Lab 2. Simple Shell

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. 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.

Simple Shell

A C program that provides the basic operations of a command line shell is supplied below. 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()`.

(How to Create a First C Program on Linux: http://www.linfo.org/create_c1.html)

```
#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. */
```

```c
void setup(char inputBuffer[], char *args[],int *background)
{
     int length, /* # of characters in the command line */
          i,      /* loop index for accessing inputBuffer array */
          start,  /* index where beginning of next command parameter is */
          ct;     /* index of where to place the next parameter into args[] */

    ct = 0;

    /* read what the user enters on the command line */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);

    start = -1;
    if (length == 0)
        exit(0);            /* ^d was entered, end of user command stream */
    if (length < 0){
        perror("error reading the command");
        exit(-1);           /* terminate with error code of -1 */
    }

    /* examine every character in the inputBuffer */
    for (i = 0; i < length; i++) {
        switch (inputBuffer[i]){
        case ' ':
        case '\t' :               /* argument separators */
            if(start != -1){
                args[ct] = &inputBuffer[start];    /* set up pointer */
                ct++;
            }
            inputBuffer[i] = '\0'; /* add a null char; make a C string */
            start = -1;
            break;

        case '\n':                 /* should be the final char examined */
            if (start != -1){
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            args[ct] = NULL; /* no more arguments to this command */
            break;

        case '&':
            *background = 1;
            inputBuffer[i] = '\0';
            break;

        default :             /* some other character */
            if (start == -1)
                start = i;
      }
    }
    args[ct] = NULL; /* just in case the input line was > 80 */
}
```

```
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->");
     fflush(stdout);

     /* 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 == 0, the parent will wait,
         otherwise the parent will continue to the next iteration. */
  }
}
```

Outline of simple shell

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.)

Creating a Child Process

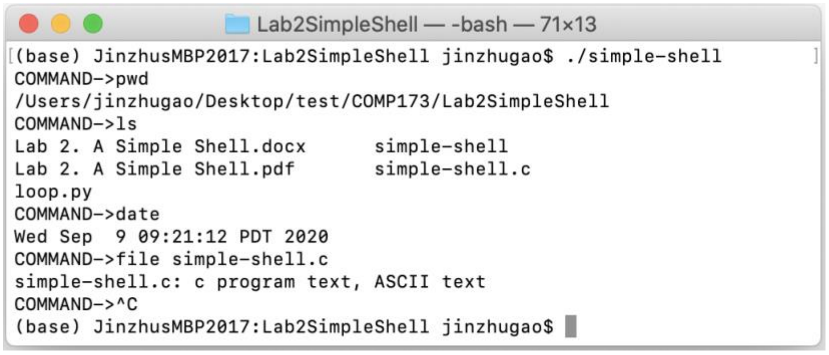
The major part of this project is to modify the `main()` function so that upon returning from `setup()`, a child process is forked and executes the command specified by the user.

