An Introduction to
"Executing Commands in UNIX"

The **fork(), execve(), and wait () system calls** are used in UNIX shell programs (also called command line interpreters) to execute commands. A shell program allows each interactive user to issue commands to the OS and the OS to respond directly to the user.

A shell program should be able to execute **any command** given to it even if the program that implements the command is buggy. Operating systems are built so that if a process is executing a program that contains a fatal error, then the process is terminated. If the shell process were to call the command code directly, and the command code contained a fatal error, then the shell process would terminate. At the user interface, you would call the bad command, and the shell would suddenly terminate! This can be avoided by designing the shell so that it creates a child process to execute each command. Then if the command execution fails, the OS will terminate the child process but the shell process will continue to execute. Now, if you call a bad command, the shell process can simply report the command failure and be ready to accept a new command.

Next we will design a program, called ***launch***, that behaves like a shell program. You can use this code as part of your solution to the UNIX shell project.

The *launch* program reads a list of commands from a file, then executes each command just as a shell would execute it. Each command line in the file will be the same as a line you would type to the shell, such as

ls

to list all the files in the current directory. If you wanted to list the files with the "long" listing that shows the permissions, owner, and the time that the file was last accessed, you would type

ls -1

The **syntax for a command line** is simple: When our launch program sees a command line such as

a.out foo 100

it will parse it, placing each "word" in the line into a string that will be kept in an array of strings, *char *argv[]*. This command line would be parsed so that

argv[0] = "a.out"
argv[l] = "foo"
argv[2] = "100"

You may recognize the *argv* name from writing C programs in your previous programming classes. For example, when you write a C program and you want the shell to pass parameters (from the command line) to your program, you declare the function prototype for your main program with a line like

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

In the shell case, when a user enters a command line of the form

a.out foo 100

it is intended to mean that a main program with a standard main header is to be executed with *argc* set to 3 (since there are three words on the command line), with the *argv[]* array initialized as shown above. The *a.out* main program will then interpret the first parameter (*argv[1]*) as, say, a file name, and the second parameter (*argv[2]*) as, say, an integer record count. When the shell passes these arguments to the *a.out* main program, it will treat *argv[1]* and *argv[2]* as strings.

Our command launch program is **required to read command lines from the file**, and then to parse them so that it has a value for *int argc*, and for *char * argv[]*. Here is a C data structure that we can use to save these arguments:

```
struct command_t {
        char *name;
        int argc;
        char *argv[MAX_ARGS];
} ;
```

We added the *char *name* field to the *struct*, since we are going to use it to keep the name of the file that contains the binary program for the command.
Of course this *char *name* is redundant since it is the same string as *argv[0]* (the first word on the command line). Here is a function that parses a command line, returning the result in the *struct command_t *cmd* argument:

```
#include <string.h>

/* Determine command name and construct the parameter list.  This function will build
   argv[] and set the argc value.

   argc is the number of "tokens" or words on the command line and argv[] is an array of
   strings (pointers to char *).

  The last element in argv[] must be NULL. As we scan the command line  from the left,
  the first token goes in argv[0], the second in argv[l], and so on. Each time we add a
  token to argv[], we increment argc.
*/
```

```
int parseCommand(char  *cLine, struct command_t  *cmd) {
        int argc;
        char **clPtr;
        /* Initialization */
        clPtr = &cLine;         /* cLine is the command line */
        argc = 0;
        cmd->argv[argc] = (char *) malloc(MAX_ARG_LEN);

        /* Fill argv[] */
        while((cmd->argv[argc] = strsep(clPtr, WHITESPACE)) != NULL)
        {
                cmd->argv[++argc] = (char *) malloc(MAX_ARG_LEN);
        }

        /* Set the command name and argc */
        cmd->argc = argc - 1;
        cmd->name = (char *) malloc(sizeof(cmd->argv[0]));
        strcpy(cmd->name, cmd->argv[0]);
        return 1;
}
```

Next, we will write the **main program** that executes the binary object program in the file identified by *argv[0]*. To do this, our (parent) shell process will create a child process, then use a form of the *execve()* system call to load and execute the named file. The parent process will "launch" the executable file in a child process so that even if the executable file contains a fatal error (which destroys the process executing it), the parent process can continue executing other commands.

Our program needs to know the name of a file, say *launch_set* that contains a list of command lines. For example, the *launch_set* file might contain the following lines:

```
date
gcc   main.c
mv   a.out   foobar
cd
ls -1
```

If the name of the *launch_set* file was passed to our program, it should execute the 5 commands using 5 different child processes. Suppose we pass the file name to our launch program as a shell argument, meaning that our launch program is executed with a shell command such as

```
launch  launch_set
```

Here is a plan for our program to solve the problem:

1. Read the command line parameter (*launch_set* in the example).
2. Open the file that contains the set of commands.
3. For every command:
    - Read a command from the file.
    - Parse the command so we know the name of the program and its arguments.
    - Create a new process to execute the command.
4. Terminate after all commands have finished.

Here is a solution that uses the *parseCommand()* function above. In this solution, we use a variant of *execve()* called *int execvp (char *file, char **argv)* (see the man page for *execvp()* for details).

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

#define     MAX_ARGS          64
#define    MAX_ARG_LEN    16
#define    MAX_LINE_LEN    80
#define   WHITESPACE        " .,\t\n"

struct command_t {
        char *name;
        int argc;
        char *argv[MAX_ARG];
} ;

/* Function prototypes */
int parseCommand(char *, struct command_t *);

int main(int argc, char *argv[])
{
        int i;
        int pid, numChildren;
        int status;
        FILE *fid;
        char cmdLine[MAX_LINE_LEN];
        struct command_t command;

        /* Read the command line parameters */
        if ( argc != 2) {
                fprintf(stderr, "Usage: launch <launch_set_filename>\n");
                exit ( 0);
        }
```

```
        /* Open a file that contains a set of commands */
        fid = fopen(argv[1], "r");

        /* Process each command in the launch file */
        numChildren = 0;
        while(fgets(cmdLine, MAX_LINE_LEN, fid) != NULL) {
                parseCommand(cmdLine, &command);
                command.argv[command.argc] = NULL;

                /* Create a child process to execute the command */
                if((pid = fork()) == 0) {
                        /* Child executing command */
                        execvp(command.name, command.argv);
                }
                /* Parent continuing to the next command in the file */
                numChildren++;
        }
        printf("\n\nlaunch: Launched %d commands\n", numChildren);

        /* Terminate after all children have terminated */
        for(i = 0; i < numChildren; i++) {
                wait(&status);
                /* Should free dynamic storage in command data structure */
        }
        printf("\n\nlaunch: Terminating successfully\n");
        return 0;
}
```

Notice that the first few lines of the main program check to be sure that the *launch_set* file name was passed to the program by the shell (using *argv[1]*). Then it opens the named file so that the main loop can process each command line in the file.

The while loop is executed once for each command in the file. It gets a command line (using a *stdio* library function, *fgets()*. Next it parses the command line to define the *argv[]* array and the name of the file containing the code for the command. The *execvp()* command requires that the *argv[]* array be terminated with a *NULL* pointer, so we do that next. The next couple of lines create the child process with *fork()*, and then direct the new child to use *execvp()* to execute the command. While the child is executing the command line, the parent increments the number of children it created, and then concurrently goes to the top of the while loop to launch the next command. If this launch program were executed with a *launch_set* containing 5 commands, then there could be 5 child processes and the parent process all executing concurrently.