Working with Processes

This assignment has two parts. Please submit one compressed file that contains all required files.

Part 1 Objective:

To pass information from the shell command line to a running program and to understand the operating system's role during this interaction.

Part 1 Description

Most of the utilities/commands in Linux are written in C. Many of these commands accept input(s) from the user, in the form of flags and arguments, and react differently to different inputs (e.g. gcc -o temp sample1.c). The OS also maintains a list of variables about the system running environment such as your home directory, terminal type, search path,… etc. The values are collectively known as the environment.

The OS reads the user input from the command line, creates standard structures, and passes them and the environment variables to the command. The programmer of this command can access these structures by declaring arguments in the definition of the main program. A common convention uses **`argc`** and **`argv, envp`** as follows:

```
void main(int argc, char **argv, char **envp) {….}
```

where:

- **`int argc`:** argument count, which is the number of tokens typed on the command line.
- **`char **argv`:** argument vector, which is an array of these tokens.
- **`Char **envp`:** environment variables and their values.

These variables (argc, argv, and **`envp`**) are now local variables the programmer can read and use to direct the execution of the command.

Submission:

Please compress all your source code files into one file and upload it to Blackboard. Name your files to easily reference different parts of the lab (e.g., "Part_1_1.c"; "Part_2_1_b.c") or follow the names in each part.

Part 1:

1-1. Print the arguments passed through the command line or the environment variables:

Your first deliverable is a C program (MyCommand1.c) that prints one of 2 outputs:

- If any arguments are passed to MyCommand1, it prints all the arguments that are passed (see figures below)

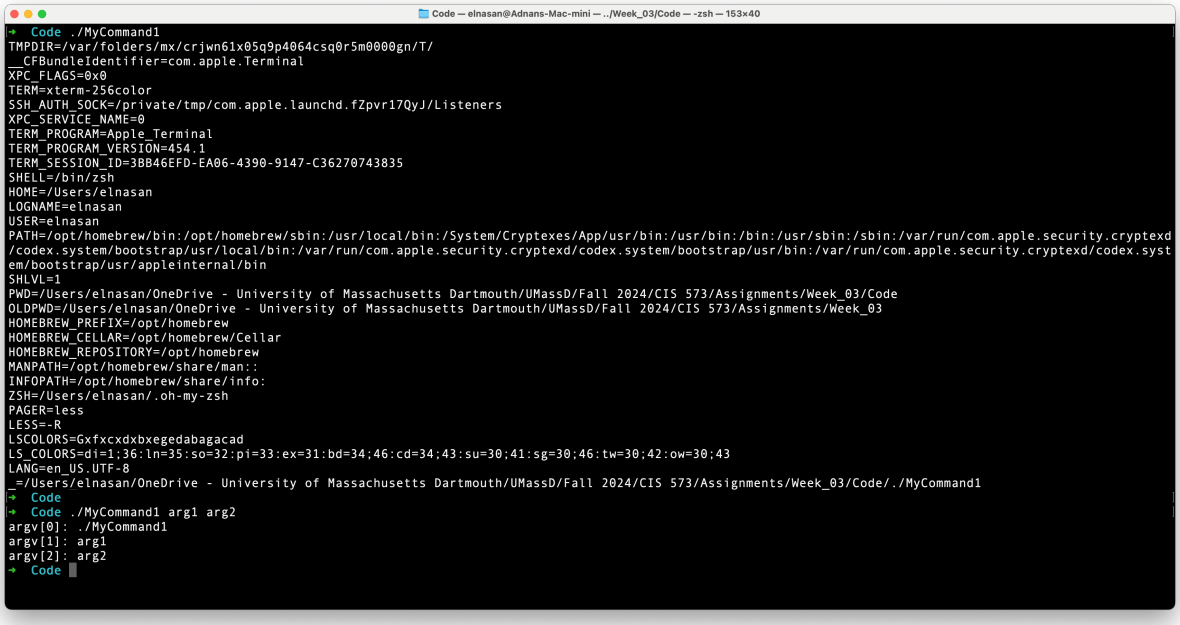
```
$MyCommand1 Arg1 Arg2

argv[0]: MyCommand1
argv[1]: Arg1
argv[2]: Arg2
```

```
$MyCommand1 Arg1 Arg2 Arg3 Arg4

argv[0]: MyCommand1
argv[1]: Arg1
argv[2]: Arg2
argv[3]: Arg3
argv[4]: Arg4
```

