Rutgers ECE 434, Spring 2019 Prof. Maria Striki

Project 1: LINUX OS, Processes and Inter Process Communication

Issue Date: Friday 02-15-2019, Due Date: Tuesday March 5th 2019, 8.00pm

Total Points (18 + 18 + 14 = 50 points)

Problem 1 (Introduction) (18 points)

Part 1: File Vanished (3 points):
Let's assume that you have worked for many days now on a program named myfunc.c. While still working on it, you have accidentally issued some wrong commands or pressed certain combination of keys and you find that your file myfunc.c is lost. By looking into the history of your shell though, you find that the last command issued was:
`$ gcc –Wall –o myfunc.c myfunc.c`
Can you explain what has happened?

Solution:

Part 2: Concatenating to files into a third one (12 points):
You are asked to write a program that generates an output file, the context of which is derived by concatenating the contexts of two input files. Your program will be labeled myfiles and will take two or three arguments. The first and second arguments must be the input files, while the third argument must be the output file. The default name for the output file is myfile.out

Note 1 (6 pts): If your program is run without the proper arguments, you must make sure that the proper message appears on screen that guides the user into how many and what sort of arguments are required. If one of the input files provided in the arguments does not exist, then the program must print on screen the corresponding error message.

Note 2 (3 pts): Execute your program using OS instruction `strace` (research on how this works). Copy the output of `strace` that is produced from your code.

Note 3 (3 pts): Modify your initial code to support indefinite number of input files. The last argument though is always the output file.

For your implementation you are expected to use the following functions:
---- `void WriteInFile (int fd, const char *buff, int len)`: writes data to file descriptor fd.
---- `void CopyFile (int fd, const char *file_in)`: writes the contexts from file named file_in to file descriptor fd. WriteInFile is called from within CopyFile.

Below we provide one example of similar execution:

```
$ ./myfiles A1
Usage: ./myfiles file_in_1 file_in_2 [file_out (default:myfile.out)]
```

```
$ ./myfiles A1 A2
A1: No such file or directory

$ echo 'Data for file_in_1 This is OS 434 Sp19,' > A1
$ echo 'data for file_in_2 but also OS 579 Sp 19!' > A2
$ ./myfiles A1 A2
$ cat myfile.out
Data for file_in_1 This is OS 434 Sp19,
data for file_in_2 but also OS 579 Sp 19!

$ ./myfiles A1 A2 A3
$ cat A3
Data for file_in_1 This is OS 434 Sp19,
data for file_in_2 but also OS 579 Sp 19!
```

Solution:

Part 3: Using strace to dig deeper (3 points):
Let's work more on `strace` with objective to identify which system call is used to implement `strace`. Also consider the following: by typing:
`$ strace –o myfile`
The output of the instruction is set into file `myfile`.

Solution:

Problem 2: Introduction to multi-process environments and IPC (18 points)

Given a list of integers find the Minimum, Maximum, and the Sum of the numbers. The list will be in a text file you generate. You may use any form of inter-process communication (IPC) to partition the file/array into chunks and distribute work to more than one processes (if there are multiple ones) (e.g., pipes, shared memory, or additional (perhaps more sophisticated) inherent process system calls).
Remark: Record the time it takes for each of the programs to run and comment on your observations. Try it on lists of size 10,100, 1k, 10k,100k.
Hint: You should first load the file into an array then start working on the data.

Input Format:
Input will be in a text file. Each integer will be separated by a newline character (\n).

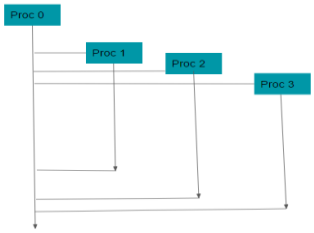
Exp:

100
20

Output Format: You should print out the results in a text file. Every process that is created should print out their own process id and their parent's process id. Then once you have computed the final max, min, and sum, you will print those out. Please follow this format:

Hi I'm process 2 and my parent is 1.
Hi I'm process 3 and my parent is 2.
Hi I'm process 4 and my parent is 2.
Max=50
Min=1
Sum=371

Part A (4 pts): Write a program to answer this problem using only one process.
Part B (6 pts) : Write a program to answer this problem using multiple processes where each process spawns at most one process. (Like DFS).



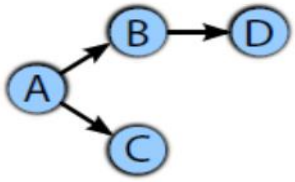
Scheme 1: Random Tree of processes

Part C (8 pts): Write a program to answer this problem using multiple processes where the first process spawns multiple processes and they (the children) spawn their own multiple processes, and so on and so forth. Your ultimate goal should be to produce such a hierarchy of processes (process tree) that produces the final result (compared to Parts A and B) faster. Can you create an arbitrary number of processes or are there any limitations? If you do find you have limitations in your version of Linux OS, please show (e.g. via a printscreen or a file) what are your exact limitations in increasing the number of processes you will be generating. Why is this so?

Solution:

Problem 3: (Building a given Process Tree) (14 points)

In this exercise you are asked to write a program which generates the following process tree (Scheme 1).



Scheme 1: A given process tree.

