Imperial College London

Department of Computing

Exploring Asynchronous Operating System Design Using Rust

Background and Progress Report

Author:
Yifan Wu

Supervisor:
Lluis Vilanova

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

June 2022

Contents

Chapter 1

Introduction

System call is one of the most important components in the Operating System (OS) since it serves as the interface between the OS and applications. Applications can request services provided by the OS via system calls, and the OS is responsible for allocating resources to the applications according to their requests while the security requirements including the isolation between applications must be guaranteed.

In the classical OS design, system calls are synchronous. To be specific, the application should execute a dedicated instruction provided by the Instruction Set Architecture (ISA) such as *syscall* on x86_64 platform or *ecall* on RISC-V platform. After that, the execution of the application is immediately stopped and an interrupt is triggered and captured by the processor. Then, the processor mode or the privilege level is switched from the user mode to the kernel mode, which means that the control is transferred to the code of the OS. The OS saves the execution context of the application, handles the system call requested by the application and then restores the execution context of the application. Finally, another dedicated instruction (e.g., *sysret* on x86_64 platform or *sret* on RISC-V platform) is executed, leading to the result that the processor mode is switched back to the user mode and the execution of the application is resumed.

During a system call, we have to take the overhead of saving and restoring the application context which requires storing several general-purpose registers to the memory or vice versa. However, there is also a much higher indirect overhead on a switch of the processor mode which we tend to ignore. This overhead is called **processor state pollution**, which is mentioned in [8]. We know that there are many mechanisms in modern processor architecture which accelerate the instruction execution assuming the temporal and spatial locality of accessed data. For example, multiple levels of data and instruction caches and prefetching are used to reduce the memory access latency on average; Translate Look-aside Buffer (TLB) can eliminate unnecessary page table traversals when translating a virtual address to a physical address in a virtual address space; branch prediction can help us quickly determine whether we should take the branch and what the branch target is. These components are called processor states and they can be trained as a flow of execution is consecutively executed on the processor to learn and record some information about the locality of the execution. Thus, the performance of this execution in terms of Instructions Per Cycle (IPC) can be maximized.

Why are the processor states polluted on a system call? Imagine that the processor has run an application for a period of time, now the processor states contain some information about the locality of the application's execution so that the application's code can be executed efficiently. If the processor traps into kernel mode to handle a system call, then the processor starts to work on a kernel stack and accesses kernel data structures which are completely isolated from the data accessed by the application, which means that the processor states have to be trained from scratch to fit into kernel's execution. Since the capacity of the processor states are limited, these kernel's locality information will substitute application's locality information, which we call the pollution of the application's processor state. After the system call returns and the application's execution is resumed, the polluted processor state will degrade the performance of the application's execution until the impact of the kernel's execution diminishes. We can see that the performance of both the application's and kernel's execution suffers from the processor state pollution, which is proven by the results of experiments [8]: the kernel-mode IPC is significantly lower than the user-mode IPC(up to 8x slower), and the degradation of user-mode IPC is up to 60% where the processor state pollution dominates the degradation if the system calls are invoked at a medium frequency.

How can we mitigate this problem? In general, we need to reduce the frequency of trapping into the kernel mode. There have been a number of works[8, 5, 3] which tried to achieve this goal. Among them, FlexSC [8] was a very successful attempt which made the system calls asynchronous. We are more familiar with synchronous system calls. This is because they are simpler and they can provide maximized transparency for applications: The application believe that the control is immediately transferred to the kernel after executing the dedicated instruction, and it is immediately returned back after the system call returns, which is as easy and natural as a subroutine call. However, given that the overhead of synchronous system calls is substantial, it is reasonable to sacrifice the transparency in pursuit of performance.

Apparently, asynchronous system calls are more complicated since applications should also be aware of these system calls. In FlexSC, there is a threading library called FlexSC-threads which is compatible with multithreading applications and responsible for transferring synchronous system calls issued by applications to asynchronous system calls. On a system call, FlexSC-threads only add a system call entry in system call pages shared between the application and the kernel rather than trap into the kernel mode. At the same time, FlexSC-threads blocks the current user-level thread and switch to another thread which is ready. If all user-level threads are waiting for the result of system calls, FlexSC-threads issues a synchronous system call to trap into the kernel mode. The kernel looks at the system call entries submitted by the application, handles these system calls accordingly and writes the results back to these system call entries. After at least one system call has completed, the control is given back to the application. Then, it is FlexSC-thread's turn to wake up threads according to the system call entries. We can see that FlexSC-threads also serves as a user-level scheduler. Additionally, FlexSC requires some modifications in the Linux kernel.

Although FlexSC significantly reduces the number of switches of processor mode, it still has some limitations. Firstly, it did not fully take advantage of the asynchronous system call interface, which allows the OS to reorder incoming system call requests according to requirements. FlexSC only handles system calls as the order they are issued, but obviously it could have done better if it can schedule system calls intentionally, especially in a multicore environment. Secondly, FlexSC uses threads as scheduling units in the OS, which leads to a higher memory consumption and context switch overhead. We want to use stackless coroutine instead since we believe that it is a more promising programming model and it is relatively easy to use in some programming languages like Rust.

We aims to further improve the OS implementation based on the asynchronous system call interface introduced by FlexSC in this project. In other words, the target of this project is designing a new asynchronous OS architecture which can handle asynchronous system call requests efficiently. The efficiency can be evaluated using the following kernel-side metrics:

- better core specialization and data locality
- lower synchronization overhead
- lower memory consumption and context switch overhead based on stackless coroutines
- be flexible enough to work on different hardware platforms including symmetric and asymmetric multicore environments

Although we prioritize the performance, we will also consider the compatibility with existing applications if possible.

The main challenge of this project is how we can reorder and distribute incoming system call requests to multicores to satisfy these performance requirements. A simple solution is just dedicating every core to a set of system calls. For example, core 0 is responsible for memory management while core 1 takes the responsibility for I/O operations. In this way, every core will execute and access a fixed set of code and data structures for a long time, which increases core specialization and data locality. For the reason that a type of system calls which access specific data structures protected by a group of locks are only handled on a single core, the data contention is also alleviated.

Inspired by the SEDA architecture[9], we can come up with a fine-grained architecture. We can divide the OS into several asynchronous modules, each of which represents a specific kind of kernel functionality. These modules can communicate with each other via asynchronous function calls. Every asynchronous module is an event-driven system which manages multiple coroutines which only executes this module's code. Every asynchronous module has its own scheduler and there is also a global scheduler which allocates CPU resources to these modules. Such architecture can allow us to utilize resource more flexibly. For example, we can let modules related to I/O run on cores with a low frequency since these modules are not CPU-bound. Mention that this architecture can provide more flexibility even if the OS only supports synchronous system calls, so it is not limited by asynchronous system calls to some extent.

Chapter 2

Background

2.1 Technical Background

2.1.1 Coroutines

Coroutine is a programming model of cooperative multitasking. The execution of one coroutine can never be interrupted until itself decides to yield its control over the CPU. There are 2 kinds of coroutines: stackful and stackless coroutines, with the former being similar to threads since we need to allocate a stack for each stackful coroutine. Oppositely, a group of stackless coroutines only require a single stack since they use the stack alternatively. A typical implementation of a stackless coroutine is a finite state machine which only records the execution context at yield points. Other temporary data is stored on the shared stack. We can see that the stackless coroutine reduces the total memory consumption. We also expect that the context switch overhead between stackless coroutines to be lower under efficient implementation and compiler optimizations. This is the reason why we use stackless coroutines as the task unit instead of stackful coroutines.

2.1.2 Asynchronous Programming in Rust

In Rust[7], the **Future** trait describes the behaviour of a stackless coroutine (or a finite state machine). There is only one method called **poll**, which means that the stackless coroutine tries to make some progress. If it successfully completes, the poll method returns **Poll::Ready** with the results. Otherwise, the coroutine has to wait for an external event to go ahead. In this case, a **Poll::Pending** is returned.

