A Design and Implementation of Rust Coroutine with priority in Operating System

Wenzhi Wang
Department of Information Engineering, Capital Normal University
Beijing, China
wwzcherry@gmail.com

Abstract

Non-preemptive cooperative scheduling of the coroutine is an effective tool for concurrent programming due to its low cost. Still, there is currently a lack of flexible algorithms for scheduling. Moreover, the current research and application of coroutines are mainly concentrated in user-mode applications, and the scheduling of coroutines is ultimately a behavior within a single process.

This paper proposes a priority-based coroutine model and implements it as a library which can achieve priority-based cooperative scheduling based on the Rust language. And by introducing the priority bitmap of the coroutine into the kernel, the kernel can perceive the existence of the coroutines in the user mode through the priorities of the coroutines, thereby intervening in the coroutine scheduling of the whole system.

Further, we compare the performance difference between the OS that introduces the coroutine library to use the coroutine and the OS to use the thread through experiments. The results show that with sufficient concurrency, the performance of the coroutine is significantly better than the thread.

Keywords: Coroutines, Asynchronous Programming, Operating System, Scheduler, Rust

1 Introduction

In modern programming, coroutines are primarily designed and used as an abstraction for non-blocking asynchronous programming. Asynchronous procedures [12] are procedures whose progression may involve waiting for external events to transpire but allow for other components of the program to execute while waiting. And the cooperative switching between coroutines will not cause the switching of the page table (process) nor the stack (thread) and does not require the participation of the kernel. The stack-switching overhead of multithreading technology will become the system's performance bottleneck due to the continuous increase of concurrency. At this time, coroutines will be an effective solution [1] .

1.1 Coroutines and Related Work

The earliest description of coroutines was proposed by Melvin Conway [3] in 1958, and it is a popular abstraction in computer science that has developed over the years. In the most general sense, coroutines [8, 12] are a generalization of subroutines, equipped with the ability to suspend their own execution, and be resumed by another part of the program later, at which point the coroutine continues execution at the point it previously suspended itself, up until the next suspension or the termination of the coroutine.

The **stacked coroutine** [13] is similar to the thread which holds the call information on the stack. As a coroutine is created, stack memory is allocated to hold the context of the coroutine, and if a child coroutine is created, it continues to be pushed to the stack. The feature of the stacked coroutine is that users do not need to worry about the scheduling. The disadvantage is that it is difficult to allocate a reasonable stack size. The state-of-art languages that use stacked coroutines are Lua [4] and Golang [10].

Stackless coroutines are used like functions. which store coroutine local variable in heap memory. Since there is no stack, subroutines can be woken up anywhere in the program. The stackless coroutine is characterized by the low overhead of coroutine switching. The disadvantage is that coroutine scheduling is unbalanced and users need to intervene in the scheduling. Popular languages such as C++ [2], Rust [14], Kolint [5] and Python [11] use stackless coroutines. The coroutine model in this paper is also stackless.

The above large-scale practical applications are based on previous work, and research in recent years is constantly exploring more possibilities for coroutines. Corobase [6] adopts the coroutine-to-transaction model to hide data stalls during data fetching. Zhu et al. [15] introduce the coroutine-to-SQL model to reduce the waiting cost of interactive transactions. FaSST [7] and Grappa [9] use the C++ coroutine package to hide the RDMA network latency.

But without exception, none of the above coroutines support priority scheduling. When faced with multiple ready coroutines, there is no way to distinguish their execution order by urgency. To make up for this shortcoming is one of the tasks of this paper.

1.2 Rust and rCore OS

For the above programming languages, we need to choose one to implement the coroutine model. For performance reasons, we first exclude golang and kotlin due to their garbage collectors, and python. We choose the latter between C++ and Rust due to memory safety concerns, as its ownership model largely eliminates the need for manual reference counting. Based on the above advantages, Rust is very suitable

for the development of low-level software, such as the cryptocurrency Libra developed by Facebook (Meta), and the new generation of cross-platform operating system Fuchsia developed by Google.

