Homework #2 Solution

Question 1 (4 pt.) Shell

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. This array of arguments is then printed through the `print_args` function, implemented in question 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:

- `strtoken`: splits a string into substrings given a delimiter character, which in our case is the space character (' '). See `man` pages for more details.
- `strdup`: allocates a new region of memory and duplicates a string. Every string extracted from `in` should be duplicated before inserted in `argv`.

This is an example of the program execution:

```
$ ./q3
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 `q3.c`, which should compile and run without modifications.

Solution (only function get_args shown)

```
int get_args(char *in, char **argv, int max_args)
{
      // Initialize number of arguments
      int argc = 0;

      // Extract arguments
      while (1)
      {
            // Get a token, and set 'in' to NULL for next time
            char *token = strtok(in, " ");
            in = NULL;

            // Done if no more tokens
            if (!token)
                  break;

            // One more token
            argv[argc] = strdup(token);
            argc++;

            // Check if maximum number of tokens was reached
            if (argc >= max_args)
                  break;
      }

      // Return number of arguments
      return argc;
}
```

Question 2 (6 pt.)

Write a program that runs commands `ls -l` and `wc`, where the standard output of the first command is connected to the standard input of the second. When each of these two processes finish, a message should be displayed saying `Process <pid> finished`, where `<pid>` is the process identifier of the finishing process. Note that the fact that we need to display a message when the child processes finish forces the parent process to keep its original process image until the end, that is, the parent process should not run `exec()`. You have two options:

1. Have the parent process spawn one child process executing command `ls -l`, and have the child process spawn a grandchild process executing `wc`.
2. Have the parent process spawn two child processes, one executing command `ls -l`, the other one executing command `wc`.

For the questions below where you need to attach text or diagrams, include one **single PDF file** and upload it on Blackboard. You can either export it from a word editor software, or scan hand-written answers.

a) For each of the options above, write a timeline, similar to those presented in class, where the interaction between the parent and child processes is illustrated. The timeline should include all system call invocations and the beginning and end of the lifetime of each process.

b) From options 1 and 2, which one do you think is preferable? Can you identify any advantages or disadvantages of one versus the other?

c) Choose only one of the options above and implement a full program in file `q2.c`. Upload this file on Blackboard. The output of your program should be something like this:

```
$ ./q2
      5      38     189
Process 3764 finished
Process 3765 finished
```

Solution

The solution shown below chooses option 2, where the parent process spawns two child threads. A disadvantage of this solution versus option 1 can be seen in the creation of the pipe. The parent process is creating the pipe, even though this process does not use it at all (you can see how the parent closes both ends of the pipe in the last `else` block). However, this is necessary in order for both children to inherit the same pipe and communicate through it. On the contrary, option 1 would allow the pipe to be created by the first child, which would be inherited directly by the grandchild process.

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

int main(int argc, char **argv)
{
	// Create pipe
	int fds[2];
	int err = pipe(fds);
	if (err == -1)
	{
		perror("pipe");
		return 1;
	}

	// Spawn child
	int child_pid1 = fork();
	if (child_pid1 == 0)
	{
		// Close read end of pipe
		close(fds[0]);

		// Duplicate write end of pipe in standard output
		close(1);
		dup(fds[1]);

		// First child launches command "ls -l"
		char *argv[3];
		argv[0] = "ls";
		argv[1] = "-l";
		argv[2] = NULL;
		execvp(argv[0], argv);
	}
	else
	{
		// Parent process launches second child
		int child_pid2 = fork();
		if (child_pid2 == 0)
		{
			// Close write end of pipe
			close(fds[1]);

			// Duplicate read end of pipe in standard input
			close(0);
			dup(fds[0]);

			// Second child launches command "wc"
			char *argv[2];
			argv[0] = "wc";
			argv[1] = NULL;
			execvp(argv[0], argv);
		}
```

```
        else
        {
            // Parent does not need pipe
            close(fds[0]);
            close(fds[1]);

            // Parent process waits for both children.
            int i;
            for (i = 0; i < 2; i++)
            {
                int child_pid = wait(NULL);
                printf("Process %d finished\n", child_pid);
            }
        }
    }
    return 0;
}
```