Enforcement of Execution Properties in Linux

William A. Gozlan
william.gozlan@wustl.edu

Washington University in St. Louis
McKelvey School of Engineering
Department of Computer Science and Engineering

Advised by Dr. Christopher Gill and Marion Sudvarg

Defended December 2, 2022

Abstract

In this project, we explore how to apply execution properties from other architectures to Linux. We discuss some ideas, implement, and test them, including a client-server architecture based on CAmkES and the seL4 Microkernel. We see how related processes can constrain execution on behalf of each other in this way. We also extend to applying these ideas in LITMUS-RT and comparing how and to what extent LITMUS handles execution properties when compared to Linux. Many new ideas for future works are also explained.

1 Background

To begin, I will provide some background information so that the more intricate parts of this project are clearer. These may be a review for readers that have experience in real-time systems, the C programming language, and operating systems.

For my own background information, I referenced numerous books at the outset of this project. This includes *Hard Real-Time Computing Systems* by Giorgio C. Buttazzo, *Pthreads Programming* by Bradford Nichols, and *Programming with POSIX Threads* by David Butenhof. Additionally, the research papers *A Concurrency Framework for Priority-Aware Intercomponent Requests in CAmkES on seL4* by Marion Sudvarg and Chris Gill, and *Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time* by Anna Lyons et al. serve as excellent background for this project. Many of these concepts and ideas were discussed further through weekly meetings.

1.1 Real-Time Systems

Intel defines real-time systems as "Any information processing system with hardware and software components that perform real-time application functions and can respond to events within predictable and specific time constraints.[1]" For the purposes of this project, real-time systems is constraining the execution time of programs so that we have a bound on how long a computation will take to run.

This has applications in control systems, robotics, and self-driving vehicles. As an example to see how and why, let's consider an automatic steering and acceleration system in a self-driving car. Suppose that this system read inputs off sensors and adjusts steering accordingly with a simple algorithm. If we are waiting on this steering system to compute, we may not have the ability to adjust the speed, since we are busy executing the steering code. Real-time systems can fix this by not only limiting how long execution takes, but also ensuring the computation finishes.

[1] "Real-time Systems Overview and Examples". In: URL: `https://www.intel.com/content/www/us/en/robotics/real-time-systems.html`.

1.2 Threads in C

While C does not directly support threading via language standards, the POSIX standard defines POSIX threads, known as pthreads, for all multi-threading needs. We can create a new thread in C with the `pthread_create` function. Since there are no higher-level wrappers for abstraction, it is up to the programmer to correctly use threads. Part of this is clean-up, and it is common to wait for threads to finish before exiting the program with the `pthread_join` function. Again, all of this must be managed by the programmer which makes threading in C not trivial.

If we want multiple processes to be able to access the same memory, we can use shared memory. The basic idea of shared memory is to create and resize a memory region in one process, then access the same region from another process. The C functions that allow for this are `shm_open`, `ftruncate`, and `mmap`. Specific details on shared memory and these functions to set it up can be found in the Linux man pages.

1.3 Server-Client Framework with FIFOs

Recall that a FIFO concept is a first-in-first-out data structure in which the oldest item is used and removed first. A Linux FIFO is a named pipe where multiple threads can read and write data to share it while keeping order. Using the `mkfifo` wrapper function in C, we can set up FIFOs between different programs to communicate by sending and receiving data!

We can use this ideology to build a server and client architecture in which multiple concurrent clients can connect to a single server, with the order of the requests maintained. The server can then process these queued requests one at a time. Figure 1 illustrates this below:

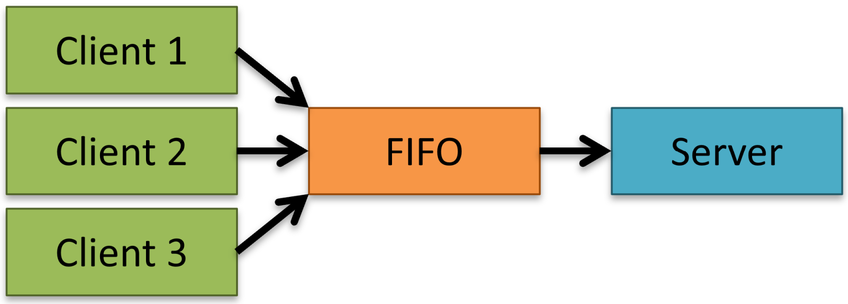


Figure 1: Simple Client-Server Architecture using a FIFO

2 Introduction

With the background information set, we will now lay out the initial motivation and goals for the project. Additionally, we had many changes as ideas evolved over the yearlong project, so I address these. Overall, the project did lead to meaningful outcomes, which are nice to keep in mind when considering the entire evolution.

2.1 Motivation

The main motivation for this project was the seL4 Microkernel. seL4 is an operating system microkernel, so it has just the bare minimums to make it run, making it lightweight. seL4 focuses on performance, since it is a microkernel, and security, with process isolation beyond standard Linux. This includes, but is not limited to, address spaces with enhanced memory protection, threads and scheduling protection using temporal protection, and controlled communication across isolation boundaries for interprocess communication. As you can see, many aspects of the system were considered for performance and security in seL4.

