2025 / 08 / 28-29

Rustでつくる
オペレーティングシステム
SWEST 27

高野祐輝, TIER IV

This research is based on results obtained from a project, JPNP21027, subsidized by the New Energy and Industrial Technology Development Organization (NEDO).

Rustでつくるオペレーティングシステム

SWEST 27
髙野 祐輝, TIER IV

Self Introduction

01

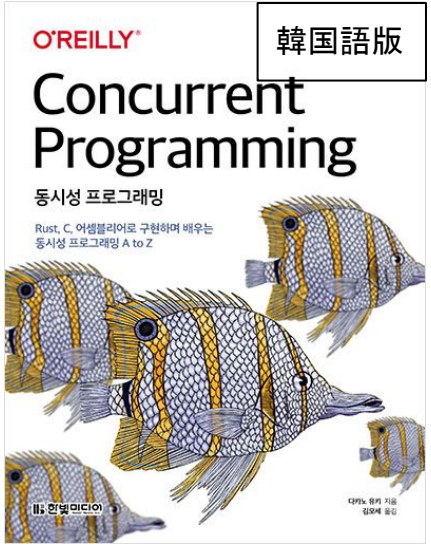
Self Introduction

Name: Yuuki Takano (高野 祐輝)

Affiliation: TIER IV, Inc.

Books:『ゼロから学ぶRust』講談社, 2022,

『並行プログラミング入門』オライリー・ジャパン, 2021



Background and Motivation

02

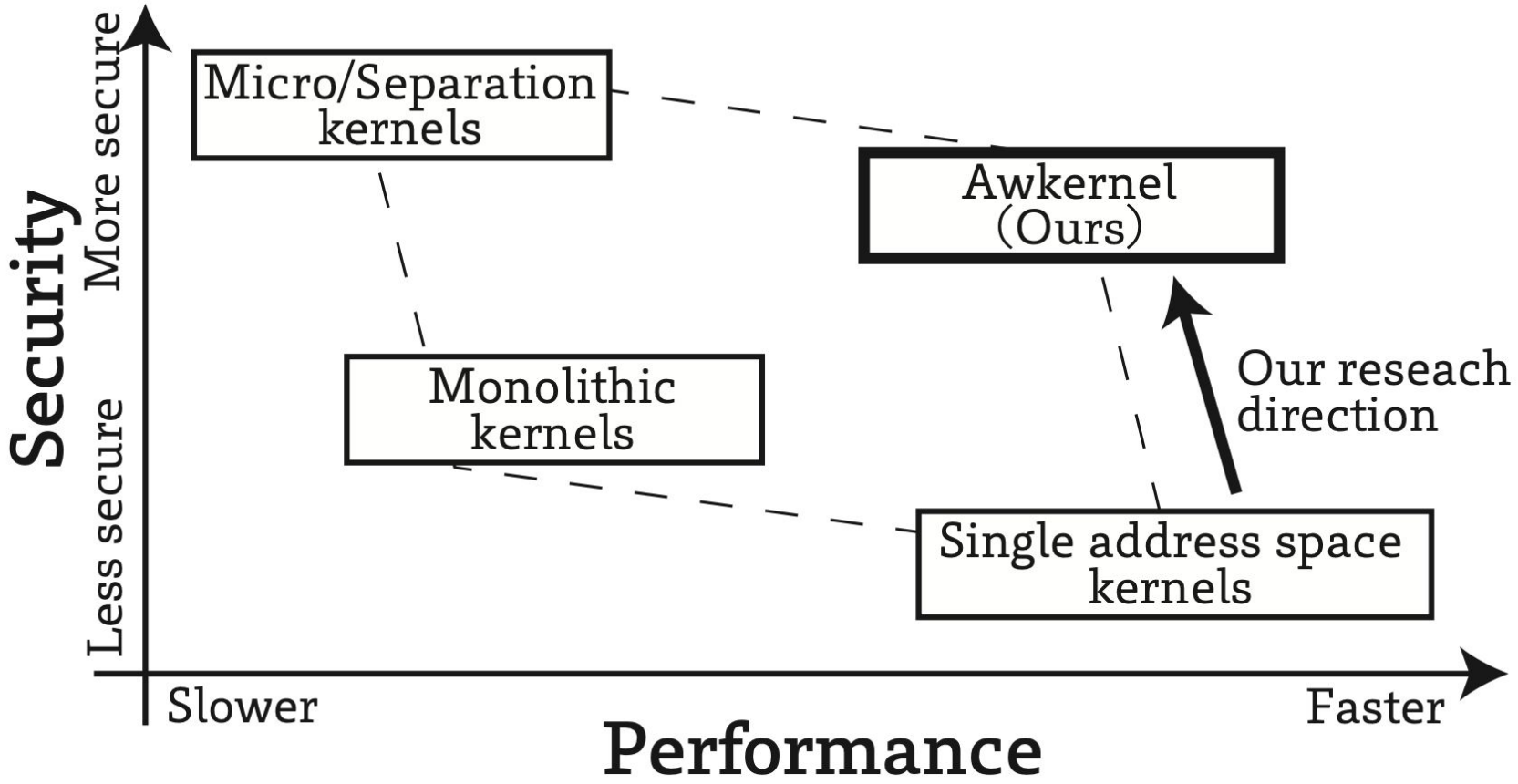
Background

1. Automobile system meets micro service
2. Microservices are facing performance penalty of inter-process communications
3. Automobile system requires security and safety
4. We propose Awkernel, a secure single-layered operating system, as a high performance microservice infrastructure for automobile system

Awkernel

1. Single address space
2. Memory space isolation by using Rust's type system
3. Lightweight Formal Methods
4. Preemptible async/await APIs
5. Microkernel style services by async/await tasks

Classification of OSes and Our Direction



Related Work

03

Operating System Written in Rust

1. RedLeaf (USENIX OSDI 2022), Single address space
2. Theseus (USENIX OSDI 2022), Single address space
3. Rust for Linux, monolithic kernel, https://rust-for-linux.com/
4. Windows Kernel, monolithic kernel
   https://www.theregister.com/2023/04/27/microsoft_windows_rust/
5. Redox OS, micro kernel, https://www.redox-os.org/

