Final Assignment

1. Introduction

As you learned in class, serving user requests is an important role of the OS kernel. The kernel exposes its services to user code through system calls. For example, C programmers can create new processes and finish them via the `fork()` and `exit()` system calls, respectively. The goal of this assignment is implementing a new set of services for monitoring and controlling the scheduling of normal (that is, non real-time) user processes. To this end, you will implement two new system calls: `get_vruntime()` and `increment_vruntime()`.

But before diving in, we start with a detailed walk-through of Linux kernel development. You should work on the Ubuntu virtual machine that was installed in HW0.

1. Introduction

In this assignment you will not write or modify kernel code. You will write only user mode applications. Your mission will be to write simple grade server and client applications for the course operating systems.

A grade server stores grades for all students and allows clients to read and modify the grades.

There are two kinds of clients:

- A teaching assistant (TA): can read and modify grades.
- A student: can only read his grade.

2. Detailed Description

Note: strings which are written in *italic* are variables that can have different values (as explained for each one), while regular (non italic) strings should be used (or printed) as is.

The client application:

The client application will be called "GradeClient" and receives 2 parameters in the command line:

- GradeClient *hostname port*
  - *hostname* – name of the grade server (e.g.localhost)
  - *port* – port on which the grade server is listening.

The client application consists of 2 processes: **command line interpreter** and **communication process**. The command line interpreter receives commands from the user and the communication process handles communication with the server. The two processes communicate using an unnamed pipe.

When the application starts the command line interpreter prints the prompt "> " and waits for commands from the user (the prompt is printed after each command), while the communication process connects to the server and promotes to it user commands.

Users can send one of the following commands:

- Login *id password* : The user (student or TA) tries to login to the grade server with the given id and password. If login succeeds the application will print "Welcome Student *id*" or "Welcome TA *id*" accordingly. Otherwise, it will print "Wrong user information". Login will succeed even if a user with the same details is logged on from another client. It will fail only in the following cases:
  - There is a user already logged in from this client.
  - The id doesn't exist in the server.
  - The password is incorrect.

The following commands (except Exit) are allowed only if a user is logged in. Otherwise, the message "Not logged in" will be printed. You may assume that command length including the parameters doesn't exceed 256 chars.

- ReadGrade : Print the grade of the logged in student. The operation can be done only by a student. If the operation was requested by a TA then the message "Missing argument" will be printed.
- ReadGrade *id*: Print the grade of the specified id. The operation can be done only by a TA. If there is no student with the given id then the message "Invalid id" will be printed. If the operation was requested by a student then the message "Action not allowed" will be printed.

· GradeList : Print a list of all the grades in the server. The operation can be done only by a TA. If the operation was requested by a student then the message “Action not allowed” will be printed. The list will be ordered in an ascending id order, and will be printed in the following format:

id: *grade*

id: *grade*

…

- UpdateGrade *id grade* : Update The grade of *id* in the server. If there was no such id in the server then it will be added. The operation can be done only by a TA. If the operation was requested by a student then the message “Action not allowed” will be printed.
- Logout : The user will be disconnected from the grade server. The operation will fail if no user is logged in. If logout succeeds the application will print “Good bye *id*”. Otherwise, it will print “Not logged in”.

Note: a Logout command does not terminate the connection with the server. It only means that this user logged out.

- Exit : The client application will exit (both processes will finish). If a user was connected, the application will logout before it exits.

If the command is not one of the above the application will print “Wrong Input”.

Example:

```
GradeClient localhost 12345

 > Login 234567890 student1

Welcome Student 234567890

> ReadGrade

74

> UpdateGrade 234567890 98

Action not allowed

> Logout
```

```
Good bye 234567890

> Hi

 Wrong Input

> ReadGrade 234567890

Not logged in

> Login 011223344 TA1

Welcome TA 011223344

> fd

Wrong Input

> UpdateGrade 234567890 0

> ReadGrade 234567890

0

> GradeList

 234567890: 0

123456789: 83

> Exit
```

The server application:

Introduction: Thread pool

If a server wants to work efficiently, the main thread can only receive new connections and for each connection it creates a thread to handle it. Although creating a thread to implement a task is cheaper than a process, but it is still an operation that takes resources and time. In order to solve this problem, the concept of thread pool was introduced. Instead of creating a thread for each task, the main thread creates a constant number of working threads during its initialization, and they live all the time the server is up. When a task is received it is inserted to a task list and a working thread is waked up to handle it. When no task is available the working threads will do nothing (wait on a condition variable).

The Grade Server

The server application will be called “GradeServer” and receive 1 parameter in the command line:

- GradeServer *port*
  - *port* – server port number

The server is a multi-threaded application in which the main thread listens for new connections and a pool of working threads handles user connections.

When the server starts, the **main thread** will:

- initialize the data structures of the server.
  - The ids and passwords of the TAs are read from a file called assistants.txt which resides in the same directory of the server application. Each line of the file has a single id:password couple (No spaces, id is nine digits). You may assume the file is in the correct format.
  - The ids and passwords of the students are read from a file called students.txt which resides in the same directory of the server application. Each line of the file has a single id:password couple (No spaces). You may assume the file is in the correct format.
  - The grades of all students (from the file students.txt) are initialized to 0.
- Create N=5 working threads which will be ready to handle clients.
- Open a socket with the port specified in the command line.
- The server will run an infinite loop in which it will accept users’ connections. For each connection a new task is added to the task list and a working thread is waked up to handle the task.
- The server ends only when it receives a kill command or Ctrl-C from the command prompt. The signal handler should close the socket and exit.

All **Working threads** will run the same function. The function runs an infinite loop in which it takes a task from the task list and handles the user’s queries. When the list is empty the thread will wait on a condition variable until new tasks are inserted. All working threads work with the same global tables and should synchronize access to them.

3. Notes and Tips:

1. You must use condition variables for the workers thread waiting for new tasks (semaphores are not allowed).

2. You should write a portable code in the communication between the server and the clients (use htonl/s and ntohl/s functions when needed).

3. You should write in C (or C++ if you know it) only.

4. You should close sockets after you finish using them.

5. The kernel has a TIME_WAIT before it has totally released a socket. Look at this explanation: http://hea-www.harvard.edu/~fine/Tech/addrinuse.html .You can use any of the suggested "Strategies for Avoidance", or simply ignore this problem.

6. Don't add any printouts beyond that is requested, otherwise, you might fail in the automatic check ("Wrong Input" means only the string "Wrong Input" and not "wrong input" nor "bad input"…).

4. Submission:

- **Wet submission:** You should submit a zip (**important!!!** Use zip and not gzip, gzips are not acceptable) file containing (at least) the following files:
  - All your source and header files.
  - (recommended) makefile – a single makefile that creates the applications "GradeServer" and "GradeClient" as explained in the exercise (**important!!!** make sure you know how to write a makefile that creates two executables). It should look like:

```
all: GradeServer GradeClient



GradeServer: grade_server.c

gcc -lpthread -o GradeServer program1.c



GradeClient: grade_client.c
```

```
gcc -lpthread -o GradeClient grade_client.c
```

- submitters.txt – names of the group members as in previous exercises.

.

Have a nice connection,

Leonid