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}
```

Programmers can provide their own implementation of the runtime to manage these coroutines which implements the Future trait. Typically, every coroutine managed by the runtime is the root node of a Future tree, with each node on the tree implementing the Future trait. For each kind of external events, programmers design a dedicated struct and implement Future trait for it by themselves. They are the leaves in the Future tree. After that, the Rust compiler can combine these sub state machines into a large one level by level using the **async fn** and **.await** keywords until a Root-level Future is generated.

The runtime is an event-driven system which can be divided into 2 parts: **Executor** and **Reactor**. The runtime runs an event loop. In each iteration, the Reactor analyzes external events it receives and wakeup coroutines accordingly while the Execution selects a coroutine which is ready to execute, i.e, calling its **poll** method provided by the Future trait and blocking the current coroutine if **Poll::Pending** is returned.

2.1.3 zCore

zCore[10] reimplements the system calls of Zircon[11], the kernel of Google's next generation OS called Fuchsia[4] following Android, in Rust[7] with its asynchronous programming features. Because of its great modular design, it is also compatible with the Linux kernel and it can run on x86_64 and RISC-V bare-metal platforms and also Linux and MacOS host OS.

Currently it only supports synchronous system calls, but its internal implementation is asynchronous based on coroutines. Kernel-visible threads in classical OS design are substituted with coroutines and these coroutines are managed by a runtime inside zCore.

We decided to apply our design on zCore.

2.2 Academic Background

Here are some previous academic works related to this project:

2.2.1 Cohort Scheduling

Cohort scheduling[6] indicated that for a server application, handling a cohort of similar requests together by collecting and scheduling the requests can significantly reduce the cost per request in terms of cycles as the cohort size increases. This is because similar request tend to access the same group of data structures so that it can contribute to the data locality. We also take advantage of this observation to design our own architecture, but the difference is that we try to apply this idea in OS design based on asynchronous system calls and the huge impact of processor state pollution is considered.

The staged computation model[6] is similar to our asynchronous module architecture. However, the synchronization control per module is more difficult in kernel since it involves multitasking. Derived from the staged computation model, the Staged Event-Driven Architecture (SEDA) [9] use controllers and filters to handle excessive workloads gracefully, which we can also utilize to provide more flexibility and allow the OS to work on more platforms.

2.2.2 System Call Aggregation

In order to mitigate the impact of processor state pollution, another idea in addition to asynchronous system calls is to submit as much requests as possible on a single trap into the kernel, which is called system call aggregation. For example, the Effective System Call Aggregation (ESCA) [3] allows applications to submit a batch of system calls before trapping into the kernel. Anycall [5] allows applications to submit a piece of bytecode to the kernel so that the in-kernel compiler can translate the bytecode into native instructions or kernel operations. Although these methods can reduce the times of processor mode switches, they are difficult to use and not transparent to applications.

2.2.3 Minimizing Granularity of Locks in Kernel

To eliminate the unnecessary synchronization overhead in the kernel, Corey OS[2] allows threads from the same process to explicitly indicate which resources it actually needs to access via modified system calls. This can reduce the in-kernel locks' granularity to a appropriate level so that the synchronization overhead can be mitigated. This idea is orthogonal to our project.

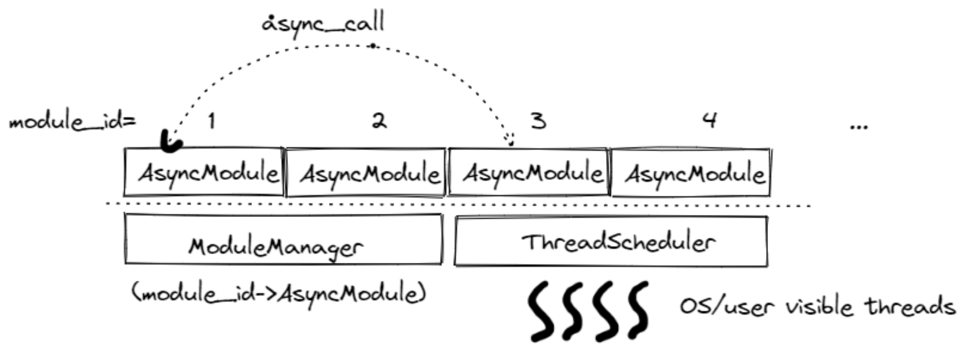
Chapter 3

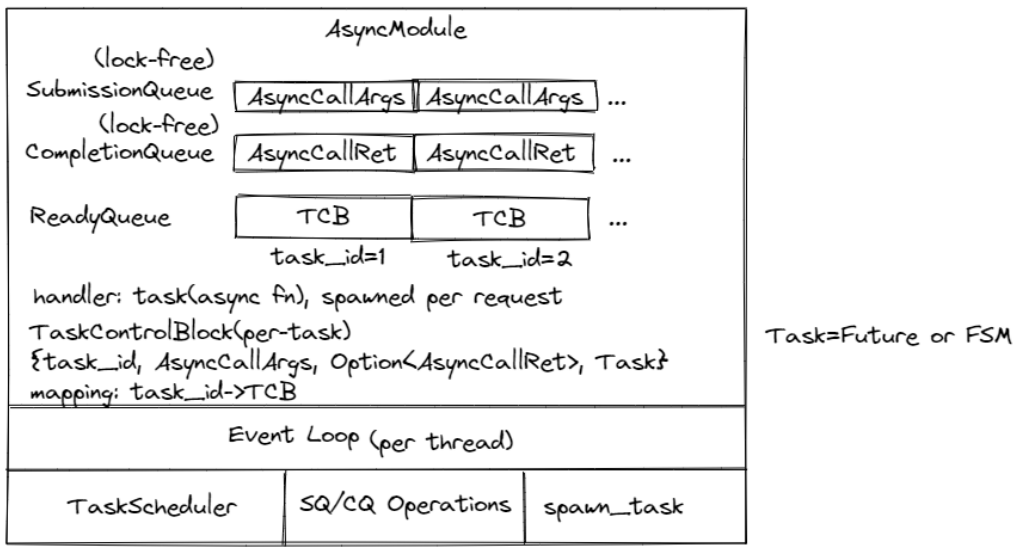
Progress

We have completed a proof-of-concept prototype with respect to the main mechanism of our asynchronous module design, i.e., the asynchronous call mechanism, in Rust. The prototype can be found in [1]. To run the demo, just install Rust[7] and **cargo run** under the root directory.

We create 2 asynchronous modules, which run asynchronous functions called **handler1** and **handler2** defined in **src/main.rs**, respectively. The **handler1** issues an asynchronous request to **handler2**.

Here are some details about the design:





handler: task(async fn), spawned per request
TaskControlBlock(per-task)
{task_id, AsyncCallArgs, Option<AsyncCallRet>, Task}
mapping: task_id->TCB