- If no arguments are passed to MyCommand1 then it prints all environment variables (see figure below)



1-2. Populate the values/strings passed through the command line into a fixed-size argument array

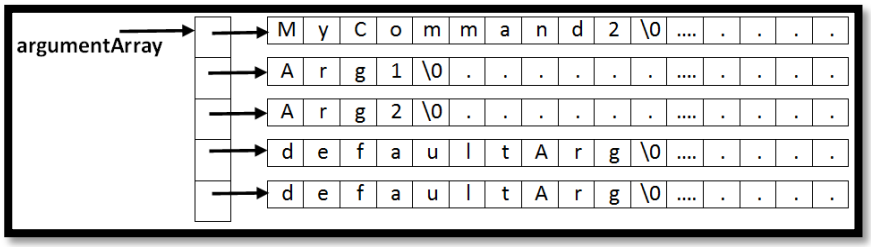
Your next deliverable is a C program (`myCommand2.c`) that:

- Takes strings as arguments.
- Creates a fixed-size argumentArray to hold the arguments passed through argv.
- Populates each entry in the fixed-size argumentArray with a default value “defaultArg”.
- Copies the individual arguments from argv into argumentArray, overwriting the default values.
- Prints the copied arguments and/or remaining default values in argumentArray.
- Assume:
  - argumentArray is defined as follows:

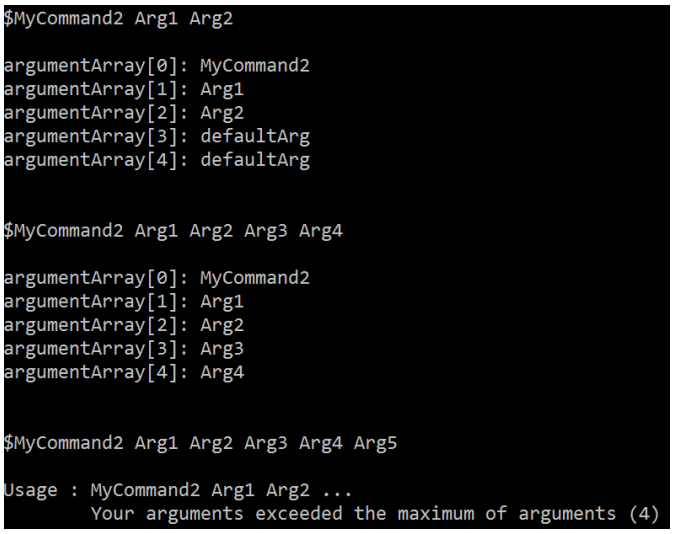
```
#define MAX_NUM_OF_ARGS 5
#define MAX_ARG_SIZE 256
char argumentArray[MAX_NUM_OF_ARGS][MAX_ARG_SIZE];
```

  - The first argument in the argument array is the command itself: (**`argumentArray[0]`** corresponds to **`argv[0]`**)

For example: **`$MyCommand2 Arg1 Arg2`** **//results below**



- Print an error message if the user enters too many arguments (see below)



Part 2 Objective

To work with Linux process management system calls and to understand the relation between processes.

Part 2 Description

Creating a process in modern operating systems usually includes creating a process control block (PCB) and loading an executable image of the new program into memory. To be able to pass arguments and return status information between processes, operating systems maintain organizational information that relates processes to each other. When a process terminates, its PCB is deallocated, and its memory is marked as available so the OS can reuse it.

Throughout the process lifetime, the OS keeps track of a process by maintaining its PCB which has many identifying information about the process. This includes the process ID (PID), its parent's ID (PPID), open file descriptors, the process's state and values of the CPU registers last time the process ran on the CPU. The PID, is a unique number that identifies the process throughout its lifetime while its parent's PID (PPID) is usually used to pass status information from a terminating child process to its parent. PIDs may be reused after a process terminates, but this usually happens after an extended period and/or when the system runs out of numbers to use.

