Rutgers ECE 434, Spring 2019 Prof. Maria Striki

Project 2: LINUX Inter Process Communication and Signals

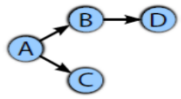
Issue Date: Monday 03-11-2019, Due Date: ~~Sat March 30th~~ Monday Apr 8th 2019,

Total Points (20 + 20 + 10 + 40 = 90 points)

Problem 1 (Arbitrary Process Tree generation) (20 points)

Write a program which generates arbitrary process trees from a given input file. Your program is based on a recursive function which will be called for every tree node. If the tree node has children, the function will create them and will wait until they are terminated. If the tree node does not have children, the function will call `sleep()` with a predefined argument.

The input file contains the description of a tree, node after node, starting from the root. For every node, you must specify its name, the number of children and the names of the children. You must decide how to represent the nodes and their names in your input file so that they can be uniquely mapped to the tree under consideration. For instance, how to distinguish between DFS vs. BFS nodes?



Scheme 1: Example of a process tree.

Under one scenario, as an example, a process tree may be described as follows:

```
A 2 B C
B 1 D
D 0
C 0
```

Remark 1: Ensure you do not generate more processes than what your system can handle. So, please justify in your report, what the size of the process tree you selected is and why.

Remark 2: Every tree node is defined by struct: `tree_node`, which contains the number of children (*`children_no`*), and the name of the node and pointer to the area where contiguous *children_no* structs are placed, one for every child node. Remark 2 helps understand how to represent the remaining tree when stored in the memory of an internal process tree node. You should not pass part of the original file as "file", but pass only the amount of information required stored as linked list or some other structure.

Remark 3: Write a function that reads the tree from file, constructs its representation in memory and returns a pointer to the root: `read_tree_file (const char *filename)`. Also write a function that runs the tree starting from *root* and prints its elements: `print_tree (struct tree_node *root)`.

Remark 4: You must build the process tree, not just a tree data structure. However, for every process tree node you are building you must pass on the information of the remaining sub-tree recursively. Also, you must print out the tree after creation using the process tree. For this, you will need some form of IPC for interaction across processes (go up or down the tree). You may choose to print in a recursive fashion.

Question: What is the order of appearance of start and termination messages from processes and why?

Solution:

Problem 2: (**Handling and Sending Signals) (20 points)**

Expand the previous problem so that the processes can be handled via the use of signals, in order to print their messages in a depth-first fashion. Every process creates its own children-processes and suspends execution until it receives the appropriate resume signal (SIGCONT). When a process receives SIGCONT it prints the corresponding message and activates its child processes one after another. It monitors and waits for their terminations and prints the corresponding diagnostics. The process evolves recursively, starting from the root process (what type of traversal?). The initial process of the program sends SIGCONT to the root process, after displaying the process tree to the user.

In the example of scheme 1, the activation messages are printed in order: A, B, D, C

Remarks: A process suspends its execution with the use of sys call: `raise (SIGSTOP)` (if raise does not work check to find what alternative system call you may equivalently use). But before it does so, it validates that all the children have suspended their execution as well. This validation is conducted by means of a function named: `wait_for_children()`. To build the latter you may use function `explain_wait_status()` following sys call `wait()` or `waitpid()`.

During your program execution you may send manual messages via the use of instruction `kill` from another window. You may use the command: `strace –f –p <pid>` to monitor the system calls coming from process `<pid>` and all its children processes.

Questions:

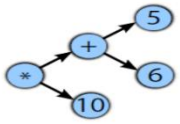
Q1: We have used `sleep()` in the previous parts to synchronize processes. What is the advantage of the use of signals?

Q2: What can the role of function `wait_for_children()`be? What benefit does it ensure and what potential problem could its omission bring about?

Solution:

Problem 3: (**Parallel calculation of numerical expression**). **(10 points)**

Expand the program of Problem 1 for the purpose of computing trees that represent numerical expressions. For example, the tree of schema 2 represents the expression: `10 x (5+6)`.



Scheme 2: Process tree for numerical expression: `10 x (5+6)`.

The provided input files and hence the process tree that are derived have the following limitations:

1) Every internal node has exactly two children and represents one of the two operands: “+” / “*”.
2) The name of every leaf node is an integer number.

The evaluation of the expression will be conducted in parallel, creating one process for every tree node. Every leaf process return to the parent process the numerical value that corresponds to it. Every non-leaf process receives the evaluation of the sub-expressions coming from its children processes, calculates the value corresponding to its own node and returns the value to its parent process. The root process returns the final result to the initial process of the program, which prints/displays it on the screen.

Test Output: Among the random expressions you are to generate, you should also generate the following and provide the proper output:

```
10 x [(2 x (5+6)) + (3 x (2+3))] x [[(4 x (8+5)) + (5 x (2+4))]
```

Remark 1: You must use Linux pipes for the communication between parent and children processes. Every process must monitor its children for their termination and print the corresponding diagnostic in order to detect programming errors in a timely manner. Intermediate printouts facilitate debugging.

Remark 2: You have the option to utilize different forms of IPC every time if you want. An obvious method for IPC is pipes. Can you identify more? Do you want to use them? If not, why not?

Question: How many pipes per process do we need to use in this problem? Would it be possible for every parent process to use only one pipe for all the children processes? Or in general, can we use only one pipe for every numerical operand?