In addition, the *async/await* programming model provided by Rust language has strong support for asynchrony, and this feature can be used to create asynchronous functions easily. During its execution, it will exit many times in the form of function return, and then continue to execute after being woken up. We can simply process the asynchronous function to obtain the data structure used to describe the coroutine, that is, the coroutine control block.

rCore OS is a Unix-like operating system developed by Tsinghua University and runs on the RISC-V platform. Many designs of its kernel make full use of the features of the Rust language to facilitate the extension of asynchronous mechanisms. It supports processes and threads but does not support coroutines. This article applies the implemented coroutine library to rCore OS and designs some experiments to compare the performance differences between threads and coroutines in rCore OS.

1.3 Contributions and Paper Organization

The main work of this paper has two points:

1. This paper proposes a coroutine model based on Rust language and implements it as a function library in Rust language. It is worth noting that this function library is completely implemented based on the core library of Rust, so it can be introduced and used by the kernel so that the kernel can also create and run coroutines. The standard library of Rust needs the support of the operating system, so it cannot be applied to the kernel. This is an essential difference from some existing coroutine libraries (such as tokio) that can only run in user mode.

2. This paper introduces priority for coroutines and implements priority scheduling of coroutines concerning Linux's O(1) scheduling algorithm. Based on the coroutine priority bitmap of each process, the coroutines in the process are scheduled and executed according to the priority; based on the global priority bitmap of the kernel, after each clock interruption, the kernel will schedule the process with the highest priority coroutine, the process will later schedule and run its own highest priority coroutine, which is also the highest priority coroutine in the entire system. This mechanism makes coroutine scheduling no longer an independent behavior within each process, but the kernel can uniformly control the coroutines of all processes through priority.

This paper will present the design and implementation details of the coroutine model in the next second section, give a performance comparison experiment in the third section and make a conclusion in the fourth section.

2 Design and Implementation

This section describes the overall framework and design details of this coroutine model. We first introduce the model and its fundamental operating mechanism, then introduce the usage and functions of the interface provided by this model, then we will introduce the design and implementation of the priority scheduling of the coroutine, and finally, we will introduce the design of the wake up mechanism of the coroutine.

2.1 Overview

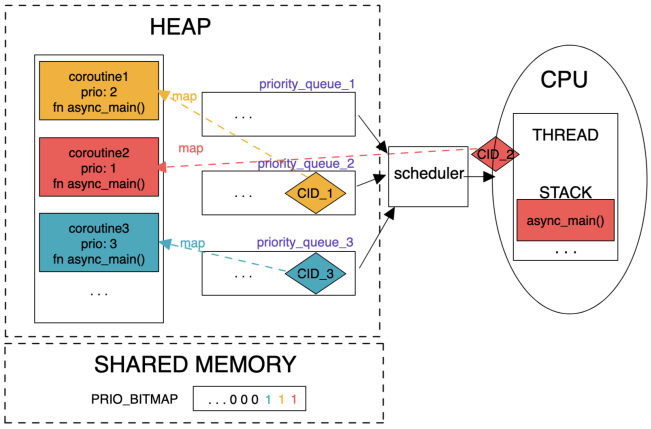

Figure 1: The architecture of the coroutine model in this paper

As shown in *Figure 1*, the Rust language provides asynchronous functions, and the coroutine library provides priority and coroutine ID fields and encapsulates them into a data structure called a coroutine control block. Each coroutine control block represents a coroutine. And a coroutine can be regarded as a scheduling object corresponding to a global asynchronous function one-to-one. We use the coroutine ID as an index to store all the coroutines in a hash table on the process's heap memory. We can get or delete its corresponding coroutine control block through the coroutine ID.Considering that the coroutine will enter and exit the scheduler many times during its life cycle due to waiting for external events, compared to the scheduler directly managing the coroutine control block, it can reduce the memory overhead when the scheduler works.

The coroutine scheduler provides two necessary operations: push and pop, which are used to insert and delete coroutines, and implement the coroutine scheduling algorithm. For priority scheduling, we will set up multiple queues, each with a different priority, and store the coroutine ID of the coroutine at the corresponding priority in it. Query these queues from high to low priority to complete priority scheduling.

We use the priority bitmap to describe the priority layout of the coroutines within each process. We also use a bitmap in the kernel to describe the priority layout of the coroutines of all processes. The kernel can perceive the existence of coroutines and their priority through these bitmaps, and formulate the scheduling strategy of the process, thereby indirectly intervening in the coroutine scheduling in the user mode. This makes coroutine scheduling no longer an independent behavior within each process but allows them to be considered uniformly from a global perspective. Its design and implementation will be introduced in Section 2.3. The above is the coroutine structure within the process.