An Empirical Study of Rust-for-Linux

Latency of e1000 device driver

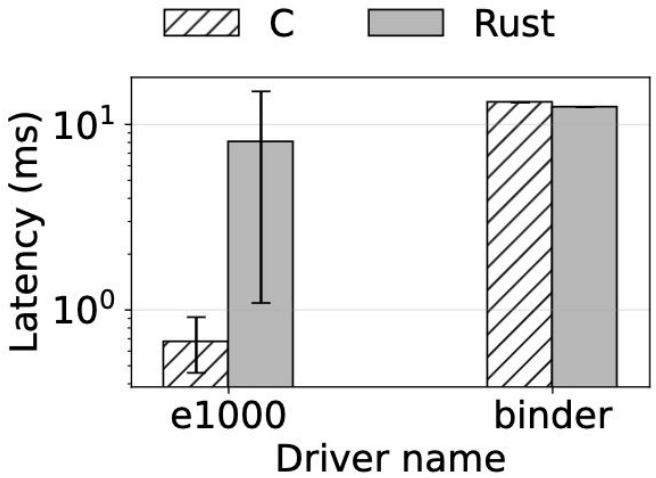


Figure 8: The latencies between Rust and C drivers. Rust `e1000` driver is significantly slower because it lacks advanced features such as `prefetch`.

Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An Empirical Study of Rust- for-Linux: The Success, Dissatisfaction, and Compromise. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 425–443, Santa Clara, CA, July 2024. USENIX Association.

An Empirical Study of Rust-for-Linux

Code Quality

Table 6: The code quality measurement. % means coverage. *RFL* achieves 100% documentation coverage and least CI errors per 10K LoC.

Subsystems	Docs%	CI errors/10K LoC
RFL	100%	3.8
ebpf	15%	7.5
io_uring	31%	11.9

Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An Empirical Study of Rust- for-Linux: The Success, Dissatisfaction, and Compromise. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 425–443, Santa Clara, CA, July 2024. USENIX Association.

Other Operating Systems

1. FlexOS, configurable, (ACM ASPLOS 2022)
2. seL4, micro kernel
3. Unikraft, single address space

FlexOS

1. Isolation domains and safety mechanisms can be flexibly configurable.
2. FlexOS can define isolation domains, and a process chooses an isolation domain when executing.
3. FlexOS can choose whether using safety mechanisms like address space layout randomization (ASLR).

Hugo Lefeuvre, Vlad-Andrei Badoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: to- wards flexible OS isolation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, edi- tors, ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022, pages 467–482. ACM, 2022.

FlexOS: Performance Evaluation

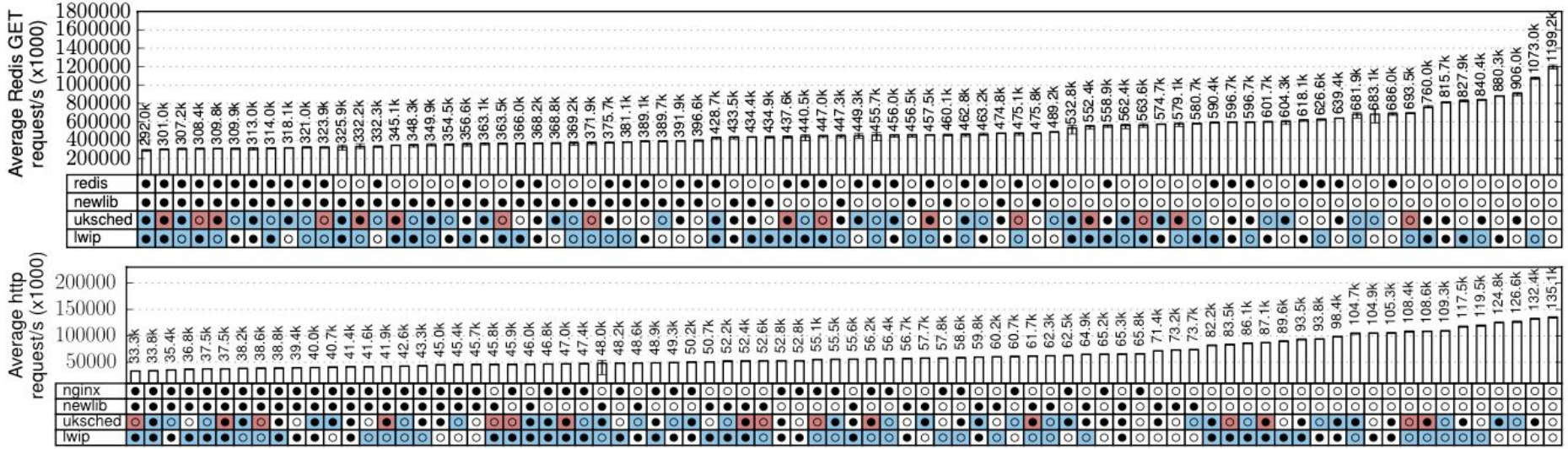


Figure 6: Redis (top) and Nginx (bottom) performance for a range of configurations. Components are on the left. Software hardening can be enabled [•] or disabled [○] for each component. The white/blue/red color indicates the compartment the component is placed into. Isolation is achieved with MPK and DSS.

← strong isolation and safety mechanisms **weak isolation and safety mechanisms →**

Hugo Lefeuvre, Vlad-Andrei Badoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: to- wards flexible OS isolation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, edi- tors, ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022, pages 467–482. ACM, 2022.

Rust

—

04

Programming Language Rust

1. Safe: type safety, memory safety
2. Fast: as fast as C/C++
3. Rich ecosystem: documentation comment, sophisticated build system
4. Suitable for real-time system: no garbage collection

Why Rust?

1. Protecting critical infrastructures

   ex: nuclear power plants
   A nuclear power plant was attacked by using a buffer overrun vulnerability
   Rust can prevent (stuxnet).

2. Mitigating security risk

3. Reduce development cost

   Debugging accounts for the majority of development time.

