. Call exec() and re-initialise the context of ‘N’ with that of program ‘Z’

Applications of the System() Function

System() can be used for the variety of interesting applications. One example usage could be writing a program that will measure the execution time of any other given program.

Before executing the code given below, create a C program that has at least three nested loops each having around 500 iterations. Save the executable with file name “sample”.

Program to find the execution time of another program

```
# include<stdio.h>
# include<sys/time.h>
int main()
{
struct timeval start,finish;

gettimeofday(&start,NULL);
system("./sample");
gettimeofday(&finish,NULL);

printf("Took %f seconds\n",finish.tv_sec-start.tv_sec+1e-
6*(finish.tv_usec-start.tv_usec));
```

```
return 0;
}
```

In the above program our focus is on the system() function call and not on the timeval structure.

Program Components: Arguments and Environment

When a UNIX Program is executed, it receives two collections of data from the process that invoked it: the arguments and the environment. To C programs, they both are in form of an array of character pointers, all but the last of which point to a NULL terminated character string. The last pointer is NULL. A count of the number of arguments is also passed on. Recall that the actual syntax of main() is as follows:

```
int main(
     int argc,        /* Argument Count */
     char *argv[]     /* Array of Argument Strings */
)
```

The count “argc” does not include the NULL pointer that terminates the “argv” array. If the program does not take command line arguments we usually omit both the count and the array and write “void” instead.

In addition to the parameters of a main() function, there is also a global variable called “environ” that points to an array of environment strings also NULL terminated.

```
extern char **environ;   /* Environment Array */
```

Each argument string can be anything at all, as long as it’s NULL-terminated. But environment strings are more constrained. Each is in the form name=value with NULL byte after the value. To see the values of the current execution environment the following code can be used:

Run the program as: `./a.out 1 2 3 4 5`

```
# include<stdio.h>

extern char **environ;

int main(int argc, char *argv[])
{
int i;
```

```
printf("Arguments to this Program\n");
for(i=0;i<argc;i++)
{
      printf("%s\n",argv[i]);
}

printf("Environment Listing of this program\n");
for(i=0;environ[i]!=NULL;i++)
{
      printf("%s\n",environ[i]);
}
return 0;
}
```

The output of the above program gives a listing of all the environment variables. However most of the time, we would be interested in the value of a single variable. In that case, the getenv() function can be used as follows:

```
int main(void)
{
      char *s;
      s = getenv(“LOGNAME”);

      if(s==NULL)
            printf(“Variable Not Found\n”);
      else
            printf(“Value is %s \n”, s);

      return 0;
}
```

It is important to understand that the arguments hold user data (specific to a particular program) while the environment variables usually hold system data (not specific to any program)

Exec Family

Actually there is no system call named “exec”. The so-called “exec” system calls are a set of six, with names of the form execAB, where ‘A’ is either ‘l’ or ‘v’, depending on whether the arguments are directly in the call (list) or in an array (vector), and ‘B’ is either absent, a ‘p’ to indicate that the PATH environment variable should be used to search for the program, or an ‘e’ to indicate that a particular environment is to be used. Thus, the six names are

1. execl

2. execv
3. execlp
4. execvp
5. execle
6. execve

execl()system call

```
int execl (
const char *path, /* Program pathname */
const char *arg0, /* First Argument(filename) */
const char *arg1, /* Second Argument(optional) */
…                 /* Remaining Arguments (if any) */
(char *) NULL     /* Arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

The first argument, ‘path’ should contain the complete path of the executable program file (the path should also include the file name).

When execl() is called the context of the process is overwritten. The code\instructions of the process is now replaced with the instructions of the executable in ‘path’. The user-data of the process is also replaced with the data of the program in ‘path’, thereby reinitializing the stack.

After replacing the context, the new program begins its execution from the top, i.e., from its main function.

All of the other arguments except *'path'* are optional. In fact, even the first argument containing the file name is just a convention and not a mandate. All arguments other than *'path'* are collected into an array of character pointers: the last argument in the call, which should be a NULL stops the collection and terminates the array. The new program accesses these arguments via the *argc* and *argv* arguments of the main function (as shown in the preceding example).

The environment pointed to by *'environ'* is also passed to the new program and is accessible via its own *'environ'* pointer or with the *getenv()* function.

When execl() is called, the user-data and instructions change in the context of the process. However, system-data is not replaced completely. Certain system-data such as process-id, parent-process-id, current and root directory, priority, accumulated execution time do not change. The open file descriptors remain available even after a new program is loaded.

How can a execl() call send its return value? Recall whenever a systemcall or function is called, the return address is saved in the stack, which is a part of the process' context. After executing the function or system call the return address is popped from the stack. This mechanism cannot work for the execl() call because calling execl(), reintialises the stack with the data from the new program and hence there is no way to pop the saved return address from stack. This means that there can be no return from a execl() call, if the call has been successful. When the execl() fails, the stack would not have been re-intialised hence a error value of -1 is returned on failure.

execl() Example to invoke user executables

Before executing the code given below, create a simple hello world C program and save the executable as sample

