Homework #3

Question 1 (2 pt.) Shell

Modify functions `get_args` and `print_args` implemented in the previous homework assignment to work with null-terminated arrays, where the size of the array is defined by its first 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 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.

Question 2 (4 pt.) Shell

The following data structures represent a command line, composed of multiple sub-commands separated by 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 (first argument), and populates the `Command` data structure, passed by reference in the second argument. 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 (use calls to `readArgs`).

- `void printCommand(struct Command *command)`

  This function prints all arguments for each sub-command of the command passed by reference. The function can invoke `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. Name it `q2.c` and upload it on Blackboard. This is an execution example:

```
$ ./q2
Enter command: ls -l | wc | hello how are you

Command 0:
argv[0] = 'ls'
argv[1] = '-l'

Command 1:
argv[0] = 'wc'

Command 2:
argv[0] = 'hello'
argv[1] = 'how'
argv[2] = 'are'
argv[3] = 'you'
```

Question 3 (4 pt.)

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 carries out 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, the other as a child-to-parent channel. At each synchronization point, a process can simply send one character, no matter which one, to the other process through a pipe.

Upload your code in file `q3.c`. This is the exact (and deterministic) output that your program should provide:

```
$ ./q3
1. Parent
1. Child
2. Parent
2. Child
3. Parent
3. Child
4. Parent
4. Child
5. Parent
5. Child
```