Linux provides a parent-child relation between processes. A process can become a parent process by creating new processes, child processes, which can create their own child processes as well. The parent process ideally should wait for its child processes before it terminates so it gets back the return and/or status information from the child processes. Linux provides many system calls to manage processes throughout their lifetime and they include:

- `pid_t fork(void);` http://man7.org/linux/man-pages/man2/fork.2.html
- `pid_t vfork(void);` http://man7.org/linux/man-pages/man2/vfork.2.html
- `pid_t getpid(void);` http://man7.org/linux/man-pages/man2/getpid.2.html
- `pid_t getppid(void);`
- `pid_t wait(int *wstatus);` http://man7.org/linux/man-pages/man2/waitpid.2.html
- `pid_t waitpid(pid_t pid, int *wstatus, int options);`

- `unsigned int sleep(unsigned int seconds);` // library function
  http://man7.org/linux/man-pages/man3/sleep.3.html

File processing:

- `int open(const char *pathname, int flags);` https://man7.org/linux/man-pages/man2/open.2.html
- `int close(int fd);` https://man7.org/linux/man-pages/man2/close.2.html
- `int read(int fd, void *buf, size_t count);` https://man7.org/linux/man-pages/man2/read.2.html
- `int write(int fd, const void *buf, size_t count);` https://man7.org/linux/man-pages/man2/write.2.html

Part 2 Submission:

- *Submit your program(s) in text format so the TA can run them.*
- *Answers to the questions below in a separate file.*

2-1. Process Creation in Linux:

Run the code below in your shell and answer the questions that follow:

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

int main()
{
    int i;
    for (i = 0; i < 1; i++)
    {
        if(fork());
            printf("Hello 573 (from process %d)\n", getpid());
    }
    sleep(20);
    return 0;
}

```

Hint: Use shell command `pstree` to capture the relations between the different processes of parts a and b. Run two shells: on the first shell, use the command ps to get your shell process PID, and then run the code. On the other shell, run `pstree -p <pid>`

a. When the program runs, how many processes are created (including the program itself)? How many "Hello ...." are printed? Capture the output of pstree.
b. Change the value of "i" to 2. How many processes are created? And how many "Hello ...." are printed? Capture the output of pstree.
c. What is the relation between the number of consecutive fork calls and the total number of processes created? Capture the output of pstree.
d. Delete the ";" after the fork() statement and rerun parts a through c. Document the results. Why are the results different with and without the semicolon?

2-2. What gets inherited by child processes (Inheritance.c):

In Linux, a newly created process inherits many attributes from its parent including the parent's argument and environment variables and its open file descriptors. Update the skeleton program to test what attributes are inherited by a child process.

Please provide 3 different C programs, one for each test:

```
int printList(char **someList)
{
     int index = 0;
     while (someList[index] != NULL)
     {
          printf("%s\n", someList[index]);
          index++;
     }
     return index;
}

int main(int argc, char **argv, char **envp)
{…
}
```

- Test `argv`: use `fork` to create a new process which will print its copy of `argv`. The parent process should also print its own copy of `argv`. Run the program and direct the output to a temp file. Open the temp file and check if both child and parent processes print the same set of arguments. Make sure the parent waits for its child before it terminates.
- Test `envp`: use `fork` to create a new process which will print its copy of `envp`. The parent process should also print its own copy of `envp`. Run the program and direct the output to a temp file. Open the temp file and check if both child and parent processes print the same set of arguments. Make sure the parent waits for its child before it terminates.
- Test open file descriptors:
  - in the parent process, use a system call to create a new data file and open it for writing, then create a new process.
  - The child process should write to the file using the file descriptor that was created in the parent process.
  - The parent process should wait for its child then write to the file it created earlier.
  - After the parent process finishes writing, it should close the file and exit. After running your program, open the data file and verify the what's written.

(see below for what each process should print. What the child process prints should start with a [c] while what the parent process prints should start with a [p])

The run below shows PIDs when run on my machine.

```
        [c] this is process 38762 and my parent is 38761
[p] this is process 38761 and my parent is 37538
```