The processes that you generate must remain active for some considerable window of time in order for the user to be able to observe the tree. The leaf process executes a call to: `sleep()`. The internal process awaits of the termination of all its children processes. Every process prints a corresponding message every time it transitions to another phase (for example: start, loop to wait for children termination, allowing its own termination), so that the validation of the correct program operation is feasible. In order to separate the processes, please make sure that every process **terminates with a different return code**. In this example, one scenario can be:

`A = 2, B = 4, C = 6, D = 10.`

At this point, you may find helpful a number of auxiliary functions for process handling, such as those that:

1) have to do with identifying the circumstances under which a child process terminated (included in your ppt slides),
2) display the process tree starting from the root process (included in the appendix),
3) are related with different ways of recursively traversing a tree once the whole process tree is generated, etc.

Questions:

1. What happens if root process A is terminated prematurely, by issuing: `kill -KILL <pid>`?
2. What happens if you display the process tree with root `pstree(getpid())` instead of `pstree(pid())` in main()? Which other processes appear in the tree and why?
3. What is the maximum random tree you can generate? Why?

Solution:

What to turn in:

- C files for each problem
- A makefile in order to run your programs.
- Input text file (your test case)
- Output text file (for your test case)
- **Report**: Explain design decisions (fewer vs. more processes, process structure, etc.). Elaborate on what you have learned from each problem. Answer the question(s) below each part/subproblem. Also, please consider providing a very detailed report, as along with your C file deliverables, it corresponds to a substantial portion of your grade.

Logistics:

- For Project 1 please work in groups of 4-5 students.
- You are expected to work on this project using LINUX OS

- For those that do not have access to LINUX in their laptop, you may use one of the solution that I have already posted online regarding how to get access to a LINUX platform.
- Make ONE submission per group. In this submission provide a table of contribution for each member that worked on this project.
- Only students that may be left without peers will be allowed to work in groups of 2 or 3.
- Do not collaborate with other groups. Groups that have copied from each other will BOTH get zero points for this project (as a warning) no matter which copied from another, and will also incur more substantial consequences.

APPENDIX

Useful Links:

https://www.gnu.org/software/libc/manual/html_node/Generating-Signals.html#Generating-Signals

https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html#Pipes-and-FIFOs

https://www.gnu.org/software/libc/manual/html_node/Creating-a-Process.html#Creating-a-Process

https://www.gnu.org/software/libc/manual/html_node/Process-Completion.html#Process-Completion

https://en.wikipedia.org/wiki/Depth-first_search

Useful Auxiliary Functions and Definitions

explain_wait_status ()

```
void explain_wait_status(pid_t pid, int status)
{
  if (WIFEXITED(status))
    fprintf(stderr, "Child with PID = %ld terminated normally, exit status = %d\n",
         (long)pid, WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
    fprintf(stderr, "Child with PID = %ld was terminated by a signal, signo = %d\n",
         (long)pid, WTERMSIG(status));
  else if (WIFSTOPPED(status))
    fprintf(stderr, "Child with PID = %ld has been stopped by a signal, signo = %d\n",
         (long)pid, WSTOPSIG(status));
  else {
    fprintf(stderr, "%s: Internal error: Unhandled case, PID = %ld, status = %d\n",
         __func__, (long)pid, status);
    exit(1);
  }
  fflush(stderr);
}
```

Example:

```
pid = wait(&status);
explain_wait_status(pid, status);
if (WIFEXITED(status) || WIFSIGNALED(status))
         --processes_alive;
```

Example of handling SIGCHLD

```
void sigchld_handler(int signum)
{
        pid_t p;
        int status;

        /*
         * Something has happened to one of the children.
         * We use waitpid() with the WUNTRACED flag, instead of wait(), because
         * SIGCHLD may have been received for a stopped, not dead child.
         *
         * A single SIGCHLD may be received if many processes die at the same time.
         * We use waitpid() with the WNOHANG flag in a loop, to make sure all
         * children are taken care of before leaving the handler.
         */

        do {
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);
                if (p < 0) {
                        perror("waitpid");
                        exit(1);
                }
                explain_wait_status(p, status);

                if (WIFEXITED(status) || WIFSIGNALED(status))
                        /* A child has died */
                if (WIFSTOPPED(status))
                        /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
         } while (p > 0);
}
```

Auxiliary Functions and Operations on Trees and Tree Nodes

```
struct tree_node {
        unsigned          nr_children;
        char              name[NODE_NAME_SIZE];
        struct tree_node  *children;
};
```

```
static void
__print_tree(struct tree_node *root, int level)
{
        int i;
        for (i=0; i<level; i++)
                printf("\t");
        printf("%s\n", root->name);

        for (i=0; i < root->nr_children; i++){
                __print_tree(root->children + i, level + 1);
        }
}

void
print_tree(struct tree_node *root)
{
        __print_tree(root, 0);
}
```

How to write a MakeFile (Example):

```
$ cat Makefile
# a simple Makefile

CC = gcc
CFLAGS = -Wall -O2

all: fork-example

fork-example: fork-example.o proc-common.o
 <Tab> $(CC) -o fork-example fork-example.o proc-common.o

proc-common.o: proc-common.c proc-common.h
 <Tab> $(CC) $(CFLAGS) -o proc-common.o -c proc-common.c

fork-example.o: fork-example.c proc-common.h
 <Tab> $(CC) $(CFLAGS) -o fork-example.o -c fork-example.c

clean:
 <Tab> rm -f fork-example proc-common.o fork-example.o
```

```
$ make
gcc -Wall -O2 -o fork-example.o -c fork-example.c
gcc -Wall -O2 -o proc-common.o -c proc-common.c
gcc -o fork-example fork-example.o proc-common.o
```