Instructions for ELEC-C7310 course assignments

These instructions give the minimum requirements for the assignments. There are two assignments on this course, both of which are meant to exercise programming with the functions shown in the lectures. To be accepted a returned assignment should fulfill the following minimum requirements.

Both assignments have common minimum requirements:

1. Two-to-four page learning diary that is a free form description of the phases of programming done for the assignment, explaining the encountered problems and the found solutions. There should be some pondering on the problems, solutions and other interesting things. The learning diary should also detail how the code was tested, what works and what doesn't, and some plans on how the assignment could be made better (for further development.) The learning diary should be either UNIX-form text file or a PDF document.
2. Well documented source code that:

- Is modular, consists of multiple modules and at least one own library.
- Compiles without warnings with "-Wall -pedantic" parameters.
- Writes a log file of what it does.
- Makes a clean exit if user presses CTRL-C.
- Handles failed function calls gracefully and meticulously.

3. A working Makefile that compiles all the components and the program itself, and cleans useless object files.
4. Short usage instructions and description of what the program does in general.
5. All the previous in one .tar.gz package ("tar czvf as.tar.gz directory/"):

- The filename must be "studentnumber-asNUMBER.tar.gz"
- The directory inside tha package must be "studentnumber-asNUM"

In other words, put the files into a directory according to your student number and assignment number and and run command "tar czvf 123456-as1.tar.gz 123456-as1/" This package should be uploaded using mycourses.aalto.fi, preferable before the deadline.

Fulfilling the minimum and part specific requirements does not automatically grant points, but rather the variety and the care taken with the programming and learning diary. Meeting the minimum requirements allows to proceed and potentially pass the course. Points are given on a scale from 1 to 10. Each participant is entitled to personal feedback on their work.

Few notes

- Read this instruction carefully. It's futile to lose points just because you didn't read this instruction properly. If something's unclear, ask.
- Take advantage of the things taught on lectures, for example, how to write a proper signal handler.
- Preferably write a simple, requirement-filling program, than a huge and fanciful but "somewhat" unfinished one.
- Start early, there is really a lot of time, but when programming one often encounters puzzling problems that take a lot of time to fix.
- Other functions that were not covered on the lectures are allowed to use. The main idea is to fulfill the requirements, for example, reading and writing can be done with FILE streams also, if you feel like it.

Part 1 specific requirements

In addition to the common requirements, the program must use all of the following features:

- Writing to a file (writing to a log file at least) and reading from a file (if nothing else, reading a configuration file).
- Signal handling (for example, controlled shutdown when CTRL-C is pressed).
- Creation of child process(es).

And some of the following:

- Code replacement with exec().
- Use of memory mapped files.
- File locking.
- Asynchronous or non-blocking I/O.

Try to adapt these concepts into the assignment topic that you select.

Part 2 specific requirements

In addition to the common requirements, the program must use:

- Communication between processes or programs
- Thread creation
- Use of mutex
- Thread synchronization

Try to adapt these concepts into the assignment topic that you select.

Clarification

There are two assignments, both are mandatory. The assignments listed below are topics for each assignment. Select *one topic* for *each assignment*, implement it so that the requirements given above are fulfilled and return the code and the documentation to Optima. To reiterate, you only need to do two programs, one for each assignmen. Requirements are different for each assignment: The first assignment is about covering many APIs and the second assignment is about threads and synchronization.

Deadlines

01 Nov 15 at 23.59 1st assignment 01 Dec 15 at 23.59 2nd assignment

Grading principle

- Learning diary 2pts
- Source code 3pts
- Miscellaneous 2pts
- Fulfulling requirements 3pts

Max 10 points in total. Range is from 0 to 10, and return may be failed if it completely misses some portion (e.g. doesn't do anything required)

Late returns: Minus one point per day.

Topics for part 1 assignments

Data transfer

Program takes two parameters, two file names (file1, file2). Program starts a child process which opens the file given as it's first parameter (file1). Child them reads from the file and sends the file to the parent process, which writes it to the file given as the second argument (file2).

Transfer the data between the processes by morse coding it so that the child sends characters encoded in morse code to the parent. You can for example define that USR1 signal is "short" and USR2 signal is "long". How will you tell apart single characters or even words? Help on Morse code can be found in:

- https://web.archive.org/web/20130102133344/http://aimo.kareltek.fi/~reni/morse-en.php (english)
- https://web.archive.org/web/20130102133349/http://aimo.kareltek.fi/~reni/morse.php (same in finnish)
- http://morsecode.scphillips.com/morse.html

Error case has it's own Morse code. Note that newline code is missing, invent your own substitute. If file1 doesn't exist, parent process must be notified of this.

Consider what would be an efficient solution and what only gets the work done but takes a lot of time. Do you need some kind of flow control?

Possible implementation methods:

- Keep short pauses between characters and measure the time using gettimeofday().
- Use some additional signals. At least SIGALARM and SIGINT could be used. Though then pressing CTRL-C wouldn't end program, but it doesn't matter here.

Program code "cleaning"

Make a program that is given file names as command line parameters. The program starts a child process for each file. The child processes will remove every empty and comment line from the file and write the thus "cleaned" code into a new file with name "original_name.clean". Comment line can be either:

- One starting with characters "//" and ending with new line (naturally)
- One starting with characters "/*" and ending with corresponding "*/".

