Rutgers ECE 434, Spring 2022 Prof. Maria Striki

Project 1: LINUX OS, Processes and Inter Process Communication

Issue Date: Friday 03-05-2022, Due Date: Wed March 23th 2022, 8.00pm

Total Points (18 + 18 = 36 points)

Very useful resources for writing C code:

https://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm
https://www.cprogramming.com/tutorial/c-tutorial.html
Programming Language: by Brian W. Kerninghan, Dennis M.Ritchie

NOTE: You are to submit 2 out of the 3 items for a MAXIMUM of 36 points. You may choose to do two project questions or one project and one presentation question. For the presentation question please read first: research project ideas and guidelines.

Problem 1: Introduction to multi-process environments and IPC (18 points)

Problem Introduction and Description: Generate a text file and populate it with L (>=3,000) positive integers, randomly generated. Identify the MAXIMUM, the AVERAGE, and a number of "H" GIVEN HIDDEN KEY INTEGERs **out of the 10 hidden keys randomly placed in the file**. Use pipes (or shared memory – if you are familiar with it) to partition the file/array into chunks and distribute work to more than one processes.

You must "hide" your secret keys into your list/file. The hidden keys will be **5 negative integers** (with value -1) inserted at random positions into your file. But then, after creation of the file, you "forget" where the hidden keys are located. You only know that you must look for "H" secret keys.
Arguments L and H are provided at run time and are handled by `main (int argc, char *argv[]).`
You can also ask this information from the keyboard and have your program read/scanf the information entered from the user. (how to use argc-argv[] -- read the related section on tutorialspoint site provided above).

Objective: Use multiple processes "PN" (where PN: Process Number argument), use different process layout strategies (DFS, tree: see next page), and IPC to speed up your search on the Maximum element, the Average element, and the "H" hidden elements. You can get "PN" from the arguments list at run time (or read it from the user's input later). Impose that your program generates no more than PN processes.

Remark: Record the time it takes for each of the programs to run and comment on your observations. Try it on lists of size L: 5k, 10k,100k, 1M.
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). Example:

```
100
20
35
```

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, avg, and hidden keys, 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, Avg =36 (fictitious numbers).