The execution of functions must depend on the stack, the same is valid for asynchronous functions. For this, we have the following design: When a coroutine needs to be executed, we

start the coroutine executor, which is essentially a function. The coroutine executor will take out all the coroutines in the ready state and run them in a loop. The coroutine will use this thread's stack to execute its asynchronous function. When the coroutine is completed or suspended due to waiting for external events, it will exit the occupied thread stack by returning the function. At this time, the thread running the coroutine scheduler can take out the next coroutine from the scheduler to run.

The coroutine needs to be woken up after it yields because it waits for external events. Wake-up is a key mechanism of the coroutine. This part is described in detail in Section 2.4.

2.2 Interface

The coroutine model will be presented as a function library, which only provides an interface-*coroutine_run*, it will be used to create coroutines and start the coroutine executor.

```
1 // Algorithm 1
2 fn coroutine_run(async_main, prio = DEFAULT){
3     create_coroutine(async_main, prio);
4     if get_coroutine_size() <= 1:
5         run();
6 }
```

We use the *async* keyword provided by the Rust language to create an asynchronous function, where the asynchronous code statement needs to be decorated with the *await* keyword, and this asynchronous function will run as the main function of the coroutine. When no coroutine priority is specified it will be given the default value. If we need to execute a batch of coroutines, this interface should be nested in an asynchronous function to create coroutines in batches. When the first coroutine is created, the interface needs to call the run function to start the coroutine executor; when the number of coroutines is greater than 1, it means that the executor has been started, and this interface only needs to complete the creation of the coroutine.

2.3 Priority and Schedule

We handle the scheduling order of coroutines based on priority. Within a process, coroutines are executed from high to low priority; between processes, the process with the highest priority coroutine will be scheduled first.

2.3.1 Coroutine scheduling within a process

Each coroutine has a priority. After the coroutine is created, it will be inserted into the corresponding priority queue and wait to be scheduled for execution. The scheduler scans the queues according to the priority from high to low, and searches for the first non-empty queue to complete the priority scheduling. Only the coroutines in the ready state are saved in the queues, and the coroutines in the waiting state will be reinserted into the queue after being awakened, which ensures the query efficiency of the scheduler.

2.3.2 Coroutine Scheduling of the Whole System

Based on the previous design, the priority scheduling of coroutines can be implemented within the process. Now, we extend it to the entire system to perform priority scheduling on the coroutines of all processes.

2.3.2.1 Priority Bitmap

We use bitmaps to describe the priority layout of coroutines.

Each process will maintain a bitmap whose length is the number of priorities of the coroutines, and the value of each bit is 0 or 1, indicating whether there is a coroutine in the ready state of that priority.

The kernel of a modern operating system basically has an independent address space and page table, so the kernel also has a priority bitmap to represent the priority information of the coroutines it manages, called the kernel bitmap.

In addition, we set an extra bitmap in the kernel to represent the priority information of the coroutines of the whole system, which is called the system bitmap. But the 0 or 1 of each bit represents whether there are coroutines of this priority in all processes and the kernel of the entire system.

According to the definition of the system bitmap, we can know that its value is the result of the OR operations of all processes and kernel bitmaps-as long as at least one coroutine of at least one process belongs to this priority, the corresponding bit in the system bitmap is 1; and only when all processes do not have coroutines of this priority, the corresponding bit in the system bitmap will be 0. As shown in Figure, it will be calculated when the clock interrupt arrives.

Now we explain how the kernel gets all the bitmaps. Our approach is to specify a virtual address as the address of the process accessing the bitmap in user mode. Then, when the process is initialized, apply to the kernel for a free physical page, map the specified virtual address to the starting address of the physical page, and grant the process permission to read and write this address. In this way, the process can read and write the bitmap in user mode, and the kernel can also directly access the bitmap of the process. We map each process ID with the physical address of its process bitmap, and we can get all process bitmaps in the kernel by enumerating the IDs of the processes.