Solution:

Problem 4: (**Project 1 – Problem 1 with Signals**). **(40 points)**

This problem is a slight elaboration of your Project 1- Problem 2 and Project 2 - Problem 1.

Please expand your Project 1/Problem 2 – Part C to do the following tasks: **(16 pts)**

a) **(4 marks)** The IPC is now to be conducted with signals and signal handlers (no pipes or shared Memory). Review the available signals and find how to pass information from the parent to the child and vice versa through signals/signal handlers. Rewrite and compile the code appropriately.

b) **(12 marks)** Your parent process becomes now very impatient…. It does not want to wait for any child process longer than 3+6 x ll seconds (where ll is iteration of computation from the parent to the farthest descendat: ll=0 at the leaf processes and increases by one, once we go to the immediate parent level). If a child process takes longer than this, do not incorporate its contribution to the final result if the child process has a faster responding sibling.
What should the contribution get substituted with in each case?

The parent should "mark" the slow behaving processes and "notify" all its children processes (if any) about "who" this misbehaving sibling process(es) is(are). Once a child process is notified of a misbehaving sibling process, it attempts to "suspend" it EITHER by sending the corresponding signal OR by sending a custom signal with a customized handler that emulates/executes suspension but also updates a data structure that contains the `pid` of the process which executed the handler (if feasible). At time 3 + 6 x ll + 2 seconds, the parent checks on the status of the misbehaving child(ren) and if implemented, on the updated data structure of the reporter siblings.

If the status of the misbehaving child(ren) is suspended, then the contribution of that child is not incorporated to the parent's result indeed. If in addition, the data structure of the reporter siblings is populated with more than 2 elements then the parent additionally terminates that child. If the status of the misbehaving child is not yet suspended, then at time 3 + 6 x ll +4 seconds, the parent checks for an actual contribution from that child and sends it to its own parent, otherwise, as discussed, the contribution of this child is not considered. However, the parent does not terminate the child. When the root of the overall tree is reached, print the count of the terminated processes and the count of the live processes. Then, have the root invoke sys call wait to release resources of live processes.

Question: How many process resources will be released? Why?

Solution:

Please expand your Project 2/Problem 1 to do the following tasks: **(24 pts)**

Change the random process tree you have used before to a random tree of threads instead. Therefore, modify your code to generate threads instead of processes. Can you actually create a tree of threads? Or if not, can you find a way to make threads have such a relationship by having one thread hold the proper info w.r.t. to the remaining threads (i.e., other threads ids, children threads ids, parent threads ids, etc etc)?

c) **(24 marks)** The user running this program got very impatient while waiting for all these calculations to be conducted, and decides to be done with the whole experiment by playing around with various signals and/or interrupts, and in particular with the following: i) CTL-C, ii) SIG_QUIT, iii) SIG_STOP, iv) SIG_STP, v) SIG_ABRT, vi) SIGTERM, vii) SIG_KILL, viii) SIG_EGV, viii) select your own interrupt with your customized handler.

Experiment 1) (4 pts) Send manually the above sequence of signals to the process (main function), one signal every time. Record the messages you get on the screen and describe the behavior of your program. One way to check on the status of your threads (if your process is still alive) is to ask the threads send printouts with their id and status. Those that do are still alive and unblocked.

Experiment 2) (6 pts) Now send the above signals to a specific thread id (have the process or another thread send the signals to a specific thread, one by one every time and observe. Experiment with each signal at a time, record the messages you get on the screen and describe the behavior of your program. Does your program completely stop at all times, for all signals? Does one thread only get suspended or stopped or all the threads or process do? Report your findings. One way to check on the status of your threads (if your process is still alive) is to ask the threads send printouts with their id and status. Those that do are still alive and unblocked.

Experiment 3) (14 pts) Modify and run your program so that it accommodates the options below: Choose three threads to send those signals to from another thread. Omit this time around SIG_STOP and SIG_KILL. The thread will be sending all the six signals to each other thread 5 times in a row (write a loop of 5 iterations in which you will be sending all 6 signals in every iteration). The thread that receives the signals will do the following:

1) **(4 pts)** Has defined its own handler for every signal that mainly updates the thread status, prints on screen the number of incoming signal, the id of the sender, the receiver, and if

possible the number of times this signal currently sent. Use function signal() to run the handler, and do not re-install the handler within signal. What is the behavior of your program? How many times is each signal in each iteration going to run?

2) **(4 pts)** Now do exactly as above, use function `signal()` for the handler but re-install the handler within the handler. Run the program as prescribed and printout the results. Are there any differences in execution 2) with execution 1)? Describe in detail, provide evidence, justify why.

3) **(6 pts)** Now do exactly as above, also add in the handler of every signal, along with the rest of the instructions, a system call `sleep(3)`. However, you are to use function `sigaction ()` to define and install the handler. Set the `sa_mask` to block the last three signals (SIG_ABRT, SIG_EGV, your own custom signal). What do you observe when running your code now? What are the differences? Can you justify your differences by explaining how `sa_mask` operates? Run the program as prescribed and printout the results.

Please provide detailed code for this problem (you have ample degree of freedom) and a very thorough report with your results and your justifications of the results.

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 2 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 solutions 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
```