操作系统 Lab 05 内核中的进程与线程

姓名: 雍崔扬
学号: 21307140051

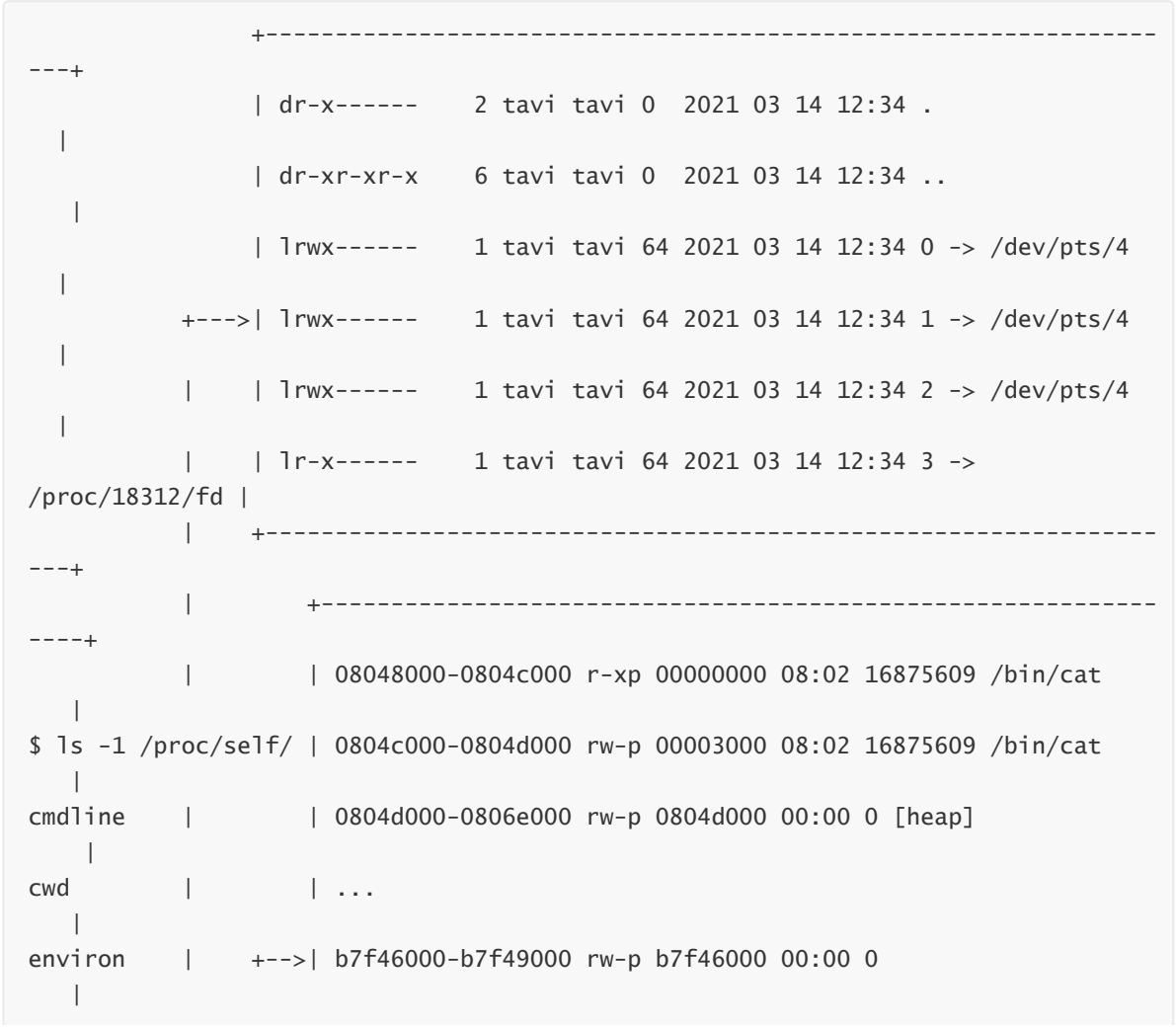
0. 基础知识

0.1 进程

进程是将多种资源组合在一起的操作系统抽象：

- 地址空间
- 打开的文件描述符
- 信号量
- 共享内存区域
- 计时器
- 信号处理程序
- 许多其他资源和状态信息

所有这些信息都保存在 Process Control Group (PCB) 之中.
在 Linux 中，即 `struct task_struct`
在内核中，我们可以通过 `task_struct` 来访问进程的所有信息;
而在用户空间，我们可以通过 `/proc` 文件系统来访问进程的信息.
如下所示 (`/proc/self/` 会被内核自动翻译为 `/proc/<current pid>/`):