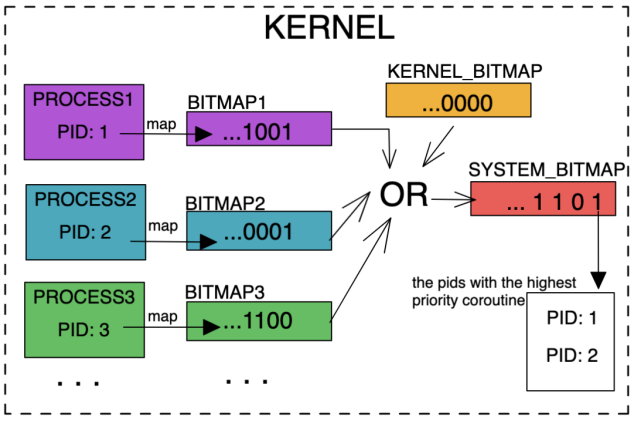

Figure 2: The structure of priority bitmaps in the kernel

2.3.2.2 Global Scheduling Based on Priority Bitmap

Based on the above bitmaps, we can guarantee the global priority scheduling when the clock interrupt arrives: the system bitmap will be updated when the clock interrupt arrives,

and then the process or kernel with the highest priority coroutine can be scheduled to run according to the result, and the highest priority coroutine will be scheduled to run in this address space, We call this **relaxed global priority scheduling**.

Within one clock cycle, the changes in the priority layout of coroutines caused by the creation, wake-up and deletion of coroutines are synchronized to the system bitmap in time, so that the process with the highest priority coroutine is always prioritized scheduling, we call this **strict global priority scheduling**. Our specific measures are: the bitmap of the process can always be updated in real time by the execution of the coroutines, so before each execution of the coroutine, compare the highest priority recorded in the process bitmap and the system bitmap, If and only if there is a coroutine with a higher priority recorded in the system bitmap, it will enter the kernel through a system call to update the system bitmap, and then reschedule according to the result.

Obviously, adopting strict global priority scheduling will ensure that the highest priority coroutine in the system is always executed first, but it will also introduce a lot of overhead by frequently entering the kernel within a clock cycle, and relaxed scheduling is the opposite. Fortunately, which scheduling strategy to adopt is the behavior of the coroutine executor itself, and we can control which strategy it chooses through a parameter according to needs.

2.4 Coroutine Manager and Asynchronization

The Coroutine Manager is responsible for managing the coroutine control blocks, the scheduler, and the coroutine wakeup mechanism introduced in this section.

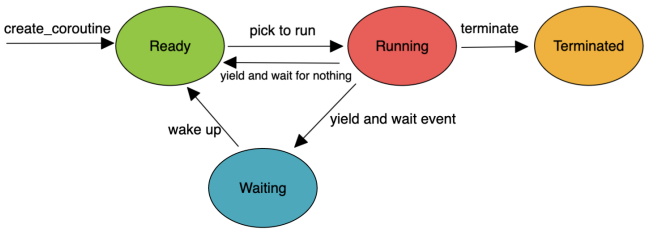

Figure 3: The state transitions of a coroutine during its lifetime

Only the IDs of the coroutines in the ready state are saved in the priority queue of the scheduler, so the wake-up operation can be completed by inserting the IDs of the coroutines awakened from the waiting state into the queue, and it only needs the ID and priority of the coroutine, and the address of the queue. We can encapsulate these three variables into a data structure called TaskWaker, and establish a mapping between the coroutine ID and the corresponding TaskWaker. So when a coroutine needs to be woken up, only its coroutine ID is needed. We will check whether its TaskWaker is created before each coroutine is executed, and delete it when the coroutine is deleted. Obviously, TaskWaker should be managed by the coroutine manager.

Now the coroutine executing the external event can use TaskWaker to wake up the suspended coroutine. But the key question is how it gets the ID of the coroutine it is going to wake up. We introduce a data structure called WakerMap here, which maintains the mapping of integer key values to coroutine IDs. For the waiting coroutine and the coroutine that performs the wake-up operation, they need to pass in the same key value when they are created (this requires the use of a global counter with mutually exclusive access within the process) to establish a connection. The waiting coroutine will insert the key and ID into WakerMap, and the wake-up coroutine can access TaskWaker and WakerMap through the same key to wake it up. If the wake-up coroutine is executed first, a null value will be read when accessing the WakerMap and the wake-up operation will be omitted, and then the waiting coroutine will not be suspended because the waiting event has already arrived (for example, the buffer has written enough data).

