Introduction 介绍

Up until now, we have only been working with **blocking I/O**. When a user requests data, the call to `read()`, `write()`, or `ioctl()` blocks until the I/O operation completes. For example, reading from a network socket may block while the kernel waits for data to arrive. From the program's perspective, execution is paused until the operation finishes.

到目前为止，我们只处理**阻塞 I/O**。当用户请求数据时，对 `read ()` 、 `write ()` 或 `ioctl ()` 的调用将阻塞，直到 I/O 作完成。例如，在内核等待数据到达时，从网络套接字读取可能会阻塞。从程序的角度来看，执行将暂停，直到作完成。

In practice, processes often have other work they could do while waiting. To support this, we can use **non-blocking I/O**. When a file descriptor is opened with the `O_NONBLOCK` flag, I/O calls return immediately if the device is not ready, with an appropriate error code (e.g., `EAGAIN` for reads or `EBUSY` for writes). This allows programs to continue executing and later check whether the result is ready, or use event notification mechanisms such as `select()`, `poll()`, or `kqueue()` to be informed when I/O can proceed.

在实践中，流程通常在等待期间可以做其他工作。为了支持这一点，我们可以使用**非阻塞 I/O**。当使用 `O_NONBLOCK` 标志打开文件描述符时，如果设备未准备就绪，I/O 调用会立即返回，并带有适当的错误代码（例如，用于读取的 `EAGAIN` 或用于写入的 `EBUSY`）。这允许程序继续执行并稍后检查结果是否准备就绪，或者使用事件通知机制（例如 `select ()` 、 `poll ()` 或 `kqueue ()` 在 I/O 可以继续时收到通知。

For this prac, you will:

对于此练习，您将：

- Move the `ioctl()` you wrote in prac 6 and split it into separate `read` and `write` calls.

  移动您在实践 6 中编写的 `ioctl ()` 并将其拆分为单独的`读` `写`调用。
- Implement **non-blocking behavior** for these calls.

  为这些调用实现**非阻塞行为** 。
- Add an **event system** so that a user program can poll the driver to check if a particular event is ready.

  添加**事件系统** ，以便用户程序可以轮询驱动程序以检查特定事件是否准备就绪。

Boilerplate 样板

For this prac, instead of working from scratch or using a skeleton patch, you will refactor your solution from prac 6.

对于此练习，您将从实践 6 重构解决方案，而不是从头开始工作或使用骨架补丁。

If you're happy with your prac 6 solution, you can complete this prac using it:

如果您对实践 6 解决方案感到满意，您可以使用它来完成此实践：

```
$ git checkout -b p7 p6
```

Otherwise, if you prefer to use our example solution, check out a fresh branch and apply the p7 base patch, which contains the provided solution for prac 6:

否则，如果您更喜欢使用我们的示例解决方案，请签出一个新分支并应用 p7 基础补丁，其中包含为 prac 6 提供的解决方案：

```
$ git checkout -b p6 openbsd-7.7
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/p7-base-patch.patch
$ git am < p7-base-patch.patch && rm p7-base-patch.patch
```

Now that you have a starting point, let's begin refactoring the driver.

现在你已经有了起点，让我们开始重构驱动程序。

Adding Read and Write Entry Points

添加读写入口点

For this prac, instead of using the `ioctl` we implemented in Prac 6, we will transfer the functionality into the read and write entry points of our driver. Currently, your ioctl should sleep on the ready state of the driver before creating a request for the device to compute the stats, then wait again until it is done. Since we would like processes to opt out of blocking on the driver state, we need to split this functionality into separate `read()` and `write()` calls.

对于此练习，我们将不使用我们在 Prac 6 中实现的 `ioctl`，而是将功能转移到驱动程序的读写入口点中。目前，在为设备创建计算统计信息的请求之前，ioctl 应处于驱动程序的就绪状态，然后再次等待，直到完成。由于我们希望进程选择退出对驱动程序状态的阻塞，因此我们需要将此功能拆分为单独的 `read ()` 和 `write ()` 调用。

Adding new entry points 添加新的入口点

The first step is to update the driver to perform the same tasks as before, but with the functionality split between read and write. However, right now the device will return `ENODEV` (operation not supported for device) if we attempt to read or write, even if we implement `p6statsread()` and `p6statswrite()`.

第一步是更新驱动程序以执行与以前相同的任务，但功能在读取和写入之间拆分。但是，现在，如果我们尝试读取或写入，即使我们实现了 `p6statsread ()` 和 `p6statswrite ()，` 设备也会返回 `ONEDEV` （设备不支持作)。

To fix this, we need to tell the kernel that these functions exist by updating our character device switch table (cdevsw).

为了解决这个问题，我们需要通过更新我们的字符设备切换表 （cdevsw） 来告诉内核这些函数的存在。

From Prac 5, you may recall defining the p5 device in `<sys/conf.h>`. The Prac 6 and Prac 7 base patches already define `p6stats`, but if you check the current definition you'll see it only sets up open, close, and ioctl in `/usr/src/sys/sys/conf.h`:

在实践 5 中，您可能还记得在 `<sys/conf.h>` 中定义了 p5 设备。Prac 6 和 Prac 7 基础补丁已经定义了 `p6stats`，但如果您检查当前的定义，您会发现它仅在 `/usr/src/sys/sys/conf.h` 中设置了 open、close 和 ioctl：

```
/* open, close, ioctl */
#define cdev_p6stats_init(c,n) { \
    dev_init(c,n,open), dev_init(c,n,close), (dev_type_read((*))) enodev, \
    (dev_type_write((*))) enodev, dev_init(c,n,ioctl), \
    (dev_type_stop((*))) enodev, 0, \
    (dev_type_mmap((*))) enodev }
#endif
```

Let's update this to include `read`, `write`, and `kqfilter`. (We'll implement `kqfilter` later.) For now, keep your ioctl implementation — when refactoring, it's best to change things in small chunks to reduce errors.

让我们更新它以包括 `read`、`write` 和 `kqfilter`。（我们稍后会实现 `kqfilter`。现在，保留你的 ioctl 实现——重构时，最好以小块的形式更改内容以减少错误。

Looking at the `ujoy` driver (a joystick driver), we can see an example that implements all the needed entry points:

查看 `ujoy` 驱动程序（一个纵杆驱动程序），我们可以看到一个实现所有需要的入口点的示例：

```
/* open, close, read, write, ioctl, kqfilter */
#define cdev_ujoy_init(c,n) { \
    dev_init(c,n,open), dev_init(c,uhid,close), dev_init(c,uhid,read), \
    dev_init(c,uhid,write), dev_init(c,ujoy,ioctl), \
    (dev_type_stop((*))) enodev, 0, \
    (dev_type_mmap((*))) enodev, 0, 0, dev_init(c,uhid,kqfilter) }
#endif
```

This should leave us with:

这应该给我们留下：

```
/* open, close, read, write, ioctl, kqfilter */
#define cdev_p6stats_init(c,n) { \
    dev_init(c,n,open), dev_init(c,n,close), dev_init(c,n,read), \
    dev_init(c,n,write), dev_init(c,n,ioctl), \
    (dev_type_stop((*))) enodev, 0, \
    (dev_type_mmap((*))) enodev, 0, 0, dev_init(c,n,kqfilter) }
#endif
```

Here we simply use n in place of `uhid` / `ujoy`, meaning our entry points will be named `p6statsxxx` as usual.

在这里，我们简单地使用 n 代替 `uhid` / `ujoy`，这意味着我们的入口点将像往常一样命名为 `p6statsxxx`。

Adding function stubs 添加函数存根

Now that we've told autoconf to expect these functions, we can add stubs to our driver. You may recall the character device switch table (`cdevsw`) from Applied Class 5 or Prac 5:

现在我们已经告诉 autoconf 期待这些函数，我们可以向驱动程序添加存根。您可能还记得应用类 5 或实践 5 中的字符设备开关表 （`cdevsw`）:

```
/*
 * Character device switch table
 */
struct cdevsw {
    int     (*d_open)(dev_t dev, int oflags, int devtype,
                     struct proc *p);
    int     (*d_close)(dev_t dev, int fflag, int devtype,
                     struct proc *);
    int     (*d_read)(dev_t dev, struct uio *uio, int ioflag);
    int     (*d_write)(dev_t dev, struct uio *uio, int ioflag);
    int     (*d_ioctl)(dev_t dev, u_long cmd, caddr_t data,
                     int fflag, struct proc *p);
    int     (*d_stop)(struct tty *tp, int rw);
    struct tty *
        (*d_tty)(dev_t dev);
    paddr_t    (*d_mmap)(dev_t, off_t, int);
    u_int    d_type;
    u_int    d_flags;
    int     (*d_kqfilter)(dev_t dev, struct knote *kn);
};
```

We want to add stubs for read, write, and kqfilter so the driver compiles. Add them anywhere in your `p6stats.c` file:

我们想添加用于读取、写入和 kqfilter 的存根，以便驱动程序编译。将它们添加到 `p6stats.c` 文件中的任何位置:

```
int
p6statswrite(dev_t dev, struct uio *uio, int flags)
{
    return (ENODEV);
}

int
p6statsread(dev_t dev, struct uio *uio, int flags)
{
    return (ENODEV);
}

int
p6statskqfilter(dev_t dev, struct knote *kn)
{
    return (ENODEV);
}
```

Recompiling the kernel 重新编译内核

Now let's recompile the kernel to verify everything builds. Since we updated `conf.h`, we must reconfigure `GENERIC.MP` before compiling. Use the full process you should already be familiar with:

现在让我们重新编译内核以验证所有构建。由于我们更新了 `conf.h`，因此我们必须重新配置 `GENERIC。MP` 在编译之前。使用您应该已经熟悉的完整流程：

```
$ cd /usr/src/sys/arch/amd64/conf
$ config GENERIC.MP
$ cd ../compile/GENERIC.MP
$ make -j4
$ doas make install
$ doas reboot
```

[Ex1] Implementing Read and Write

[例1]实现读写

Now that we have added our function stubs, it's time to start implementing them. We want to split the functionality between the read and write entry points of our driver.

现在我们已经添加了函数存根，是时候开始实现它们了。我们希望在驱动程序的读取和写入入口点之间拆分功能。

For the **write** call, userland will provide a buffer containing an array of `uint64_t`. The driver begins by validating this input: the buffer size must be a multiple of `uint64_t`, and the total number of inputs must not exceed `P6_R_ICOUNT_MAX`. If the device is currently busy, the process should sleep until the device signals readiness via some ready state.

对于**写入**调用，userland 将提供一个包含 `uint64_t` 数组的缓冲区。驱动程序首先验证此输入：缓冲区大小必须是 `uint64_t` 的倍数，并且输入总数不得超过 `P6_R_ICOUNT_MAX`。如果设备当前处于繁忙状态，则进程应处于睡眠状态，直到设备通过某个就绪状态发出就绪信号。

Once the device is available, the driver sets up DMA. First, the userland buffer is copied into memory that can be accessed by DMA, creating the input DMA map. Then, a corresponding output DMA buffer is allocated and mapped to hold the results, which will be returned in a `struct p6stats_output`. Both DMA maps must be synchronized to maintain cache coherency before the transfer begins.

设备可用后，驱动程序将设置 DMA。首先，将用户空间缓冲区复制到可由 DMA 访问的内存中，从而创建输入 DMA 映射。然后，分配并映射相应的输出 DMA 缓冲区以保存结果，这些结果将在结构 `p6stats_output` 中返回。在传输开始之前，必须同步两个 DMA 映射以保持缓存一致性。

After preparing memory, the write call should finish by programming the relevant registers in the device and marking the driver as busy so the computation can proceed.

准备好内存后，写入调用应通过对设备中的相关寄存器进行编程并将驱动程序标记为繁忙来完成，以便计算可以继续进行。

Error handling in the write path must cover several cases (including but not limited to):

写入路径中的错误处理必须涵盖多种情况（包括但不限于）：

- `EINVAL` – the input buffer is misaligned or of an invalid size

  `EINVAL` – 输入缓冲区未对齐或大小无效
- `ENOMEM` – too many inputs were requested

  `ENOMEM` – 请求的输入过多
- `EIO` – DMA allocation or mapping fails

  `EIO` – DMA 分配或映射失败

The **read** call works in the opposite direction. Userland provides a buffer large enough to hold a `struct p6stats_output`:

读取调用以相反的方向工作。Userland 提供了一个足够大的缓冲区来容纳`结构 p6stats_output`：

```
struct p6stats_output {
    uint64_t po_count;
    uint64_t po_sum;
    uint64_t po_mean;
    uint64_t po_median;
    uint8_t  po_rsvd[8];
};
```

The read call should begin by retrieving the softc. Assuming the state is valid, it must then synchronize the output DMA map so the results are visible to the CPU. The driver copies the results into the buffer provided by userland so that the user receives their output. After the transfer, the DMA maps are unloaded and cleaned up, and the driver resets its internal state to READY, allowing new requests to be submitted.

读取调用应从检索 softc 开始。假设状态有效，则必须同步输出 DMA 映射，以便 CPU 可以看到结果。驱动程序将结果复制到 userland 提供的缓冲区中，以便用户接收其输出。传输后，DMA 映射将卸载和清理，驱动程序将其内部状态重置为 READY，从而允许提交新请求。

Error handling in the read path must cover several cases (including but not limited to):

读取路径中的错误处理必须涵盖多种情况（包括但不限于）：

- `ENOBUFS` – the provided buffer is insufficient

  `ENOBUFS` – 提供的缓冲区不足
- `ENOMSG` – there is nothing to read

  `ENOMSG` – 没有什么可读的
- `EIO` – DMA-related failures

  `EIO` – 与 DMA 相关的故障

Hints 提示

Think carefully about where your DMA buffers should be allocated and what information the driver must keep track of. The `bus_dma(9)` family of functions will be key to setting up, synchronizing, and tearing down mappings.

请仔细考虑应在何处分配 DMA 缓冲区，以及驱动程序必须跟踪哪些信息。`bus_dma（9）` 系列函数将是设置、同步和拆除映射的关键。

Be prepared to make small changes elsewhere in the driver to support these features. Refactoring often requires adding bits of code in multiple places to ensure functionality stays consistent.

准备好在驱动程序中的其他位置进行小的更改以支持这些功能。重构通常需要在多个位置添加代码位，以确保功能保持一致。

Finally, avoid creating your own `uios` unless you are directly passing the user's `uio` structure into your functions. These represent input given by userland, and we generally don't construct them ourselves.

最后，避免创建自己的 `uio，` 除非直接将用户的 `uio` 结构传递到函数中。这些代表用户空间给出的输入，我们通常不会自己构建它们。

With this structure in place, the next step will be updating the `p6stats` test program from the previous practical so that it can exercise the kernel driver.

有了这个结构，下一步将是从之前的实践中更新 `p6stats` 测试程序，以便它可以运行内核驱动程序。

[Ex2] Userland Test Program

[例2]用户区测试计划

Now that you have implemented the read and write entry points for your `p6stats` driver, the next step is to verify that it behaves as expected. To do this, you need a userland program that can exercise the driver's functionality.

现在您已经实现了 `p6stats` 驱动程序的读写入口点，下一步是验证它的行为是否按预期运行。为此，您需要一个可以行使驱动程序功能的用户空间程序。

The idea is to extend the `p6stats` program you wrote back in prac 6 so that it can interact directly with your driver. The program should be able to run in read-only mode, write-only mode, or in the default read–write mode when no explicit flags are given.

这个想法是扩展您在实践 6 中编写的 `p6stats` 程序，以便它可以直接与您的驱动程序交互。程序应该能够在只读模式、只写模式或默认读写模式下运行，如果没有给出显式标志。

To start, the program will need to open the `/dev/p6stats` device file. Command-line flags will control how the program operates: `-r` for when you want to read, `-w` for when you want to write, and `-rw` for when you want both. When in write mode, the program should accept a list of numbers as command-line arguments, up to a maximum number of values.

首先，程序需要打开 `/dev/p6stats` 设备文件。命令行标志将控制程序的运行方式：`-r` 表示要读取时，`-w` 表示要写入时，- `rw` 表示两者都想要时。在写入模式下，程序应接受数字列表作为命令行参数，最多值数。

These numbers must be passed to the driver as an array of `uint64_t`. In read mode, the program should fetch results from the driver and display them in a clear format, showing the count, sum, mean, and median.

这些数字必须作为 `uint64_t` 数组传递给驱动程序。在读取模式下，程序应从驱动程序获取结果并以清晰的格式显示它们，显示计数、总和、平均值和中位数。

The test program also needs to handle errors gracefully. For example, it should report when write mode is requested but no numbers are provided, or when a general I/O error occurs. Providing meaningful messages will make it easier to debug both your program and your driver.

测试程序还需要优雅地处理错误。例如，当请求写入模式但未提供数字时，或者发生常规 I/O 错误时，它应该报告。提供有意义的消息将使调试程序和驱动程序变得更加容易。

To make the program usable, format the output in a straightforward way so the statistical results are obvious to whoever is running the test. For example, the expected usage could look like this:

要使程序可用，请以简单的方式格式化输出，以便运行测试的人都能清楚地看到统计结果。例如，预期的用法可能如下所示：

```
./p6stats 10 20 30 40        # defaults to read + write
./p6stats -r                 # read only
./p6stats -w 5 15 25         # write only
./p6stats -rw 1 2 3 4 5      # explicit read + write
```

Hints 提示

Use `getopt()` for command-line parsing to handle the `-r` and `-w` flags cleanly.

使用 `getopt ()` 进行命令行解析，以干净地处理 `-r` 和 `-w` 标志。

Choose the appropriate flags to `open()` (`O_RDONLY`, `O_WRONLY`, or `O_RDWR`) depending on the requested mode.

根据请求的模式选择适当的标志来 `打开 ()` （`O_RDONLY`、`O_WRONLY` 或 `O_RDWR`）。

When writing, make sure to send your data as an array of `uint64_t`.

写入时，请确保将数据作为 `uint64_t` 数组发送。

When reading, allocate a `struct p6stats_output` and fill it with the data returned from the driver before printing the results.

读取时，在打印结果之前分配结构 `p6stats_output` 并使用驱动程序返回的数据填充它。

Solution 溶液

Here is our solution for the userland program:

以下是我们针对用户土地计划的解决方案：

▼ 扩大

⚠ You will need to move `P6_R_ICOUNT_MAX` into your `p6statsvar.h` file and run `doas make includes` in `/usr/src/includes` for the program to compile properly.

您需要将 `P6_R_ICOUNT_MAX` 移动到 您的 `p6statsvar.h` 文件中，并在 `/usr/src/includes` 中运行 `doas make includes`，以便程序正确编译。

And for the kernel driver

对于内核驱动程序

见相关代码文件Userland_Test_Program1.c和Userland_Test_Program2.c

[Ex 3] Non-Blocking IO [例 3]非阻塞 IO

Now that the driver has been refactored to handle write requests and read them back, the next step is to add support for non-blocking operation. From the user's perspective, this is enabled by specifying the `O_NONBLOCK` flag when opening the file descriptor:

现在驱动程序已被重构以处理写入请求并读回它们，下一步是添加对非阻塞作的支持。从用户的角度来看，这是通过在打开文件描述符时指定 `O_NONBLOCK` 标志来启用的：

```
int fd = open("/dev/p6stats", O_RDWR | O_NONBLOCK);

int fd = open (“/dev/p6stats”,  O_RDWR |O_NONBLOCK) ;
```

Once the file descriptor is opened with this flag, all subsequent `read()` and `write()` calls on that descriptor will automatically use non-blocking semantics. Inside the kernel, this flag is passed to the driver's entry points (`d_read` and `d_write`) as `IO_NDELAY` through the `ioflag` parameter. Your driver must check for this flag and decide whether to block or return immediately when the device is not ready.