A key idea in seL4 is a notion of budget and period, which are bandwidth limitations of CPU time that the kernel enforces per process. So a process will only use its budgeted microseconds out of the period microseconds of CPU time. For example, a budget of 10 microseconds and a period of 100 microseconds allows that task to run for 10% of the CPU time.

Additionally, seL4 comes with capabilities to control system resources. These capabilities can be thought of as pointers with access privileges, though the formal definition is "a unique, unforgeable token that gives the possessor permission to access an entity or object in system.[2] "

Finally, CAmkES (component architecture for microkernel-based embedded systems) is a seL4 framework for the client-server architecture we saw previously using FIFOs.

2.2 Goals and Pivots

We originally sought to use CAmkES to explore the extent to which execution properties can (or cannot) be ensured. However, goals shifted early on to bring ideas from seL4 to Linux, such as capabilities and budget constraints. We thus explore these ideas on Linux and investigate how and to what extent we can keep these ideas.

The CAmkES ideology of a client-server architecture pattern remained, and this idea was a core part of the project. The basic idea is the following. First, a server thread runs with privileges for execution. Then, a client thread connects to the server with a request for execution. The server then runs computations on behalf of the client's request. On completion, the server sends the computation result back to the client.

2.3 Outcomes

We achieved this basic notion, described above, with some further properties as well. While a server program runs to do work on behalf of a client, the server's execution from the client should inherit its execution properties (priority, budget, etc.). This was a key outcome of the project. It may be useful to keep this in mind in the next section.

3 Program Architecture Development

With the background knowledge, motivations, and goals set, we will now go discuss the development of a server-client architecture program that maintains the ideas from seL4 but on the standard Linux platform. This includes the design and implementation for both a server program and a client program, though they both interact at runtime.

3.1 Basic Overview

As hinted earlier, we will use a Linux FIFO to queue client request connections and accept them as they arrive. In other words, the server will poll for these client requests. The server is responsible for just initializing itself and waiting for a client to connect.

The client, on the other hand, is responsible for setting itself up and connecting to the server, to run some computation it wants to run but does not have the privileges to do. This computation is assumed to be something computationally intensive, so the client is not permitted to run it on its own; it must have the server do it on its behalf and wait for the result, which the server should send back.

When client connects to the server and the server accepts a client connection, the server is agreeing to do the computation for that client then send the result back to the client which polls for a result. Of note, the requests are on the same system, so internet connections are not made, but rather local connections with the FIFO.

This concept is illustrated below in Figure 2, which is read with time going from top to bottom.

The various stages of execution for both programs is shown in white, and the arrows indicate the request direction. We see that the server initializes itself reasonably quickly, then polls until the client makes the request, at which time it computes. When the computation is complete, it sends back the result to the client, which finishes, while the server loops back to poll for the next client connection.

Since the client waits on the server in the second half of the timeline, we have a two way communication channel. As such, two FIFOs are used. The first is the servers FIFO and that was already discussed. The second is from the client, and the role of it is so that the client can block on a `read`

[2] "Capabilities". In: URL: `https://docs.sel4.systems/Tutorials/capabilities.html`.

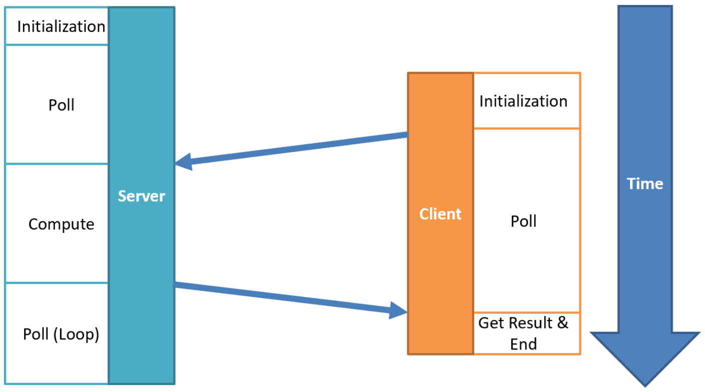


Figure 2: Client-Server Architecture Timing Overview

system call to it, so it knows when the server completes the computation. In other words, the client makes a `read` system call to its own FIFO, while in the polling loop. When the server finished that clients request, it writes to the clients FIFO to indicate completion, for which the client is then notified as the read call completes.

This means we have a FIFO for the server and a FIFO for each client. As such, for N clients we would have N+1 FIFOs.

3.2 Workload

Desiring a heavy, CPU bound workload, we chose matrix multiplication as it is a natural choice for such idea. The matrix multiplication code is part of the server program, and the client treats it as a black box since it does not do the multiplication itself. The client sets up the matrix coefficients in its initialization phase.

Since the client and server are separate programs, and thus run as separate processes, they typically cannot access each others memory space without a memory access violation which would cause a segmentation fault. This is where POSIX shared memory comes into play! The client creates a shared memory region and places the matrix coefficients into it; the server then accesses this region, and stores the result back into it, and signals the client it finished, once the matrix multiply is complete.

When a client makes a request, it includes the matrix size n, for the $n \times n$ matrix multiplication it wants to do. Matrix coefficient values need not be sent as they are in shared memory. As such, we share data, and only pass pointers to that data between the processes.

It is at this point when we realized we can draw ideas from seL4 into Linux, based on this shared memory. Linux shared memory can be seen as a file by looking at the Linux virtual file system. In seL4, on the other hand, capabilities are used so other processes not only do not have permission to access the shared region of memory, but they do not know it even exists. This idea could be brought to Linux by hiding elements of the virtual file system using scoping with Linux namespaces. While we did not have time to implement this, it could be a very interesting future work.

3.3 Achieving Parallelism with Threading

The design and implementation described thus far works perfectly, but has a big caveat. A client connecting to the server would have to wait for all the current client connections to complete before it gets a turn for the server to even acknowledge that it is there.

A natural solution to this problem is to achieve a parallel architecture by using threads. Simply put, when a client connect, the server should launch a new thread, with `pthread_create` for that clients computation to be completed in. The server thread itself should focus on polling for new connections and launching worker threads to do the matrix multiplication, instead of doing it itself. This allows for multiple client requests to come in at the same time and be processed, with no perceived waiting.

Threading and parallelism always introduces the risk of data hazards (races). However, all clients and their respective threads launched are independent. Additionally, the client fills the shared memory before making a request to the server, at which point it never writes to memory again. As such, we have a fully thread safe architecture without the need from locking, which would introduce additional overhead and slowdown.

3.4 Applying Real-Time Components: Scheduling and Priority

Thus far we have considered all clients have equal priority on the system. However, suppose some are more time sensitive than others, and must be done sooner. We can achieve this by using Linux priority values. At first, we used Linux user priority values to denote priority, which have a range [-20, 19]. However, we switched to Linux real-time priority values, with range [0, 99], instead, as they allow for real-time scheduling policies.

The client is responsible for sending its priority value to the server upon connecting, along with the matrix size n as previously mentioned. As such, we assume clients to be "nice", in that they will not simply use the highest priority.

In terms of Linux schedulers, we have used and tested two different ones: `SCHED_FIFO` and `SCHED_RR`. The scheduler can be set with a command line flag in the server, but more on this later. FIFO scheduling involves using the first task in a queue (with highest priority) and running that first. Round Robin scheduling is similar, but adds time slices for fairness, and cycles through them in order based on real-time priority value.

3.5 Visualizing the Updated Architecture

With all these new ideas in place, Figure 3 below illustrates how two clients may be connected at once. Again, recall time moves from top to bottom.

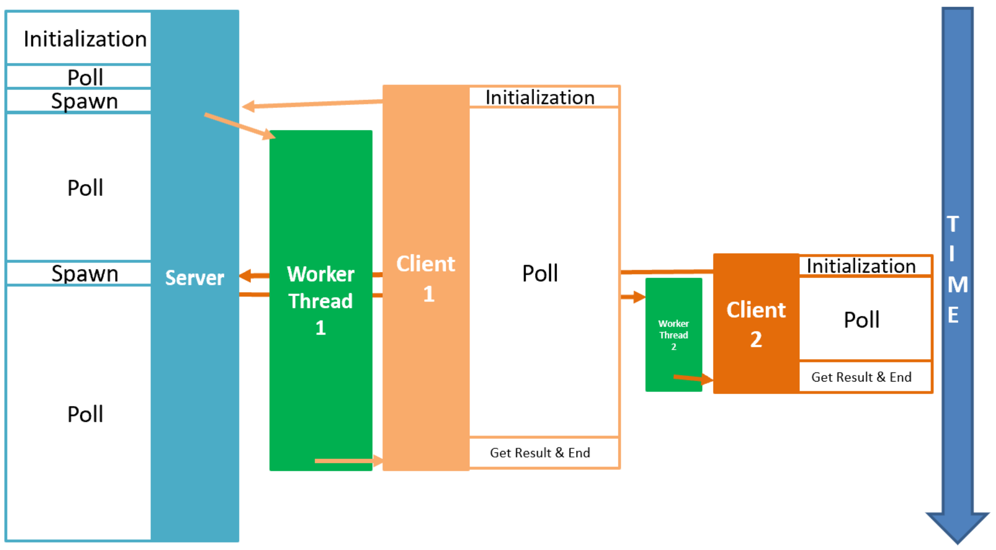


Figure 3: Client-Server Architecture Timing Overview

Here, we see that client 1 is connected before client 2. Yet, client 2 is able to connect to the server while client 1 is still connected and waiting for data. This illustrated the parallelism aspect. Additionally, client 2 connects after 1, yet is finished before. The is because client 2 has a smaller matrix size and a higher priority than client 1, and thus client 2 takes less time.

3.6 Constraining Clients with Linus Control Groups

We would like for clients, and the server code executed on their behalf, execution to be constrained. While priority values and scheduling policy decide what runs first, we seek a bandwidth limitation for that process, with regard to overall CPU time, much like the notion of budget and period from seL4.

We can accomplish this by applying Linux control groups (cgroups). Linux control groups limit system resources for a processes. While they can limit many aspects of the systems resources, like memory and I/O, we are interested in using the to constrain CPU time. To constrain CPU execution with cgroups, budget and period values are used, just like seL4.

A new architectural pattern we propose here, is that when a client connects to the server, the launched server thread should join the clients cgroup. As such, system resources continue to be limited on the server, since the computation is for the client. In other words, the clients execution should be constrained all the way through.

While we got a basic version of this working, there sadly was not enough time to go further with the idea, due to LITMUS, which is discussed later. We leave this as a future work, as well as expanding to constrain other system resources, like memory and I/O, also with cgroups.

3.7 Design Considerations

It is at this point that we can take a step back and address a few design choices made along the way.

i. One issue with using FIFO's for multiplexed polling is that both ends of the FIFO must be simultaneously open before we can process inputs/outputs of it. As such, a read or write will not show up unless the other end of the FIFO is open. As a workaround, we use a separate thread in the server to hold the other end of the FIFO open. This is a common workaround, and works to get the intended result.

ii. Though the server is multi-threaded, we wanted to have both a single-core and a multi-core version. We achieve this with an optional flag passed to the server program. For the single-core version, all spawned threads are pinned to CPU core zero, with the exception of the thread that holds the FIFO open, since that is not part of what we are testing. For the multi-core version, the scheduler decides which of the four cores to use.

iii. In thinking about allocating matrices for the matrix multiplication, either static allocation and dynamic allocation would be suitable. We first built and tested static allocation, since it is simpler. However, we later built a dynamic version, which is used in testing. If time permitted, we thought to compare the performance of these two, but it could be left as a future work.

iv. For a matrix multiplication, there is a simple, naive method of multiplication as well as a cache optimal method that uses blocking to reduce the number of cache misses. We briefly looked at both of these and compared them, using a preprocessor directive to change the option between the two. Results showed that for a size 500 square matrix, the cache optimal version had a 60x less miss rate compared to the naive matrix multiply. This value increased 135x less miss rate for size 1000 matrix. We saw that matrix multiplication dominated the cache performance over scheduling policy, which did not lead to a significant change. The data for this was collected using hardware performance counters for recording the number of cache hits and misses. While we did not continue down this path for this project, it could be continued as a future work.

v. Finally, it is important to address how the server program should end. To end the server program cleanly, we registered a signal handler to set an atomic flag which is checked for in the server loop. When this flag is set, the server does the cleanup before returning. We maintained a Linked List of client connections and ensure those end before ending the program, to avoid thread leaks. Similarly, we deallocate all memory to avoid memory leaks, as well as close resources like the FIFO and shared memory.

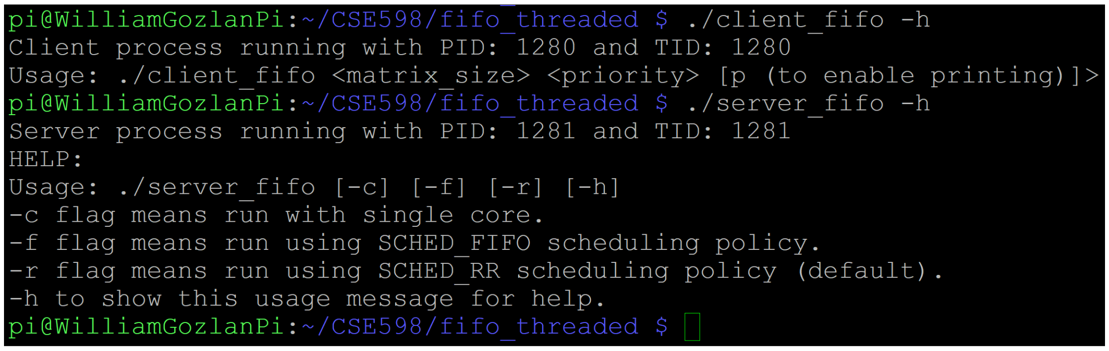
4 Testing, Tracing, and Validation

With such an involved architecture, we wanted to ensure correctness of design and implementation through testing and validation.

All testing, code compilation, running, and tracing was completed on a Raspberry Pi Model 3B+. Since testing was a large endeavor, testing incrementally proved to be the way to go. We first tested a single core, non-threaded version. Then, stepped this up to implemented threading, but used a single-core by pinning all threads to core zero. Finally, the place where most problems can appear is the multi-core, multi-threaded version. However, this incremental testing allowed us to fix small problems before they became bigger and much harder to find.