3 PERFORMANCE COMPARISON

We introduce this coroutine library on rCore OS to compare the performance between coroutines and threads. Each experiment creates multiple processes to run, and each process runs coroutines or threads, which will perform some of the same work. Two times are collected at the beginning and end of the process, and the difference between them is used as the running time of each test. We will compare multiple runtimes under multiple concurrency levels, and summarize the performance characteristics of coroutines and threads from them.

3.1 Experiment A: Test in User Mode without the Syscall

In this experiment, we test the performance of coroutines and threads in user mode, and will not actively call system calls to enter the kernel. In order to reflect the asynchronous characteristics of coroutines and the context switching of threads, coroutines or threads will mutually exclusive read and write a global variable.Specifically, multiple coroutines or threads will be created in the process, and they will be numbered sequentially starting from 1, and the value of the global variable will be initialized to 0. When all coroutines or threads are created, the main thread of the process will modify the value of the global variable to 1. Each coroutine or thread will read the global variable, and if the value is equal to their own number, they will add 1 to the value of the global variable. Otherwise, the coroutine will be suspended through the asynchronous mechanism, and the thread will be switched through the yield system call.

As can be seen from [Figure 5], when the concurrency is small, the total execution time of coroutines is greater than that of threads. The total execution time of coroutines increases steadily as the number of concurrency increases, while the total execution time of threads increases rapidly and exceeds that of coroutines. It can be seen from [Table 1] that at the beginning of this experiment, that is, when the concurrency is 200, the performance of coroutines is weaker than that of threads; When the concurrency is greater than or equal to 800, the performance of the coroutines begins to exceed that of the threads; when the maximum concurrency of the test is 4000, the total execution time of the coroutine is about 26% of the

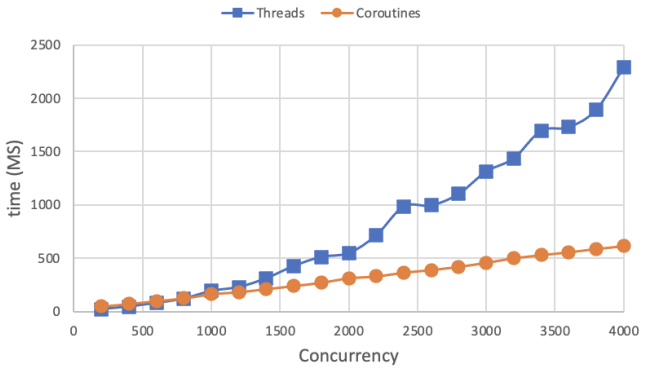

Figure 4: Performance comparison of coroutines and threads in user mode

Concurrency	Threads (MS)	Coroutines (MS)
200	22	49
800	122	124
1000	193	164
4000	2290	616

Figure 5: Test data of key nodes, including starting point, ending point, and the turning point where the coroutine performance exceeds the thread

thread.

3.2 Experiment B: Test in user mode with syscall

This experiment is carried out in user mode, and each coroutine or thread will use read and write system calls, so this experiment involves the kernel. This experiment is carried out in user mode, and each coroutine or thread will use read and write system calls to enter the kernel.

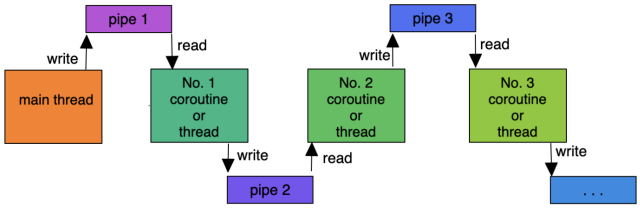

Figure 6: experiment B, the read and write ends of the pipe are located in two coroutines or threads, respectively.