CVE VS. Protection Level: Definition

1. Rare and Difficult (RD): Issues which are rare and difficult to find、but the language can prevent these issues
2. Safeguarded (SG): Issues can be prevented by using the language, but additional countermeasures will be required in some cases
3. Unprotected (UP): Issues the language cannot prevent

Tiago Espinha Gasiba, Sathwik Amburi, and Andrei-Cristian Iosif, Can Secure Software be Developed in Rust? On Vulnerabilities and Secure Coding Guidelines, International Journal on Advances in Security, vol 17 no 1 & 2, year 2024

CVE VS. Protection Level: Comparison

Tiago Espinha Gasiba, Sathwik Amburi, and Andrei-Cristian Iosif, Can Secure Software be Developed in Rust? On Vulnerabilities and Secure Coding Guidelines, International Journal on Advances in Security, vol 17 no 1 & 2, year 2024

TABLE I
SANS TOP 25 CWE VS. PROTECTION LEVELS IN RUST

CWE ID	Short Description	RD	SG	UP
CWE-787	Out-of-bounds Write	●		
CWE-79	Cross-site Scripting			●
CWE-89	SQL Injection		●	
CWE-20	Improper Input Validation		●	
CWE-125	Out-of-bounds Read	●		
CWE-78	OS Command Injection		●	
CWE-416	Use After Free	●		
CWE-22	Path Traversal			●
CWE-352	Cross-Site Request Forgery			●
CWE-434	Unrestricted Dangerous File Upload			●
CWE-476	NULL Pointer Dereference	●		
CWE-502	Deserialization of Untrusted Data			●
CWE-190	Integer Overflow or Wraparound		●	
CWE-287	Improper Authentication			●
CWE-798	Use of Hard-coded Credentials			●
CWE-862	Missing Authorization			●
CWE-77	Command Injection		●	
CWE-306	Missing Critical Function Authentication			●
CWE-119	Buffer Overflow	●		
CWE-276	Incorrect Default Permissions			●
CWE-918	Server-Side Request Forgery			●
CWE-362	Race Condition	●		
CWE-400	Uncontrolled Resource Consumption		●	
CWE-611	Improper Restriction of XXE			●
CWE-94	Code Injection		●	
		24%	28%	48%

TABLE II
SANS TOP 25 CWE VS. PROTECTION LEVELS IN C, C++, AND JAVA

CWE	C			C++			Java		
	RD	SG	UP	RD	SG	UP	RD	SG	UP
CWE-787			●		●		●		
CWE-79			●			●			●
CWE-89			●		●			●	
CWE-20			●			●		●	
CWE-125			●			●	●		
CWE-78			●			●		●	
CWE-416			●		●		●		
CWE-22			●			●			●
CWE-352			●			●			●
CWE-434			●			●			●
CWE-476			●		●		●		
CWE-502			●			●			●
CWE-190			●			●			●
CWE-287			●			●			●
CWE-798			●			●			●
CWE-862			●			●			●
CWE-77			●			●		●	
CWE-306			●			●			●
CWE-119			●		●		●		
CWE-276			●			●			●
CWE-918			●			●			●
CWE-362			●			●		●	
CWE-400			●		●			●	
CWE-611			●			●			●
CWE-94			●			●		●	
	0%	0%	100%	0%	24%	76%	20%	28%	52%

Static analysis against C/C++

They prepared several bugs, and evaluated static analysis tools of C/C++.

TABLE II: CHECKING IF SCA/DCA TOOLS FIND MEMORY BUGS.

Bug	cppcheck	splint	GNU C/C++ sanitize
0	✓	✓	✓
1	✓	✓	✓
2	✓	✓	✓
3	missed	✓	missed
4	missed	✓	compile err.
5	✓	✓	missed
6	✓	✓	missed
7	missed	✓	missed
8	✓	✓	partly
9	partly	✓	partly
10	✓	missed	missed
11	missed	✓	missed
12	✓	✓	missed
13	missed	✓	partly
14	✓	incompatible	✓
15	missed	incompatible	missed

Mario Noseda, Fabian Frei, Andreas Rüst, and Simon Künzli, Rust for Secure IoT Applications: Why C Is Getting Rusty, Embedded World Conference, Nuremberg, Germany, 21-23 June 2022.

Pitfalls of Static Analyzers

1. Many false positives
2. Need much effort to apply static analyzers

C, Zig, and Rust Memory Safety Comparison

TABLE III: C, ZIG, AND RUST MEMORY SAFETY COMPARISON.

Issue	Zig (release-safe)	Rust (release)	C
Out-of-bounds R/W	Run time	Run time	None
Null dereference	Run time[1]	Run time[1]	None
Type confusion	Run time[1,2]	Run time[1]	None
Integer overflow	Run time	Run time[1]	None
Use-after-free	None[1]	Compile time	None
Double free	None[1]	Compile time	None
Invalid stack R/W	None	Compile time	None
Uninit. memory	None	Compile time	None
Data race	None	Compile time	None

1: Some restrictions apply, 2: Partial

Mario Noseda, Fabian Frei, Andreas Rüst, and Simon Künzli, Rust for Secure IoT Applications: Why C Is Getting Rusty, Embedded World Conference, Nuremberg, Germany, 21-23 June 2022.

Crypto Benchmark

TABLE IV: RELATIVE DIFFERENCE IN EXECUTION TIME FOR CRYPTOGRAPHIC ALGORITHMS WHEN SWITCHING FROM MBEDTLS (C) TO RUSTCRYPTO (RUST).

Algorithm	From C to Rust
SHA256 (16 B)	- 13 %
SHA256 (64 KiB)	- 9 %
AES128-CCM (16 B)	+ 145 %
AES128-CCM (64 KiB)	+ 73 %
AES128-GCM (16 B)	+ 101 %
AES128-GCM (64 KiB)	+ 20 %
CHACHA20-POLY1305 (16 B)	- 53 %
CHACHA20-POLY1305 (64 KiB)	- 52 %

Mario Noseda, Fabian Frei, Andreas Rüst, and Simon Künzli, Rust for Secure IoT Applications: Why C Is Getting Rusty, Embedded World Conference, Nuremberg, Germany, 21-23 June 2022.

Must Use Rust?

1. If you have a complete understanding of C/C++, then Rust is unnecessary.
2. If you work together with junior developers, Rust should be useful.