4.1 Basic and Advanced Functional Testing

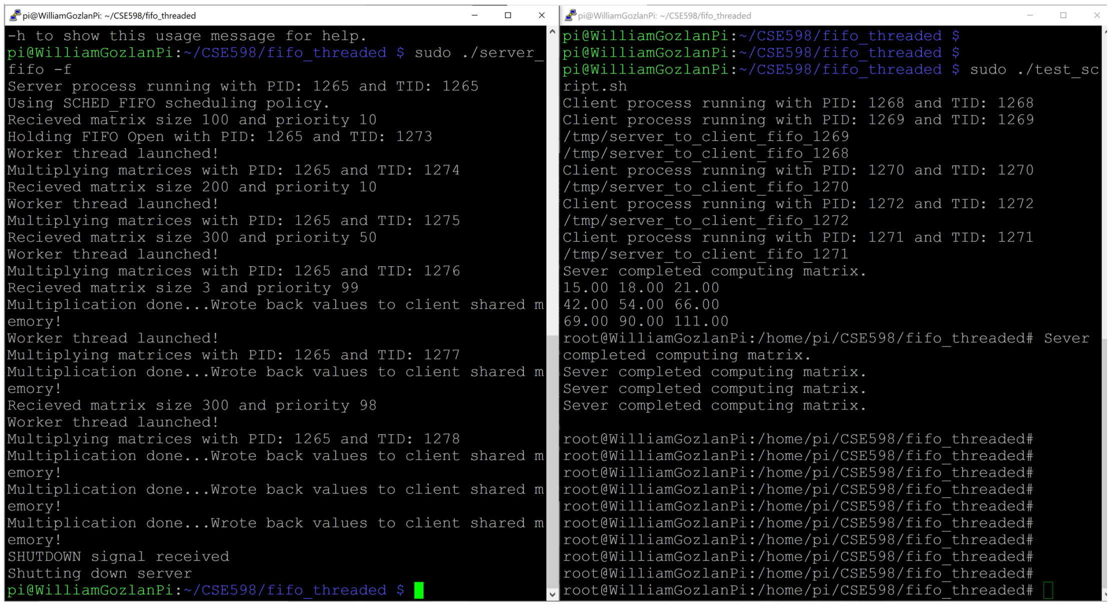
Figure 4 below illustrates the usage message of both the server and client programs, so you can see what options they accept.



```
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $ ./client_fifo -h
Client process running with PID: 1280 and TID: 1280
Usage: ./client_fifo <matrix_size> <priority> [p (to enable printing)]>
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $ ./server_fifo -h
Server process running with PID: 1281 and TID: 1281
HELP:
Usage: ./server_fifo [-c] [-f] [-r] [-h]
-c flag means run with single core.
-f flag means run using SCHED_FIFO scheduling policy.
-r flag means run using SCHED_RR scheduling policy (default).
-h to show this usage message for help.
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $
```

Figure 4: Usage Message of Server and Client Programs

We used tracing and Kernelshark to validate the schedulers and priority, and ensured everything is working as expected. We also created a verbose mode to print Process ID's and crossed referenced those with Kernelshark, on a per-thread basis to ensure all threads are correctly made and closed. To ensure that the server is truly able to handle parallel requests, a simple bash script bombards it with parallel requests. Figure 5 below shows the output of this bash script run, using verbose mode. The server program is on the left and five client programs connect at once with the bash script on the left of the figure. Of note, it is of a multi-core run using the `SCHED_FIFO` scheduling policy.



```
-h to show this usage message for help.
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $ sudo ./server_
fifo -f
Server process running with PID: 1265 and TID: 1265
Using SCHED_FIFO scheduling policy.
Recieved matrix size 100 and priority 10
Holding FIFO Open with PID: 1265 and TID: 1273
Worker thread launched!
Multiplying matrices with PID: 1265 and TID: 1274
Recieved matrix size 200 and priority 10
Worker thread launched!
Multiplying matrices with PID: 1265 and TID: 1275
Recieved matrix size 300 and priority 50
Worker thread launched!
Multiplying matrices with PID: 1265 and TID: 1276
Recieved matrix size 3 and priority 99
Multiplication done...Wrote back values to client shared m
emory!
Worker thread launched!
Multiplying matrices with PID: 1265 and TID: 1277
Multiplication done...Wrote back values to client shared m
emory!
Recieved matrix size 300 and priority 98
Worker thread launched!
Multiplying matrices with PID: 1265 and TID: 1278
Multiplication done...Wrote back values to client shared m
emory!
Multiplication done...Wrote back values to client shared m
emory!
Multiplication done...Wrote back values to client shared m
emory!
SHUTDOWN signal received
Shutting down server
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $
```

```
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $
pi@WilliamGozlanPi:~/CSE598/fifo_threaded $ sudo ./test_sc
ript.sh
Client process running with PID: 1268 and TID: 1268
Client process running with PID: 1269 and TID: 1269
/tmp/server_to_client_fifo_1269
/tmp/server_to_client_fifo_1268
Client process running with PID: 1270 and TID: 1270
/tmp/server_to_client_fifo_1270
Client process running with PID: 1272 and TID: 1272
/tmp/server_to_client_fifo_1272
Client process running with PID: 1271 and TID: 1271
/tmp/server_to_client_fifo_1271
Sever completed computing matrix.
15.00 18.00 21.00
42.00 54.00 66.00
69.00 90.00 111.00
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded# Sever
completed computing matrix.
Sever completed computing matrix.
Sever completed computing matrix.
Sever completed computing matrix.

root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi:/home/pi/CSE598/fifo_threaded#
```

Figure 5: Output of Test Validation Run

We use five concurrent clients since it is enough for each client to consume a core with one needing to share a core, since we are on a four-core machine. Yet, it is not too much that the output is overwhelming.

All these tests ultimately passed which validated the design and implementation.

4.2 Real-Time Component Testing

In tracing the server and client's execution, we generated many traces. Here, we will go over some of these traces that validate the real-time components of real-time priority and real-time scheduling policy.

Of note, we modified the view of Kernelshark to focus on a per-thread view rather than the default core view. To do this, deselect all cores in the plots menu, and select all tasks you made from the plots menu.