```
event_loop {
    foreach SQE: spawn a task, add to ready queue
    foreach CQE: wakeup a blocked task, update TCB.AsyncCallRet, add to ready queue
    TaskScheduler selects a(or a bunch of) task
    match task.run()
        Ready=> write to caller's CQ, deallocate this task
        Pending=> do nothing
}

async_call {
    write to callee's SQ
    block current task
    callee writes to caller's CQ on completion
    task is woken up in caller's event_loop.foreach CQE
    task is selected by the scheduler
    task runs again, and this time it does not block
}
```

```
AsyncCallArgs{
    caller_module_id, //find CQ on completion
    caller_task_id, //find task to wakeup on completion
    callee_module_id, //find SQ on request
    data: [usize; 5],
}
AsyncCallReturnValue{
    caller_task_id, //find task to wakeup on completion
    status,
}
```

Chapter 4

Future Plan

- $01/06 \sim 15/06$: Support asynchronous system calls in zCore.
- $16/06 \sim 01/07$: Coarse-grained separation based on system calls.
- $02/07 \sim 15/07$: Fine-grained separation based on asynchronous modules.
- $16/07 \sim 01/08$: Exploring advanced scheduling policies.
- $02/08 \sim 31/08$: Evaluation writing the final report.

References

[1] *async-modules*. https://github.com/wyfcyx/async_modules. Accessed: 2022-06-01.

[2] Silas Boyd-Wickizer et al. “Corey: An Operating System for Many Cores.” In: *OSDI*. Vol. 8. 2008, pp. 43–57.

[3] Yu-Cheng Cheng, Ching-Chun Jim Huang, and Chia-Heng Tu. “ESCA: Effective System Call Aggregation for Event-Driven Servers”. In: *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2022, pp. 18–25.

[4] *fuchsia*. https://fuchsia.dev/. Accessed: 2022-06-01.

[5] Luis Gerhorst et al. “AnyCall: Fast and Flexible System-Call Aggregation”. In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 2021, pp. 1–8.

[6] James R Larus and Michael Parkes. “Using cohort scheduling to enhance server performance”. In: *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. 2001, pp. 182–187.

[7] *Rust*. https://www.rust-lang.org/. Accessed: 2022-06-01.

[8] Livio Soares and Michael Stumm. “{FlexSC}: Flexible System Call Scheduling with {Exception-Less} System Calls”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[9] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An architecture for well-conditioned, scalable internet services”. In: *ACM SIGOPS operating systems review* 35.5 (2001), pp. 230–243.

[10] *zCore*. https://github.com/rcore-os/zCore. Accessed: 2022-06-01.

[11] *zircon*. https://fuchsia.dev/fuchsia-src/concepts/kernel. Accessed: 2022-06-01.