**Institute of Technology of Cambodia**

Department of Information and Communication Engineering

TP07

“***Understanding the process with C programing language***”

Prof.

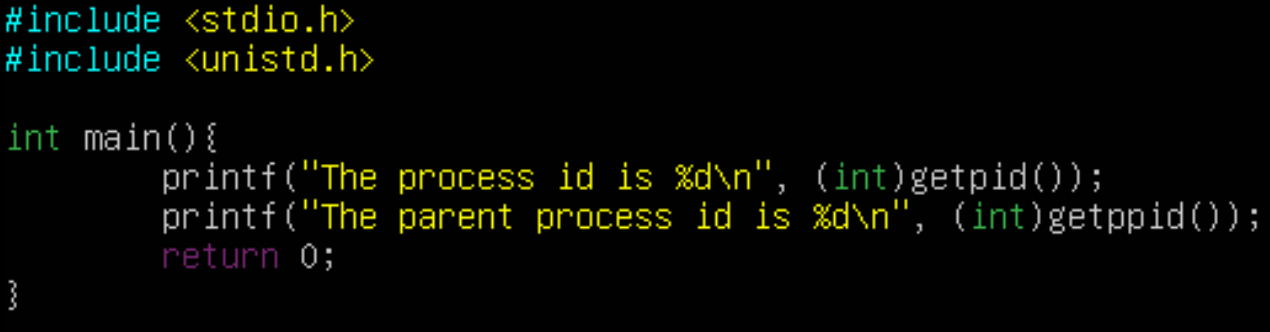
***KHEM Thay***

By  
***RITHYNY Lyhab***

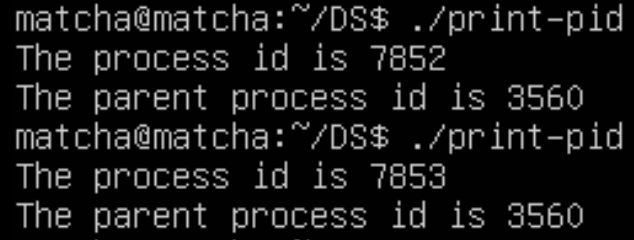
1. Monitoring Process

Process is a running instance of program which has a unique PID (Process Identifier). Each process also has a parent process whose ID is PPID (Parent Process Identifier).

* Listing 1: (print-pid.c) Printing the process ID using program below:



* Result:

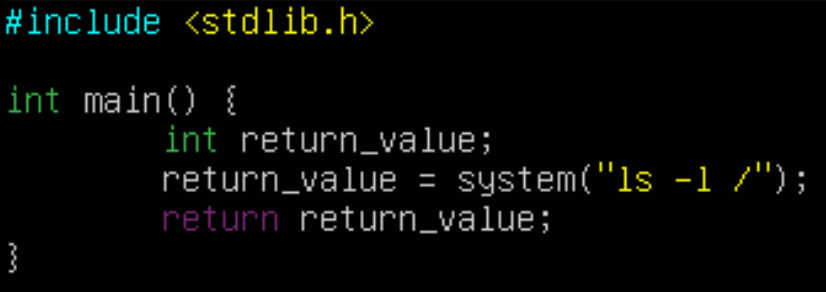


This program prints out the current PID and its own PPID when it is run. Note that each running instance of program produces different values of PID as shown in the result where PPID stays the same.

1. Create Process

Processes can be created in two different methods with the first method being used with the dependency on the system() library function, which allows you to execute a shell command in the software environment. This method is quite simple but is rarely used due to its ineffectiveness and significant risk to the system security.

* Listing 2: (system.c) Using the system() library function:

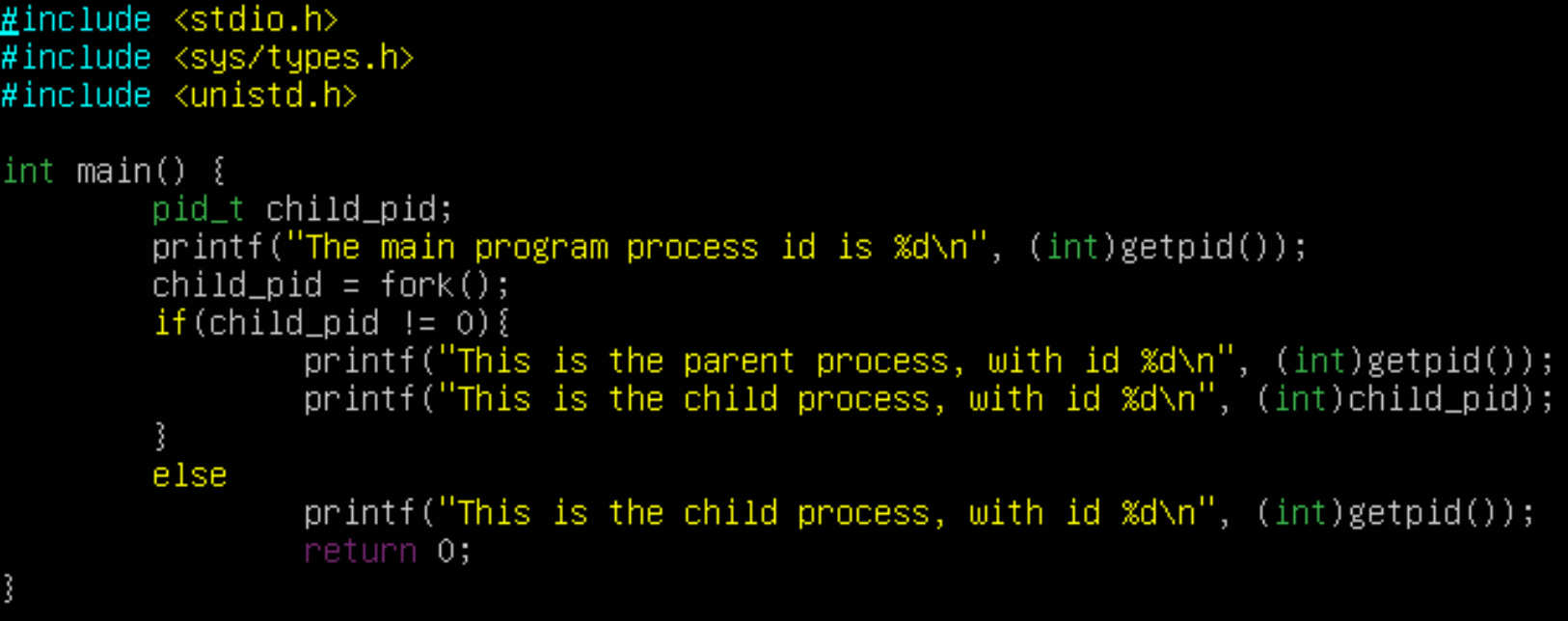


* Result:

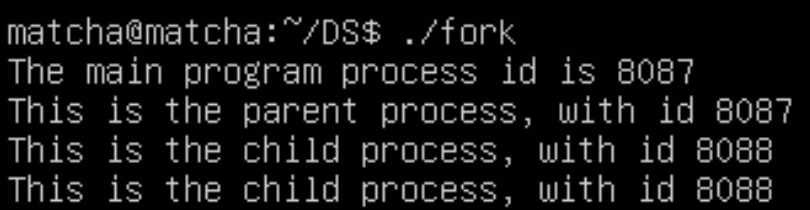
****

This program uses the system() function to execute the command “ls -l /”.

* Listing 3: (fork.c) Using the fork() to create a child process:



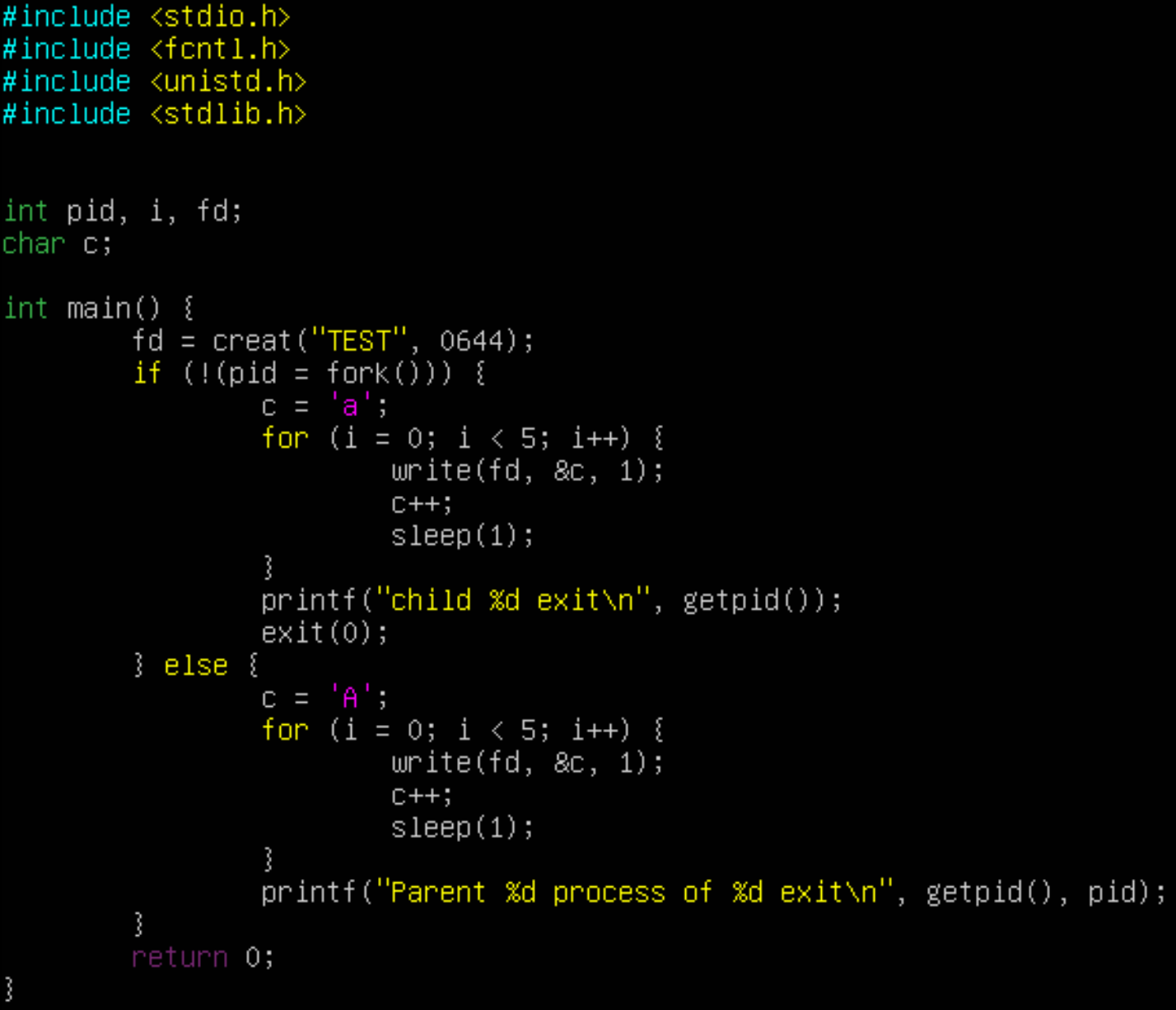
* Result:



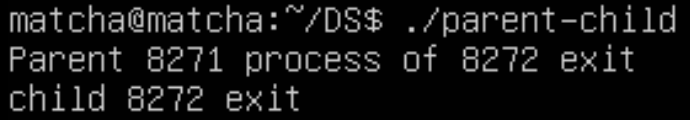
This program demonstrates the use of the fork() system call to create a child process:

1. The program prints the process ID of the main (parent) process using `getpid()`.
2. It uses `fork()` to create a new process (child process).
3. The parent process prints its own process ID and the process ID of the child process.
4. The child process prints its own process ID.

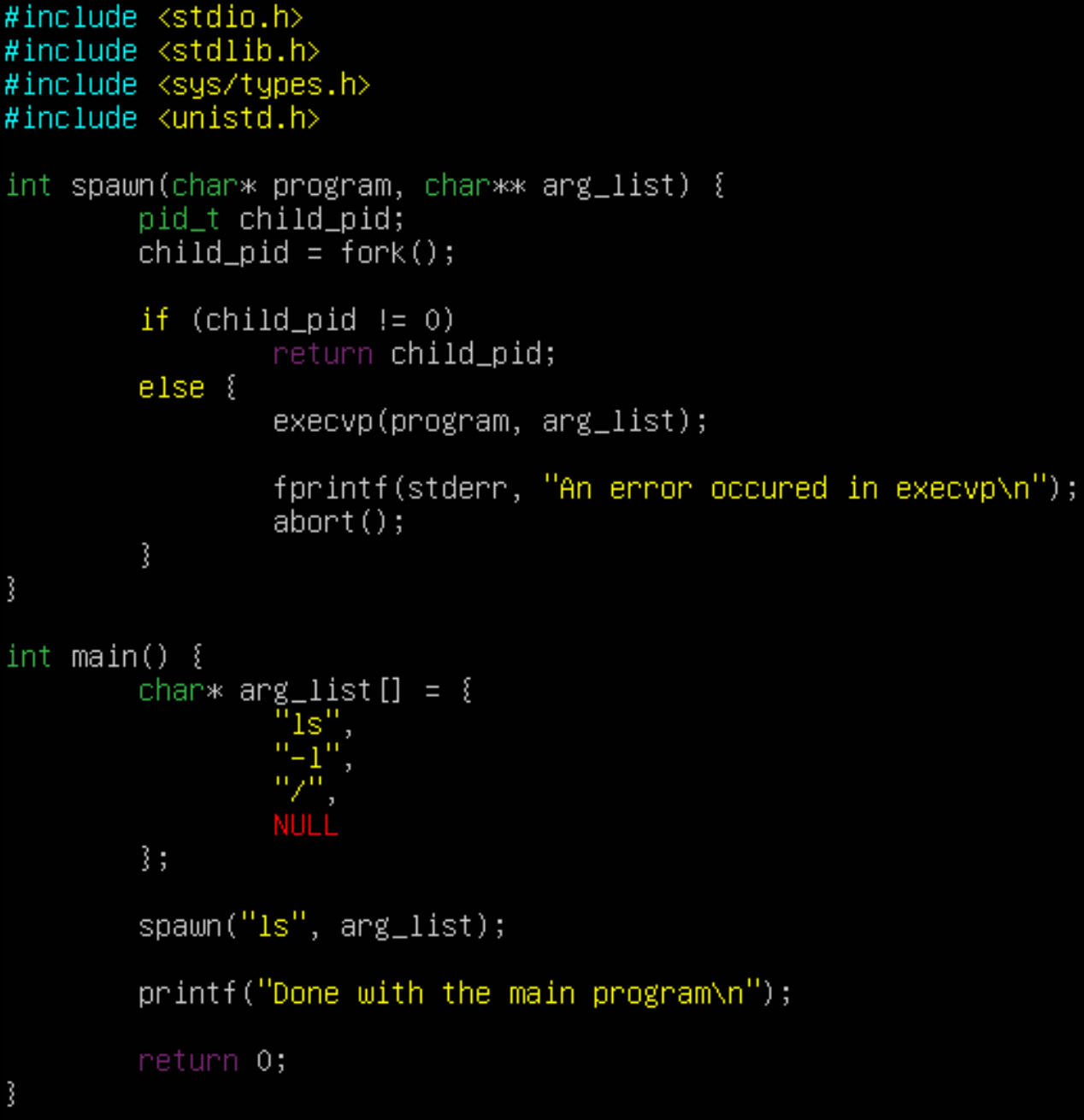
* Listing 4: (parent-child.c) Processes running after fork() completes:



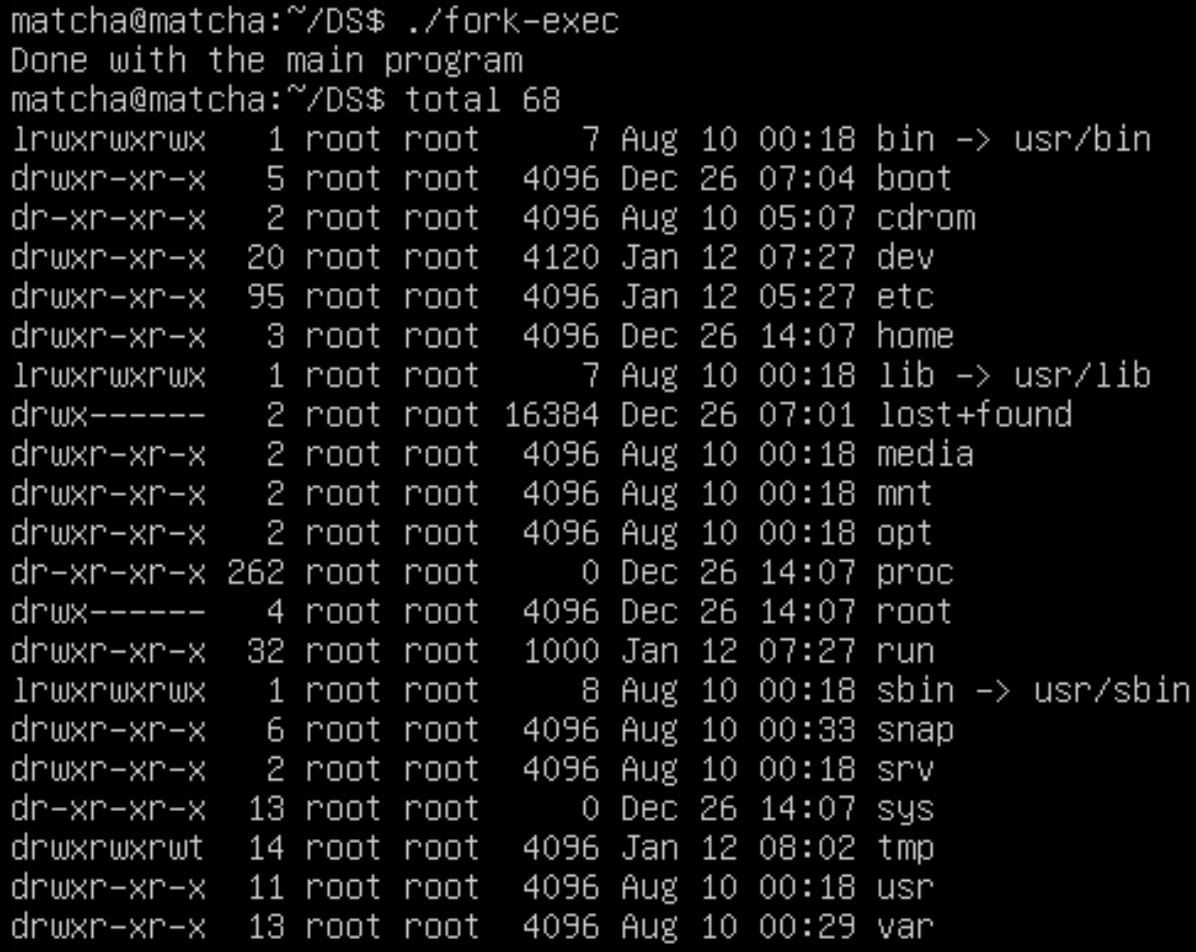
* Result:



* This program creates a file named "TEST" and employs inter-process communication using the fork() system call. It spawns a child process and a parent process. Both processes write characters to the file in parallel. The child process writes lowercase letters ('a' to 'e'), and the parent process writes uppercase letters ('A' to 'E'). The program introduces a 1-second delay between character writes. After completion, the program prints messages indicating the exit of the parent and child processes.
* Listing 5: (fork-exec.c) Using fork() and exec() together:



* Result:

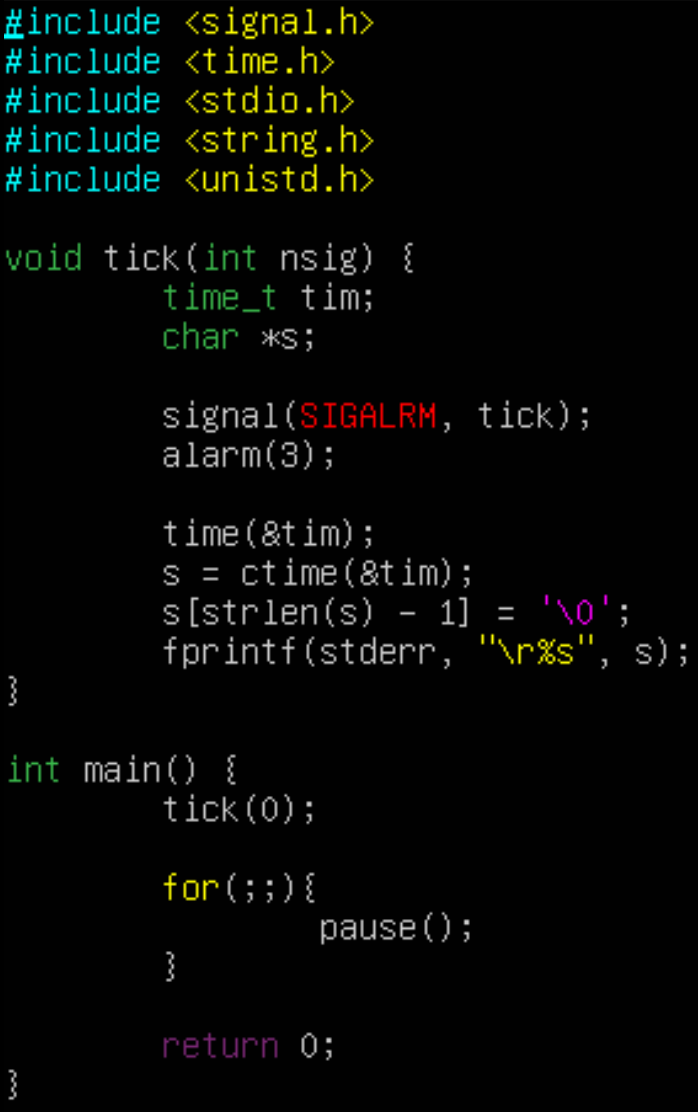


* This program defines a `spawn` function that creates a child process using `fork()` and executes a specified program with arguments using `execvp()`. The `main` function uses this to spawn a child process that runs the "ls -l /" command. The program demonstrates basic process creation and execution of external commands.

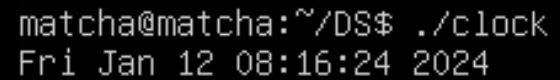
1. Signal

A signal is a special message sent to a process. Processes can communicate with each other via signals.

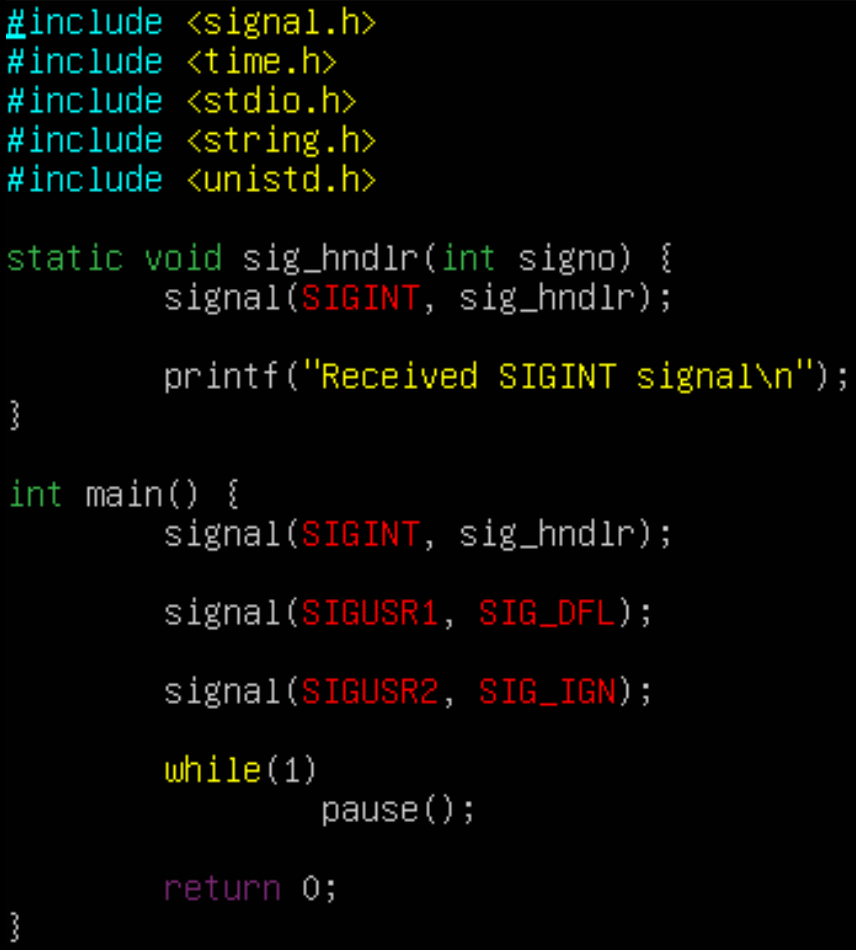
* Listing 6: (clock.c) The simplest implementation of an electronic clock:



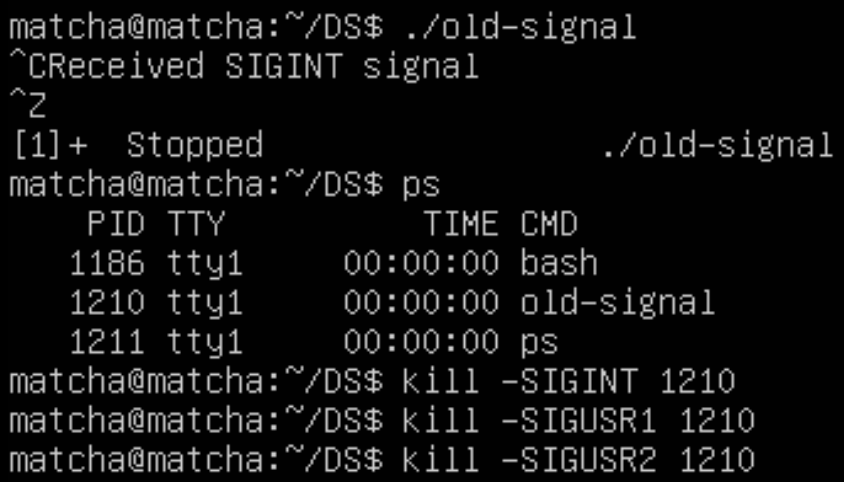
* Result:



* This program creates a simple clock that updates every 3 seconds. It uses signal handling to catch the SIGALRM signal, which is triggered by the alarm function every 3 seconds. The clock continuously displays the current time on the same line in the terminal. The program enters an infinite loop and pauses until a signal is received, ensuring that the clock continues to update periodically.
* Listing 7: (old-signal.c) Setting and checking signal disposition:



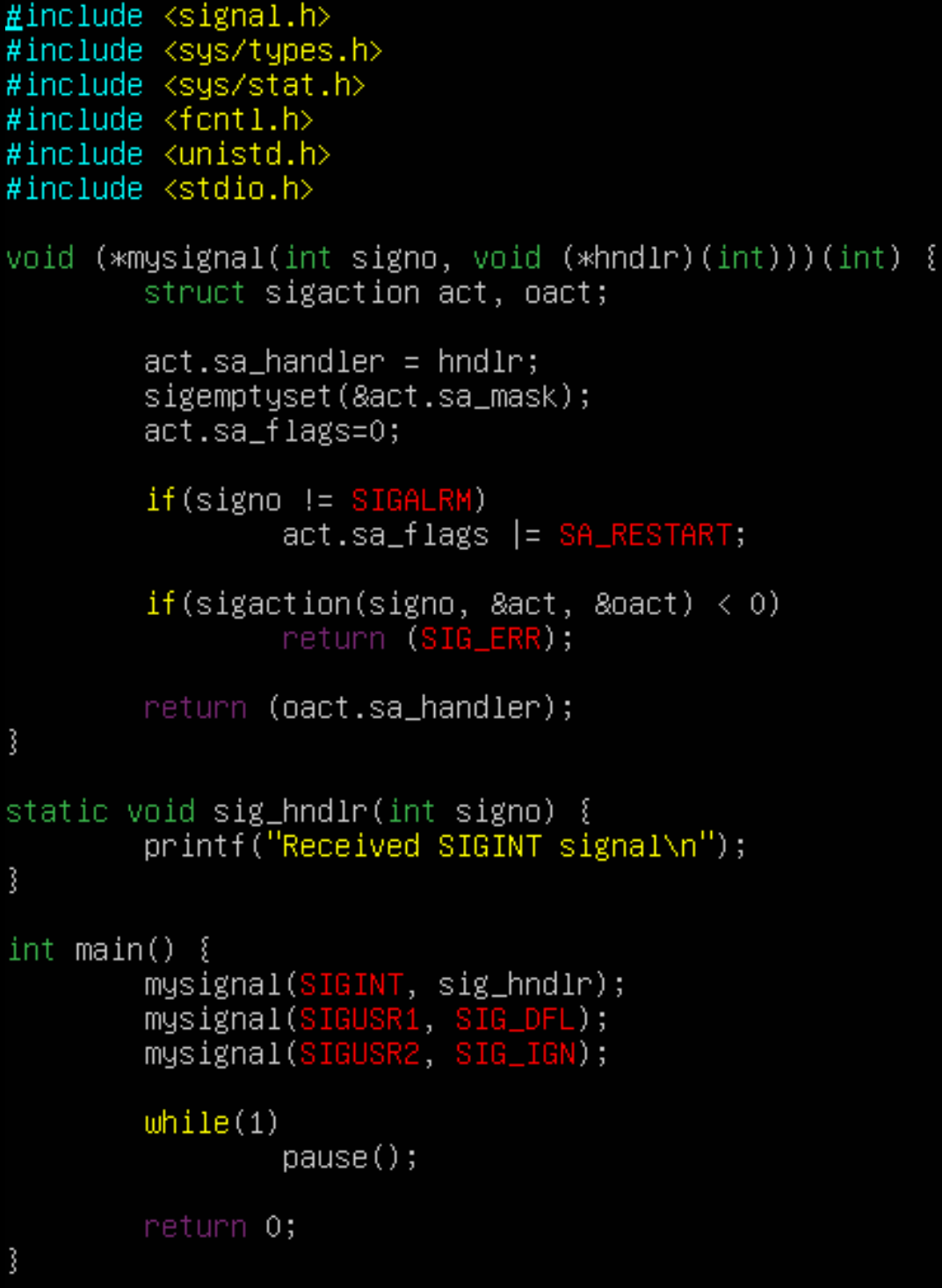
* Result:



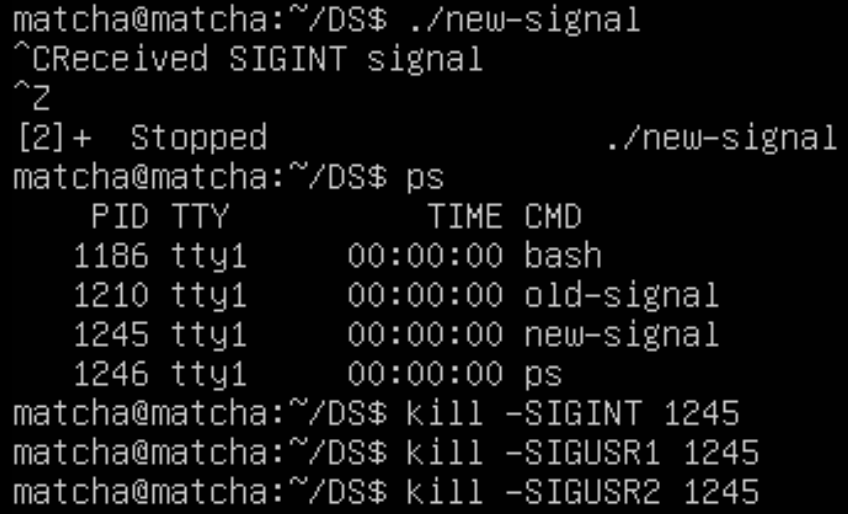
* The program then enters an infinite loop using pause(), effectively waiting for signals to arrive. This allows you to observe the behavior of the program when different signals are sent to it.

1. Reliable signals

* Listing 8: (new-signal.c) Using reliable signals:



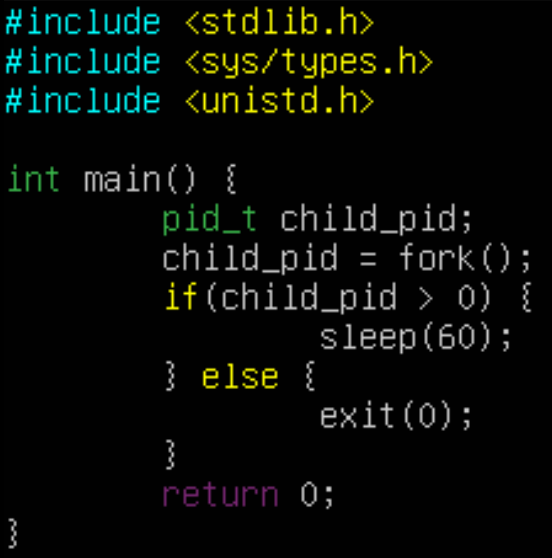
* Result:



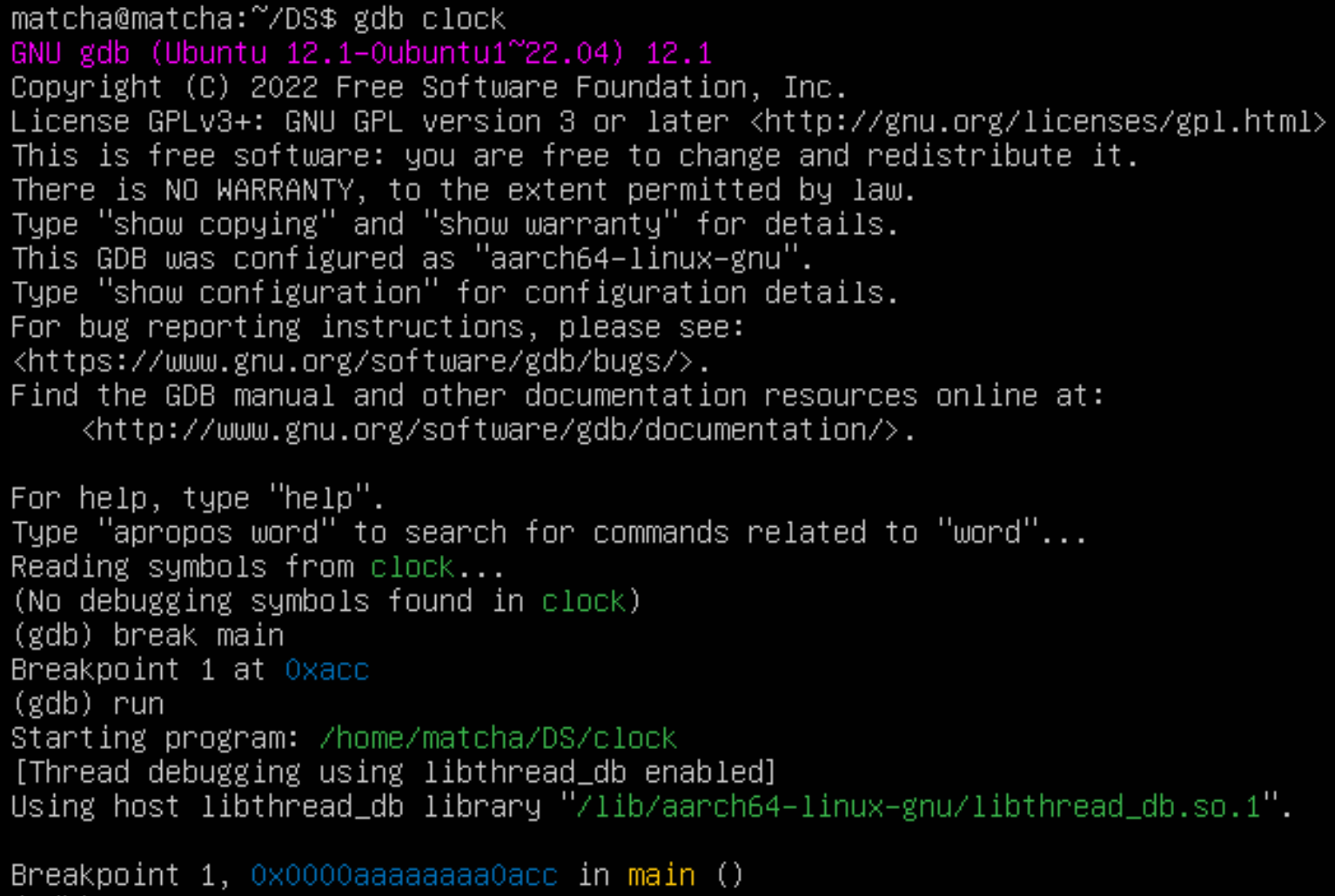
* This program demonstrates a custom implementation of a signal-handling function called mysignal. The mysignal function allows the user to set up signal dispositions for specific signals, similar to the standard signal function.