Shown below in Figure 6 is the Kernelshark trace when five clients of varying priority and matrix size connect at once, with the server using a FIFO scheduling policy and the single-core option, so all threads are pinned to run on core zero. On the left, we see the thread names, with the big blue thread being the loop to hold the FIFO open in a separate thread. The green traces below that is the server processing client requests, specifically all five at once. We see that only one runs at a time since we only are using a single core here. Moreover, threads run to competition due to FIFO policy, unless a higher priority thread comes along. This does occur for the last two clients on the bottom; the first client begins execution until the last client takes over execution and runs to completion, by which time the first thread resumes. The second thread did have a higher priority value and thus ran first.

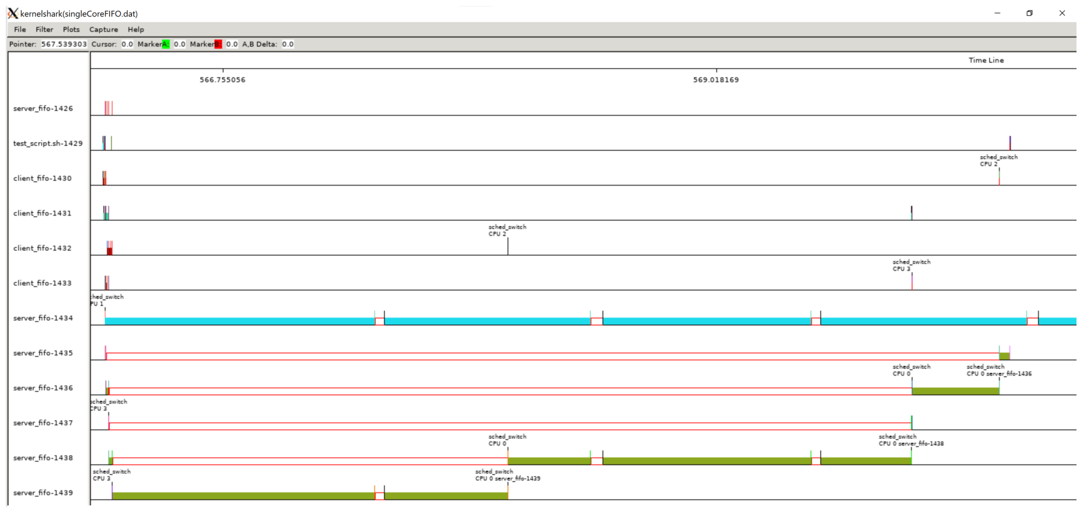


Figure 6: Kernelshark Output of Single-Core FIFO Run

Similarly, we can now look at a multi-core trace using round-robin scheduling, again with five concurrent client connections. Figure 7 shows the Kernelshark trace of this run below, with a specific portion highlighted in red:

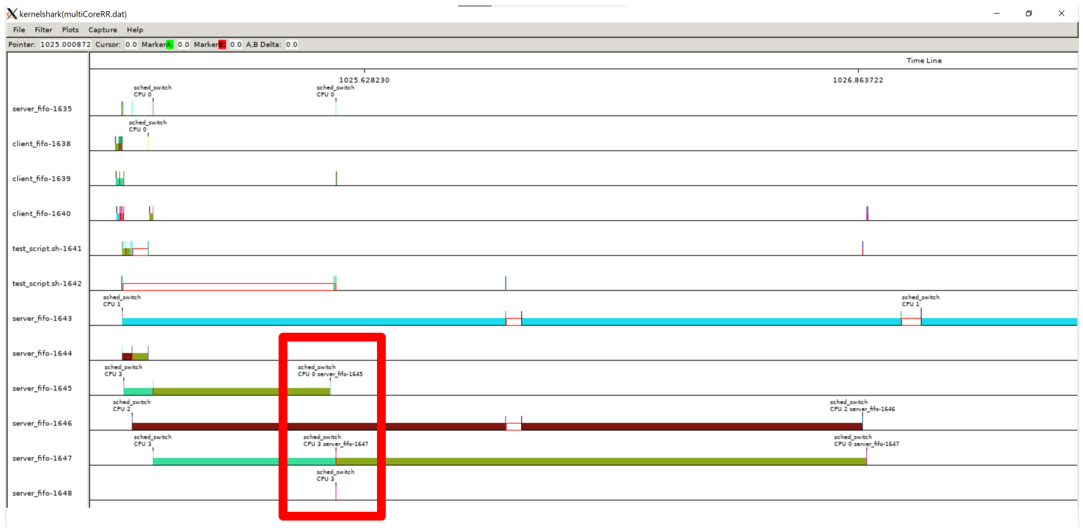


Figure 7: Kernelshark Output of Multi-Core Round-Robin Run

Zooming into the red highlighted portion of Figure 7 above, we see the following, which is shown in Figure 8.