Awkernel

05

Awkernel (revisit)

1. Single address space
2. Memory space isolation by using Rust's type system
3. Lightweight Formal Methods
4. Preemptible async/await APIs
5. Microkernel style services by async/await tasks

Awkernel is open source software

https://github.com/tier4/awkernel

Search “awkernel” at GitHub.

Comparison

	HW Isolation	Type System Isolation (Partially)	Type System Isolation	Single Address Space	Formal Methods
Awkernel			✓	✓	✓
seL4	✓				✓
FlexOS	✓ (configurable)			✓ (configurable)	
Singularity		✓			
RedLeaf			✓	✓	
Theseus			✓	✓	
Unikernel				✓	

Memory Space Isolation

- Type system based isolation (Type safe programming language), Our approach
  - Pros
    - No runtime error
    - Least performance penalty
    - Single Address Space
  - Cons
    - Limited support of programming languages
- Hardware based isolation (MMU)
  - Pros
    - Support any executable binary
  - Cons
    - Runtime error
    - Performance penalty

Formal Methods

- Model Checking, Our Approach
  - Pros: Lightweight
  - Cons: Does not prove theorem
- Theorem provering
  - Pros: Provable
  - Cons: Difficult to use, Heavyweight
  - seL4 required 20 person-years for verification

Applying Formal Methods to Basic Functions

Targets	Properties	Tools
MCSLock	Mutual exclusion on weak memory model	loom
RWLock	Mutual exclusion on weak memory model	loom
Store and restore registers of preemption (AArch64 and x86_64)	Registers are properly restored	TLA+
Delta list	Timers are really invoked	Kani
Ring queue	Push/pop operations are really FIFO	Kani
Scheduler and CPU sleep	Work conservation	SPIN
Scheduler	Priority	SPIN

Test of Async/await Scheduler

1. We translate source code in Rust to Promela, and test it by using SPIN
   a. Promela: Language for specification
   b. SPIN: Model checker
2. Test properties
   a. Starvation-free
   b. Eventually all tasks will be terminated

Rust vs Promela

```rust
fn get_next(&self) -> Option<Arc<Task>> {
    let mut node = MCSNode::new();
    let mut data = self.data.lock(&mut node);

    // Pop a task from the run queue.
    let data = match data.as_mut() {
        Some(data) => data,
        None => return None,
    };

    loop {
        let task = data.queue.pop_front()?;

        // Make the state of the task Running.
        {
            let mut node = MCSNode::new();
            let mut task_info = task.info.lock(&mut
node);

            if matches!(task_info.state,
State::Terminated | State::Panicked) {
                continue;
            }

            task_info.state = State::Running;
        }

        return Some(task);
    }
}
```

```
inline get_next(tid) {
   lock(tid, lock_scheduler);

   int head;

start_get_next:

   if
       :: atomic { queue ? [head] -> queue ? head };
           lock(tid, lock_info[head]);

           if
               :: tasks[head].state == Terminated ->
                   unlock(tid, lock_info[head]);
                   goto start_get_next;
               :: else -> skip;
           fi

           tasks[head].state = Running;

           printf("Running: task = %d, state = %d\n", head,
tasks[head].state);

           unlock(tid, lock_info[head]);
           unlock(tid, lock_scheduler);

           result_next[tid] = head;
       :: else ->
           unlock(tid, lock_scheduler);
           result_next[tid] = -1;
   fi
}
```

Async/await of Awkernel

06

Kernel-level async/await

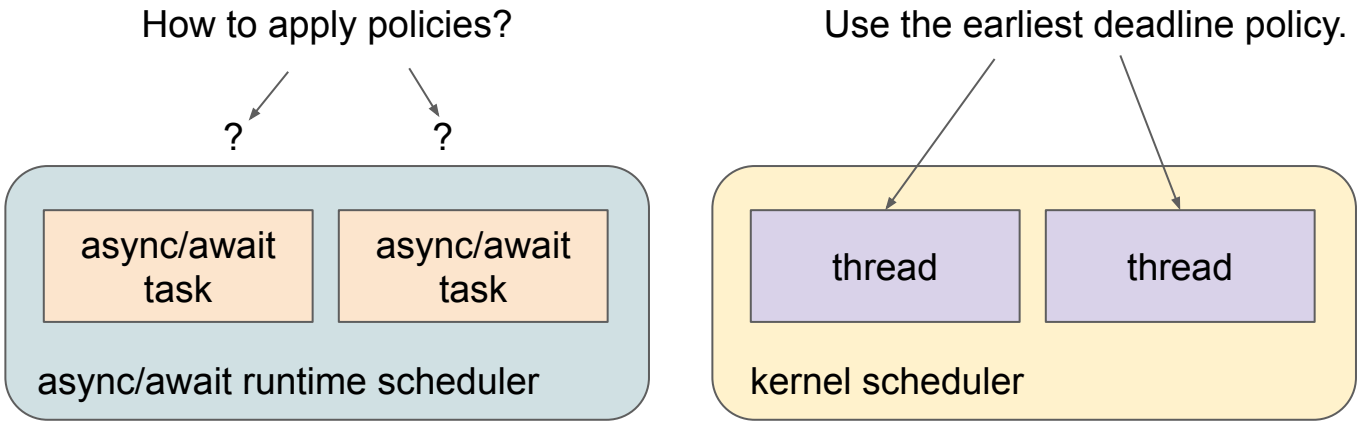
1. Awkernel provides async/await runtime in kernel.
2. Why async/await?
   a. Cyber-physical systems are inherently asynchronous.
   b. Callback based programs are suffer from the callback hell.
3. Problems of conventional async/await runtime libraries.
   a. Non-preemptible
   b. Multiple scheduling domain

Problems regarding non-preemptible task schedulers

1. Almost real-time scheduling algorithms depend on preemption.
   Example: Earliest deadline first
2. Conventional async/await mechanisms cannot take advantage of real-time scheduling algorithms.