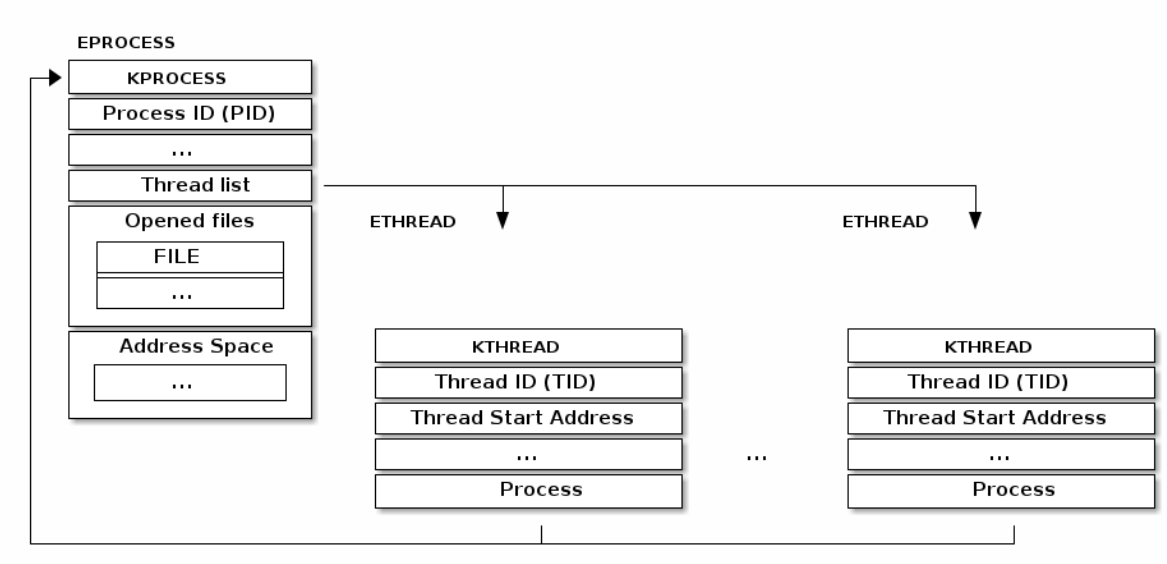
```
exe         |     |    | b7f59000-b7f5b000 rw-p b7f59000 00:00 0
    |
fd --------+     |    | b7f5b000-b7f77000 r-xp 00000000 08:02 11601524 /lib/ld-
2.7.so   |
fdinfo           |    | b7f77000-b7f79000 rw-p 0001b000 08:02 11601524 /lib/ld-
2.7.so   |
maps ------------+    | bfa05000-bfa1a000 rw-p bffeb000 00:00 0 [stack]
    |
mem                   | ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]
     |
root                 +-------------------------------------------------------
----+
stat                  +----------------------------+
statm                 |  Name: cat                 |
status ------+        |  State: R (running)        |
task         |        |  Tgid: 18205               |
wchan        +------->|  Pid: 18205                |
                      |  PPid: 18133               |
                      |  Uid: 1000 1000 1000 1000  |
                      |  Gid: 1000 1000 1000 1000  |
                      +----------------------------+
```

0.2 线程

内核在分配 CPU 资源时，调度器在不同的线程之中进行选择，线程是调度的基本单位.
线程具有以下特征:

- 每个线程都有自己的堆/栈和寄存器上下文
- 线程在进程上下文中运行，同一进程内的所有线程共享了许多资源
- 内核调度线程而不是进程，并且用户级线程 (例如纤程 Fiber、协程 Coroutine 等) 在内核中不可见

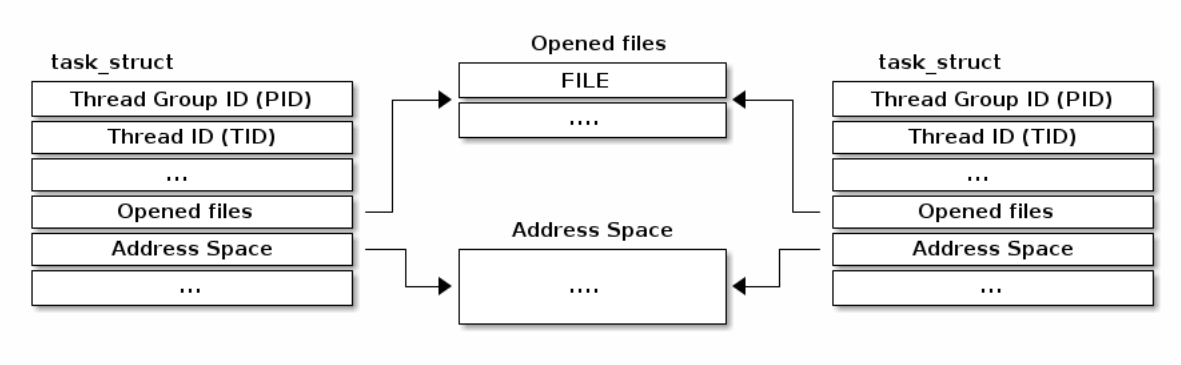
由于线程是依附于进程上下文而存在的，因此通常我们会定义一个 `thread` 结构体，
然后在进程的结构体中将线程们链接成一个链表.
例如，Windows 内核使用这样的实现:



Linux 对线程的实现有所不同.
在 Linux 中调度的基本单位称为任务 (即 `struct task_struct`)，不论线程还是进程都由一个 `task_struct` 来表示.
如果 (抽象来说) 两个线程属于同一个进程，那么他们的 `struct task struct` 中许多资源都将指向同