一旦使用此标志打开文件描述符，该描述符上的所有后续 `read ()` 和 `write ()` 调用都将自动使用非阻塞语义。在内核内部，此标志通过 `ioflag` 参数 `IO_NDELAY` 传递到驱动程序的入口点（`d_read` 和 `d_write`）。驱动程序必须检查此标志，并决定在设备未准备就绪时是阻止还是立即返回。

The benefit of this approach is that user programs can continue executing while the kernel manages I/O in the background. Instead of waiting for the device, the program can use internal buffers and retry later when the device is ready.

这种方法的好处是，当内核在后台管理 I/O 时，用户程序可以继续执行。程序可以使用内部缓冲区，并在设备准备就绪后重试，而不是等待设备。

For the **non-blocking write** path, if the `IO_NDELAY` flag is set, the driver must not sleep waiting on the ready state. If the device is busy, the call should return immediately with `EBUSY`.

对于**非阻塞写入**路径，如果设置了 `IO_NDELAY` 标志，则驱动程序不得在就绪状态下休眠等待。如果设备繁忙，呼叫应立即返回 `EBUSY`。

For the **non-blocking read** path, if the `IO_NDELAY` flag is set, the driver must not wait for a result to become available. If no computation has been completed or no data has been written, the call should return immediately with `EAGAIN`.

对于**非阻塞读取**路径，如果设置了 `IO_NDELAY` 标志，则驱动程序不得等待结果可用。如果尚未完成计算或未写入任何数据，则调用应立即返回 `EAGAIN。`

It is acceptable for the driver to acquire and briefly wait on a mutex in order to protect shared data structures, but it must not block indefinitely when non-blocking mode has been requested.

驱动程序可以获取互斥体并短暂等待互斥锁以保护共享数据结构，但在请求非阻塞模式时不得无限期地阻止。

Testing 测试

To test this functionality, extend your user program from the previous exercise with an additional `-n` flag. When this flag is provided, the program should open the device file using `O_NONBLOCK` so that reads and writes are performed in non-blocking mode.

要测试此功能，请使用额外的 `-n` 标志从上一个练习中扩展您的用户程序。提供此标志时，程序应使用 `O_NONBLOCK` 打开设备文件，以便在非阻塞模式下执行读取和写入。

▼ 扩大

Our solution for the user program added:

我们的用户程序解决方案添加了：

```
...
switch (opt) {
            ...
            case 'n': nonblock = 1; break;
            default: usage(argv[0]);
      }
...

if (nonblock)
    flags |= O_NONBLOCK;

fd = open("/dev/p6stats", flags);
```

Our solution involved the following modifications in write:

我们的解决方案涉及以下写入修改：

```
    if (ISSET(flags, IO_NDELAY)) {
        if (sc->sc_state != P6_S_DONE) {
            error = EBUSY;
            goto unlock;
        }
    } else {
        /* Wait for our result */
        while (sc->sc_state != P6_S_DONE) {
            error = msleep_nsec(&sc->sc_state, &sc->sc_mtx,
                PRIBIO|PCATCH, "p6wait", INFSLP);
            if (error!= 0)
                goto unlock;
        }
    }
```

and the following modifications in read:

以及以下修改：

```
if (ISSET(flags, IO_NDELAY)) {
    if (sc->sc_state != P6_S_READY) {
        error = EAGAIN;
        goto unlock;
    }
}
else {
    /* Block until we have ready state */
    while (sc->sc_state != P6_S_READY) {
        error = msleep_nsec(&sc->sc_state, &sc->sc_mtx,
            PRIBIO|PCATCH, "p6rdy", INFSLP);
        if (error != 0) {
            goto unlock;
        }
    }
}
```

Kqfilter and Events Kqfilter 和事件

Now that we have implemented non-blocking I/O, we want a way for user programs to efficiently poll the state of our driver without performing a read or write, and to support event-driven I/O. In OpenBSD, this can be achieved using kqueue filters. Kqueue filters allow user processes to check whether a device is ready for reading or writing without blocking, making them especially useful when handling multiple file descriptors or devices simultaneously. By implementing kqueue filters in your driver, you can provide non-blocking, event-driven access to the device.

