Northeastern University
College of Engineering
Department of Electrical & Computer Engineering

EECE7376: Operating Systems: Interface and Implementation

Homework 2

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 (30 Points)

The program below creates a parent and a child process, each of which prints a message five times into the standard output and makes sure that the `printf()` buffers are flushed after each call, using `fflush()`. If you run this program, you will probably observe that several messages from each process are printed at once before you see any message from the other process. The reason is that each process can print several messages before the OS performs a context switch.

```
int main()
{
	int pid = fork();
	int i;

	if (pid == 0)
	{
		// Child
		for (i = 0; i < 5; i++)
		{
			printf("%d. Child\n", i + 1);
			fflush(stdout);
		}
	}
	else
	{
		// Parent
		for (i = 0; i < 5; i++)
		{
			printf("%d. Parent\n", i + 1);
			fflush(stdout);
		}
		wait(NULL);
	}
}
```

Modify the code above to force the OS to perform a context switch after each message is printed, by synchronizing the parent and child processes with **two pipes**, one serving as a **parent-to-child** communication channel, and the other as a **child-to-parent** channel. At each synchronization point, a process can simply “write” one character (any character) to the other process through a pipe and “read” from the other pipe a character from the other process.

This is the exact, deterministic output that your program should provide (assuming your program executable file is named p1):

```
$ ./p1
1. Parent
1. Child
2. Parent
2. Child
3. Parent
3. Child
4. Parent
4. Child
5. Parent
5. Child
```

Attach the full program in a file named `hw2pr1.c`

Problem 2 (35 Points)

Add support for compound commands that use pipes to communicate two or more processes. Follow these steps:

a) Implement functions `ReadArgs` and `PrintArgs` as extension of the functions `get_args` and `print_args` implemented in the previous homework assignment. The new functions work with arrays that have their last element set to `NULL`. The new prototype for these functions will be the following:

- `void ReadArgs(char *in, char **argv, int size)`

  Argument `size` represents the number of elements allocated for `argv` by the function caller. The function should guarantee that the array is null-terminated under any circumstances, even if the number of tokens in string `in` exceeds `size`. Notice that this function does not return the number of arguments extracted from `in` anymore.

- `void PrintArgs(char **argv)`

  This function does not need the number of arguments to be passed to the function anymore. Instead, the function will stop printing arguments as soon as the `NULL` element is found.

b) The following data structures represent a command line, composed of multiple sub-commands separated by the pipes ("`|`" character). Each command has an array of at most `MAX_SUB_COMMANDS` sub-commands. Another field named `num_sub_commands` indicates how many sub-commands are present.

Each sub-command contains a field called `line` containing the entire sub-command as a C string, as well as a null-terminated array of at most `MAX_ARGS – 1` arguments (one array element is reserved for `NULL`).

```
#define MAX_SUB_COMMANDS    5
#define MAX_ARGS            10

struct SubCommand  {
        char *line;
        char *argv[MAX_ARGS];
};

struct Command {
        struct SubCommand sub_commands[MAX_SUB_COMMANDS];
        int num_sub_commands;
};
```

Write the following two functions:

- `void ReadCommand(char *line, struct Command *command)`

  This function takes an entire command line (in the `char *line` argument), and populates the `Command` data structure, passed by reference in the second argument (`struct Command *command`). The function body has two parts: First, the line is split into sub-strings with `strtok` using the "`|`" character delimiter, and each sub-string is duplicated and stored into the sub-command's `line` field. Second, all sub-commands are processed, and their `argv` fields are populated by calling `ReadArgs`.

- `void PrintCommand(struct Command *command)`

  This function prints all arguments for each sub-command of the command passed by reference. The function invokes `PrintArgs` internally.

Write a `main()` function that asks the user for an input string, and dumps all sub-commands and their arguments, by invoking the two functions above.

This is a sample run output of the program (assuming your program executable file is named p2):

```
$ ./p2
Enter a string: cat list.txt | sort | uniq
Command 0:
argv[0] = 'cat'
argv[1] = 'list.txt'

Command 1:
argv[0] = 'sort'

Command 2:
argv[0] = 'uniq'
```

Test your program with more examples that cover different scenarios of the possible inputs. Attach the full program in a file named `hw2pr2.c`

Problem 3 (35 Points)

Extend data type `struct Command` from the previous problem with the following fields:

```
char *stdin_redirect;
char *stdout_redirect;
int background;
```

Fields `stdin_redirect` and `stdout_redirect` are strings indicating file names to which the standard input of the first sub-command and the standard output of the last sub-command will be redirected, respectively. Remember operators `<` and `>` are used in the shell to redirect the standard input and output, respectively. If any of these redirections is not provided by the user, the corresponding field in the structure will take a value of `NULL`.

Field `background` is a flag indicating whether the process should run in the background. If the user adds the `&` operator at the end of the line, `background` should be set to 1. Implement the following functions:

a) `void ReadRedirectsAndBackground(struct Command *command);`

This function populates fields `stdin_redirect`, `stdout_redirect`, and `background` for the `command` passed by reference in the function argument. The function assumes that all other fields of the `command` structure have already been populated, as a result to a previous invocation to `ReadCommand`. The function should internally scan the arguments from the last sub-command in reverse order, extracting trailing `&`, `> file`, or `< file` patterns in a loop.

b) `void PrintCommand(struct Command *command);`

Extend this function to dump the value of `stdin_redirect`, `stdout_redirect`, and `background` for the command passed by reference in its first and only argument. Make sure that you consider the special cases where the redirection fields are set to `NULL`.

Write a main program that reads a command line from the standard input, creates the command object, and dumps all its fields, by invoking the previous functions.

This is an example of the execution of your program (assuming your program executable file is named p3):

```
$ ./p3
Enter command: a | b c > output.txt < input.txt &

Command 0:
argv[0] = 'a'

Command 1:
argv[0] = 'b'
argv[1] = 'c'

Redirect stdin: input.txt
Redirect stdout: output.txt
Background: yes
```

Test your program with more examples that cover different scenarios of the possible inputs. Attach the full program in a file named `hw2pr3.c`