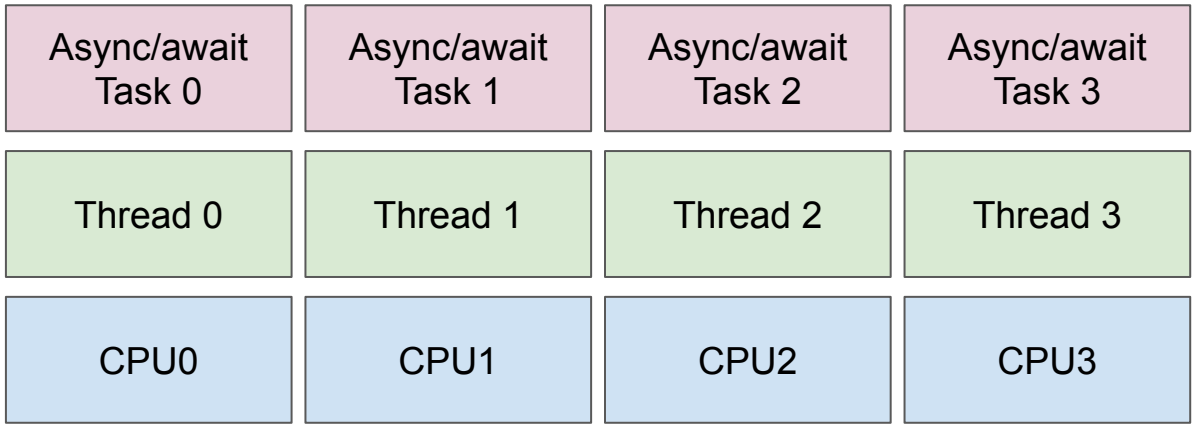
The multiple scheduling domain problem

- It is hard to apply scheduling policies to async/await tasks.

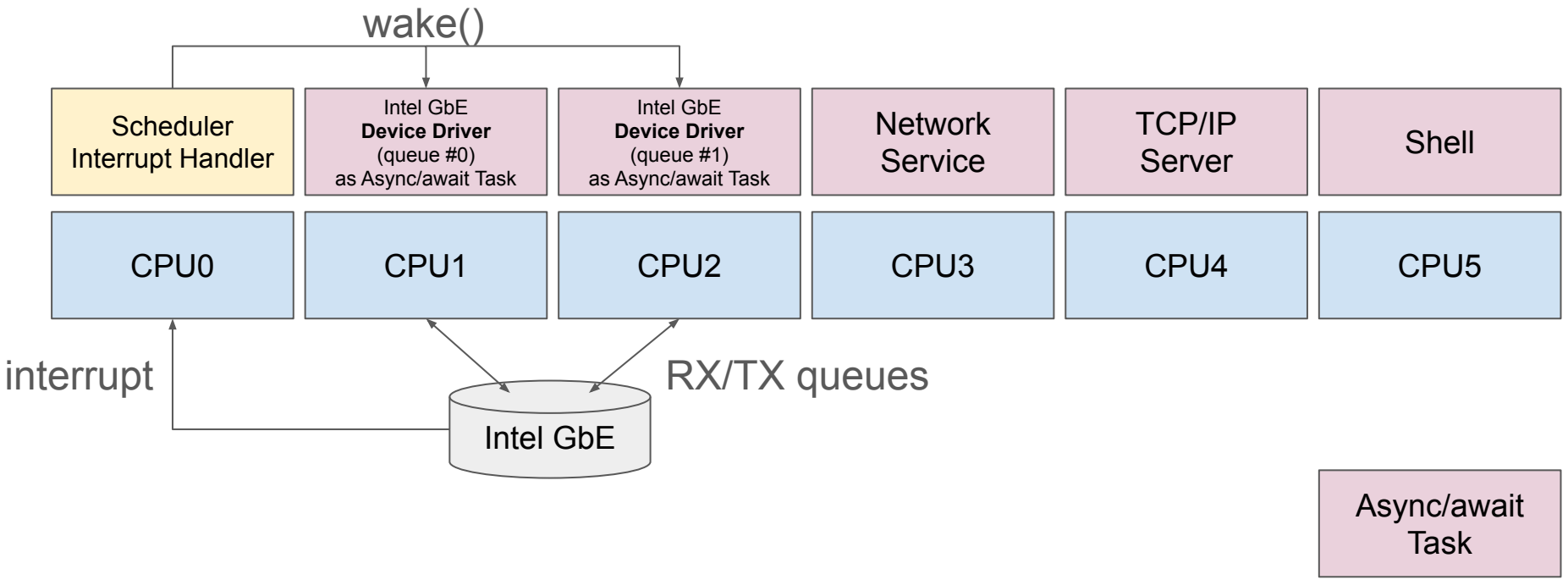


Conventional Async/await

1. Async/await tasks are on top of worker threads
2. Cooperative multitasking
3. Multiple scheduling domain



Microkernel Style Services



In-kernel Preemptible Async/await

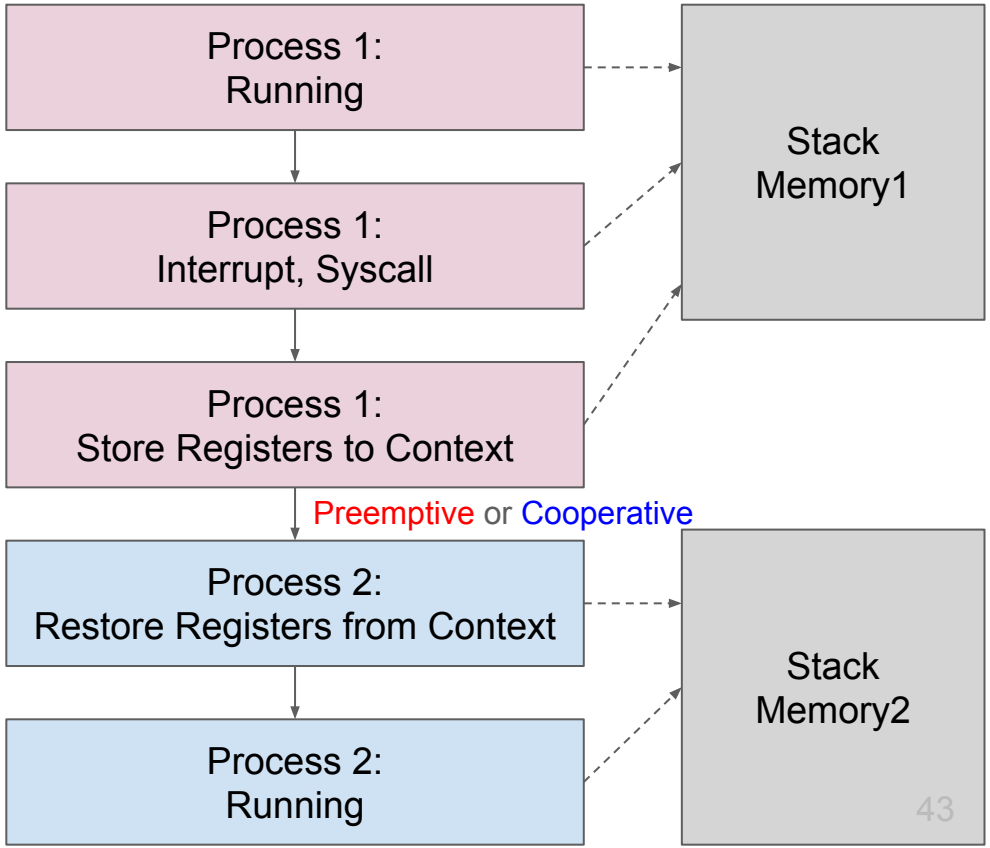
1. Preemptible
2. Single scheduling domain
3. Can specify a scheduling policy to a task