The experiments we designed are shown in *Figure 6*. The experimental steps are as follows:

1. This experiment is created as processes, and each process creates N coroutines or threads to do the same work. These N coroutines or threads are numbered from 1 to N, and the value of the number is passed in as a parameter. Get a start time when the process starts, get an end time before the process exits, and use the time difference between the two as their total execution time;

2. We create N+1 pipes, numbered from 1 to N+1, pipes can be read and written by different coroutines or threads. The working method of the used pipe is: only when the write end is closed, the read operation of this pipe can read the data and then complete the read operation; when the write end is not closed, the read operation will cause thread switching in the kernel;

3. Each coroutine or thread reads and writes to the pipes. Specifically, the coroutine or thread numbered i (1¡= i ¡= N) will read data from the pipe numbered i. After its write end is closed, the coroutine or thread can successfully read the data, then write data to the pipe numbered i + 1 and close the write end of the pipe numbered i + 1. Then the work of this coroutine or thread is completed, and the coroutine or thread numbered i + 1 can start the read operation. It is equivalent to using pipes to connect all coroutines or threads in series, and the read and write operations to the same pipe are located in two adjacent coroutines or threads.

4. After all coroutines or threads are created, the main thread of the test program will directly write data to pipe 1 and then close its write end. After that, with the scheduling and execution of coroutines or threads, the data will be read from the pipe numbered 1 by the coroutine or thread numbered 1, and then written to pipe 2, and the coroutine or thread numbered 2 will read the data from pipe 2, and then write to pipe 3, and so on;

5. Pipes read and write the same amount of data per test. We tested coroutines and threads on three data sizes of 1B, 256B, and 4096B, respectively.

6. In the experiment, the concurrency of coroutines and threads is increased from 200 to 4000, and the step size is 200;

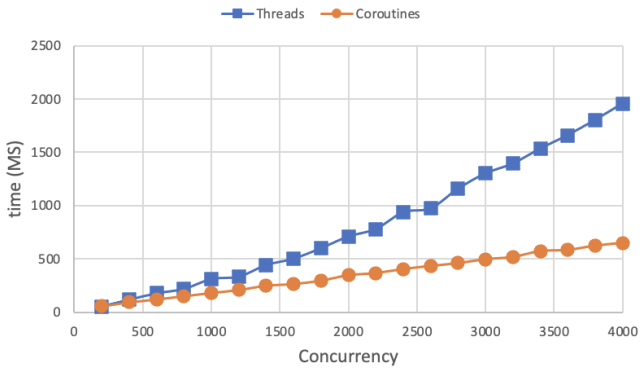

Figure 7: The amount of data read and written by the pipe is 1B

It can be seen from *Figure 7, 8, 9* that when the concurrency is small, the total execution time of coroutines and threads is very close. As the amount of concurrency increases, the total execution time of threads increases rapidly and exceeds that of coroutines. It can be seen from *Figure 10, 11, 12* that in this experiment when the concurrency is 200, the performance of the coroutine is weaker than that of the thread; when the concurrency is greater than or equal to 400, the performance of the coroutine begins to exceed that of the thread; When

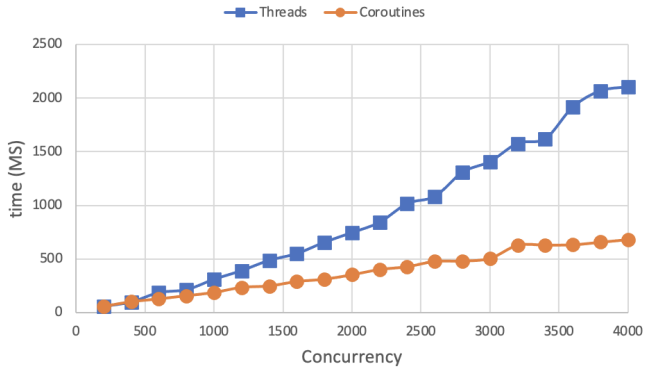

Figure 8: The amount of data read and written by the pipe is 256B

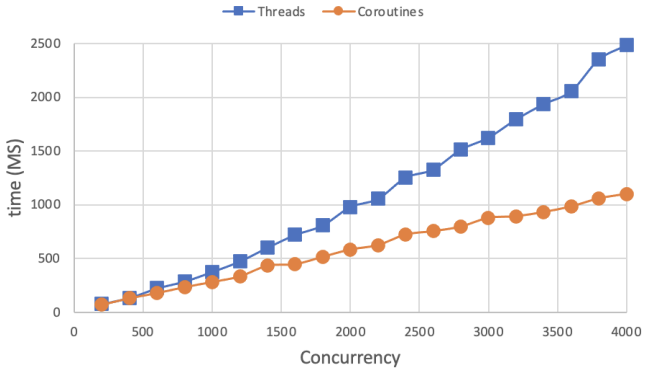

Figure 9: The amount of data read and written by the pipe is 4096B

the tested maximum concurrency of 4000 is reached, the total execution time of the coroutines is about *32% - 44%* of the threads.