1. Completion of processes

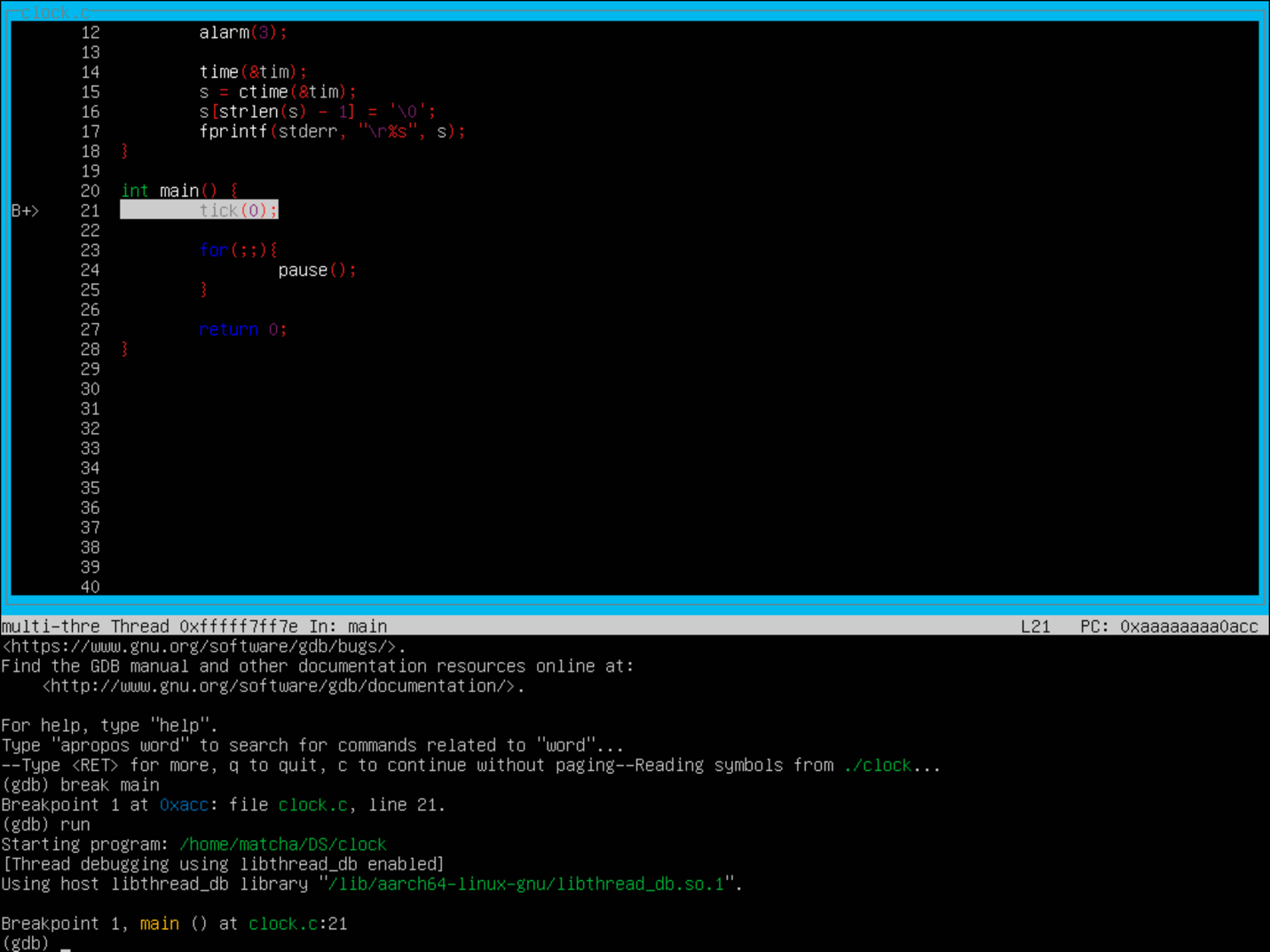
* Listing 9: (zombie.c):
* The program demonstrates a scenario involving zombie processes in Unix-like operating systems. A zombie process is a child process that has terminated but whose exit status has not yet been read by its parent process



1. Debugging programs using a GBD debugger

GDB debugger is used to debug the console programs.

1. Debugging programs using a GBD TUI debugger



1. Work task
2. Commands interpreter designed to work with processes:

* Process Status:
* ps: Display information about currently running processes.
* ps aux: Show detailed information, including the process tree.
* Kill:
* kill PID: Send a signal to a process to terminate based on its PID.
* kill -9 PID: Forcefully terminate a process based on its PID.
* Top:
* top: Display dynamic, real-time information about the system, including processes.
* Pkill:
* pkill -TERM process\_name: Send signals to processes based on their name.
* pkill -KILL process\_name: Forcefully terminate processes based on their name.
* Killall:
* killall process\_name: Terminate all processes with a specified name.
* Pstree:
* pstree: Display a tree diagram of processes, showing their hierarchical relationship.

1. Crosspile and test programs in the language of SI using library functions and system calls to create and manage processes:

* Cross-compiling is commonly used in scenarios where you develop on one platform (host machine) and generate binaries for another platform (target machine), such as embedded systems or different hardware architectures.
* By cross-compiling, cross-compilers are needed inclusive of:
  + Arm-linux-gnueabihf-gcc (for ARM Architecture)
  + Aarch64-linux-gnu-gcc (for AArch64 Architecture)

To crosspile programs, instead of using normal GCC, the commands are as such:

* Arm-linux-gnueabihf-gcc -o file\_name file\_name.si
* Aarch64-linux-gnu-gcc -o file\_name file\_name.si