

一种基于Rust语言的操作系统内核服务可靠性增强设计与实现

邓荣达

2025-06-06



CONTENTS

- 绪论
- 内核服务可靠性增强设计
- 实验分析
- 总结展望



01

绪论

绪论

- 研究背景
- 相关工作
- 研究目的
- 本文工作



研究背景

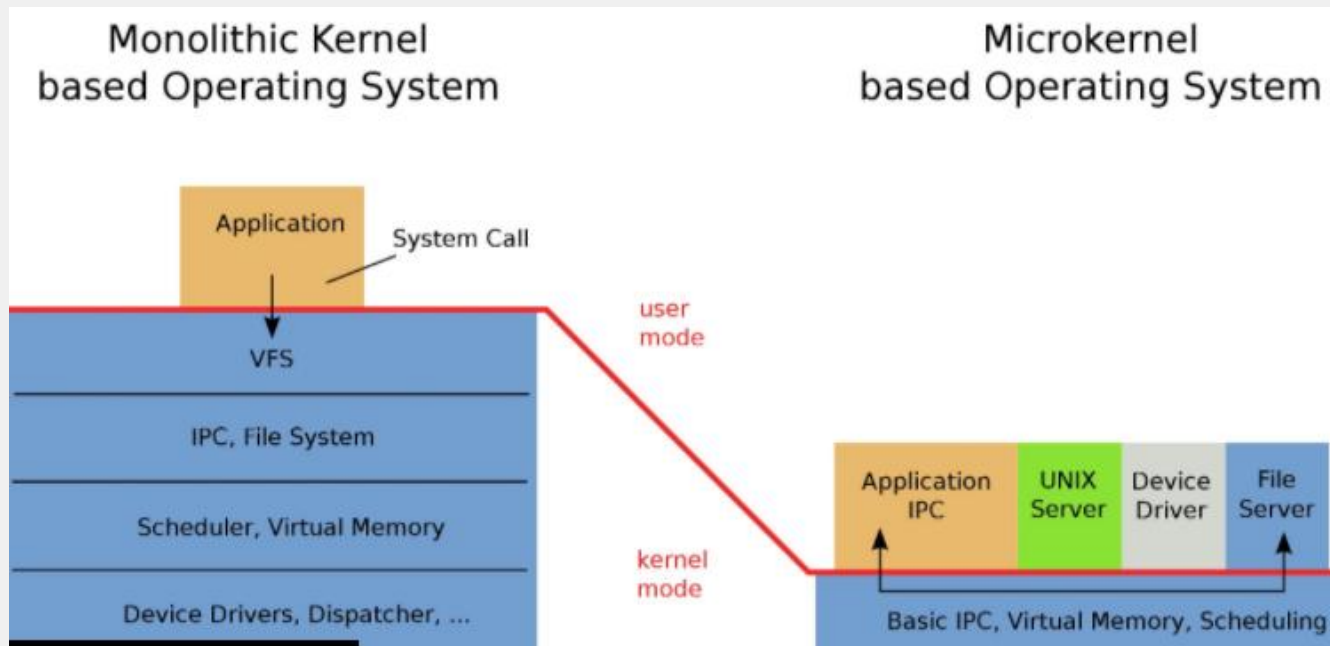
- 操作系统可靠性
- 微内核架构
- Rust内存安全语言
- 基于Rust语言改进内核架构

研究背景：操作系统可靠性

- 可靠性是操作系统的重要设计目标
 - 1996 年欧洲空间局（ESA）阿丽亚娜 5 型火箭（Ariane 5）由于软件缺陷而爆炸。
- 良好的可靠性设计可以帮助操作系统持续为用户提供服务。
- 操作系统内核架构是影响操作系统可靠性的重要因素。

研究背景：微内核架构

- 具备**良好的隔离性**，很大程度隔离内核错误的传播
- 比宏内核**更高的可靠性**。
- 通过**进程间通信 (IPC)** 交互，引入了**显著的性能开销**。



研究背景：Rust内存安全语言

- 系统级语言，高效，性能接近C语言。
- 语言层面保证内存安全、并发安全。
- 良好的异步支持（async Rust）。
- 表达能力强，且有丰富的第三方库。

研究背景：基于Rust语言改进内核架构

- 内存安全性
 - 在同一地址空间实现内存隔离
 - 优化微内核性能，用户态服务下沉内核态，由Rust保证访存隔离。
- 无栈异步协程支持
 - 支持一个线程中的多个异步任务并发
 - 优化线程模型，一个内核线程服务多个用户线程