Task Spawning

```
awkernel_async_lib::spawn(
    "task name".into(),
    async { /* do something */ },
    SchedulerType::RR, // Round robin scheduler
)
.await;
```

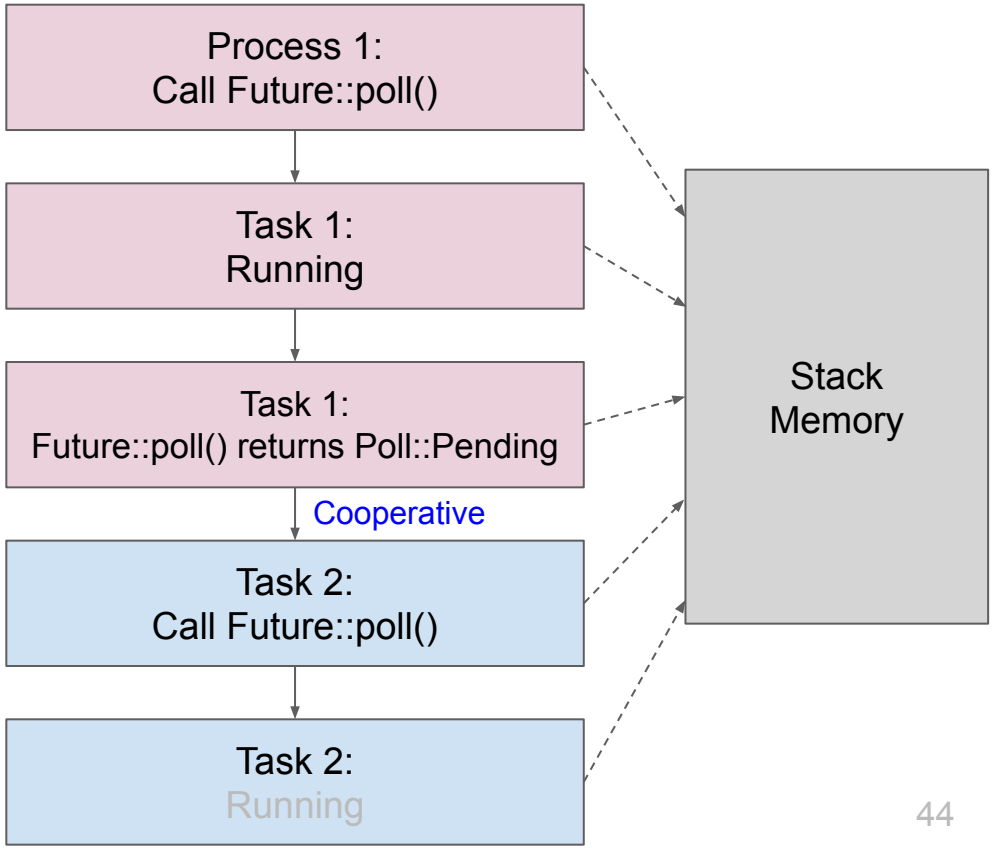
Context Switch of Conventional OSes

- Processes switch context cooperatively or preemptively.
- Store/restore registers when context switch.

