- Owner UID
- Size
- Number of attaches

Project—UNIX Shell and History Feature

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface provides the user a prompt after which the next command is entered. The example below illustrates the prompt `sh>` and the user's next command: `cat prog.c`. This command displays the file `prog.c` on the terminal using the UNIX `cat` command.

```
sh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (i.e. `cat prog.c`), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to what is illustrated in Figure 3.11. However, UNIX shells typically also allow the child process to run in the background—or concurrently—as well by specifying the ampersand (`&`) at the end of the command. By rewriting the above command as

```
sh> cat prog.c &
```

the parent and child processes now run concurrently.

The separate child process is created using the `fork()` system call and the user's command is executed by using one of the system calls in the `exec()` family (as described in Section 3.3.1).

Simple Shell

A C program that provides the basic operations of a command line shell is supplied in Figure 3.25. This program is composed of two functions: `main()` and `setup()`. The `setup()` function reads in the user's next command (which can be up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. (If the command is to be run in the background, it will end with '&', and `setup()` will update the parameter `background` so the `main()` function can act accordingly. This program is terminated when the user enters `<Control><D>` and `setup()` then invokes `exit()`.

The `main()` function presents the prompt `COMMAND->` and then invokes `setup()`, which waits for the user to enter a command. The contents of the command entered by the user is loaded into the `args` array. For example, if the user enters `ls -l` at the `COMMAND->` prompt, `args[0]` becomes equal to the string `ls` and `args[1]` is set to the string to `-l`. (By "string", we mean a null-terminated, C-style string variable.)

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

#define MAX_LINE 80

/** setup() reads in the next command line, separating it into
distinct tokens using whitespace as delimiters.
setup() modifies the args parameter so that it holds pointers
to the null-terminated strings that are the tokens in the most
recent user command line as well as a NULL pointer, indicating
the end of the argument list, which comes after the string
pointers that have been assigned to args. */

void setup(char inputBuffer[], char *args[],int *background)
{
   /** full source code available online */
}

int main(void)
{
char inputBuffer[MAX_LINE]; /* buffer to hold command entered */
int background; /* equals 1 if a command is followed by '&' */
char *args[MAX_LINE/2 + 1]; /* command line arguments */

   while (1) {
      background = 0;
      printf(" COMMAND->");
      /* setup() calls exit() when Control-D is entered */
      setup(inputBuffer, args, &background);

      /** the steps are:
      (1) fork a child process using fork()
      (2) the child process will invoke execvp()
      (3) if background == 1, the parent will wait,
      otherwise it will invoke the setup() function again. */
   }
}
```

Figure 3.25 Outline of simple shell.

This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

Creating a Child Process

The first part of this project is to modify the `main()` function in Figure 3.25 so that upon returning from `setup()`, a child process is forked and executes the command specified by the user.

As noted above, the `setup()` function loads the contents of the `args` array with the command specified by the user. This `args` array will be passed to the `execvp()` function, which has the following interface:

```
execvp(char *command, char *params[]);
```

where `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0],args);` be sure to check the value of `background` to determine if the parent process is to wait for the child to exit or not.

Creating a History Feature

The next task is to modify the program in Figure 3.25 so that it provides a *history* feature that allows the user access up to the 10 most recently entered commands. These commands will be numbered starting at 1 and will continue to grow larger even past 10, e.g. if the user has entered 35 commands, the 10 most recent commands should be numbered 26 to 35. This history feature will be implementing using a few different techniques.

First, the user will be able to list these commands when he/she presses `<Control> <C>`, which is the `SIGINT` signal. UNIX systems use **signals** to notify a process that a particular event has occurred. Signals may be either synchronous or asynchronous, depending upon the source and the reason for the event being signaled. Once a signal has been generated by the occurrence of a certain event (e.g., division by zero, illegal memory access, user entering `<Control> <C>`, etc.), the signal is delivered to a process where it must be **handled**. A process receiving a signal may handle it by one of the following techniques:

- Ignoring the signal
- using the default signal handler, or
- providing a separate signal-handling function.