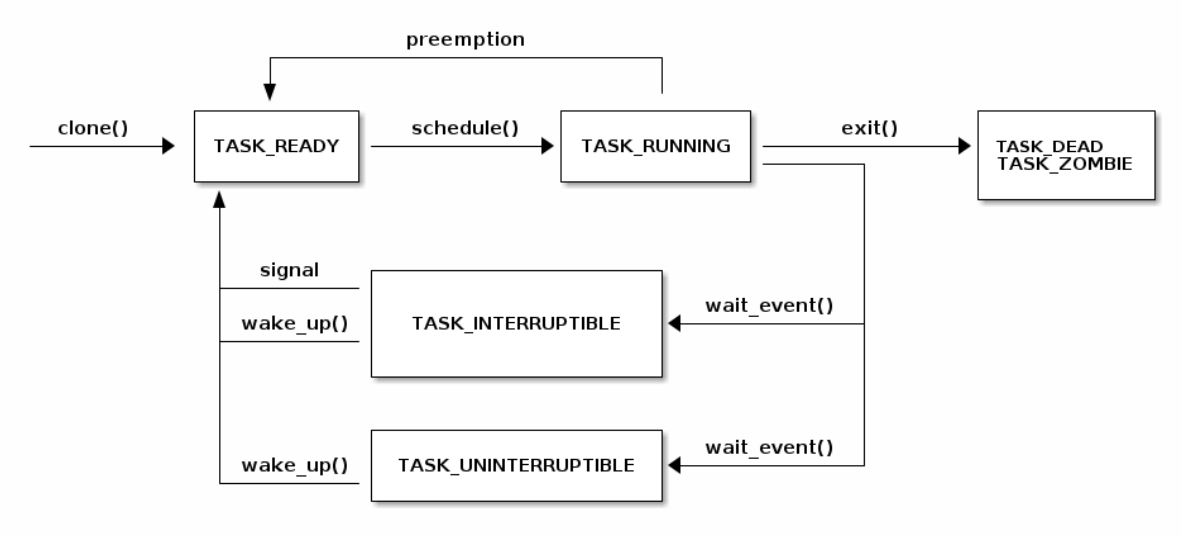
一个资源结构实例，
比如虚拟内存空间、文件描述符等.

可以说 Linux 不存在线程的概念，只存在任务的概念.
我们可以控制两个任务共享哪些资源，从而有了更加灵活的线程管理方式.



0.3 线程的阻塞和唤醒

下图显示了任务 (线程) 状态及其之间可能的转换:



- `TASK_READY`: 任务已经准备好运行，但是还没有被调度器选中
- `TASK_RUNNING`: 任务正在运行
- `TASK_INTERRUPTIBLE`: 任务正在睡眠中 (阻塞中)，并等待某个条件达成 (比如 I/O 数据抵达)，可以被信号唤醒
- `TASK_UNINTERRUPTIBLE`: 任务正在睡眠等待某个条件达成，但是不会被信号唤醒
- `TASK_DEAD` / `TASK_ZOMBIE`: 任务已经终止

阻塞当前线程是我们需要执行的一项重要操作，以实现高效的任务调度——我们希望在 I/O 操作执行时运行其他线程.
为了实现这一点，内核中的 `wait_event` 函数会进行以下操作:

- 将当前线程状态设置为 `TASK_UINTERRUPTIBLE` 或 `TASK_INTERRUPTIBLE`
- 将任务添加到等待队列
- 调用调度程序，从 `READY` 队列中挑选一个新任务
- 进行上下文切换到新任务

而与之对应的 `wake_up` 函数则会:

- 从等待队列中选择一个任务
- 设置任务状态为 `TASK_READY`

- 将任务插入到调度程序的 `READY` 队列中

Task 1: 打乒乓

1.1 `ioctl` 接口

`ioctl` (输入/输出控制) 是一个系统调用，允许用户空间程序与内核模块或设备驱动程序进行通信.
它提供了一种机制，允许在用户空间和内核空间之间传递控制指令和数据，
使应用程序能够以普通系统调用 (如 `read()` 或 `write()`) 无法实现的方式与硬件设备或内核级功能进行交互.

```
long ioctl(int fd, unsigned long request, ...);
```

- `fd` : 要交互的设备或文件的文件描述符
- `request` : 命令编号 (或操作)，定义了要执行的操作
- `...` : 可选参数，具体取决于特定的 `ioctl` 请求，可能包括指向在用户空间和内核空间之间传递数据的指针.

每个 `ioctl` 请求通过唯一的命令号来标识，通常通过宏定义来编码多个组件:

```
#define _IO(type, nr)        /* 为不传输数据的命令创建 ioctl 编号 */
#define _IOW(type, nr, size)  /* 为写命令创建 ioctl 编号 */
#define _IOR(type, nr, size)  /* 为读命令创建 ioctl 编号 */
#define _IOWR(type, nr, size) /* 为既读又写的命令创建 ioctl 编号 */
```

- `type` : 设备或功能组的唯一标识符
- `nr` : 标识特定操作的编号
- `size` : 传输数据的大小 (仅对读/写命令有效)

1.2 Task

在本实验中，我们将创建两个用户进程，一个进程打印 `ping` ，另一个进程打印 `pong` .
我们将实现一个内核模块，通过 `wait_event_interruptible` 和 `wake_up` 函数 (其实是宏) 来向用户态提供类似于同步的功能.
一个线程等待另一个线程将球打过来 (表现为某个 bool flag 取反)，
醒来后把球打回去 (取反那个 flag) 并再把对手叫醒，然后自己睡去.
在这个过程中，两个任务将交替运行.

`wait_event_interruptible` 函数的原型如下:

```
long wait_event_interruptible(wait_queue_head_t *q, int condition);
```

我们需要传入一个等待队列头 `q` 和一个条件 `condition` .
如果 `condition` 为真，那么函数会立即返回;
否则，函数会将当前任务的状态设置为 `TASK_INTERRUPTIBLE` ，并将任务添加到等待队列中.
当 `condition` 变为真时，我们可以调用 `wake_up` 函数来唤醒等待队列中已经满足条件的任务.

`wake_up` 函数的原型如下:

```
void wake_up(wait_queue_head_t *q);
```

我们需要传入一个等待队列头 `q`，函数会将等待队列中，
满足条件任务状态设置为 `TASK_READY`，并将任务插入到调度程序的 `READY` 队列中.

除此以外，我们需要用 `init_waitqueue_head` 函数来初始化等待队列头：

```
void init_waitqueue_head(wait_queue_head_t *q);
```

任务:

1. **代码准备**
   将分发的代码框架 `lab5` 目录复制到 `skels` 目录下.
2. **阅读框架与写代码**
   - 阅读 `pingpong_user` 和 `pingpong` 下的程序代码，了解程序的功能.
   - 完成 `pingpong/pingpong.c` 代码中的三个 TODO.
3. **编译** (工作环境下)
   - 在 `skels/Kbuild` 目录下添加 `./lab5/pingpong/` 目录 (具体格式参照文件)
   - 在工作目录 `linux/tools/labs` 下执行 `make build` 编译内核模块
     (位于 `lab5/pingpong_user` 目录的测试程序会在 `make build` 时自动编译)
4. **模块加载与运行测试程序** (虚拟环境下)
   - 在工作目录 `linux/tools/labs` 下 `make console` 启动虚拟环境，加载内核模块
   - 进入 `pingpong_user` 目录，执行 `./ping &` 以及 `./pong &`

测试:
如果代码正确，你将看到 `ping` 和 `pong` 交替打印.

1.3 Implementation

`pingpong_ioctl` **函数:**
它检查命令 (`cmd`)，并根据接收到的请求执行相应的操作.

- ① 当 `cmd` 为 `PING_CMD` 时
  它首先进入一个 `while` 循环，检查 `pingpong_flag` 是否为 `1`
  如果是 `1`，说明是 "pong" 进程的轮次，"ping" 进程会被加入到等待队列 `wait_queue`，直到被唤醒.
  唤醒条件是 `pingpong_flag == 0`
  一旦 "ping" 进程被唤醒，它会打印出 `Ping!` 并将 `pingpong_flag` 设置为 `1`
- ② 当 `cmd` 为 `PONG_CMD` 时
  它首先进入一个 `while` 循环，检查 `pingpong_flag` 是否为 `0`
  如果是 `0`，说明是 "ping" 进程的轮次，"pong" 进程会被加入到等待队列 `wait_queue`，直到被唤醒.
  唤醒条件是 `pingpong_flag == 1`
  一旦 "pong" 进程被唤醒，它会打印出 `Pong!` 并将 `pingpong_flag` 设置为 `0`

```
/* ioctl handler function */
static long pingpong_ioctl(struct file *file, unsigned int cmd, unsigned long
arg) {
    switch (cmd) {
        case PING_CMD:
```

```
            /* TODO */
            /* Wait for pong process to wake up */
            while (pingpong_flag == 1) {
                wait_event_interruptible(wait_queue, pingpong_flag == 0);  //
Wait until flag is 0
            }
            pingpong_flag = 1;  // Set flag to 1, indicating it's now pong's
turn to print
            wake_up(&wait_queue);  // Wake up the pong process
            break;

        case PONG_CMD:
            /* TODO */
            /* Wait for ping process to wake up */
            while (pingpong_flag == 0) {
                wait_event_interruptible(wait_queue, pingpong_flag == 1);  //
Wait until flag is 1
            }
            pingpong_flag = 0;  // Set flag to 0, indicating it's now ping's
turn to print
            wake_up(&wait_queue);  // Wake up the ping process
            break;

        default:
            return -EINVAL;
    }
    return 0;
}
```

`pingpong_init` **函数:**
这是内核模块的初始化函数，`static` 标记它仅在定义它的文件内可见.
而 `__int` 表示该函数仅用于模块初始化，在初始化完成后，内核会丢弃该函数.
我们使用 `init_waitqueue_head()` 函数初始化等待队列 `wait_queue`
最后使用 `misc_register` 函数初始化设备 `pingpong_dev` (由 `pingpong_device` 结构体定义)

```
/* Module initialization */
static int __init pingpong_init(void) {
    int ret;

    /* TODO: Initialize the wait_queue */
    init_waitqueue_head(&wait_queue);

    /* Register the misc device */
    ret = misc_register(&pingpong_device);
    if (ret) {
        printk(KERN_ERR "Failed to register misc device\n");
        return ret;
    }

    printk(KERN_INFO "PingPong device initialized with /dev/%s\n", DEVICE_NAME);
    return 0;
}
```

模块加载与运行测试程序 (虚拟环境下)

- 在工作目录 `linux/tools/labs` 下 `make console` 启动虚拟环境，加载内核模块

- 进入 `pingpong_user` 目录，执行 `./ping &` 以及 `./pong &`

```
root@qemux86:~# root
-sh: root: not found
root@qemux86:~# cd ./skels/lab5/pingpong
root@qemux86:~/skels/lab5/pingpong# insmod pingpong.ko
pingpong: loading out-of-tree module taints kernel.
PingPong device initialized with /dev/pingpong_dev
root@qemux86:~/skels/lab5/pingpong# lsmod pingpong
    Tainted: G
pingpong 16384 0 - Live 0xe0875000 (O)
root@qemux86:~/skels/lab5/pingpong# cd ..
root@qemux86:~/skels/lab5# cd ./pingpong_user
root@qemux86:~/skels/lab5/pingpong_user# ./ping & ./pong &
root@qemux86:~/skels/lab5/pingpong_user# Ping!
Pong!
Ping!
Pong!
Ping!
Pong!
Ping!
```

Task 2: Kernel Threads

有的时候内核需要执行一些任务，但是这些任务并不需要与用户空间进行交互.
内核线程是一种特殊的任务，它类似一个内核的 GC 线程，
在背景中运行并负责一些诸如内存合并 (kcompactd)、内存交换 (kswapd) 等任务.
本质上，内核线程是仅在内核模式下运行且没有用户地址空间或其他用户属性的线程.

2.1 内核线程的相关 API

我们可以使用 `kthread_create` 函数来创建一个内核线程:

```
#include <linux/kthread.h>

struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data, const char namefmt[],
...);
```

- `threadfn` 是将由内核线程运行的函数
- `data` 是要发送给函数的参数
- `namefmt` 代表内核线程名称，会在 ps/top 中展示;
  你可以在这里使用 `%d`、`%s` 等，这些序列将根据标准 printf 语法进行替换.

比如，下面这行代码将会创建一个名为 `mykthread0` 的内核线程:

```
kthread_create (f, NULL, "%skthread%d", "my", 0);
```

刚刚创建的内核线程并不会立即运行，而是处于 `TASK_INTERRUPTIBLE` 状态.
我们需要调用 `wake_up_process` 函数来唤醒它:

```
#include <linux/sched.h>

int wake_up_process(struct task_struct *p);
```

你可能会好奇这和 `wake_up` 有什么区别.
`wake_up` 用于唤醒等待队列中满足条件的任务，而 `wake_up_process` 用于唤醒一个特定的任务，并且后者是强制的.
不过，有一个函数提供了上面两个函数的功能: `kthread_run`
这个函数会创建一个内核线程并立即唤醒它:

```
struct task_struct * kthread_run(int (*threadfn)(void *data)
                                 void *data, const char namefmt[], ...);
```

关于内核线程，有一些需要注意的地方:

- 无法访问用户地址空间 (即使使用 `copy_from_user`、`copy_to_user`)，因为内核线程没有用户地址空间
- 无法长时间运行 busy waiting 代码.
  如果内核编译时没有选上抢占 (preemptive)，那么 busy waiting 代码会导致该线程一直占用 CPU 不释放
- 可以调用阻塞操作哦
- 可以使用自旋锁，但如果锁的保持时间很长，建议使用互斥锁

内核线程的终止是自愿的，通过调用 `do_exit()` 来终止线程:

```
fastcall NORET_TYPE void do_exit(long code);
```

大多数内核线程程序都使用相同的框架来实现，这样可以避免一些常见错误的错误.
我们给出一套框架代码:

```
#include <linux/kthread.h>

DECLARE_WAIT_QUEUE_HEAD(wq);

// list events to be processed by kernel thread
struct list_head events_list;
struct spin_lock events_lock;


// structure describing the event to be processed
struct event {
    struct list_head lh;
    bool stop;
    //...
};

struct event* get_next_event(void)
{
    struct event *e;

    spin_lock(&events_lock);
    e = list_first_entry(&events_list, struct event*, lh);
    if (e)
        list_del(&e->lh);
```

```
    spin_unlock(&events_lock);

    return e
}

int my_thread_f(void *data)
{
    struct event *e;

    while (true) {
        wait_event(wq, (e = get_next_event));

        /* Event processing */

        if (e->stop)
            break;
    }

    do_exit(0);
}

/* start and start kthread */
kthread_run(my_thread_f, NULL, "%skthread%d", "my", 0);
```

当我们想给内核线程发送一个事件时，我们可以使用以下代码：

```
void send_event(struct event *ev)
{
    spin_lock(&events_lock);
    list_add(&ev->lh, &events_list);
    spin_unlock(&events_lock);
    wake_up(&wq);
}
```

注: 上面的代码框架中，`get_next_event` 函数会从事件列表中取出一个事件，如果列表为空则会阻塞.
`send_event` 函数会将一个事件插入到事件列表中，并唤醒内核线程.
但这些并不会在实验任务中用到 :)

2.2 atomics

在内核中，我们经常会使用原子操作来保证数据的一致性.
原子操作是一种不可分割的操作，它可以保证在多线程环境下数据的一致性.
在 Linux 内核中，原子操作是通过 `atomic_t` 类型来实现的.
我们可以使用以下函数来操作 `atomic_t` 类型的变量:

```
#include <asm/atomic.h>

void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_cmpxchg(atomic_t *v, int old, int new);
```

2.3 Task

任务:

1. **代码准备**
   在工作目录下执行 `LABS=deferred_work/6-kthread make skels` 获取代码框架.
2. **写代码**
   - 完成 `kthread/kthread.c` 代码中的 TODO
   - 你需要在 `kthread_init` 函数中创建一个内核线程，并完成同步，使 `my_thread_f` 能在模块卸载时被正常结束.
   - 你会用到上一个 task 中的 `wake_up`、`wait_event` 函数，以及用于同步的 `atomic_t` 相关 API
   - 其实这个 task 和上一个 task 差不多，如果遇到问题你可以想想上一个 task 是如何实现的
3. **编译** (工作环境下)
   - 在工作目录下执行 `make build` 编译内核模块
4. **模块加载与卸载** (虚拟环境下)
   - 启动虚拟环境，加载内核模块
   - 卸载内核模块

测试:
如果代码正确，你可以在加载和卸载模块时看到以下输出:

```
root@qemux86:~# insmod skels/deferred_work/6-kthread/kthread.ko
[kthread_init] Init module
[my_thread_f] Current process id is 248 (my_thread_f)
root@qemux86:~# rmmod skels/deferred_work/6-kthread/kthread.ko
[my_thread_f] Exiting
[kthread_exit] Exit module
```

2.4 Implementation

① kthread_init 函数:
这是内核模块的初始化函数，`static` 标记它仅在定义它的文件内可见.
而 `__int` 表示该函数仅用于模块初始化，在初始化完成后，内核会丢弃该函数.