Parent process waits until the child processes have died before quitting.

File processing

Same as the previous but each file is passed through some more complex operation - design it yourself.

Bit more complex "Hello World"

Make a version of the classical "Hello World" program that is as complex as possible. The classical "Hello World" prints "Hello World" on the screen and exits. The program could, for example, read some value from file system, create child processes based on the value and the child processes could then be ordered, for example, with signals to write one character each and quit. Anything can be used as long as the requirements of the assignments are filled, that is, many different methods are used to accomplish the result.

Performance evaluation

Compare performance differences between memory mapped files, regular I/O and/or asynchronous I/O, etc. Make a program that, for example, opens a file, finds a string from a file, replaces it with another string and writes it back. Make two programs, one that does it with regular I/O and another that uses memory mapping or asynchronous I/O, for both reading and writing. How do you read and write files? A character, a line, a block or a file at a time? What difference would those make? Why?

Compare programs based on memory usage, execution time (wall clock time and CPU time). For testing purposes, create (or find) a large enough file so that some differences emerge. You should also note that Linux "optimizes" handling files sometimes... How?

Learning diary for this assignment should contain descriptions of comparisons do e and what was learned.

You may invent some other performance comparison, but agree on the topic first with the lecturer.

Topics for part 2 assignments

Pick one topic and implement it and also fulfill the common minimum requirements given on the assignment page.

Printout capturing

Make a library that can capture normal output (for example, printf()) of other programs to a separate output process or thread. This output process should continue to post-process the output of programs by, for example, writing it out to a log file. Other kinds of post-processing is possible, like sending the output over network sockets (not part of this course, not necessary to implement.) The basic idea is the same as with command line "program > log.txt". The implementation should be as simple as possible to add to existing programs, and it must also support child processes spawned by the programs. Name of the log file should be able to be given in the old program, and if the file already exists, new messages should be appended to the end. stderr stream must not be captured but is to be left untouched.

Write also a threaded program to test the library.

Data transfer

This is somewhat like the morse code signaling topic earlier. Make two programs. Program A can be started multiple times, as independent processes. Each opens a text file given as a parameter (file1) and reads from it and transfers the contents to program B (a daemon) that writes the sent data into new files.

Transfer the data using FIFO pipes so that program B is the coordinator and gives writing permissions one by one to the senders. Data transfer must happen in parallel, not so that one sender is fully served while others wait for the first one to finish. When given permission the sender can write a given maximum amount at a time to the pipe. Which kinds of methods can be used to implement this kind of synchronization? You can assume that there are no more than 256 senders.

Own syslog daemon

Write a re-implementation of the syslog daemon and an associated library. The daemon should wait for input from some sort of pipe. The client applications are linked with your library. The daemon should add date and time with millisecond precision to the messages. The program that sends the log messages should know nothing of the implementation of the daemon or even of the pipes that are used to relay the messages. The library should handle all this. The implementation

must support multiple simultaenous senders. (Note: This requires some special work, write a "stresstester" to your daemon and check if it can handle a lot of workload.) In addition, messages from different sources should be differentiated, for example identical messages from two different senders should be differentiable. Log messages should resemble something like

DATE TIME WHO MESSAGE

Example:

Dec 24 12:00:01.250 clientname_and/or_pid This is my message

My own AIO functions

Make your own implementation of the asynchronous I/O functions introduced in the part two of the lectures. Make a library that includes the similar functions and implement the same kind of functionality as was shown on the lectures. The idea is not to make functions that call the actual standard functions, but to make your own re-implementation of them. Function call lio_listio is not required to be implemented.

You must also write a test program that tests the implemented library. It should be challenging to synchronize the test program and the library, and arranging and using the shared memory.

ThreadBank application

Make an application that implements a rudimentary bank. A bank contains N service desks, each of which should run in it's own thread. Each thread waits from messages from it's own message queue. Based on queue lengths in a shared memory pool the client selects the desk with the shortest queue and waits until the desk is free (if there's a previous client) or starts communication with the desk using the message queue, one service request at a time. The accounts have R/W locks.

The bank servers maintain the collected balance of the bank, that is the sum of all balances in the bank.

In addition there is a master thread that queries the balance of each desk, that is how much the clients have deposited and withdrawn funds. The master thread can overtake all queued customers but must not interrupt the current client. The desks should not proceed with other clients until all the desks have reported to the master thread.

Clients can send, for example, the following commands:

"l 1": give balance of account 1 "w 1 123": withdraw 123 euros from account 1 "t 1 2 123": transfer 123 euros from account 1 to account 2 "d 1 234": deposit 234 euros to account 1

Commands are given on command line.

For more challenge, add some kind of bank statement that can be queried. Bank statement is not an absolute requirement for this work and is not required for full points.

Performance comparison

This is topic is challenging. The idea is to compare the performance differences between processes and threads. This should be done by measuring the time required to do e.g. following things:

- Process creation vs thread creation
- Process switching vs thread switching (this part is voluntary)
- Size test: semaphore size vs mutex size
- Semaphore time-to-acquire vs mutex time-to-acquire (when empty)
- Semaphore time-to-acquire vs mutex time-to-acquire when one process/thread is holding it and then releasing it.
- Practical test: copying multiple files (select some BIG files) using processes / threads
- Practical test: transfer a file between processes / threads using:

  1) pipes
  2) message queues
  3) shared memory