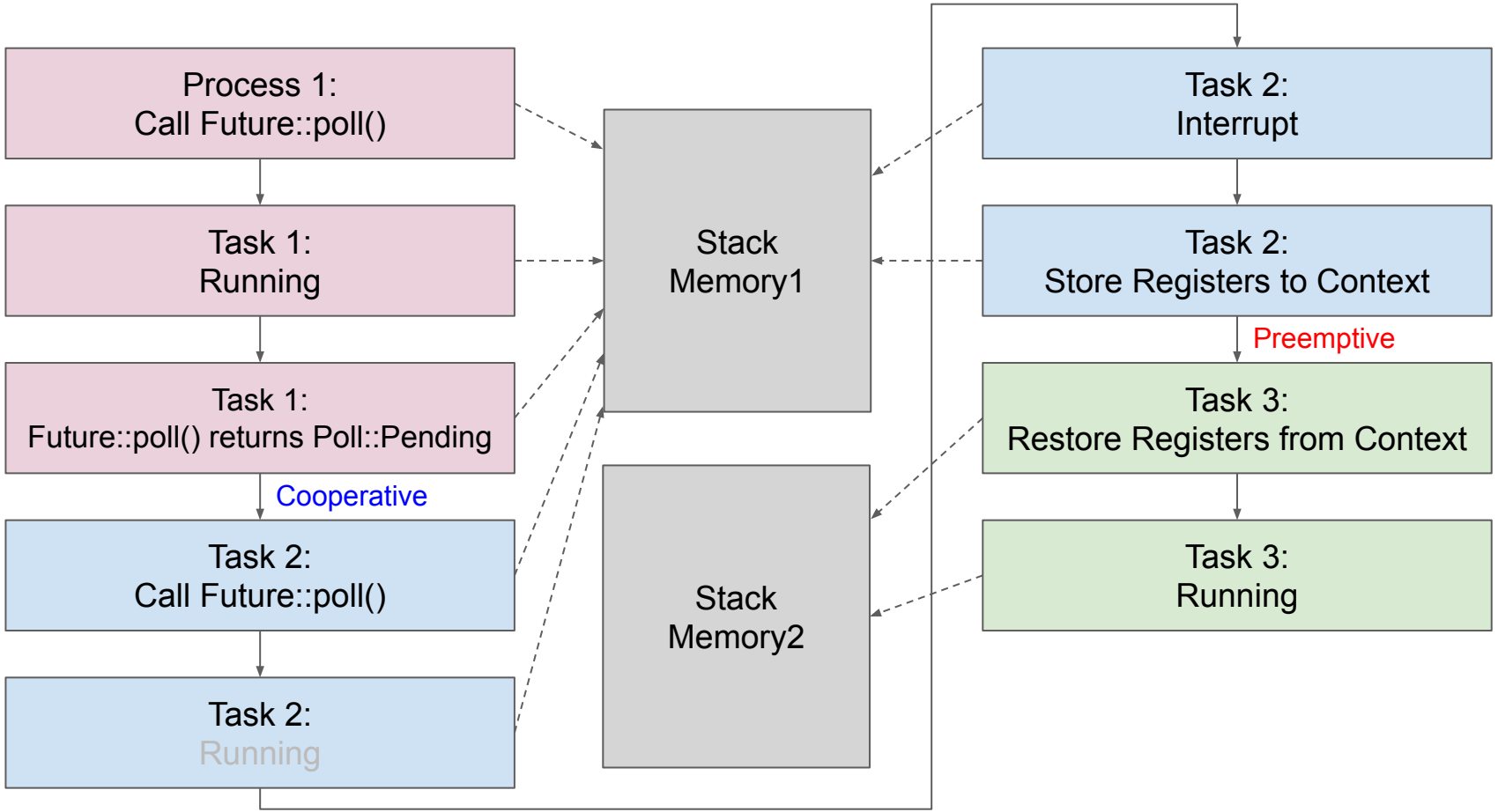


Context Switch of Async/await in Rust

- Async/await tasks switch context cooperatively.
- Do not store and restore registers.
- Use a shared stack memory.



Context Switch of Awkernel

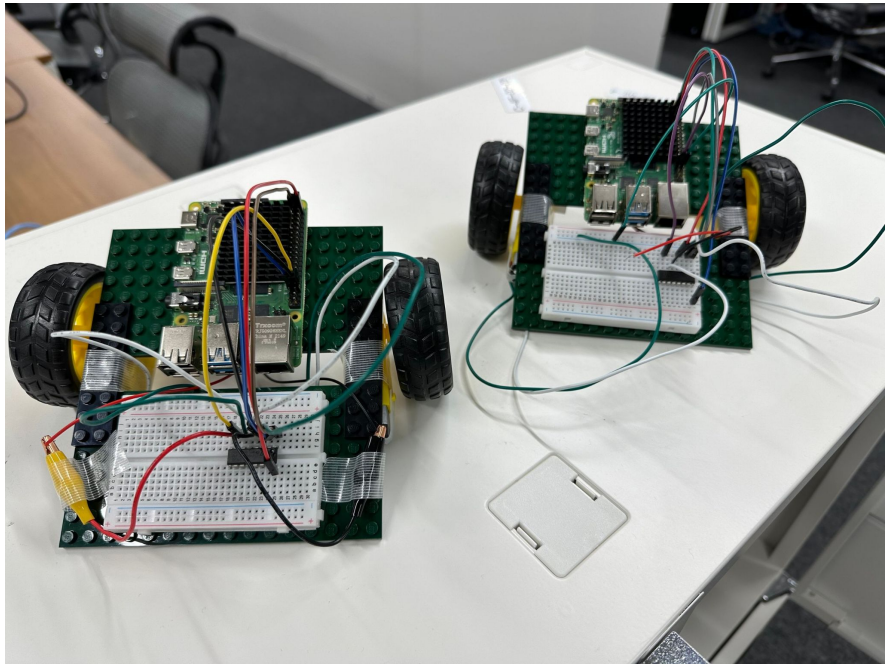


Implementation

07

Target Architectures

- x86_64
- AArch64
  - Raspberry Pi Zero 2W/3/4
  - Qemu virt



Awkernel on Raspberry Pi 4

Device Drivers

- Network Interfaces
  - Intel GbE
  - Intel 10GbE
  - Broadcom bcmnet (Raspberry Pi)
- Interrupt Controllers
  - xAPIC, x2APIC
  - bcm2835 interrupt controller
  - GICv2, GICv3 for AArch64
- Raspberry Pi's IO: GPIO, PWM, I2C, SPI

Networking

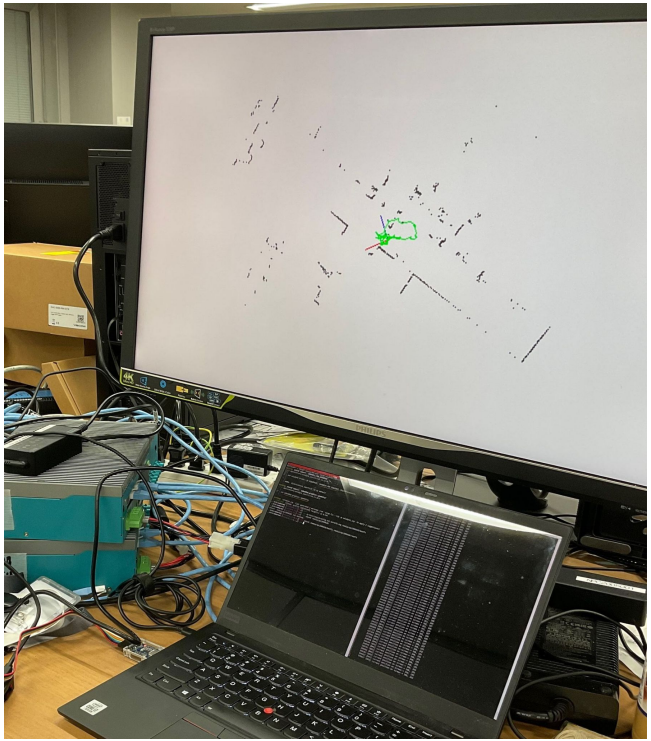
- Use external TCP/IP library
- smoltcp
  https://github.com/smoltcp-rs/smoltcp

Block Devices and File Systems

Ongoing...