- `flag_stop_thread` 标志指示内核线程是否应该停止，对应等待队列 `wq_stop_thread`

- `flag_thread_terminated` 标志指示内核线程是否已经终止，对应等待队列 `wq_thread_terminated`
- `kthread_run` 函数启动一个内核线程 (命名为 `mykthread0`) 并立即运行 `my_thread_f` 函数

```
static int __init kthread_init(void)
{
    pr_info("[kthread_init] Init module\n");

    /* TODO: Initialize the waitqueues and flags */
    init_waitqueue_head(&wq_stop_thread);
    init_waitqueue_head(&wq_thread_terminated);
    atomic_set(&flag_stop_thread, 0);  // Flag to stop the thread
    atomic_set(&flag_thread_terminated, 0);  // Flag to indicate the thread has
terminated

    /* TODO: Create and start the kernel thread */
    kthread_run(my_thread_f, NULL, "%skthread%d", "my", 0);

    return 0;
}
```

② `my_thread_f` **函数:**
这是内核线程执行的主函数.

- 内核线程 `mykthread0` 通过 `wait_event` 函数阻塞在等待队列 `wq_stop_thread` 中，并等待 `flag_stop_thread == 1` 条件
- 当条件满足时，它被唤醒，并设置 `flag_thread_terminated` 为 `1` 然后调用 `wake_up` 函数唤醒等待队列 `wq_thread_terminated` 中等待在 `flag_thread_terminated` 上的线程.
- 最后打印信息并调用 `do_exit` 函数终止线程.

```
int my_thread_f(void *data)
{
    pr_info("[my_thread_f] Current process id is %d (%s)\n", current->pid,
current->comm);

    /* TODO: Wait for a command to remove the module (i.e., flag_stop_thread to
be set) */
    wait_event(wq_stop_thread, atomic_read(&flag_stop_thread) == 1);

    /* TODO: Set flag to mark kernel thread termination */
    atomic_set(&flag_thread_terminated, 1);

    /* TODO: Notify the unload process that we have exited */
    wake_up(&wq_thread_terminated);

    pr_info("[my_thread_f] Exiting\n");
    do_exit(0);  // Exit the thread
}
```

③ `kthread_exit` **函数:**
这是内核模块的卸载函数.
它被 `__exit` 宏标记，因此当模块被卸载时会自动调用 `kthread_exit` 函数.

- 设置 `flag_stop_thread` 为 `1`，并唤醒等待队列 `wq_stop_thread` 中阻塞的 `mykthread0` 线程.

- 当前 (卸载模块的) 线程调用 `wait_event` 函数阻塞在等待队列 `wq_thread_terminated` 中，并等待 `wq_thread_terminated == 1` 条件，使得 `mykthread0` 线程继续执行.
- 最终 `flag_thread_terminated` 被 `mykthread0` 线程设置为 `1`，卸载模块的进程被唤醒.
  当 `mykthread0` 线程终止后，卸载模块的进程继续执行，打印信息并退出.

```
static void __exit kthread_exit(void)
{
    /* TODO: Notify the kernel thread that it's time to exit */
    atomic_set(&flag_stop_thread, 1);  // Set flag to stop the kernel thread
    wake_up(&wq_stop_thread);

    /* TODO: Wait for the kernel thread to exit */
    wait_event(wq_thread_terminated, atomic_read(&flag_thread_terminated) == 1);

    pr_info("[kthread_exit] Exit module\n");
}
```

测试:
启动虚拟环境，加载内核模块，再卸载内核模块.

```
qemux86 login: root
root@qemux86:~# cd ./skels/deferred_work/6-kthread
root@qemux86:~/skels/deferred_work/6-kthread# insmod kthread.ko
kthread: loading out-of-tree module taints kernel.
[kthread_init] Init module
[my_thread_f] Current process id is 267 (mykthread0)
root@qemux86:~/skels/deferred_work/6-kthread# lsmod kthread
    Tainted: G
kthread 16384 0 - Live 0xe0865000 (O)
root@qemux86:~/skels/deferred_work/6-kthread# rmmod kthread.ko
[my_thread_f] Exiting
[kthread_exit] Exit module
root@qemux86:~/skels/deferred_work/6-kthread#
```

Task 3: IO uring

本实验实现一个简易的轮询模式 (`SQPOLL`) 的 `io_uring`

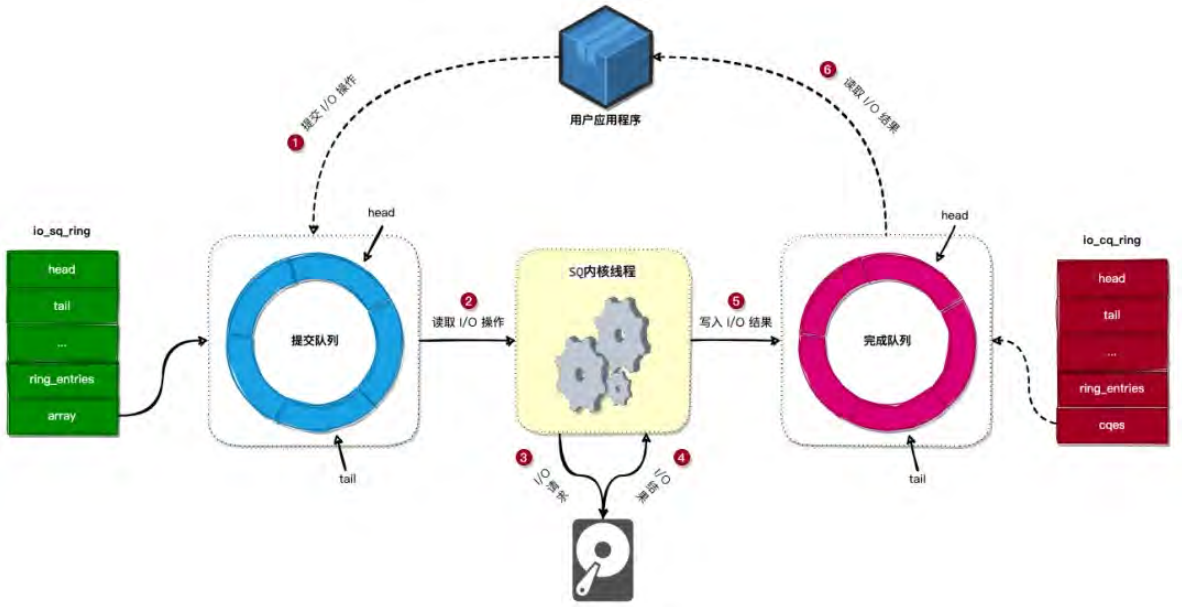
在 Lab 1 中，我们比较了 stream-oriented I/O (如 `read` 和 `write` 系统调用)
与 memory-mapped I/O (即 `mmap` 将文件映射到内存空间) 在文件读写效率上的差异.
我们发现，stream-oriented I/O 操作往往伴随着频繁的用户态与内核态切换，
这种上下文切换以及数据拷贝会显著影响性能.
在高并发场景中 (例如 Web 服务器或分布式系统)，
大量小型 I/O 请求 (如文件读写或网络传输) 更容易导致性能瓶颈.

为了解决这些问题，Linux 引入了 `io_uring`
这是一种高性能的异步 I/O 接口，通过减少用户态与内核态切换以及最小化数据拷贝等机制，显著提升了 I/O 性能.

> 在 Lab 1 中，我们已经通过 memory-mapped I/O 显著减少了用户态与内核态切换的开销，明显加快了文件读写的速度.
> 但在某些具体场景下，memory-mapped I/O 并非最优解.
> 一个 `io_uring` 优于 memory-mapped I/O 的具体案例可参考 (Qdrant under the hood: io_uring)

> 当然也存在 memory-mapped I/O 在性能优于 `io_uring` 的情况，
> 因此讨论性能时离不开具体的应用场景和运行环境.

设计: 阅读 (io_uring 原理)



3.1 实验内容

为了减少频繁的用户态与内核态切换带来的开销，`io_uring` 提供了一种轮询模式 (`SQPOLL`)
其核心工作流程如下:

1. 提交请求: 用户无需通过传统的 `read` 或 `write` 系统调用，而是将 I/O 请求的相关信息
   (如文件描述符、偏移量、目标缓冲区等) 直接写入一个与内核共享的环形缓冲区，称为**提交队列**
   (Submission Queue)
   这一机制有效避免了频繁系统调用的开销.
2. 内核线程轮询:
   在初始化 io_uring 实例时，内核会创建一个专属的内核线程 (kthread)，持续轮询提交队列.
   当检测到新的 I/O 请求时，该线程立即对请求进行处理.
3. 结果读取：请求处理完成后，内核线程将 I/O 操作的结果写入另一个环形缓冲区，即**完成队列**
   (Completion Queue)
   用户程序随后可以从完成队列中读取操作的结果，完成整个 I/O 流程.

> **环形缓冲区** (Ring Buffer) 是一种固定大小的缓冲区数据结构，
> 其逻辑上呈线性结构，但通过首尾相连的设计形成了环形.
> 环形缓冲区因其高效的数据流管理特性被广泛应用，尤其适用于生产者-消费者模型.
> 这种设计能够显著减少数据拷贝以及动态内存分配的开销，提升性能.
>
> 环形缓冲区的核心是一个固定长度的数组，同时维护两个指针: `head` 和 `tail`
> 生产者在 `tail` 指针处写入数据，消费者则从 `head` 指针处读取数据.
> 以下是写入和读取操作的简化伪代码:

```
struct RingBuffer {
 Item buffer[SIZE];
 size_t head = 0; // 初始指针位置
 size_t tail = 0;
};
// 写入数据
void RingBuffer::Write(Item data) {
 buffer[tail] = data;
 tail = (tail + 1) % SIZE; // 指针循环递增
```

```
}
// 读取数据
Item RingBuffer::Read() {
 Item data = buffer[head];
 head = (head + 1) % SIZE; // 指针循环递增
 return data;
}
```

在我们的实验中，用户需要依次向提交队列写入一系列数字 (即学号)
内核线程会逐位读取这些数字，并打印出对应的 `secret[i]` 的内容.
例如，若学号为 21307130094，
内核线程将在 `dmesg` 中依次打印 `secret[2]`、`secret[1]`、`secret[3]`，以此类推.

与 Linux 内核中的 `io_uring` 机制相比，
我们实验中要求实现的 `Baby io_uring` 进行了大量简化，主要体现在以下几点:

- 简化的提交队列格式: 提交队列的内容格式非常简单，仅包含用户希望读取的元素下标.
- 模拟 I/O 请求: 实验中并未实际发起 I/O 请求，而是通过读取 `secret` 来模拟 I/O 操作.
- 省略完成队列: 实验中不涉及创建完成队列，请求的处理结果直接打印到了 `dmesg` 中.

提示:
我们的实验框架通过 `mmap` 创建了大小为 `SHARED_MEM_SIZE` 的共享内存区域，
其中前 4 个字节用作 `head`，第 $4 \sim 8$ 个字节用作上面介绍的 `tail`，
剩余的字节用于提交队列 (内核模块源码中的 `sq_data`)
因此我们需谨慎地确定上面的 `SIZE`.