现在我们已经实现了非阻塞 I/O，我们希望用户程序能够有效地轮询驱动程序的状态，而无需执行读取或写入，并支持事件驱动的 I/O。在 OpenBSD 中，这可以使用 kqueue 过滤器来实现。Kqueue 过滤器允许用户进程检查设备是否准备好读取或写入而不会阻塞，这使得它们在同时处理多个文件描述符或设备时特别有用。通过在驱动程序中实现 kqueue 筛选器，可以提供对设备的非阻塞、事件驱动的访问。

Key Data Structures 关键数据结构

Implementing a kqueue filter in a kernel driver relies on three core structures: `struct knote`, `struct klist`, and `struct filterops`.

在内核驱动程序中实现 kqueue 过滤器依赖于三个核心结构：`struct knote`、`struct klist` 和 `struct filterops`。

`struct knote` 结构 `knote`

Each `knote` represents a single event attached by a user process. When a process calls `kqueue()` and then `kevent()` to monitor a file descriptor, the kernel creates a `knote` to track the event.

每个 `knote` 代表用户进程附加的单个事件。当进程调用 `kqueue ()` 然后调用 `kevent ()` 来监视文件描述符时，内核会创建一个 `knote` 来跟踪事件。

Important Fields in `struct knote`:

`struct knote` 中的重要字段：

Important Fields in `struct knote`:

`struct knote` 中的重要字段:

- `kn_filter` – The filter type, e.g., `EVFILT_READ` or `EVFILT_WRITE`. Determines whether the `knote` tracks read or write events.

  `kn_filter` – 过滤器类型，例如 `EVFILT_READ` 或 `EVFILT_WRITE`。确定 `knote` 是跟踪读取还是写入事件。
- `kn_flags` – Flags controlling `knote` behavior, e.g., `EV_EOF` for end-of-file, `EV_ONESHOT` for a one-time event.

  `kn_flags` – 控制 `knote` 行为的标志，例如，`EV_EOF` 文件结束，`EV_ONESHOT` 一次性事件。
- `kn_hook` – Pointer to device-specific data, usually the driver's `softc` structure.

  `kn_hook` – 指向特定于设备的数据的指针，通常是驱动程序的 `软结构` 。
- `kn_fop` – Pointer to the filter operations (`struct filterops`) for this `knote`.

  `kn_fop` – 指向此 `knote` 的过滤作（`struct filterops`）的指针。
- `kn_data` – Set by the driver; indicates bytes available to read or space available to write.

  `kn_data` – 由司机设置;表示可供读取的字节数或可供写入的空间。
- `kn_status` – Internal driver or `knote` status, used during event processing.

  `kn_status` – 内部驱动程序或 `knote` 状态，在事件处理期间使用。

Common Event Filters and Their Meaning:

常见事件过滤器及其含义:

- `EVFILT_READ` – The device has data available to read. `kn_data` indicates how many bytes can be read without blocking.

  `EVFILT_READ` – 设备具有可供读取的数据。`kn_data` 表示可以在不阻塞的情况下读取多少字节。
- `EVFILT_WRITE` – The device can accept data for writing. `kn_data` shows how many bytes can be written without blocking.

  `EVFILT_WRITE` – 设备可以接受数据写入。`kn_data` 显示可以在不阻塞的情况下写入多少字节。

Hint Parameter: 提示参数:

The `hint` is an integer passed to `knote(&sc->sc_klist, hint)` by the driver to indicate what kind of event occurred.

`提示`是驱动程序传递给 `knote（&sc->sc_klist， hint）` 的整数，用于指示发生的事件类型。

- `f_event()` receives this `hint` and determines whether the `knote` should be marked ready.

  `f_event（）` 接收此`提示`并确定是否应将 `knote` 标记为就绪。
- Drivers can define their own hints, for example:

  驱动程序可以定义自己的提示，例如:
  - `hint = 1` → new data available to read

    `提示 = 1` → 个可供读取的新数据
  - `hint = 2` → space available to write

    `hint = 2` → 可用写入空间

This allows the kernel to trigger only the relevant `knotes` without inspecting the device state.

这允许内核仅触发相关的 `knotes`，而无需检查设备状态。

`struct klist` `结构 klist`

Each device that supports kqueue maintains a `klist`:

每个支持 kqueue 的设备都维护一个 `klist`:

```
struct klist sc_klist;
```

This contains all `knotes` attached to the device. The klist provides functions for managing and notifying `knotes`:

这包含连接到设备的所有 音符 。klist 提供了管理和通知 `knote` 的功能:

- `klist_init(&sc->sc_klist)` – Initialize the list of `knotes`.

  `klist_init（&sc->sc_klist）` – 初始化 注释 列表。
- `knot(&sc->sc_klist, hint)` – Notify all `knotes` that an event has occurred.

  `knot（&sc->sc_klist， hint）` – 通知所有 `knote` 事件已发生。
- `knote_detach(&sc->sc_klist, kn)` – Remove a `knote` from the list.

  `knote_detach(&sc->sc_klist, kn)` – 从列表中 删除注释。

The `klist` allows the device to broadcast readiness events to all waiting processes efficiently.