Applications

- Velodyne VLP16 LiDAR driver
- Localization using Iterative closest first (ICP)



ICP on Awkernel

Demonstration

08

(ifconfig) function

```
> (ifconfig)
[0] igb-0000:00:1f.6:
    IPv4 address: 192.168.100.64/24
    IPv4 gateway: None
    MAC address: cc:96:e5:1c:87:c7
    Link status: Up (Full duplex), Link speed: 100 Mbps
    Capabilities: CSUM_TCPv4 CSUM_UDPv4 VLAN_MTU VLAN_HWTAGGING
    IRQs: [32]
    Poll mode: false

[1] igb-0000:5b:00.0:
    IPv4 address:
    IPv4 gateway: None
    MAC address: 98:b7:85:01:9a:08
    Link status: Up (Full duplex), Link speed: 1000 Mbps
    Capabilities: CSUM_IPv4 CSUM_TCPv4 CSUM_UDPv4 VLAN_MTU VLAN_HWTAGGING
CSUM_TCPv6 CSUM_UDPv6
    IRQs: [33, 34, 35, 36, 37, 38, 39, 40, 41]
    Poll mode: false
```

(task) function

```
> (task)
Uptime: 1498571515
Tasks:
   ID  State          #Preemption  Last Executed name
    2  Waiting                  0       17027421 network service
    3  Waiting                  4     1498556182 TCP garbage collector
    4  Waiting                  1     1497869930 network service:igb-0000:00:1f.6: IRQ = 32
    5  Waiting                  1     1461841869 network service:igb-0000:5b:00.0: IRQ = 33
    6  Waiting                  0       17044581 network service:igb-0000:5b:00.0: IRQ = 34
    7  Waiting                  0       16957795 network service:igb-0000:5b:00.0: IRQ = 35
    8  Waiting                  0       17062003 network service:igb-0000:5b:00.0: IRQ = 36
    9  Waiting                  0       16957853 network service:igb-0000:5b:00.0: IRQ = 37
   10  Waiting                  0       16957906 network service:igb-0000:5b:00.0: IRQ = 38
   11  Waiting                  0       16957959 network service:igb-0000:5b:00.0: IRQ = 39
   12  Waiting                  0       16958012 network service:igb-0000:5b:00.0: IRQ = 40
   13  Waiting                  0       21046736 network service:igb-0000:5b:00.0: IRQ = 41
```

Conclusion and Future Work

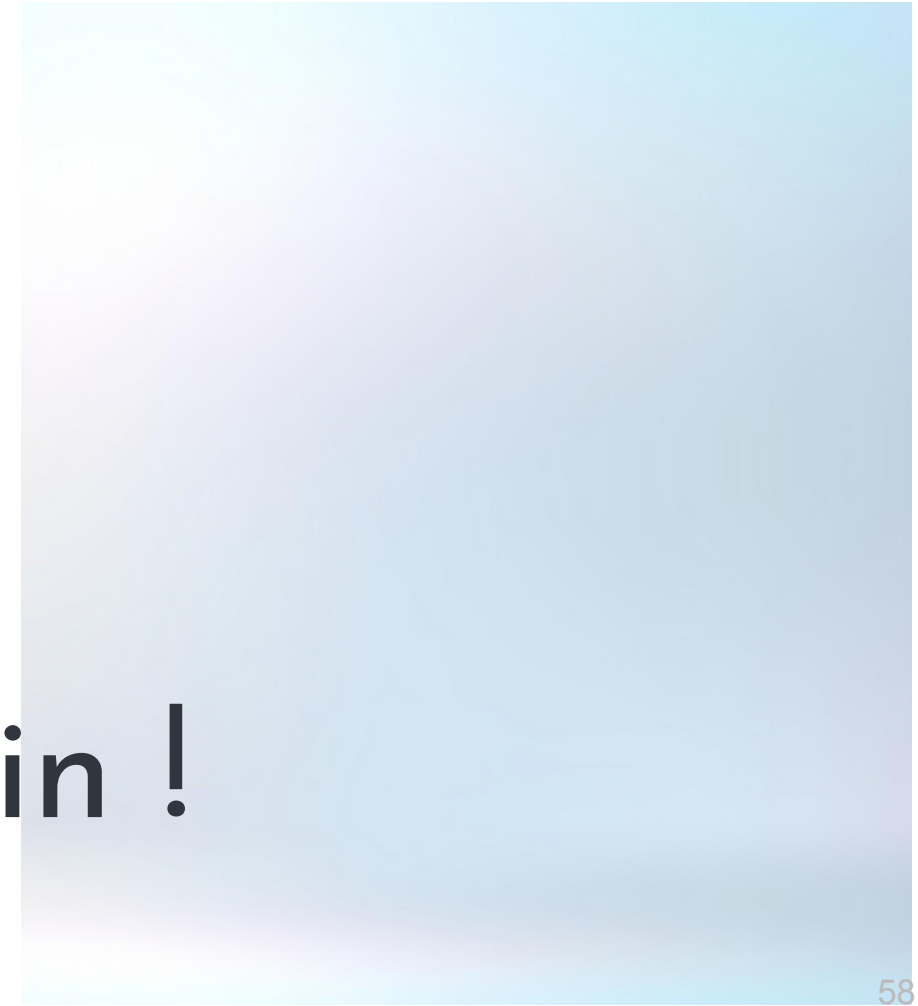
09

Conclusion

1. Rust is safe
    a. Safety of Rust have been studied by several academic papers.
    b. Rust and Java can prevent 6 and 5 of Top 25 CWEs C/C++ cannot prevent, respectively.
2. Developing Awkernel
    a. Single address space operating system
    b. Tested by using formal methods
    c. Preemptible async/await
    d. Single scheduling domain

Future Work of Awkernel

- Block device and file systems
- Performance evaluation
- Apply to autonomous driving systems



CONTACT US

https://tier4.jp/

Thanks Again !