https://folk.ntnu.no/michaeng/tdt4186_21/
michael.engel@ntnu.no

Practical exercises
Spring 2021

Practical Exercises 2

Processes

Please submit solutions on Blackboard by Thursday, 18.02.2021 12:00h

The objective of this exercise is to write a C program that implements the function of an alarm clock with *multiple settable alarm times*. Please submit a separate source code file for each of the four questions below.

2.1 Unix processes: simple alarm clock (2 points)

In this first part, write a simple alarm clock program. The program should ask the user to enter a number using `scanf(3)`, which represents a delay time in seconds. After the number is entered, the program waits for the the given amount of time (use `sleep(3)`) and "rings" when the given time has passed.

Use a loop so that after the alarm has sounded, the user is asked for a new time and a new alarm can be started.

Hints: You can indicate the ringing alarm clock by simply printing a text on the screen. If you want to create a real tone, you can use the *escape sequence* "\a" inside of a printf format string.

2.2 Multiple alarm clocks (5 points)

An extended version of your alarm clock program shall support the setting of *multiple alarms* running concurrently.

After entering a delay for an alarm, create a new child process using `fork(2)`, which is responsible for waiting the given time and then sounding the alarm. When the alarm has sounded, the child should terminate using `exit(3)`.

While the child process is running, the parent process should already prompt the user for a new alarm delay, so that the user can set an additional alarm while a previous one is still "ticking".

When creating the child process, the parent should print the child process ID. When a child process sounds an alarm, it should also print its own process ID so the user knows which alarm was activated.

Test your program to see if it works correctly with multiple alarms.

2.3 Catch the zombies! (2 points)

When you managed to get the program to compile and run correctly, observe the processes started by you using the tool `ps(1)` or `top`. You will find that the alarm clock child processes that have already rung and terminated using `exit(3)` are still listed as *zombie* processes: They remain in the system as long as the parent process does not call `wait(2)` or `waitpid(2)`.

Solve the problem of zombie processes using `waitpid(2)` (see the man page for details), e.g. directly after each user input in the parent process. When doing this, also print the PID of the "deceased" child process.

2.4 Error handling (1 point)

If you read the manpages for the various system and libc calls, you will notice that there is always a section describing possible *errors* that are returned in case the call fails. Add error handling code to all system and libc calls your program makes (you can use `perror(3)` for this) and add code to terminate your program in case of an error.