10k Problem Final Project

Ben Soer

A00843110

Table of Contents

Background

The 10k problem is a typical benchmarking problem used to determine whether a server is considered “scalable”. The purpose of the test is to execute 10 thousand client connections simultaneously transferring data to a single server. This problem has been previously explored in COMP8005 in C/C++. This implementation was done with also various server architectures so as to benchmark and compare the different architectures. This included *multi-process*, *select* and *epoll* architectures.

Inspired by the C/C++ project, this project attempts to recreate the C/C++ functionality in NASM x64 Assembly. The main learning objective is to explore and learn about various system calls available in the Linux x64 OS. Additionally, exploration into additional assembly structures and compilation methods are expected as part of the research and learning process of this project. Finally, as this is a bench marking system, some basic tests will be executed, comparing the execution times of C++ and Assembly based systems.

Implementation

Implementation of the 10k problem will be using a number of new system calls in NASM assembly. These will include *sys_fork*, *sys_select*, and *sys_epoll_create* for making system calls for the primary functions of each server. Additionally the development of a client that will simulate multiple connections needs to be developed. This will be implemented using the *sys_fork* call; allowing multiple connections to operate simultaneously on the same client program.

On top of these system calls will be the implementation and management of socket processing. Each server will have a buffer size and need to be configured to accept and process input. For this experiment, the client and server will be sending back and forth a simple message, simulating data always in transfer. With configured parameters, the data send back and forth, port numbers and IP addresses can all be adjusted in the architecture.

The client will also need to write its contents to file, adding the usage of the *sys_write* system call. This though will add the addition of file management and organizing the file system so that it can be easily interpreted by the reader, after a simulation concludes. As these files will contain stats information about the round trip times and data quantities processed, usage of addition and division functions will be used including potentially the use of floating point numbers on NASM assembly. As these have not been covered too far in depth, will add and additional challenge to the development of the client program.

Outcome

The outcome results of the project unfortunately did not meet the intended expectations due to a number of interesting and difficult factors. One of the key problems was the availability of documentation. Documentation on socket programming is heavily available, and thus the client and

multi-process assembly programs were able to be completed to large success, but documentation on implementing *epoll* and *select* server functionality was largely unavailable. The largest hindrences by the lack of documentation was the proper implementation of NASM structures and select's FD_SET macros.

NASM to our surprise supported the use of macros and structures. This widened much of the options and eased up a lot of the bit manipulation that was going to be needed to recreate them for the system calls. By creating structures, the amount of code needing writing dropped significantly. Additionally macro support allowed quick access to writing *htonl* and *htons* functions to give assembly level functionality in C like syntax. The problem though arose again in documentation as to how to properly format and code a NASM structure. Many structures found were made specifically for a particular use and thus custom implementations of structures were used for every instance a structure was required. An example below shows creating the *sockaddr_in* structure needed in order to connect to a server, and then the time structure required to benchmark the round trip time of the request

```
my_sa: istruc sockaddr_in
            at sockaddr_in.sin_family, dw PROTO_FAM
            at sockaddr_in.sin_port, dw PORT
            at sockaddr_in.sin_addr, dd LOCALHOST
            at sockaddr_in.sin_zero, dd 0, 0   ;  for struct sockaddr
        iend

tv: istruc TIMEVAL
            at  TIMEVAL.tv_sec,     dq 0
            at  TIMEVAL.tv_usec,    dq 0
        iend

tv2: istruc TIMEVAL
    at  TIMEVAL.tv_sec,     dq 0
    at  TIMEVAL.tv_usec,    dq 0
iend

%define tv.tv_sec tv+TIMEVAL.tv_sec
%define tv.tv_usec tv+TIMEVAL.tv_usec

%define tv2.tv_sec tv2+TIMEVAL.tv_sec
%define tv2.tv_usec tv2+TIMEVAL.tv_usec
```

The *TIMEVAL* structure requires *%define* calls in order to operate. Without these definitions, NASM returns a compilation error and cannot find the structures. In contrast, the *sockaddr_in* structure does not require them.

In an effort to still use these servers, C++ versions of an *epoll* and *select* server were created and then decompiled into Assembly. Here it was discovered that the g++ compiler did not event use the *epoll* and *select* system calls either, but instead called the C *epoll* and *select* library calls instead. Exploration through the assembly code to find the functionality of *select*'s FD_SET macros also proved to be ineffective due to the verbosity and memory manipulations carried out by *select* and the macros themselves. In contrast, the decompiled *epoll* server was easier to follow, but still used C library calls,

thus not posing as a solution to our implementation. Using the C library calls also appeared to remove the need for any of the structures previously needed in making system calls. The g++ decompiled assembly simply pushed its data onto appropriate registers and then called the C library function. An example of this can be seen below:

With no way of moving forward in developing the NASM *epoll* and *select* servers, the focus was redirected on benchmarking the various components that did exist. At this point, a NASM client and multi-process server exist. Additionally, C++ *epoll* and *select* servers had also been implemented. Efforts from here were then to make all components compatible with each other and then benchmark them.

The assembly client system, met a majority of the requirements for the project but lacked one of the key components. The assembly client was originally proposed to be multi-processed, but due to the inability to write content to file for reasons stated earlier, multi-processing was not possible as it would not allow processes to write to console. Due to this, the assembly client became a single processed application, and multiple instances would have to run in order to simulate multiple connections.

Below are the results of the benchmarking:

Client Program Used	Server Program Used	Average RTT
Assembly Client	Assembly Multi-Process Server	23 milliseconds
Assembly Client	C++ Epoll Server	18 – 20 milliseconds
Assembly Client	C++ Select Server	23 – 27 milliseconds

The results here show an interesting phenominon as typically *select* and *epoll* are designed to largely outperform multi-processesed architectures. These tests were conducted over localhost, with only a single client and thus show only marginal differences between the response time. From the graph though it can be observed that the assembly server performed moderately well and slightly better then the *select* server. The reason we believe this is, is because of the 1 less layer the Assembly has to deal with over the *select* C++ does. When observing the generated assembly from g++, it is obvious that many more steps happen then in contrast to the assembly code that makes up the multi-process server.

Conclusion

The end results of this project were quite disappointing compared to what was intended to be achieved. From this experience though, new insights on other NASM assembly components and coding architecture were acquired. Discovery of structures and macros provides new areas to explore and further research in. By understanding these further, returning back to this problem would be possible in future in order to properly implement the *epoll* and *select* servers using system calls. On top of this, evidence of assemblies performance even over C was able to be observed and insight into some of the workings of the g++ compiler were explored. Although an exciting and challenging experience, this project at this point in time was too ambitious and requires more work and study into the NASM assembly language before reexamining.

References

Assembly Client Code References

- https://github.com/arno01/SLAE/blob/master/exam1/shell_bind_tcp.nasm
- https://forum.nasm.us/index.php?topic=1811.0
- https://ubuntuforums.org/archive/index.php/t-1105208.html
- http://stackoverflow.com/questions/31373650/nasm-data-strings
- http://stackoverflow.com/questions/21968011/nasm-printing-unix-time
- http://syscalls.kernelgrok.com/
- http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- http://www.nasm.us/doc/nasmdoc3.html
- https://www.csee.umbc.edu/portal/help/nasm/sample.shtml
- http://stackoverflow.com/questions/3941771/nasm-random-number-generator-function

Select Server Code References

- http://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction