动态内核模块放置 - Midterm

刘晓义

2022.10.13

Motivation

微内核是好的，但是延迟大。

Motivation

微内核是好的，但是延迟大。
宏内核也是好的，但是 Syscall 慢。

Motivation

微内核是好的，但是延迟大。
宏内核也是好的，但是 Syscall 慢。

Linux 内核模块可以选择动态加载和编译进内核

Motivation

微内核是好的，但是延迟大。
宏内核也是好的，但是 Syscall 慢。

Linux 内核模块可以选择动态加载和编译进内核
能否添加一种将模块可选放置在用户态的办法？

```
modprobe some_mod
modprobe --user-process another_mod
modprobe --sync-caller and_another_mod
```

Approach

- 需要一个办法能够在用户态进程之间传递消息
- 需要一个对用户代码透明的办法，在不同消息传递方法之间切换

Approach

- 需要一个办法能够在用户态进程之间传递消息
- 需要一个对用户代码透明的办法，在不同消息传递方法之间切换

- 建立一个类似 C/S 结构的服务模型
  - A fancy way to say io_uring
- 所有模块和用户态进程都声明自己的提供的**服务**和需要的**依赖**
- 内核在加载代码段的时候动态决定调用的具体方式: similar to vDSO

Approach, Cont.

```
/**
 * Service handler,
 * - May be called by event queue.
 * - May be called directly if applicable.
 */
uint64_t meow_service_handler(message *msg);

REGISTER_SERVICE(
  0x8080808080808080ull, // Service Identifier (syscall number)
  meow_service_handler,
);

void init() {
  // Setup process states...

  // Event queue implementation provided by kernel
  INIT_SERVICES();
}
```

Approach, Cont.

```
let meow_service(args).await!;
```

Approach, Cont.

```
let meow_service(args).await!;
```

也许会被实现成:

- 直接调用（对应模块和本模块在同一地址空间，例如同时在内核中，并且要求低延迟）
- 同步 syscall（对应模块在内核中，本模块在用户态中，并且要求低延迟）
- Ring（对应模块以进程形式执行，无论是在内核还是在用户态）

Approach, Cont.

```
let meow_service(args).await!;
```

也许会被实现成:

- 直接调用（对应模块和本模块在同一地址空间，例如同时在内核中，并且要求低延迟）
- 同步 syscall（对应模块在内核中，本模块在用户态中，并且要求低延迟）
- Ring（对应模块以进程形式执行，无论是在内核还是在用户态）

实现使用 vDSO 注入

Challenges

如何实现一个高效的 Ring，尽量减少内核参与？

Challenges

如何实现一个高效的 Ring，尽量减少内核参与？

- 用户态中断
- 共享内存 + 一些额外唤醒机制

Trap fast path

传统 Signal：保存现场，切换到内核栈，然后进入内核代码处理 Syscall。完成后恢复现场，切换回用户栈。
我们希望唤醒操作尽量快

Trap fast path

传统 Signal：保存现场，切换到内核栈，然后进入内核代码处理 Syscall。完成后恢复现场，切换回用户栈。
我们希望唤醒操作尽量快

保留一个页存储所有被唤醒的目标进程，当下一次 Scheduler 被触发的时候执行（irq_work?）：

```
#define WORKQUEUE_CAP ((4096 / 8) - 1)
struct WakeupQueue {
  uint64_t cur = 0;
  uint64_t slots[WORKQUEUE_CAP];
};
```

sscratch 保存一个 Struct 的指针：

```
struct BackgroundContext {
  void *kstack_top;
  WorkQueue *wake_queue;
  // Other important stuff
}
```

Apprach, Cont.

```
trap_enter:
  beq a0, x0, wakeup_fast_path
  // ...
wakeup_fast_path:
  csrrw t0, sscratch, t0
  beq t0, x0, trap_cont

  STORE(t1, 0)
  STORE(t2, 1)
  csrr t1, scause
  bneq t1, U_CALL, trap_cont

  ld t1, WAKEUP_PAGE_BASE(t0)
  li t2, 1
  amoadd.d t2, t2, (t1)
  subi t2, t2, WORKQUEUE_CAP
  blt t2, x0, wakeup_full
```

```
addi t2, t2, WORKQUEUE_CAP
slli t2, t2, 3

addi t2, t1, t2
sd a1, (t2)

LOAD(t1, 0)
LOAD(t2, 1)
csrrw t0, sscratch, t0
sret
```

Progress

- 实现了上述 Ring 和 sched 配套的唤醒机制
- 定义了 Syscall 接口
  - 0: 唤醒
  - 1: WFI
  - 2: 同步调用其他服务
- 定义了一系列内核提供的服务
- 现在用户态进程可以**显式**异步调用服务：
  - 串口输出，位于用户态
  - brk 内存分配，由内核提供

Plan

为了保证对用户代码透明，所有服务调用都返回 Future（如果被配置成同步调用，会返回一个 Resolved Future）
内核需要根据服务调用者和被调用者之间的相对位置关系，和用户配置，决定链接什么调用代码。

- 直接函数调用方法类似动态链接（PLT indirect call）
- 异步调用在 PLT 基础上需要链接 Ring 相关的管理逻辑

Plan

为了保证对用户代码透明，所有服务调用都返回 Future（如果被配置成同步调用，会返回一个 Resolved Future）
内核需要根据服务调用者和被调用者之间的相对位置关系，和用户配置，决定链接什么调用代码。

- 直接函数调用方法类似动态链接（PLT indirect call）
- 异步调用在 PLT 基础上需要链接 Ring 相关的管理逻辑

注意到，同一个模块的所有下游使用者可以共享相同的 Ring 管理逻辑

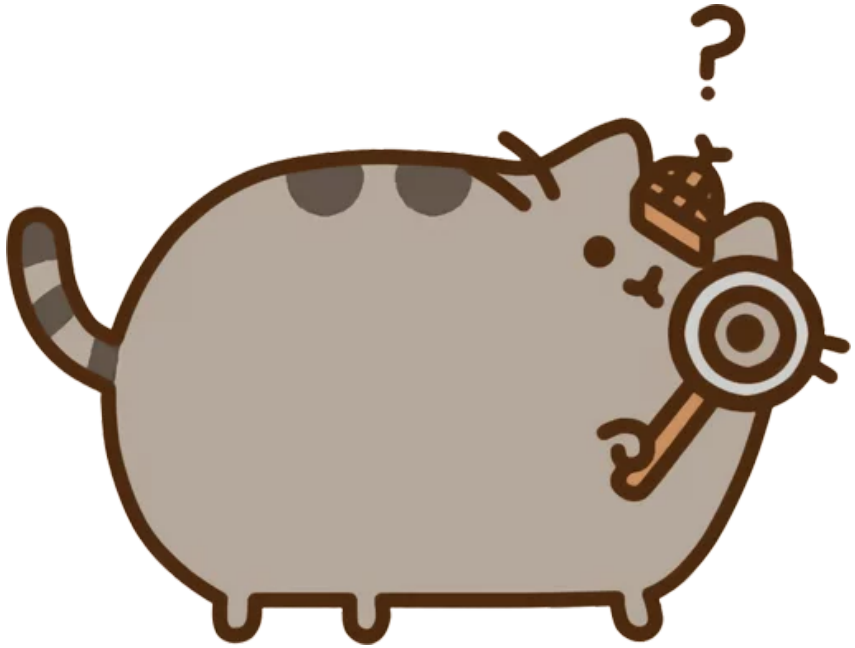
- 在编译模块的时候额外添加一个段放置这些代码（.remote.text），是由 `REGISTER_SERVICE` 生成的。
- 加载模块的时候在内核中注册
- 服务的使用者在加载时将上述代码链接进去，并且直接把 PLT 指向那里

Plan, Cont.

接下来需要干的事情：

- 动态链接基础设施
- vDSO 实现
- `.remote.text` 代码生成

That's All!



Question time!

`https://github.com/CircuitCoder/ChannelOS`