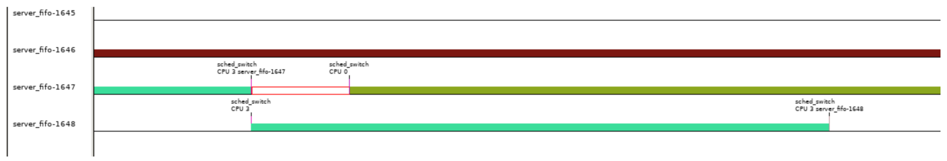


Figure 8: Zoomed-In View of Kernelshark Output of Multi-Core Round-Robin Run

Looking closer at Figure 8 we can see that the last task, which is higher priority, takes the core from the lower-priority task. Luckily for the lower priority task, however, it does not have to wait too long because Core 0, in green, becomes available, and that thread of execution is migrated there. This shows that Linux is fairly efficient at thread migration on a multi-core system.

5 Expanding to LITMUS-RT

With the design and implementation fully validated, we sought out another kernel. LITMUS Real-Time (RT) is "real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization.[3]" In other words, LITMUS is a Linux Kernel extension that seeks to bring real-time multi-processor capabilities to Linux. Originally, we looked to LITMUS to see how they may use cgroups to constrain multiple cores, which was the idea discussed earlier. However, this then transitioned to comparing LITMUS and Linux.

5.1 Installing LITMUS

Installing LITMUS was a major bottleneck. The basic idea was to clone a copy of the Linux kernel from GitHub, then cherry-pick the commits from LITMUS to apply to the original Linux kernel. Then, cross-compile the kernel for the Raspberry Pi using x86. However, since LITMUS is much older, many hardware features, like internet access using either Wi-Fi or ethernet, broke when placing the SD card onto the Raspberry Pi.

We found luck in installing LITMUS to a Raspberry Pi 2. Since the Raspberry Pi 2 is older, it did not have Wi-Fi, and the drivers for ethernet were compatible.

The installation guide for LITMUS on Raspberry Pi from the LITMUS site proved helpful overall, though. I did get in touch with the author of it, Filip Marković of Mälardalen University in Sweden, along the way to help troubleshoot some of the issues we encountered, with the hope of the guide being updated to address these.

LITMUS also required plug-ins for scheduling and tracing, which we installed. Finally, we also needed the `trace-cmd` utility on LITMUS, to trace execution. While the `apt-get` package manager did not work since the Raspberry Pi 2 is no longer supported, we installed the needed tools and dependencies from source, by cloning the repository, building, and installing locally. This was a workaround for sure, but it worked quite well.

5.2 Simple LITMUS Testing

After installing LITMUS and the various LITMUS plug-ins for scheduling, we were able to run the server-client test programs on LITMUS, and trace with Kernelshark to compare the standard Linux kernel to that of LITMUS.

Figure 9 below shows the Kernelshark traces for the same run on both a standard Linux kernel system, shown on the left side of the figure, and the LITMUS kernel, shown on the right side of the figure. It is with five clients concurrently connecting to the server, with multi-core enabled and a FIFO scheduling policy.

As you can see, they are very similar, though some differences arise. Processes to cores are shuffled around due to the scheduler deciding that at runtime. Looking at the thread that holds the FIFO

[3] "LITMUS-RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems". In: URL: `https://www.litmus-rt.org/`.

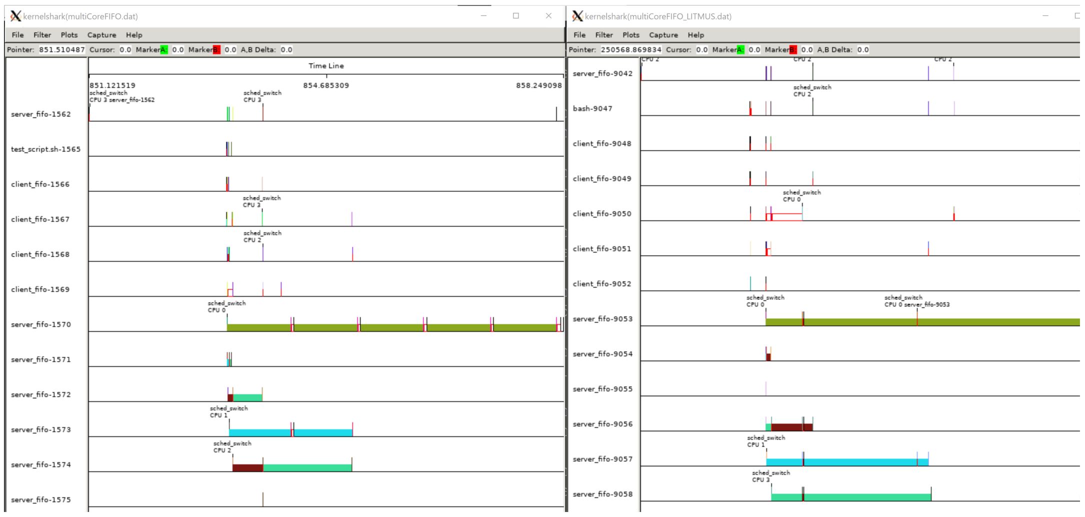


Figure 9: Kernelshark Traces for Standard Linux (left) and LITMUS (right) For The Same Run

open, which is a green trace in the middle, we see that gaps in the traces are due to global limits of execution time. The Linux one is much less than that of LITMUS, which shows that LITMUS has a larger global limits of execution time compared to standard Linux. We measured these times and they match the standards set forth in the documentation.