Hi I am process 3 and I found the hidden key in position A[65].
Hi I am process 7 and I found the hidden key in position A[123].
Hi I am process 12 and I found the hidden key in position A[204].
Hi I am process ….  (all the above are fictitious numbers).
```

Detailed Instructions on Processes Layout (14 points):

1) Write a program to solve the problem using only one process.
2) Write a program to solve the problem using multiple processes where each process spawns at most one process. (Like DFS). The maximum number of processes spawned should be NP.
3) Write a program to solve the problem using multiple processes where the first process spawns a fixed number of processes, say X, and the children spawn their own X multiple processes each, and so on and so forth. The maximum number of processes spawned should be NP. Your tree structure may be pre-defined and hence avoid using loops or recursion or counters to create a new tree. You also have the option to ask the user to provide X. At the end, you must produce at least 2 varying tree structures using X and compare with each other to address all questions.

4) Ensure that in all cases you are using the appropriate system calls for IPC and for making the processes communicate and terminate gracefully, and without zombies.
5) Your ultimate goal should be to find the process tree that produces the final result faster. Can you fine-tune the size of variable NP to do that? Consider lists of increasing sizes. There is a trade-off w.r.t. to the number of processes but there is also a trade-off w.r.t. the number of integers in the file (and increasing file sizes). What are the trade-offs you observe here?
6) Can you create an arbitrary number of processes or are there any limitations? If you find 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?
7) What is the maximum number of processes you can create before your system crashes?

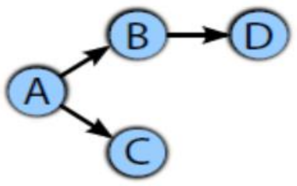
Variation of the Program – only identify the first 3 hidden elements (4 points):

Now your task is to identify only the first 3 hidden elements, not to compute the maximum and or average element. Identify and implement the changes you can make to your program that **improve its speed and ensure the following operation**: as soon as the hidden keys are identified, **your program terminates gracefully**, **without creating zombies or orphans**. You must use the proper system calls.

Solution:

Problem 2: (Building a given Process Tree) (18 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.`

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:

Problem 3: (**Research Project no 1 - Presentation**) **(18 points)**

In this question, you are asked to read the guidelines and ideas on selecting a Research topic and presenting it. You are given the freedom to work on topics that are outside the list of proposed topics. Only in that case you need to register your presentation topic using the course official online google doc in order to be approved (or not) to continue. You are to do detailed research on the topic of your selection, and prepare slides and video/recording of your presentation. You should opt for a 20-30 minute presentation/recording and your slides must be tailored accordingly.

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:

- You will work within your defined group.
- You are expected to work on this project using LINUX OS
- Make ONE submission per group. In this submission provide a table of contribution for each member that worked on this project.
- Do not collaborate with other groups. Groups that have copied from each other will BOTH get zero points for this project.

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 ()

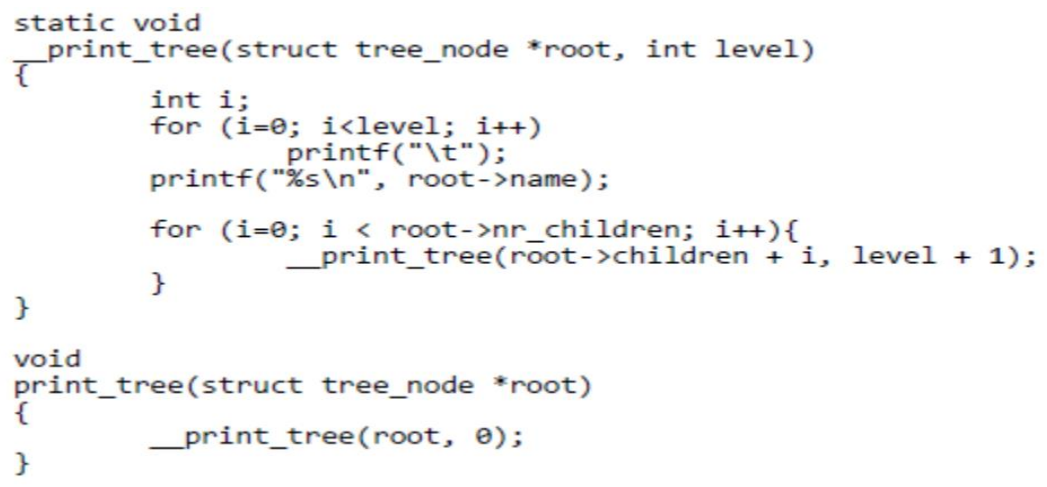
```
void explain_wait_status(pid_t pid, int status)
{
  if (WIFEXITED(status))
    fprintf(stderr, "Child with PID = %ld terminated normally, exit status = %d\n",
          (long)pid, WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
    fprintf(stderr, "Child with PID = %ld was terminated by a signal, 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;
```

Auxiliary Functions and Operations on Trees and Tree Nodes

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

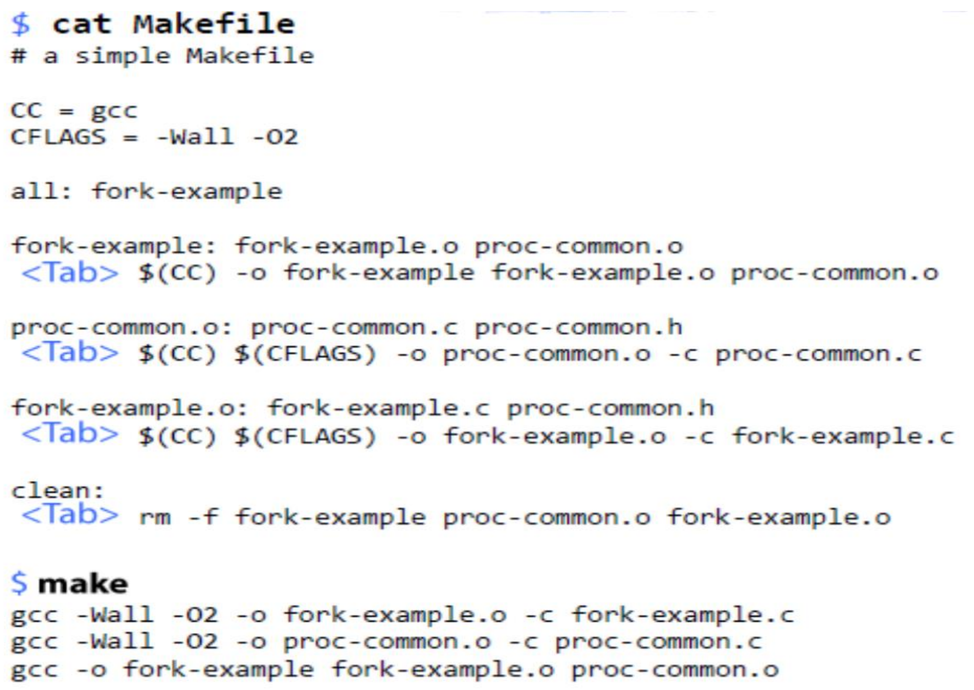


```
static void
__print_tree(struct tree_node *root, int level)
{
        int i;
        for (i=0; i<level; i++)
                printf("\t");
        printf("%s\n", root->name);

        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
```