绪论

- 研究背景
- 相关工作
- 研究目的
- 本文工作

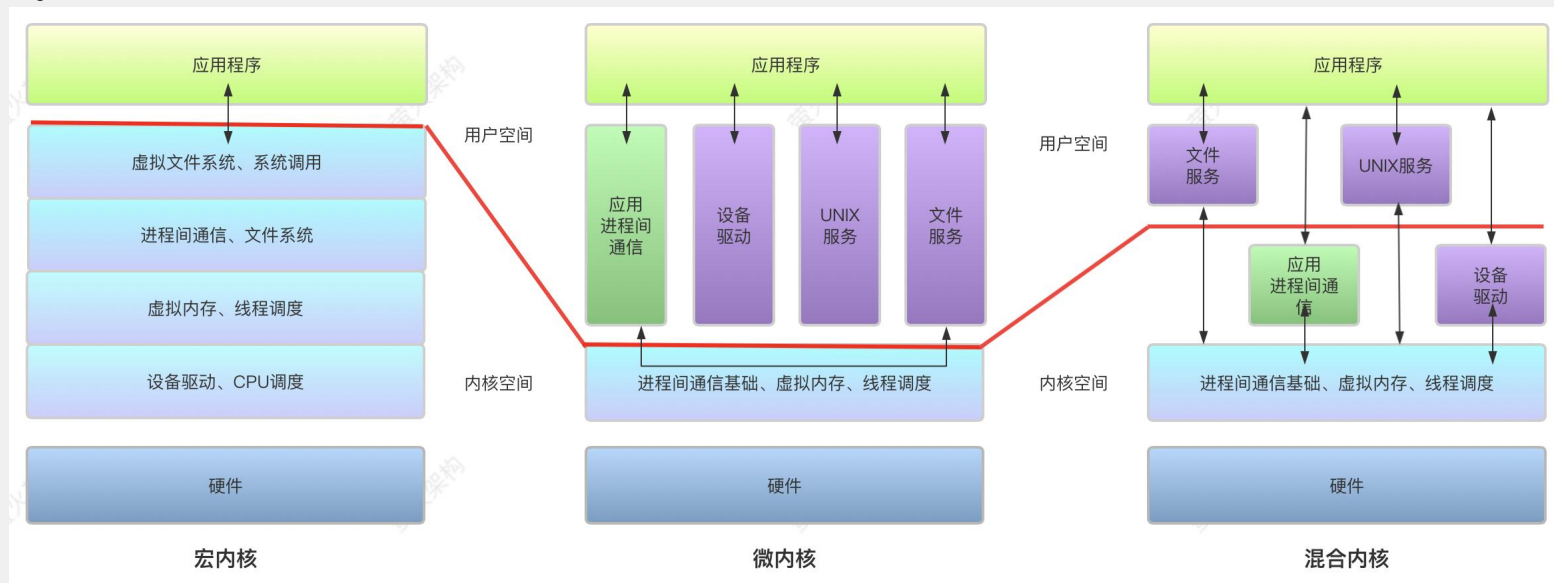


相关工作

- 主流内核架构
- 线程模型
- 基于Rust语言的内核设计

相关工作：主流内核架构

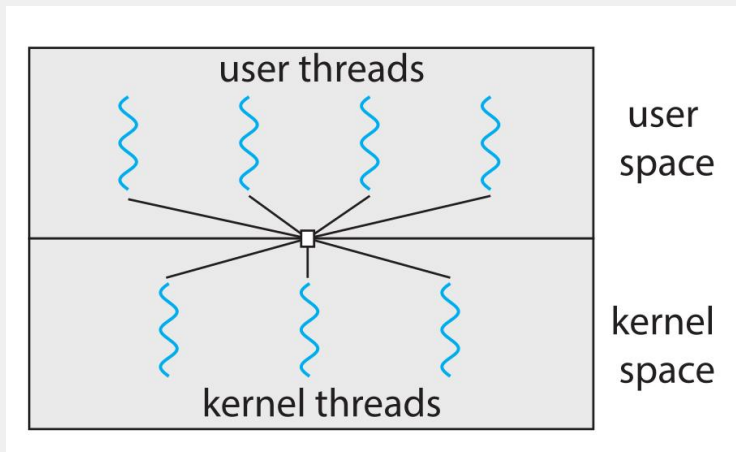
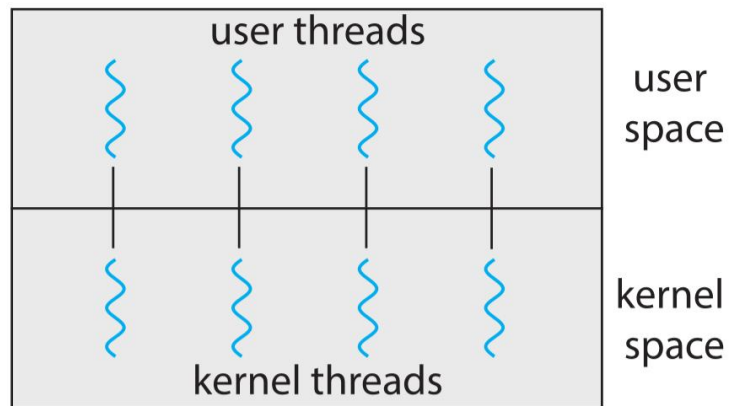
- 宏内核：所有内核服务全部集成在内核态，**性能好，可靠性、可拓展性差。**
- 微内核：核心服务在内核态，非核心服务用户态，**可靠性强，性能较差。**
- 混合内核：折中方案，非核心服务以**独立内核线程**的方式运行在内核态，**避免IPC开销。**



相关工作：线程模型

一对一线程模型：一个用户线程配备一个内核线程（有独立内核栈），内核需要**限制用户线程数量、内核线程数量多，上下文开销大。**

多对多线程模型：每个内核线程服务多个用户线程，**节省内存，节省上下文切换开销。**



相关工作：基于Rust的内核设计

- Theseus：使用Rust内存安全特性实现单地址空间隔离。
- Redox：使用Rust实现的微内核。
- Asterinas：使用Rust重新实现Linux。利用内存安全特性增强可靠性。
- 总结：多为利用内存安全特性避免内存缺陷。而**没有解决其它缺陷**(如逻辑错误触发 panic)。

绪论

- 研究背景
- 相关工作
- 研究目的
- 本文工作

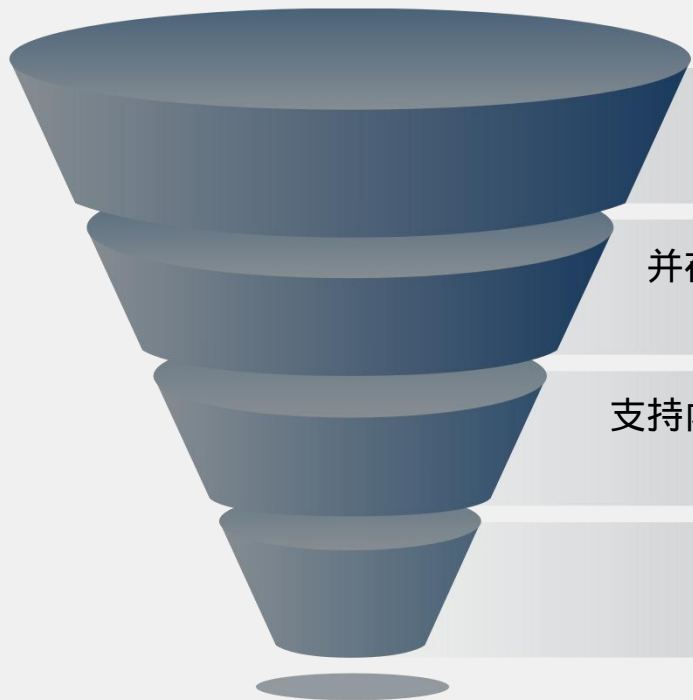
研究目的

- 实现一个混合内核，内存安全特性代替传统地址空间隔离,保证服务间隔离性的同时,避免IPC 产生的性能开销;
- 在这一点基础上,设计一种内核服务故障恢复机制,进一步提升内核可靠性,应对内存缺陷之外的其它缺陷(如逻辑错误触发 panic)。

绪论

- 研究背景
- 相关工作
- 研究目的
- 本文工作

本文工作



本课题基于Rust语言，从头设计实现了一个多
对多线程模型的混合内核，我们称其为nCore。

并在此基础上实现了一种简单的内核服务线程故障
恢复机制。

支持内存管理、多线程管理、文件系统、同步互斥、
Shell等用户程序等多个功能模块。

代码量在6500行左右。

本文工作

```
mangp@mangps:~/桌面/OS/mini-Rust-os$ cloc . --exclude-dir=crates,target,musl,tutorial
179 text files.
172 unique files.
14 files ignored.
```

```
github.com/AlDanial/cloc v 1.98 T=0.11 s (1626.0 files/s, 109641.5 lines/s)
```

Language	files	blank	comment	code
Rust	136	1151	1869	6663
Assembly	2	2	4	1312
TOML	22	25	8	201
Markdown	1	42	0	89
make	3	29	2	75
JSON	3	0	0	43
YAML	1	1	3	36
Linker Script	1	7	0	22
C	1	1	0	6
Text	2	3	0	4
SUM:	172	1261	1886	8451

代码目录树

```
.
├── boot                // 启动内核
├── crates              // 有修改的第三方库
├── executor           // 协程执行器
├── hal                // 硬件抽象层
├── hybrid-objects     // 混合内核对象
├── hybrid-syscalls    // 混合内核系统调用
├── loader             // 加载器
├── micro-objects      // 微内核对象
├── micro-syscalls     // 微内核系统调用
├── monolithic-objects // 宏内核对象
├── monolithic-syscalls // 宏内核系统调用
├── musl               // musl源码
├── Ncore              // 顶层内核
├── rcore-fs-use       // 构建文件系统
├── tutorial           // 文档
├── user-c             // C语言用户程序
├── user-components    // 用户态组件
└── user-rs           // Rust用户程序
```



02

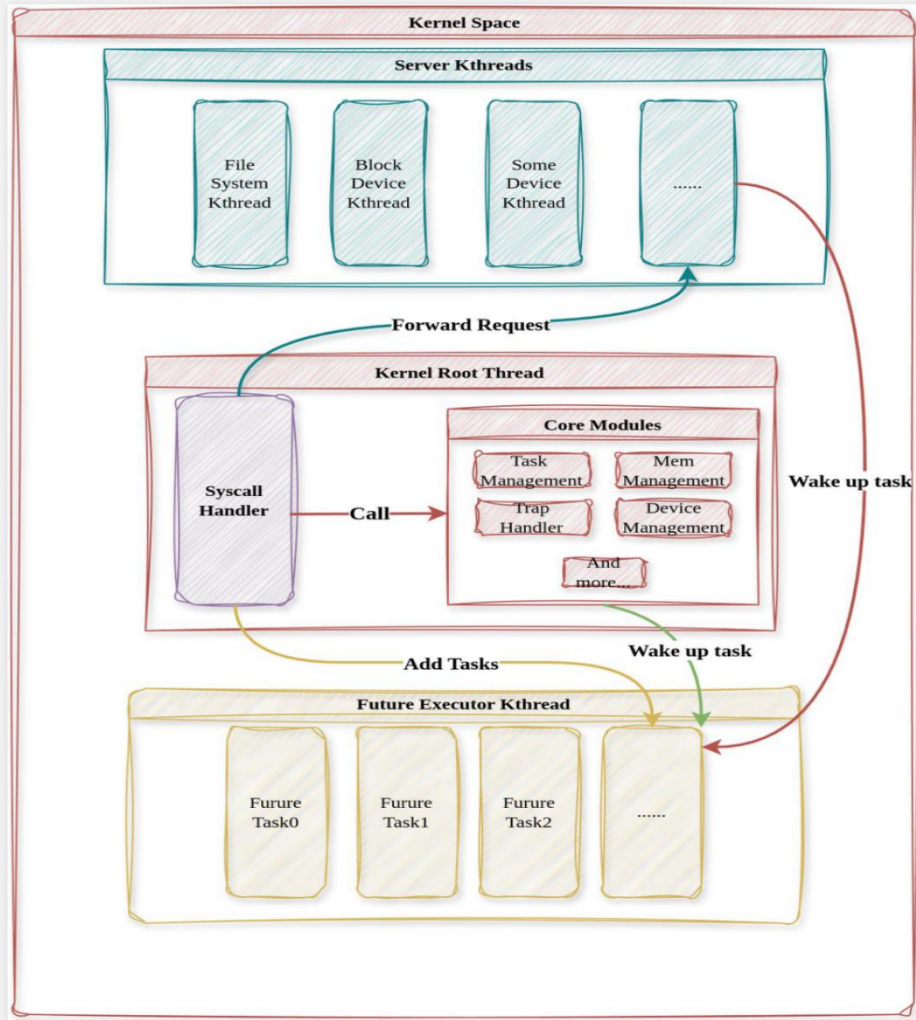
内核服务可靠性增强设计

内核服务可靠性增强设计

- 多对多线程模型
- 内核服务线程故障恢复

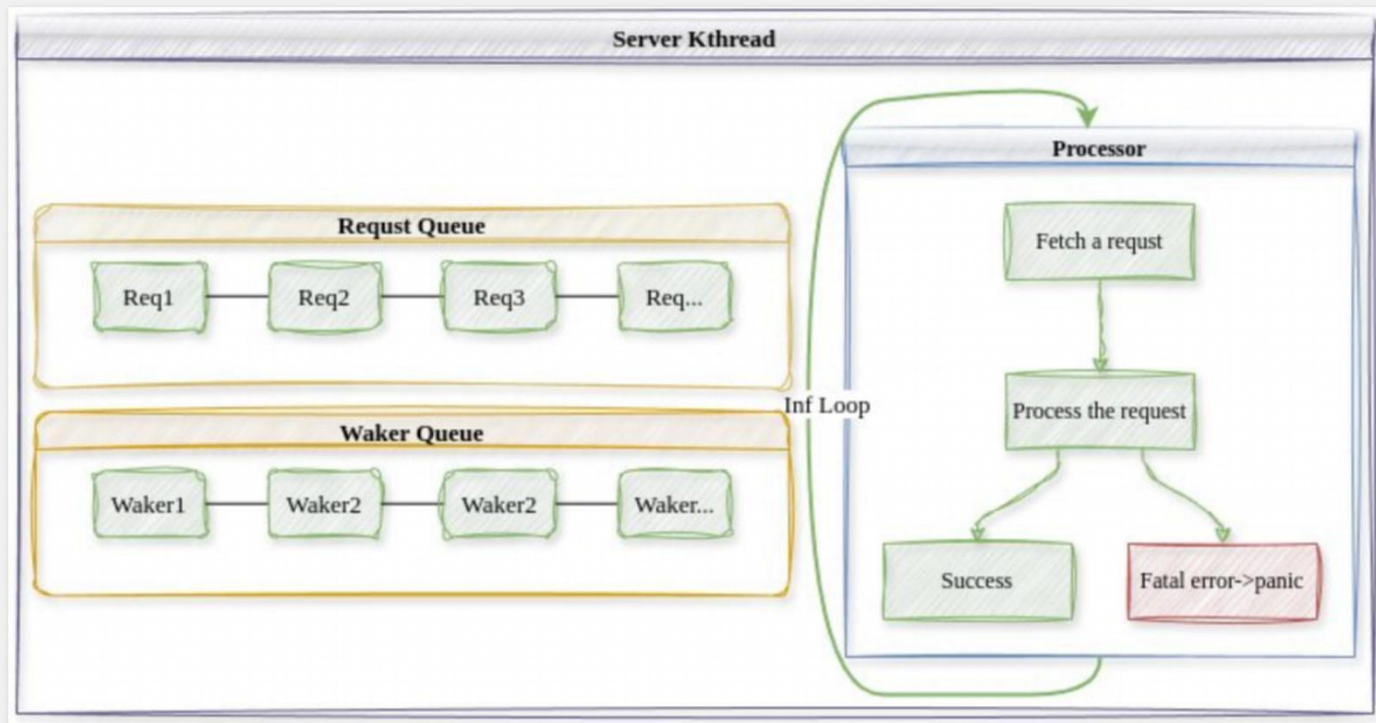
多对多线程模型

- 内核服务线程提供非核心服务（如文件系统、驱动程序等）
- 协程执行线程执行内核所有异步任务
- 微内核线程（根线程）提供核心服务（任务管理、内存管理等）
- 三种内核线程**共同服务所有用户线程**