Signals may be handled by first setting certain fields in the C structure `struct sigaction` and then passing this structure to the `sigaction()` function. Signals are defined in the include file `/usr/include/sys/signal.h`. For example, the signal `SIGINT` represents the signal for terminating a program with the control sequence `<Control> <C>`. The default signal handler for `SIGINT` is to terminate the program.

Alternatively, a program may choose to set up its own signal-handling function by setting the `sa_handler` field in `struct sigaction` to the name of the function which will handle the signal and then invoking the `sigaction()` function, passing it (1) the signal we are setting up a handler for, and (2) a pointer to `struct sigaction`.

In Figure 3.26 we show a C program that uses the function `handle_SIGINT()` for handling the `SIGINT` signal. This function prints out the message "`Caught Control C`" and then invokes the `exit()` function to terminate the program. (We must use the `write()` function for performing output rather than the more common `printf()` as the former is known as being

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

#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];

/* the signal handling function */
void handle_SIGINT()
{
   write(STDOUT_FILENO,buffer,strlen(buffer));

   exit(0);
}

int main(int argc, char *argv[])
{
   /* set up the signal handler */
   struct sigaction handler;
   handler.sa_handler = handle_SIGINT;
   sigaction(SIGINT, &handler, NULL);

   /* generate the output message */
   strcpy(buffer,"Caught Control C\n");

   /* loop until we receive <Control><C> */
   while (1)
      ;

   return 0;
}
```

Figure 3.26 Signal-handling program.

signal-safe, indicating it can be called from inside a signal-handling function; such guarantees cannot be made of `printf()`.) This program will run in the `while(1)` loop until the user enters the sequence `<Control> <C>`. When this occurs, the signal-handling function `handle_SIGINT()` is invoked.

The signal-handling function should be declared above `main()` and because control can be transferred to this function at any point, no parameters may be passed to it this function. Therefore, any data that it must access in your program must be declared globally, i.e. at the top of the source file before your function declarations. Before returning from the signal-handling function, it should reissue the command prompt.

If the user enters `<Control><C>`, the signal handler will output a list of the most recent 10 commands. With this list, the user can run any of the previous 10 commands by entering `r x` where 'x' is the first letter of that command. If more than one command starts with 'x', execute the most recent one. Also, the user should be able to run the most recent command again by just entering 'r'. You can assume that only one space will separate the 'r' and the first letter and

that the letter will be followed by '\n'. Again, 'r' alone will be immediately followed by the \n character if it is wished to execute the most recent command.

Any command that is executed in this fashion should be echoed on the user's screen and the command is also placed in the history buffer as the next command. (`r x` does not go into the history list; the actual command that it specifies, though, does.)

It the user attempts to use this history facility to run a command and the command is detected to be *erroneous*, an error message should be given to the user and the command not entered into the history list, and the `execvp()` function should not be called. (It would be nice to know about improperly formed commands that are handed off to `execvp()` that appear to look valid and are not, and not include them in the history as well, but that is beyond the capabilities of this simple shell program.) You should also modify `setup()` so it returns an `int` signifying if has successfully created a valid `args` list or not, and the `main()` should be updated accordingly.

Bibliographical Notes

Interprocess communication in the RC 4000 system was discussed by Brinch-Hansen [1970]. Schlichting and Schneider [1982] discussed asynchronous message-passing primitives. The IPC facility implemented at the user level was described by Bershad et al. [1990].

Details of interprocess communication in UNIX systems were presented by Gray [1997]. Barrera [1991] and Vahalia [1996] described interprocess communication in the Mach system. Solomon and Russinovich [2000] and Stevens [1999] outlined interprocess communication in Windows 2000 and UNIX respectively.

The implementation of RPCs was discussed by Birrell and Nelson [1984]. A design of a reliable RPC mechanism was described by Shrivastava and Panzieri [1982], and Tay and Ananda [1990] presented a survey of RPCs. Stankovic [1982] and Staunstrup [1982] discussed procedure calls versus message-passing communication. Grosso [2002] discussed RMI in significant detail. Calvert and Donahoo [2001] provided coverage of socket programming in Java.