`klist` 允许设备有效地将就绪事件广播到所有等待进程。

`struct filterops`

结构过滤器作

The `filterops` structure defines how the driver interacts with `knotes`:

`filterops` 结构定义了驱动程序如何与 `knotes` 交互:

```
struct filterops {
    int     f_flags;
    void    (*f_attach)(struct knote *kn);
    void    (*f_detach)(struct knote *kn);
    int     (*f_event)(struct knote *kn, long hint);
    void    (*f_process)(struct kevent *kev, struct knote kn*);
    void    (*f_modify)(struct knote *kn, long hint);
};
```

Operations in `struct filterops`:

`struct filterops` 中的作:

- `f_attach` – Called when a `knote` is added to the device.

  `f_attach` – 将 `knote` 添加到设备时调用。
- `f_detach` – Called when a `knote` is removed or closed.

  `f_detach` – 当 注释 被移除或关闭时调用。
- `f_event` – Determines whether the event is ready (readable or writable).

  `f_event` – 确定事件是否准备就绪（可读或可写）。
- `f_process` – Called when a userland program retrieves the event; often wraps `knote_process(kn, kdev)`.

  `f_process` – 当用户空间程序检索事件时调用;经常包装 `knote_process（kn， kdev）。`
- `f_modify` – Optional: updates filter parameters while attached.

  `f_modify` – 可选：在附加时更新筛选器参数。
- `f_flags`: `f_flags`:
  - `FILTEROP_ISFD` – the `knote` represents a file descriptor.

    `FILTEROP_ISFD` – `knote` 表示文件描述符。
  - `FILTEROP_MPSAFE` – Safe to use this operation without the kernel lock

    `FILTEROP_MPSAFE` – 无需内核锁即可安全使用此作

Any operations not needed for a particular driver can remain `NULL`.

特定驱动程序不需要的任何作都可以保持 `NULL`。

Flow of operations: 作流程:

- User calls `kqueue()` → kernel creates a `knote`.

  用户调用 `kqueue（）` →内核创建一个 `knote`。
- User calls `kevent()` → `knote` is attached to the device's `klist`.

  用户调用 `kevent（）` → `knote` 附加到设备的 `klist`。
- Driver signals an event (read/write ready) → calls `knote(&sc->sc_klist, hint)`.

  驱动程序发出事件信号（读/写就绪），→调用 `knote（&sc->sc_klist， hint）。`
- Kernel delivers events to all waiting processes.

  内核将事件传递到所有等待进程。

By combining `knote`, `klist`, and `filterops`, your driver can efficiently notify userland of I/O readiness while supporting both blocking and non-blocking modes.

通过结合 `knote`、 `klist` 和 `filterops`，您的驱动程序可以有效地通知用户 I/O 准备就绪，同时支持阻塞和非阻塞模式。

Next, we will look at a basic skeleton for the p6stats driver, including how to define `kqfilter` and `filterops` for your device.

接下来，我们将了解 p6stats 驱动程序的基本骨架，包括如何为您的设备定义 `kqfilter` 和 `filterops`。

[Ex 4] Implementing Kqfilter

[例 4]实现 Kqfilter

Now that your driver supports non-blocking reads and writes, we need a way for user processes to check if the device is ready before making a call. To do this, we will implement the `kqfilter` system call, which allows user processes to query the device state and support event-driven I/O.

现在，驱动程序支持非阻塞读取和写入，我们需要一种方法，让用户进程在进行调用之前检查设备是否已准备就绪。为此，我们将实现 `kqfilter` 系统调用，它允许用户进程查询设备状态并支持事件驱动的 I/O。

We will define the following events:

我们将定义以下事件:

Event	Reason
W	Ready to receive a completion from the device
R	Ready to make a new request to the device

Kqueue filters let user programs efficiently poll the state of your device without performing a read or write, and enable event-driven designs. Each *knote* represents a single event attached by a user process, and the kernel delivers events to all waiting processes when the device state changes.

Kqueue 过滤器允许用户程序有效地轮询设备的状态，而无需执行读取或写入，并启用事件驱动的设计。每个 *knote* 代表用户进程附加的单个事件，当设备状态发生变化时，内核会向所有等待进程传递事件。

To support this, each device that implements kqueue must maintain a `klist` of attached knotes. The first step is to add this to the driver's softc:

为了支持这一点，每个实现 kqueue 的设备都必须维护一个附加 knote 的 `klist`。第一步是将此添加到驱动程序的 softc 中:

```
/* Device softc structure with klist */
struct p6stats_softc {
    ...
    struct klist sc_klist;
};
```

Next, we define the filter operations for each of our events. At first, all function pointers will be set to `NULL` until you implement them:

接下来，我们为每个事件定义筛选作。首先，所有函数指针都将设置为 `NULL`，直到您实现它们：

```
/* filterops structures with all functions NULL initially */
struct filterops p6stats_read_filtops = {
    .f_flags = NULL,
    .f_attach = NULL,
    .f_detach = NULL,
    .f_event = NULL,
    .f_process = NULL,
    .f_modify = NULL,
};

struct filterops p6stats_write_filtops = {
    .f_flags = NULL,
    .f_attach = NULL,
    .f_detach = NULL,
    .f_event = NULL,
    .f_process = NULL,
    .f_modify = NULL,
};
```

