Process Management and Interprocess Communication

1.1 Creating a given process tree

It is requested to build a program that creates the process tree of Fig. 1. The processes are created and kept active for a certain period of time, so that the user has the ability to observe the tree.

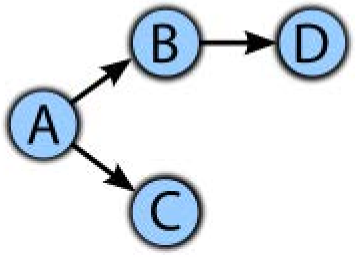

Figure 1

The processes-sheets execute the sleep () call, while the processes in intermediate nodes wait for the end of their child-processes. Each process prints an appropriate message each time it goes from phase to phase (eg, start, wait for children to finish, quit), so that the correct operation of the program can be verified. To separate the processes, each will end with a different return code: A = 16, B = 19, C = 17, D = 13.
Auxiliary code files are provided in the / home / oslab / code / forktree directory:
-fork - example.c: example of using fork (), wait ()
-proc - common. {h, c}: auxiliary functions for handling processes. The explain_wait_status () function prints an appropriate message depending on the return status of wait (). The show_pstree () function displays the process tree with a given process root.
-ask2 - fork.c: framework of the program, where you can start. Builds a child process (the root of your tree), waits for the process tree to build (sleeps for a while) and calls show_pstree ()
Questions
1. What if you terminate process A early by giving kill -KILL <pid>, where <pid> its Process ID?
2. What if you do show_pstree (getpid ()) instead of show_pstree (pid) in main ()? What additional processes appear in the tree and why?
3. In multi-user computer systems, the administrator often sets limits on the number of processes that a user can create. Why?

1.2 Creating an arbitrary process tree

It is required to build a program that creates arbitrary process trees based on an input file. The program will be based on a retrospective function, which will be called for each node of the tree. If the tree node has children, the function will create children and wait for them to terminate. If not, the function will call sleep () with a predefined argument.
The input file contains the description of a node-to-node tree, starting at the root. Each node is given its name, the number of its children and their names. The description of one node from the next is separated by a blank line. The process is repeated retrospectively for each of his children,

while the nodes must be arranged in depth. For example, the process tree of Fig. 1 emerged from the input file of Fig. 2. The ↩ symbol indicates a line break.

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

Figure 2

You are given a library tree. {H, c} which undertakes to read the input file and construct its representation in memory. Each node is defined by the struct tree_node structure, which contains the number of nr_children children, the node name, and a pointer to an area where nr_children structures are stored continuously, one for each child node. The library offers two functions:
-get_tree_from_file (const char * filename): reads the tree, constructs its representation in memory and returns a pointer to the root.
-print_tree(struct tree_node *root) : διατρέχει το δέντρο με ρίζα root και εκτυ-
πώνει τα στοιχεία του στο standard output.
Tree - example.c contains an example of using the library. It reads the log file, constructs the tree and displays it on the screen. To display it, run it following the children pointers of the struct tree_node structures.
Questions
1. In what order are the start and end messages displayed? Why?

1.3 Sending and handling signals

It is requested to extend the program of § 1.2 so that the processes are controlled using signals, to print their messages in depth (Depth-First). Each process creates its own child processes and suspends its operation until it receives an appropriate start signal (SIGCONT). When it receives it, it displays an informational message and activates the child processes in sequence: it wakes up a process, waits for it to end, and prints a corresponding diagnostic. The process evolves retroactively starting from the root process of the tree. The initial process of the program sends SIGCONT to the root process, after presenting the process tree to the user.
In the case of the tree of Fig. 1, the activation messages are printed in the order A, B, D, C.
Suggestions:
-A process inhibits the use of the raise command (SIGSTOP).
-Before doing so, it has verified that all its children have suspended their operation using the wait_for_ready_ children () function given to you. Use the explain_wait_status () function after wait () or waitpid ().
-While developing the program you can send messages manually, using the kill command from another window. Use the strace -f -p <pid> command to monitor system calls made by the <pid> process and all its children.
-Start with the ask2 - signals.c framework given to you.
Questions
1. In the previous exercises we used sleep () to synchronize the processes. What are the advantages of using signals?

2. What is the role of wait_for_ready_children ()? What does its use ensure and what problem would its omission create?

1.4 Parallel calculation of numerical expression

It της 1.2's program extension is required to compute trees representing arithmetic expressions. For example, the tree in Fig. 3 represents the expression 10 × (5 + 6).

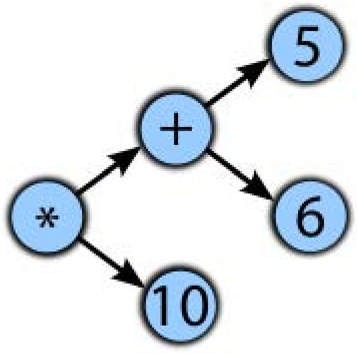

Figure 3

Consider the files - and therefore the trees produced - have the following limitations:
-Each non-terminal node ακριβώς has exactly two children, represents an operator and its name is "+" or "*".
-The name of each terminal node is an integer.
The evaluation of the expression will be done in parallel, creating a process for each node of the tree. Each terminal process returns to the parent the corresponding numeric value. Each non-terminal process receives from the child processes the values of their sub-expressions, calculates its value and returns it to its parent process. The root process returns the overall result to the original program process, which displays it on the screen.
Suggestions:
-Use Unix piping to communicate from child-to-parent processes.
-Each process must wait for its children to finish and print an appropriate diagnostic to detect programming errors in a timely manner.
-You print intermediate results from each process, it helps in debugging.
Questions
1. How many tubes are needed for this particular exercise and processing? Could each parent process use only one piping for all the processes children? In general, can only one piping be used for each arithmetic operator?
2. In a multi-processor system, more than one process can be run in parallel. In such a system, what advantage can the expression of process tree be had over the evaluation of a single process?