Northeastern University
College of Engineering
Department of Electrical & Computer Engineering

EECE7376: Operating Systems: Interface and Implementation

Homework 1

Instructions

- For programming Problems:
  1. Your code must be well commented by explaining what the lines of your program do. Have at least one comment for every 4 lines of code.
  2. At the beginning of your source code files write your full name, students ID, and any special compiling/running instruction (if any).
  3. Before submitting the source code file(s), your code must compile and tested with a standard GCC compiler (available in the CoE Linux server).
  4. Do not submit any compiled object or executable files.
- Submit the following to the homework assignment page on Canvas:
  1. Your homework report developed by a word processor (e.g., Microsoft Word) and submitted as one PDF file. For answers that require drawing and if it is difficult on you to use a drawing application, which is preferred, you can neatly hand draw "only" these diagrams, scan them, and insert the scanned images in your report document. The report includes the following (depending on the assignment contents):
     a. Answers to the non-programming Problems that show all the details of the steps you follow to reach these answers.
     b. A summary of your approach to solve the programming Problems.
     c. The screen shots of the sample run(s) of your program(s)
  2. Your well-commented programs source code files (i.e., the .cc or .cpp files).

  Do NOT submit any files (e.g., the PDF report file and the source code files) as a compressed (zipped) package. Rather, upload each file individually.

Note: You can submit multiple attempts for this homework, however, only what you submit in the last attempt will be graded. This means all required files must be included in this last submission attempt.

Problem 1 (25 Points)

Consider the following main program written in C:

```
int main(int argc, char **argv)
{
        print_args(argc, argv);
        return 0;
}
```

Function `print_args` is a function that returns no value (`void`) and takes the exact same arguments as function `main`. Internally, it traverses the array of input arguments given in array `argv` of length `argc` and prints them one by one. The following command assumes that your program was compiled into an executable file called `pr1`. The program should provide the following exact output:

```
$ ./pr1 a b c
argv[0] = './pr1'
argv[1] = 'a'
argv[2] = 'b'
argv[3] = 'c'
```

Test your program with more examples that cover different scenarios of the possible inputs. Write the content of function `print_args` and save the full source code for this program in a file named `hw1pr1.c`. You can compile it using command `gcc pr1.c -o pr1`, and run it as shown above.

Problem 2 (25 Points)

Write a C program that displays a prompt on the screen (with character '`$`'), reads a string from the user, and displays the exact same string when the user presses *Enter*. This process is repeated in an infinite loop, until the user kills the program by pressing `Ctrl+C`.

In order to read the string from the keyboard, use function `fgets` and pass `stdin` as its last argument, indicating that the standard input (keyboard) is the device from which the string should be read. Use the `man` pages to obtain full information about this function.

Notice that `fgets` will keep the *newline* character ('`\n`') introduced when pressing the *Enter* key at the end of the string. Get rid of it by replacing it with a null character ('`\0`'). Function `strlen` will be useful here. Here is an example for the program output:

```
./main
$ hello
hello
$ how are you
how are you
$ ^C                        ← Ctrl+C was pressed, program killed
```

Attach the full program in a file named `hw1pr2.c`

Problem 3 (25 Points)

Write a C program that creates a child process. The child and parent processes have the following roles:

a) The child process reads an integer value from the keyboard and returns it as the exit status of the program, either with a `return` statement in the `main` function, or with an invocation to function `exit`. The parent process waits for the child process to finish, and prints the child's exit status on the screen, which should equal the value read from the keyboard. Read the `man` pages for function `wait` to see the details on how to capture the child's exit status.

   This is an example of the program output:

   ```
   $ ./pr3
   Enter a number: 34              ← printed by child
   Child exited with status 34     ← printed by parent
   ```

   Attach the full program in a file named `hw1pr3a.c`

b) An orphan process is the one that keeps running while its parent is dead (i.e., finished its execution). Modify your `hw1pr3a.c` program so that the main program ends leaving the child as orphan. Make sure that the child code prints a message every 2 seconds indicating it is still alive. Use the command `ps -A` from another terminal to verify that the child process is running alone and then make sure to kill the orphan process using `killall <process name>`
   Attach the full program in a file named `hw1pr3b.c`

c) A zombie process is the one that dies (i.e., finishes its execution) but its parent does not check on it via `wait()`. Modify your `hw1pr3a.c` program so that the child ends executing its code while the parent is still running. Make sure that the parent code prints a message every 2 seconds indicating that it is still running. Use the command `ps -A` from another terminal to verify that the child process is a zombie (indicated by **<defunct>**). Make sure to kill your program processes using `killall <process name>`
   Attach the full program in a file named `hw1pr3c.c`

Problem 4 (25 Points)

Consider the following main program:

```
int main() {
	char s[200];
	char *argv[10];int argc;

	// Read a string from the user
	printf("Enter a string: ");
	fgets(s, sizeof s, stdin);

	// Extract arguments and print them
	argc = get_args(s, argv, 10);
	print_args(argc, argv);
}
```

From this code, you can deduce that function `get_args` takes the string entered by the user as its first argument and splits it into an array of arguments (assume we always have a maximum of 10 arguments). This array of arguments is then printed through the function `print_args`, implemented in Problem 1. This is the prototype of `get_args`:

```
int get_args(char *in, char **argv, int max_args);
```

Argument `in` is the input string. The function returns the number of arguments found in the string, and populates array `argv` with the arguments found, with a maximum of `max_args`. You can use these two functions from the C library (use `man` for more details on them):

- `strtok(char *str, const char *delim)`: splits a string into substrings (tokens) given a delimiter character, which in our case is the space character (`' '`).
  - On a first call, the function expects a *C* string as argument for `str`, whose first character is used as the starting location to scan for tokens.
  - In subsequent calls, the function expects a **null** pointer in `str` and uses the position in the original `str` right after the end of the last token as the new starting location for scanning.
- `strdup (const char *s)`: allocates a new memory space to store a duplicate string of `s` (as just copying a pointer is not the right way to copy a string.) Every string extracted from `in` should be duplicated before inserted in `argv`.

This is an example of the program execution:

```
$ ./pr4
Enter a string: hello how are you
argv[0] = 'hello'
argv[1] = 'how'
argv[2] = 'are'
argv[3] = 'you'
```

Attach the full program in a file named `hw1pr4.c`.