```
# include<stdio.h>

int main(int argc, char **argv){

execl("./sample","sample","123",”abc”,(char *)NULL);
printf("hello from execdemo\n");

return 0;
}
```

It is important to note that the printf statement in the above program will never get executed if the execl() call has been successful. This is because the code of this program would have been replaced with that from “sample” when execl() call was issued.

execl() Example to invoke UNIX commands

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

int main(int argc, char ** argv){

printf("Hello World!");
execl("/bin/echo","echo","Print","from","execl",(char *)NULL);

return 0;
}
```

In the above program we are trying to invoke the executable corresponding to the UNIX “echo” command. The arguments to the echo command are – Print, from, execl. According to convention, the second argument is the executable file name, “echo”. The output of the above program would be equivalent to typing – “echo Print from execl” on the terminal prompt.

We would expect the program to print “Hello World” also. But it does not. This is because of the buffering nature of the printf statements. Any printf statement in a C program does not immediately print its contents. Instead it stores its contents in a buffer. The buffer contents are flushed/printed just before another printf is encountered or the program exits.

When execl() is called just after the printf, the buffer is replaced with data from the new program without flushing the previous data. One way to get around this problem is forcefully flush the buffer before calling execl(), as below :

```
printf("Hello World!");
fflush(stdout);
execl("/bin/echo","echo","Print","from","execl",(char *)NULL);
```

Other exec system calls

The other exec calls are very similar to execl(). They provide the following three features that are not available in execl().

1. Arguments can be put into a vector\array instead of explicitly listing them in the exec call. This feature is useful if the arguments are not known at compile time.
2. Searching for an executable using the value of the PATH environment variable. When this feature is used we don't have to specify the complete path in the exec call.
3. Manually passing an explicit environment pointer instead of automatically using *environ.*

execl: execute file with arguments explicitly in call
execv : execute file with argument vector
execlp: execute file with arguments explicitly in call and PATH search
execvp: execute file with argument vector and PATH search
execle: execute file with argument list and manually passed environment pointer
execve: execute file with argument vector and manually passed environment pointer

```
int execv (
const char *path, /* Program pathname */
char* const argv[] /* Argument vector */
);

int execlp (
const char *file, /* Program filename */
const char *arg0, /*First Argument(filename) */
const char *arg1,
…
(char *) NULL     /* Arg list terminator */
);

int execvp (
const char *file,  /* Program filename */
char* const argv[] /* Argument vector */
);

int execle (
const char *path, /* Program pathname */
const char *arg0, /*First Argument(filename) */
const char *arg1,
…
(char *) NULL,     /* Arg list terminator */
char *const envv[]  /* Environment vector */
);

int execve (
const char *path,  /* Program pathname */
char *const argv[], /* Argument vector */
char *const envv[]  /* Environment vector */
```

```
);
```

Exercise

1. When exec is called, the open file descriptors remain accessible to the newly loaded program. Read about fcntl() function and use that function to close all the open file descriptors before a new program is loaded.
2. Write a program that scans its arguments for assignments of the form *variable=value,* updates the environment accordingly and then executes the program specified by the first non-assignment argument. The other non-assignment arguments become arguments to the invoked program. Do not use fork().

3. The features provided by the six exec calls can be easily provided with just two system calls instead of six. Design and implement two functions execvx and execlx, to replace the six system calls. In the implementation any number of exec system calls can be used.