3.2 实验任务

按照以下步骤完成实验:

- ① 代码准备:
  将代码框架 `baby_io_uring` 和 `baby_io_uring_test` 目录复制到 `skels/lab5` 目录下.
- ② 代码阅读与修改:
  阅读 `baby_io_uring/io_uring.c` 和 `baby_io_uring_test/user.c` 文件源码，
  理解代码框架，并完成 TODO 内容.
- ③ 编译 (工作环境下)
  在 `skels/Kbuild` 目录下添加 `baby_io_uring` 目录 (具体格式参照文件)
  在工作目录下执行 `make build` 编译内核模块
  (和上一个 task 一样，测试程序 `user` 理论上会自动编译，
  如果不会的话就在 `baby_io_uring_test` 目录下手动 `make build` 一下)
- ④ 模块加载与设备节点创建 (虚拟环境下)
  启动虚拟环境，加载内核模块.
  运行 `user`.
- ⑤ 验证答案 (虚拟环境下)
  查看 `dmesg` 中的内容，我们的 `secret` 是 `"HelloFudan"`

完成 `TODO` 内容:

- ① 初始化函数 `shm_device_init` 中:
  - 使用 `kzalloc` 函数分配共享内存:

```
// TODO BEGIN: Allocate shared memory with kzalloc.
shared_memory = kzalloc(SHARED_MEM_SIZE, GFP_KERNEL);
// TODO END: Allocate shared memory with kzalloc.
```

`kzalloc` 函数类似于 `kmalloc`，但在分配内存后会将其初始化为零.

- 将共享内存的前 4 字节用于存储 `ring_buffer_head` 指针，
  接下来的 4 字节存储 `ring_buffer_tail` 指针，
  其余空间用于存储提交队列的数据 (`sq_data`):

```
// TODO BEGIN: Initialize ring_buffer_head, ring_buffer_tail and
sq_data.
ring_buffer_head = (int *)shared_memory;          // First 4 bytes
ring_buffer_tail = (int *)(shared_memory + 4);    // Next 4 bytes
sq_data = (char *)(shared_memory + 8);            // Remaining bytes
// TODO END: Initialize ring_buffer_head, ring_buffer_tail and sq_data.
```

- ② 查询提交队列的内核进程运行的函数 `worker` 中:
  - 内核线程读取 `ring_buffer_head` 和 `ring_buffer_tail`，用于检查提交队列的状态:

```
// TODO BEGIN: Initiazlie head and tail.
int head = *ring_buffer_head;
int tail = *ring_buffer_tail;
// TODO END: Initiazlie head and tail.
```

  - 若 `head` 等于 `tail`，则说明缓冲区中没有未处理的请求，此时 `worker` 进程休眠一段时间.
    若 `head` 不等于 `tail`，则说明缓冲区中有未处理的请求.
    从 `sq_data` 中取出 `head` 指向的值，将其作为索引访问 `secret` 数组，并打印对应字符:
    然后更新 `ring_buffer_head`，并使用模运算保持其循环特性:

```
if (head != tail) {
    // TODO BEGIN: Read from SQ and process the request (print secret[i]
to dmesg).
    int index = sq_data[head];   // Read index from submission queue
    if (index >= 0 && index < sizeof(secret)){
        printk(KERN_INFO "shm_device: %c\n", secret[index]);
    }
    // TODO END: Read from SQ and process the request (print secret[i]
to dmesg).

    // TODO BEGIN: Update ring_buffer_head since we have processed the
request.
    *ring_buffer_head = (head + 1) % (SHARED_MEM_SIZE - 8);  // Circular
increment
    // TODO END: Update ring_buffer_head since we have processed the
request.

} else {
    msleep(100);  // Sleep if no requests
}
```

- ③ 用户程序 `user.c` 的 `main()` 函数中:
  遍历字符数组 `id` 的每个元素，将其转换为整数值后写入提交队列 `sq_data`，
  同时更新环形缓冲区的 `ring_buffer_tail` 指针，并使用模运算保持其循环特性:

```
for (int i = 0; i < len; i++) {
    // TODO BEGIN: Write the request (id[i] - '0') to sq_data (Submission
Queue, SQ).
    sq_data[*sq_tail] = id[i] - '0';  // Write index to submission queue
    *sq_tail = (*sq_tail + 1) % (SHARED_MEM_SIZE - 8);  // Circular
increment
    // TODO END: Write the request (id[i] - '0') to sq_data (Submission
Queue, SQ).
}
```

运行结果:

```
qemux86 login: root
root@qemux86:~# cd skels/lab5/baby_io_uring/
root@qemux86:~/skels/lab5/baby_io_uring# insmod io_uring.ko
shm_device: Device registered with major number 252
root@qemux86:~/skels/lab5/baby_io_uring# lsmod io_uring
    Tainted: G
io_uring 16384 0 - Live 0xe0839000 (O)
root@qemux86:~/skels/lab5/baby_io_uring# cd ../
root@qemux86:~/skels/lab5# cd baby_io_uring_test/
root@qemux86:~/skels/lab5/baby_io_uring_test# ./user
shm_device: Device opened
shm_device: mmap successful
shm_device: Device closed
shm_device: l
shm_device: e
shm_device: l
shm_device: H
shm_device: d
shm_device: e
shm_device: o
shm_device: H
shm_device: H
shm_device: F
shm_device: e
root@qemux86:~/skels/lab5/baby_io_uring_test#
```

The End