Through experiments we can see that the performance of coroutines is not necessarily better than threads in all cases. When the amount of concurrency is small, the basic overhead of supporting the coroutine running, including the coroutine scheduler and the wake-up mechanism, has a significant impact, resulting in the performance of the coroutine at this time being close to that of the thread. However, as shown in the experimental data, the usage overhead of coroutines increases linearly with the increase of concurrency, which means that the increase of concurrency does not significantly impact the average execution time and average switching time of coroutines. This is a distinguishing feature of coroutines from threads and also the significance of coroutines for concurrent programming. In contrast, the context switching cost of threads increases with the increase of concurrency, which causes the performance of coroutines to gradually exceed threads when the amount of concurrency increases, and the gap between the two becomes larger and larger.

Concurrency	Threads (MS)	Coroutines (MS)
200	54	56
400	117	93
4000	1956	651

Figure 10: Test data of key nodes with 1B data volume

Concurrency	Threads (MS)	Coroutines (MS)
200	55	57
400	119	101
4000	2103	681

Figure 11: Test data of key nodes with 256B data volume

4 Conclusions

This paper proposes and implements a coroutine model based on Rust language features. A Rust operating system can use this coroutine library to create and run coroutines that support priority scheduling in user mode and the kernel. Based on the O(1) scheduling algorithm of Linux, we designed the coroutine priority bitmap of the process and the coroutine priority bitmap of the kernel, so that the scheduling of coroutines is no longer an independent behavior within each process, but can be controlled by the kernel. This allows some critical coroutines to be scheduled preferentially in the entire system, while other coroutines can be set to a lower priority, reducing competition with high-priority coroutines, so that the processor can be assigned more flexibly and scientifically.

We have introduced this coroutine library on rCore OS and conducted performance comparison experiments. Experimental results show that the performance of coroutines is not necessarily better than threads in all cases. When the amount of concurrency is small, the execution time of coroutines will be longer than that of threads. This is because the basic overhead required to run coroutines, such as scheduling, switching, and wake-up, significantly impacts the coroutines when the number of concurrency is small. However, as the amount of concurrency increases, the overhead required for coroutines to run grows slowly while threads grow rapidly, so the gap between the two continues to grow. When the concurrency is greater than 200 - 800, the execution time of the threads gradually exceeds that of the coroutines. When the concurrency reaches 4000, the total execution time of the coroutines is about *26% - 44%* of the threads.

References

[1] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(3):1–21, 2019.

[2] Bruce Belson, Wei Xiang, Jason Holdsworth, and Bronson Philippa. C++20 coroutines on microcon-

Concurrency	Threads (MS)	Coroutines (MS)
200	78	81
400	134	132
4000	2484	1099

Figure 12: Test data of key nodes with 4096B data volume

trollers—what we learned. *IEEE Embedded Systems Letters*, 13(1):9–12, 2021.

[3] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[4] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[5] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 68–84, 2021.

[6] Yongjun He, Jiacheng Lu, and Tianzheng Wang. Corobase: Coroutine-oriented main-memory database engine. *arXiv preprint arXiv:2010.15981*, 2020.

[7] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

[8] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[9] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[10] Raghu Prabhakar and Rohit Kumar. Concurrent programming with go. Technical report, Citeseer, 2011.

[11] Christian Tismer. Continuations and stackless python. In *Proceedings of the 8th international python conference*, volume 1, 2000.

[12] Love Waern. Coroutines for simics device modeling language, 2021. `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-453889`.

[13] Xinyuan Wang and Hejiao Huang. Sgpm: A coroutine framework for transaction processing. *Parallel Computing*, 114:102980, 2022.

[14] Dorian Weber and Joachim Fischer. Process-based simulation with stackless coroutines. In *Proceedings of the 12th System Analysis and Modelling Conference*, pages 84–93, 2020.

[15] Tao Zhu, Donghui Wang, Huiqi Hu, Weining Qian, Xiaoling Wang, and Aoying Zhou. Interactive transaction processing for in-memory database system. In *International Conference on Database Systems for Advanced Applications*, pages 228–246. Springer, 2018.