Assignment 2
20-05-2019

Programmazione di Reti

Ing. Chiara Contoli, PhD
chiara.contoli@unibo.it

Corso di Laurea Triennale in
Ingegneria e Scienze Informatiche

Homework

Goal. The goal of this assignment is to emulate the Linux super-server *inetd*, with the aim of acquiring knowledge on:

- Socket programming
- Concurrent (server) programming

The **inetd** daemon, also known as “Internet super-server”, execute upon request other servers. It is constantly listening for requests, and leverage a file, *inetd.conf*, which contains and describes a list of servers via several parameters.

Two server models exist:

- Sequential: handle only one client at a time (typical of applications based on connection-less protocols, e.g., UDP)
- Concurrent: handle multiple client at the same time (typical of applications based on connection-oriented protocols, e.g., TCP)

Your assignment

To design and implement your super-server, you will leverage:

- a configuration file, which is similar to inetd.conf;
- source code (given in this zip directory) representing the “skeleton” of the super-server, which has to be properly modified (extended) to achieve the assignment task

The skeleton of the server contains the starting point of the implementation of the structure of the server, and part of the signaling handle behavior that will have to be properly completed.

The assignment contains two parts: an analysis/design phase (*Task1*) and the implementation phase (*Task2*).

Your assignment

In this assignment, you will *implement* the server-side application, i.e., the super-server, by:

- Properly defining a configuration file (see next slides for details)
- Implementing the required server-side behavior both for applications running over TCP and UDP protocols (see next slides for details)

As a last step, you will test your super-server by leveraging source code provided in this zip directory.

Super-server behavior

The configuration file that you have to write will be similar, in terms of structure and purpose, to */etc/initd.conf* file. It will contain a list of supported services that the Super-server will be able to run upon request. In particular, the file configuration file **must** be structured as follow:

- One line per service
- Each line will have the following structure:
  - Full path service name
  - Transport protocol (*tcp* | *udp*)
  - Service port number
  - Service mode (*wait*: for sequential services, *nowait*: for concurrent services)

An example of line is as follow:

./app udp 10000 wait

The line represents e C executable program over UDP provided on port 10000 that takes no argument.

Super-server behavior

The server-side behavior you are required to implement is as follow:

1) At the startup of the server, in the *main* program, the configuration file has to be opened in reading mode:
   - For each line of the file, it extracts service parameters and saves those information in a data structure properly conceived (see slide 11).
2) It creates the required socket for the service and if the service run over TCP, it will invoke also the *listen* operation. The socket file descriptor returned upon socket creation will also has to be saved in the above mentioned data structure.
3) It fills *select* parameters with all retrieved socket file descriptors.
4) It invokes *signal* operation.
5) It runs in an infinite loop based on the *select* operation.
6) Once *select* is fired, if the socket file descriptor belongs to a TCP service, the Super-server will invoke *accept* operation in order to retrieve the connected socket file descriptor.

Super-server behavior

7) It invokes *fork* operation and if the process is the

Father:

If service run over TCP, it will close the connected socket;

If service was *nowait* mode, it will go back to the *select*;

If service was *wait* mode, it will register service PID inside the data structure, it will remove its socket file descriptor from the *select* and it will go back to the select;

Son:

It closes all the standard I/O file descriptors, i.e., 0, 1, 2. If the service run over TCP, it closes the welcome socket.

It calls 3 times *dup* operation in order to associate the socket file descriptor to stind, stdout and stderr;

It invokes the *execle* operation in order to execute the requested service

You are therefore required to use 2 new operation

- *dup*
- *execle*

that are explained in the next slide.

Dup and execle operations

From <*unistd.h*>:

- int **dup**(int *oldfd*);
  - Returns a new (socket) file descriptor on success, -1 on error
  - Parameters: i) *oldfd*: file descriptor to be duplicated
  - It creates a copy of *oldfd*, and it associates to the new file descriptor the lowest integer not used
- int **execle**(const char **path*, const char **arg*, const char **arg0*, …, char * const *envp*[]);
  - Returns only if an error has occurred, -1 on error
  - Parameters: i) *path*: full path of the program together with the name of the program; ii) *arg*: name of the program; iii or more) *arg0, …argN*: a list of possible argument available to the executed program; last) *envp*: environment of the executed program
  - It replaces the current (calling) process image with a new one (this means that the new program has the same process ID of the calling one); it has a variable number of parameters, the first two are mandatory, and such list of parameters is terminated by NULL.
  - E.g.: *execle*("/bin/csh", "csh", "-i", NULL, env)

Considerations about *signal* and *wait*

In the source code provided as skeleton, you will find two others additional system call: *signal* and *wait*.

The system call ***signal*** allow to handle signals to which we are interested in, and allow to register on the operating system callback functions. In this assignment, you will handle the signal related to the death of the son process, ***SIGCLD***. Note**.** Remember that if a signal is fired, the system call *select* will be interrupted; therefore, remember to handle this event (i.e., distinguish the case of a select returning because of an error, from the case it returns because of a signal).

The system call ***wait*** allow to retrieve the PID and the reason of the death of a process. This is a blocking function, therefore has to be called **only** when you are sure that the death of a process has occurred.

Considerations about *signal* and *wait*

In the Super-server skeleton, you will see that the function registered via *signal* that act as a callback function is *handle_signal*. This means that when the death of a process occurs, *handle_signal* is called; therefore, you will **have to** call the *wait* system-call inside *handle_signal* function in order not to be blocked, and you will be able to retrieve the PID of the terminated service (i.e., of the son process).

When the service terminate, the operating system will call *handle_signal*, which will:

- Retrieve PID of the death process (service) via *wait* function;
- Given the PID, search in the data structure the structure containing the description of such service and, if this service was of type 'wait', you have to insert again its socket file descriptor inside the *select*.

What and where to save service information

Where: service information has to be saved in a data structure (use *typedef struct*), and you will **use a vector** of this data structure to store information about the services read from the configuration file (consider to have a maximum of 10 services as a limit for the size of the vector).

What: the data structure **must** contain the following information:

- Transport protocol (char[]): 'tcp' if protocol is TCP, 'udp' if protocol is UDP
- Service mode (char[]): 'wait' if service is concurrent, 'nowait' otherwise
- Service port (char[]): port on which service is available
- Service full path name (char[]): complete name of the service, including full path
- Service name (char[]): name of the service
- Socket file descriptor (int): socket descriptor associated to the port
- PID (int): PID of the process; this is meaningful only for services of type 'wait'

Your assignment

Task1: in this task, you are asked to provide a diagram (similar to those seen at lab classes, e.g., slide 29 of 2_TCP_socket.pdf) of the server side you are about to implement.

The diagram **must** contain the steps taken by the server from the system-call point of view (i.e., socket(), bind(), fork(), etc.). Very concise comments related to those steps are not mandatory but can be useful to prove that you have understood what the Super-server has to do to achieve its goal.

NOTE. This diagram is a useful starting point to be carried out prior to the next phase (Task2).

Your assignment

Task2: in this task, you are asked to implement (and test) the server side behavior, starting from the skeleton *superserver.c*, by strictly following instructions provided in previous slides. In particular:

1. To implement the general behavior of the Super-server, follow instructions at slides **6-7**;
2. To properly implement the data structure required in the source code, follow instructions at slide **11**;
3. To properly create the configuration file follow instructions at slide **5**;
4. To properly use system-call *dup, execle, signal* and *wait* follow instructions at slides **8**, **9** and **10**, respectively. Note that part of the code that leverage *signal* is already implemented; therefore, you only have to properly complete this implementation.
5. To handle the death of a process follow instructions at slide **10**.

If needed, you are allowed to use global variables.

Your assignment

To test the server behavior you will have to compile the source code provided together with the skeleton:

- udpServer.c and udpClient.c;
- tcpServer.c and tcpClient.c;

The two servers need to be compiled so that the Super-server will be able to execute them upon request. Instead, clients will be compiled and executed in order to test the Super-server.

Both clients require as input the IP address and the port of the service you want to request.

Note that the two servers **must** be included in the configuration file, so that they can be reached either in sequential and concurrent mode via TCP and UDP protocol.

Your assignment

In order to verify the correctness of the Super-server behavior, you will have to test UDP concurrent and sequential services, and also TCP concurrent and sequential services, so that you can test all possible 4 scenarios.

	Service mode	
Server type	**wait**	**nowait**
udpServer	✅	✅
tcpServer	✅	✅

For each mode, to test each server you will start 2 Clients and generate traffic towards the service.

Your assignment

	Service mode	
Server type	**wait**	**nowait**
udpServer	✅	✅
tcpServer	✅	✅

Questions:

1. Which is behavior of udpServer in wait mode? Which one in nowait mode?
2. Which is behavior of tcpServer in wait mode? Which one in nowait mode?

What to submit (and how)

Whoever fails in following these simple rules will incur a penalty in the grading process.

Submit by your institutional e-mail a zip file of a directory containing the **whole** assignment. Mail sent from others e-mail will not be considered. The *object* of the e-mail *must* indicate *which* assignment you are submitting and *all* group members full name. Also, whoever of the group member submit the assignment *must* put in CC *all* the other member of the group.

The directory **must** contain:

1. The superserver.c source code (please, properly comment your source code);
2. The configuration file in *.txt* format;
3. A report of the **whole** assignment in *.pdf* format(i.e., both Task1 and Task2); report has to be written in *neat* and *clear* Italian.

What to write in the Report (1/2)

Length is not always a synonymous of quality: keep your report relevant to the topic you are required to write about, and try to be clear and effective in your explanation.

For **Task1**, report **must** contain:

- The diagram of the Super-server behavior, together with a brief description

For **Task2**, report **must** contain:

- A description of how you designed your Super-server
- Instructions of how you compiled your superserver.c source code
- Report the tests you carried out to verify the correctness of your source code:
  - Describe how you carried out the test;
  - Describe the experienced behavior and answer questions at slide 16;
  - Support your report with screenshot of the Terminal and considerations about the experienced behavior.