https://folk.ntnu.no/michaeng/tdt4186_22/
michael.engel@ntnu.no

Practical exercises
Spring 2022

Practical Exercise 1
Unix processes

Please submit solutions on Blackboard by Monday, 21.2.2022 12:00h

1.1 A Unix-based alarm clock

The objective of this exercise is to write a C program for a Unix-like operating system that implements the function of an alarm clock.
The alarm clock program should implement the following functionality:

- Schedule a new alarm (input “s”)

  The program should then ask for the (absolute) time at which the new alarmshould ring.
- List all currently active alarms (input “l”)

  All active alarms should be displayed, each with a unique number
- Cancel a scheduled alarm (input “c”)

  The number of the alarm to cancel (see list) should be input
- Exit the alarm clock program (input “x”)

An example session could look like this:

```
$ ./alarmclock
Welcome to the alarm clock! It is currently 2022-01-31 14:15:23
Please enter "s" (schedule), "l" (list), "c" (cancel), "x" (exit)
> s
Schedule alarm at which date and time? 2022-01-31 15:15:23
Scheduling alarm in 3600 seconds
> s
Schedule alarm at which date and time? 2022-02-22 22:22:22
Scheduling alarm in 1930019 seconds
> l
Alarm 1 at 2022-01-31 15:15:23
Alarm 2 at 2022-02-22 22:22:22
> c
Cancel which alarm? 2
> l
Alarm 1 at 2022-01-31 15:15:23
> x
Goodbye!
$
```

a. Menu (20%)

Implement the main level menu by displaying a prompt and then either wait for an input of a return-terminated command character (use `scanf(3)`) or a typed character (use `getchar(3)`) that offers the four functions described above.

b. Data storage and parsing (20%)

Unix time ("timestamp") is counted in seconds since the beginning of the Unix "epoch" on January 1st 1970, 00:00:00 UTC. Unix defines a separate data type to store the number of seconds: `time_t`.[1]

Create a data structure for the alarms. The most simple form could be an array of `time_t` values, but you might need additional information per alarm later (e.g. the PID of the responsible child process or a sound to play), so an array of structs will be useful later.

Hint: It is sufficient if you implement a statically sized array and refuse new entries in the array is full.

Implement functionality for the menu as well as reading and parsing the respective input parameters.

Add newly scheduled alarm times to your array, remove them when their respective number is given (you can simply use the array index) and list them *with their human-readable time* when requested.

Time related functions and `time_t` are defined in the `time.h` header file, which you have to `#include` in your program.

Useful time functions:

- `strptime(3)` converts from a string representation of time to a `struct tm`, which has separate integer values for day/month/year as well as hour/minute/second of the entered date and time
- `mktime(3)` can be used to convert from a `struct tm` to a `time_t` value

You can also refer to https://en.wikipedia.org/wiki/C_date_and_time_functions for an overview of Unix time functions.

c. Alarm scheduling (20%)

After entering the date and time for an alarm, create a new child process using `fork(2)`, which is responsible for waiting the given time and then sounding the alarm.

The child process created by fork waits for the the given amount of time (use `sleep(3)`) and "rings" when the given time has passed. When the alarm has sounded, the child should terminate using `exit(3)`. For now, it is sufficient if "RING" is printed on the screen.

Hint: If you want to create a real tone, you can use the escape sequence "\a" inside of a printf format string.

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 additional alarms or cancel existing ones while a previous one is still "ticking".

When creating the child process, the parent needs to store the child process ID so that it can be canceled later.

Hint: You can use `time(2)` to obtain the current Unix timestamp and then calculate the difference between the alarm time and the current time as a parameter for `sleep(3)`.

d. Canceling alarms (10%)

Implement the functionality to cancel the alarm with the given number. Use the `kill(2)` system call to terminate the related child process.

e. Catch the zombies! (10%)

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.

[1] In older Unix systems, `time_t` is a *signed* 32-bit value and overflows to a negative two's complement number on January 19th, 2038 at 03:14:07 UTC. This might cause lots of trouble if any of these systems are still around by then...

f. A real alarm clock (10%)

   You can add “ringtone” functionality to your alarm clock! Instead of printing “RING” in the child process, use one of the `exec` system calls to call a command line audio player, e.g. `mpg123` on Linux or `afplay` on MacOS.

   Optional: Allow separate alarm tones for each alarm.

g. Test case documentation (10%)

   With all functionality, this is already a complex program, so it has to be tested thoroughly.

   Design four different test cases that not only test a single functionality (like scheduling an alarm), but also a combination of functions of your program.

   Describe each test case: what do you do to test this case, what does the test do, what is the expected result?

General hints:

- Time in computers is a *very* difficult topic. We do not consider different time zones, the change from and to daylight saving time, leap seconds and similar problems here – so just assume your program will not move between time zones and someone has finally decided to abandon daylight saving time... i.e., keep it as simple as possible!

  If you are interested, more information and a discussion can be found on hackernews:

  `https://news.ycombinator.com/item?id=2725015`

- We don’t give all instructions in excruciating detail here. You are expected to read man pages, to reconsult the C crash course, to read books, to experiment and – if all else fails – to ask questions!

  If you want to dig deeper, a *very* good book about Unix programming is (beware, the book has more than 1,000 pages!):

  W. Richard Stevens and Stephen A. Rego, *Advanced Programming in the UNIX Environment*, Addison-Wesley 2013, ISBN-13: 978-0-321-63773-4.

  `http://www.apuebook.com/apue3e.html`

  Another book which is often recommended, but which is more Linux-specific, is:

  Michael Kerrisk, *The Linux Programming Interface*, No Starch Press 2010, ISBN-13: 978-1-59327-220-3.

  `https://man7.org/tlpi/`