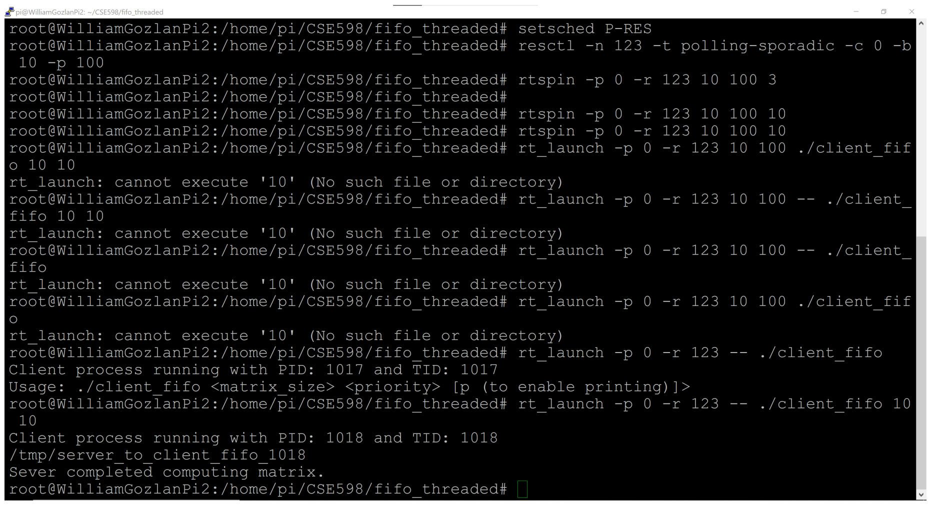
5.3 Constraining Processes with LITMUS Reservations

The LITMUS plugins include a custom LITMUS reservation scheduler called `P-RES`. This enables the use of LITMUS reservations. LITMUS reservations are entities in LITMUS, independent of processes, so you can constrain execution with a period and budget, just like in seL4. You then can launch a process to run with a previously created reservation.

We use the `resctl` command in LITMUS to create a reservation, passing in a reservation ID, that you pick, and optional period and budget values. You then launch a process specifying to use this reservation with the `rt_launch` command. Under the hood, LITMUS schedulers enforce the CPU constraints on that process. This is very analogous to what we were looking to do with cgroups in Linux, if time had permitted.

There was a delay in doing the above, due to slightly incorrect documentation in the LITMUS manual. The manual states that you should run `rt_launch` and specify a period and budget in it. However, this does not make sense as the period and budget is already in the reservation that it uses. As such, the command fails, with an unhelpful error message, if doing this exactly as written. Instead, you should *not* include a period and budget in the `rt_launch` command. Figure 10 below shows both the documented yet incorrect usage as well as the correctly inferred usage with launching a client (server run not pictured).

With the hold-up from that issue, there was no time to continue this idea. However, the goal was to modify the server so that upon a client's connection, the spawned server thread joins the same reservation as the client, thus continuing the constrained execution. This, as well as the ideas that follow, would be an excellent future work.



```
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# setsched P-RES
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# resctl -n 123 -t polling-sporadic -c 0 -b
 10 -p 100
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rtspin -p 0 -r 123 10 100 3
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded#
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rtspin -p 0 -r 123 10 100 10
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rtspin -p 0 -r 123 10 100 10
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 10 100 ./client_fif
o 10 10
rt_launch: cannot execute '10' (No such file or directory)
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 10 100 -- ./client_
fifo 10 10
rt_launch: cannot execute '10' (No such file or directory)
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 10 100 -- ./client_
fifo
rt_launch: cannot execute '10' (No such file or directory)
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 10 100 ./client_fif
o
rt_launch: cannot execute '10' (No such file or directory)
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 -- ./client_fifo
Client process running with PID: 1017 and TID: 1017
Usage: ./client_fifo <matrix_size> <priority> [p (to enable printing)]>
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded# rt_launch -p 0 -r 123 -- ./client_fifo 10
 10
Client process running with PID: 1018 and TID: 1018
/tmp/server_to_client_fifo_1018
Sever completed computing matrix.
root@WilliamGozlanPi2:/home/pi/CSE598/fifo_threaded#
```

Figure 10: Incorrect and Correct Usage of `rt_launch`

6 Conclusion

In conclusion, this project illustrated and laid the groundwork for how execution properties of seL4 can be applied in Linux. It analyzed various aspects of real-time principles to prove out initial ideas through design, testing, and experimentation.

6.1 Takeaways

This project allowed me to learn much about the world of real-time systems, execution properties, seL4, and LITMUS. I not only learned a lot of new material, but I also expanded on my previous knowledge from courses by applying and iterating on the principles in this project.

6.2 Future Works

As mentioned throughout this paper, there are many opportunities for future work. These include analyzing aspects of performance like the cache we touched on, capabilities in Linux by potentially using Linux namespaces to mask shared memory regions from other processes, and continuing the real-time execution properties in LITMUS. Another idea is using cgroups in Linux for lightweight bandwidth constraints for CPU, as well as expanding that to include I/O and memory. All in all, there are endless possibilities for where this project can go in the future.