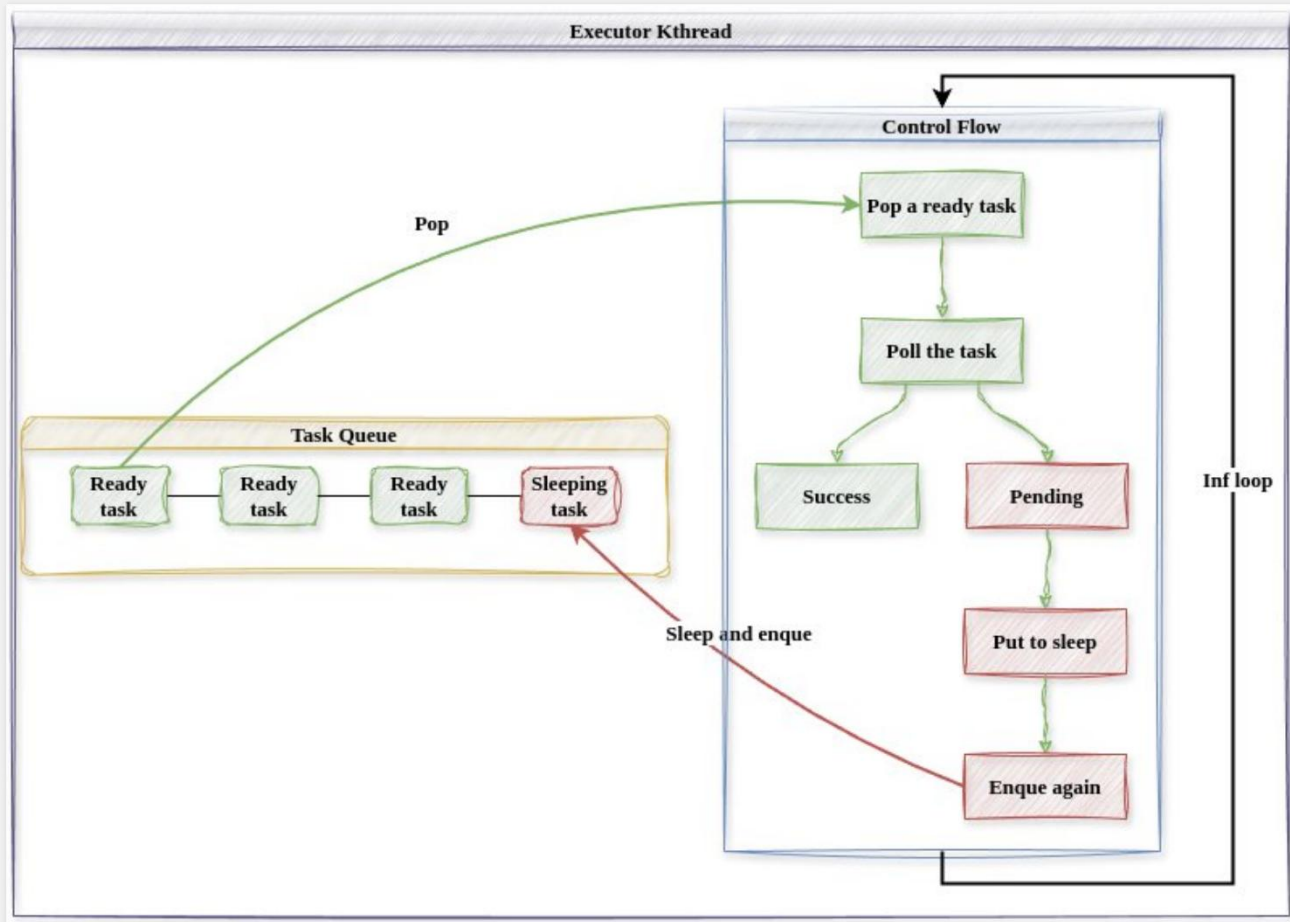
内核服务线程设计

- 提供一种特定的内核服务
- 拥有**独立的内核栈和控制流**
- 崩溃时**不影响其他内核线程和内核**



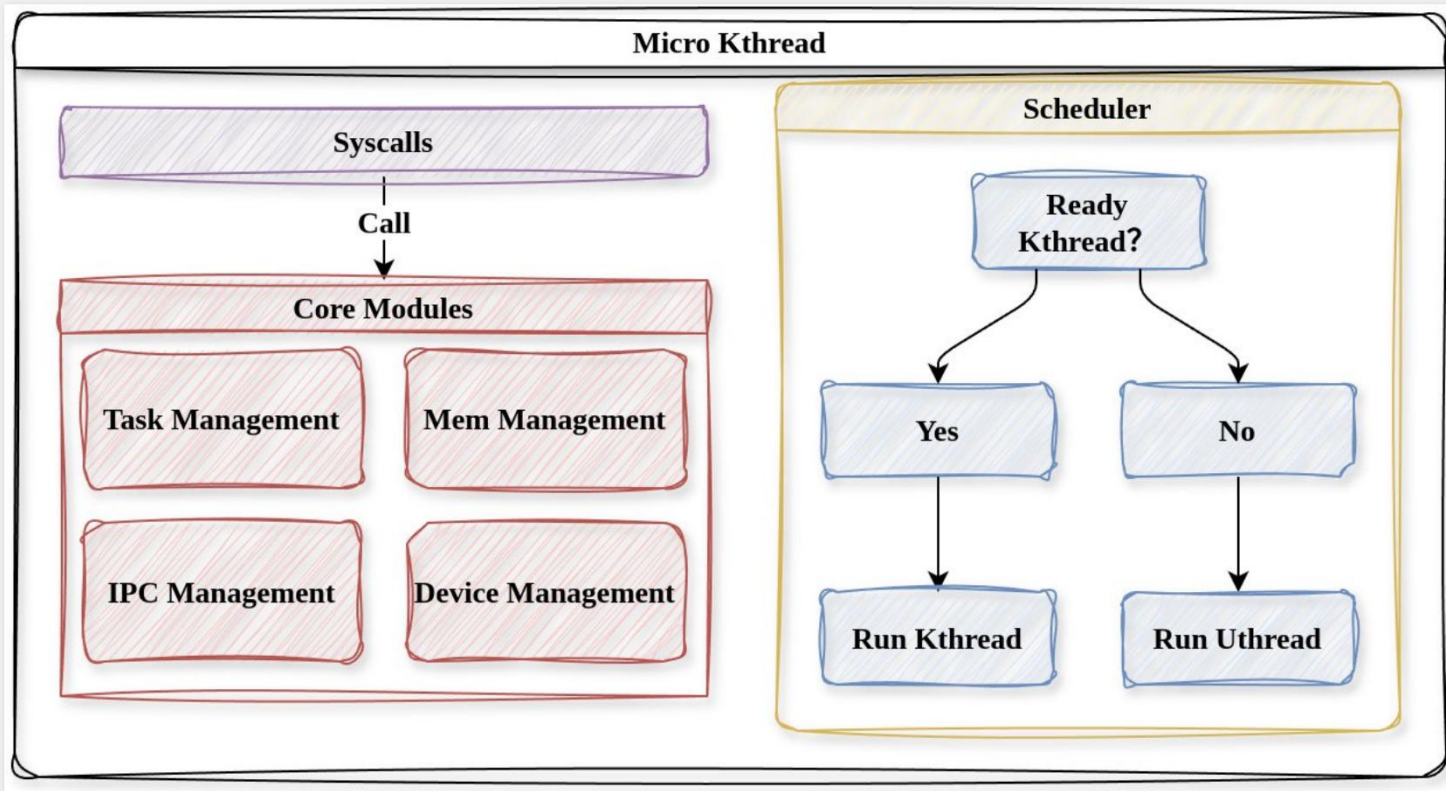
协程执行器线程设计

- 以Rust无栈异步协程代替传统内核线程
- 降低上下文切换开销
- 节省内存



微内核线程设计

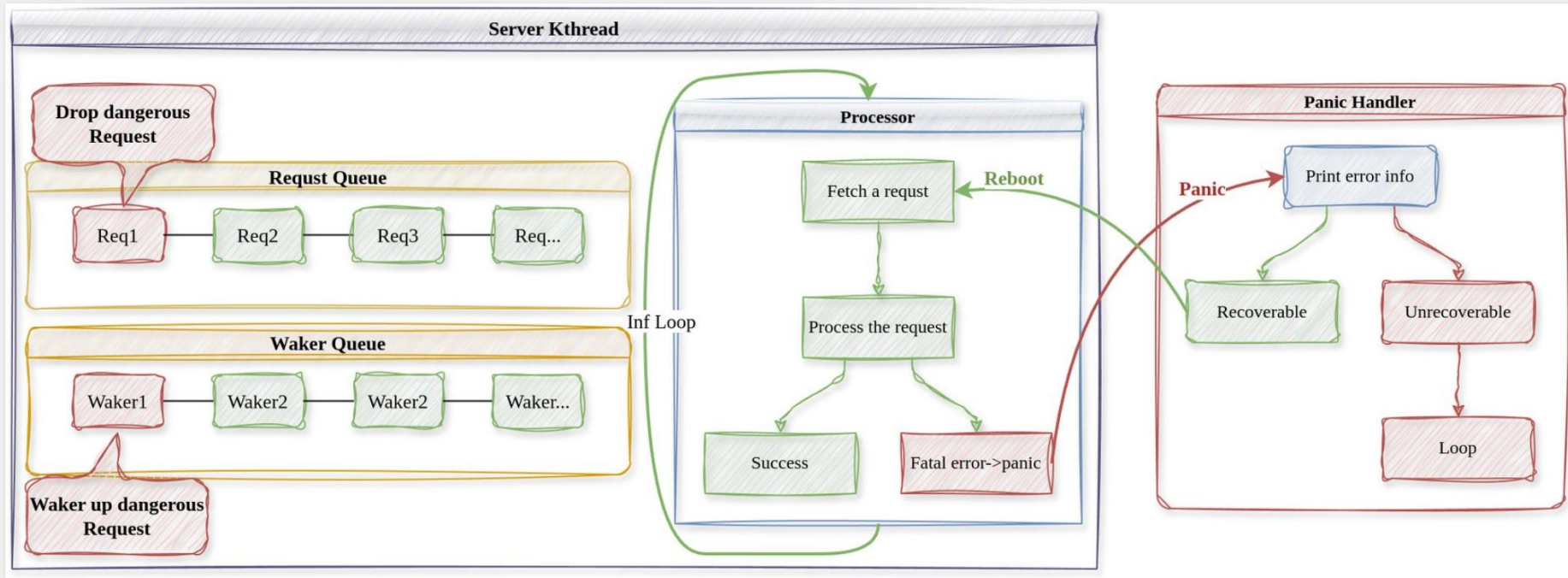
- 提供内核核心服务
- 统一调度所有内核线程和用户线程
- 转发用户线程请求



内核服务可靠性增强设计

- 多对多线程模型
- 内核服务线程故障恢复

内核服务线程故障恢复



- 内核服务线程故障时，其丢弃出错请求，内核重置内核栈和控制流。
- 任何内核服务线程故障不会影响内核和其余内核服务线程。



03

实验分析

实验分析

- 系统调用功能性测试
- 可靠性测试

系统调用功能性测试

- 手写Rust no_std环境应用程序

- 正在完成对musl-libc的支持

名称	系统调用	说明	测试方法
shell	fork、exec、pipe、dup、wait4	支持管道、重定向的shell程序	测试是否能够正确执行其他用户进程；测试是否能够正确调用实现文件重定向和管道
pipetest	open、read、write、pipe、close	父进程写管道、子进程读管道	验证读写管道的字符串是否一致
opentest	open、read、write、close	父进程写文件、子进程读文件	验证读写文件是否一致
thread_args	thread_create、thread_join	创建多个线程打印信息	验证是否能够创建并回收线程
test_condvar	mutex_create、mutex_lock、mutex_unlock、condvar_create、condvar_wait、condvar_signal、thread_create、thread_join	结合条件变量和互斥锁实现多线程的时序	验证是否能够按照时序输出
mpsc_sem	sem_create、sem_up、sem_down、thread_create、thread_join	使用信号量实现多生产者单消费者模型	验证生产者消费者是否能正常协作

可靠性测试

- 手动触发panic模拟低质量驱动程序内部发生的逻辑错误
- 测试内核服务线程崩溃后是否能够重启并恢复内核执行控制流。

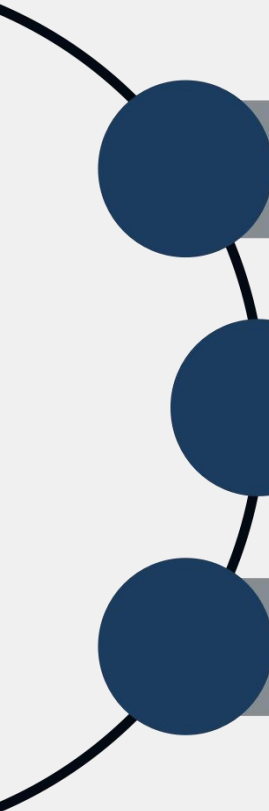
```
[Kernel] Starting main loop...
[Shell] Entered shell now!
[Shell] >> reboot
[Fs-server] Request 1 processed over!
[Fs-server] Request 2 processed over!
[Fs-server] Request 3 processed over!
[Fs-server] Request 4 processed over!
[Fs-server] Request 5 processed over!
[kernel] Panicked at hybrid-objects/src/requests/fs_processor.rs:70 [Fs Processor] Fatal error in write request!
[Panic handler] Trying to Reboot..., the dangerous request(ID: 6) will be dropped!
Kthread: Fs-server reboot success
[Executor] WaitForKthread poll pending, response_id: 6, req_id: 7
[Fs-server] Request 7 processed over!
[Fs-server] Request 8 processed over!
[Fs-server] Request 9 processed over!
[Fs-server] Request 10 processed over!
[kernel] Panicked at hybrid-objects/src/requests/fs_processor.rs:70 [Fs Processor] Fatal error in write request!
[Panic handler] Trying to Reboot..., the dangerous request(ID: 11) will be dropped!
Kthread: Fs-server reboot success
[Executor] WaitForKthread poll pending, response_id: 11, req_id: 12
[Fs-server] Request 12 processed over!
[Fs-server] Request 13 processed over!
[Fs-server] Request 14 processed over!
[Fs-server] Request 15 processed over!
[kernel] Panicked at hybrid-objects/src/requests/fs_processor.rs:70 [Fs Processor] Fatal error in write request!
[Panic handler] Trying to Reboot..., the dangerous request(ID: 16) will be dropped!
Kthread: Fs-server reboot success
[Executor] WaitForKthread poll pending, response_id: 16, req_id: 17
[Fs-server] Request 17 processed over!
[Fs-server] Request 18 processed over!
[Fs-server] Request 19 processed over!
[Fs-server] Request 20 processed over!
[kernel] Panicked at hybrid-objects/src/requests/fs_processor.rs:70 [Fs Processor] Fatal error in write request!
[Panic handler] Trying to Reboot..., the dangerous request(ID: 21) will be dropped!
Kthread: Fs-server reboot success
[Executor] WaitForKthread poll pending, response_id: 21, req_id: 22
[Fs-server] Request 22 processed over!
[Fs-server] Request 23 processed over!
Write 20 bytes, read 16 bytes, 4 requestes dropped totally
reboot test passed!
[Shell] >>
```



04

总结展望

总结展望



本文基于Rust语言的内存安全特性实现了一个多**对多线程模型**的**混合内核**，并基于此实现了内核服务**可靠性增强**，并实现了一种简单的关键系统服务**故障恢复机制**。

目前尚不支持C语言用户程序，未来应该使用原生Linux应用程序进行性能对比测试

目前的故障恢复机制只依赖于Rust语言的panic捕捉，不能适应普遍情况。

欢迎各位老师批评指正
THANKS