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[]);
```

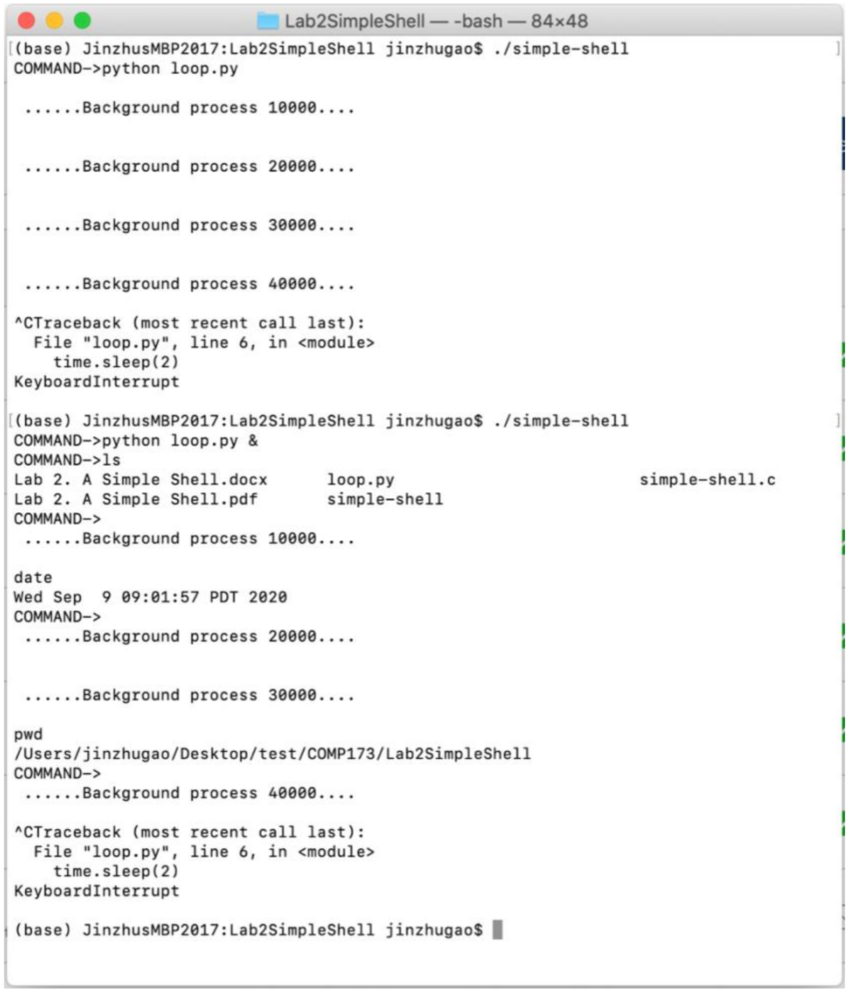
where `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0],args)`; be sure to check the value of `background` to determine if the parent process is to wait for the child to exit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

Sample Output:


Lab2SimpleShell — -bash — 71×13

```
(base) JinzhusMBP2017:Lab2SimpleShell jinzhugao$ ./simple-shell
COMMAND->pwd
/Users/jinzhugao/Desktop/test/COMP173/Lab2SimpleShell
COMMAND->ls
Lab 2. A Simple Shell.docx      simple-shell
Lab 2. A Simple Shell.pdf       simple-shell.c
loop.py
COMMAND->date
Wed Sep  9 09:21:12 PDT 2020
COMMAND->file simple-shell.c
simple-shell.c: c program text, ASCII text
COMMAND->^C
(base) JinzhusMBP2017:Lab2SimpleShell jinzhugao$
```

To verify that the parent process doesn't wait for the child process if the child process is a background process, you need to run an existing user/system program as a background process. For example, I wrote a simple python code with an infinite loop to test my simple shell by running it as a foreground or background process. Note that the screen output can be messy when both the parent process and the child process are running simultaneously.


Lab2SimpleShell — -bash — 84×48

```
(base) JinzhusMBP2017:Lab2SimpleShell jinzhugao$ ./simple-shell
COMMAND->python loop.py

 ......Background process 10000....


 ......Background process 20000....


 ......Background process 30000....


 ......Background process 40000....

^CTraceback (most recent call last):
  File "loop.py", line 6, in <module>
    time.sleep(2)
KeyboardInterrupt

(base) JinzhusMBP2017:Lab2SimpleShell jinzhugao$ ./simple-shell
COMMAND->python loop.py &
COMMAND->ls
Lab 2. A Simple Shell.docx      loop.py                         simple-shell.c
Lab 2. A Simple Shell.pdf       simple-shell
COMMAND->
 ......Background process 10000....

date
Wed Sep  9 09:01:57 PDT 2020
COMMAND->
 ......Background process 20000....


 ......Background process 30000....

pwd
/Users/jinzhugao/Desktop/test/COMP173/Lab2SimpleShell
COMMAND->
 ......Background process 40000....

^CTraceback (most recent call last):
  File "loop.py", line 6, in <module>
    time.sleep(2)
KeyboardInterrupt

(base) JinzhusMBP2017:Lab2SimpleShell jinzhugao$
```

Submissions:

- (80%) Well-commented source code.
- (20%) Write a report that
    - describes how to compile and run your program;
    - includes screenshots of your running program.

  Submit the report as a pdf file.