Finally, we add the `kqfilter` entry point. This is called when a process attaches a knote to the device:

最后，我们添加 `kqfilter` 入口点。当进程将 knote 附加到设备时，将调用此命令：

```
/* kqfilter function skeleton */
int
p6stats_kqfilter(struct file *fp, struct knote *kn)
{
    struct p6stats_softc *sc = fp->f_data; /* pointer to device softc */

    switch (kn->kn_filter) {
    case EVFILT_READ:
        /* your code here */
        break;
    case EVFILT_WRITE:
        /* your code here */
        break;
    default:
        return (EINVAL);
    }

    /* your code here */
    return (0);
}
```

This skeleton sets up the framework for event-driven access to the p6stats driver. In the next step, you will implement the actual filter logic so that user processes are notified when the device is ready to read or write.

该骨架为事件驱动的 p6stats 驱动程序访问设置了框架。在下一步中，您将实现实际的筛选逻辑，以便在设备准备好读取或写入时通知用户进程。

Your Task 您的任务

Your task is to extend the p6stats driver to support event-driven I/O using kqueue -- this means defining filter operations for both read and write events, implementing the `kqfilter` function so that knotes can be attached, and keeping track of those knotes in the device’s `klist`.

您的任务是扩展 p6stats 驱动程序以支持 kqueue 的事件驱动 I/O ——这意味着为读取和写入事件定义过滤作，实现 `kqfilter` 函数以便可以附加 knotes，并在设备的 `klist` 中跟踪这些 knotes。

When a user polls on a **write** event they should get a 1 for if it's ready, or otherwise 0 if it's not. If a user polls on a **read** event, they should be given back a number of bytes they have to read out.

当用户对**写入**事件进行轮询时，如果它准备就绪，他们应该得到 1，如果它没有准备好，他们应该得到 0。如果用户对**读取**事件进行轮询，则应返回必须读出的字节数。

Once this is in place, the driver must notify any waiting processes whenever the device becomes ready for a read or a write.

完成此作后，每当设备准备好进行读取或写入时，驱动程序都必须通知任何等待进程。

Hints 提示

When a new knote is attached, set its filterops pointer appropriately and keep a reference to the device softc in `kn->kn_hook`.

当附加新的注释时，适当地设置其 filterops 指针，并在 `kn->kn_hook` 中保留对设备 softc 的引用。

Insert the knote into the device’s `klist`, and whenever the device’s state changes, use `knote()` to notify all waiting processes.

将 knote 插入到设备的 `klist` 中，每当设备的状态发生变化时，使用 `knote ()` 通知所有等待进程。

The key part is the `f_event` callback inside your filterops: this function decides whether the device is currently ready to perform the operation.

关键部分是 filterops 内部 的 `f_event` 回调：此函数决定设备当前是否已准备好执行该作。

If you want to see concrete examples, the `if_tun` and `bpf` drivers are excellent references, and can be found in `/usr/src/sys/net/if_tun.c` and `/usr/src/sys/net/bpf.c`.

如果您想查看具体示例，`if_tun` 和 `bpf` 驱动程序是很好的参考，可以在 `/usr/src/sys/net/if_tun.c` 和 `/usr/src/sys/net/bpf.c` 中找到。

[Ex 5] Kevent In Userland

[例 5]用户区的 Kevent

Now that the driver supports `kqfilter`, we can extend the userland test program to exercise readiness checking. This allows processes to determine whether the device is ready for reading or writing without blocking, providing immediate feedback on the device state.

现在驱动程序支持 `kqfilter`，我们可以扩展用户区测试程序来进行就绪性检查。这允许进程确定设备是否已准备好读取或写入而不会阻塞，从而提供有关设备状态的即时反馈。

Your Task 您的任务

Update your test program to support two new options: `-R` to check if the device is ready for reading and `-W` to check if it is ready for writing. These options should be accepted alongside the existing `-r`, `-w`, and `-n` flags.

更新测试程序以支持两个新选项：`-R` 检查设备是否已准备好读取，`-W` 检查设备是否已准备好写入。这些选项应与现有的 `-r`、`-w` 和 `-n` 标志一起接受。

When either `-R` or `-W` is specified, your program should create a kqueue, register the appropriate `EVFILT_READ` or `EVFILT_WRITE` filters for the device, and perform a zero-timeout check to determine readiness. The program should print clear messages such as:

当指定 `-R` 或 `-W` 时，程序应创建一个 kqueue，为设备注册适当的 `EVFILT_READ` 或 `EVFILT_WRITE` 过滤器，并执行零超时检查以确定就绪情况。程序应打印清晰的消息，例如：

```
Read is ready
No read is ready
p6stats device ready for input
p6stats busy


阅读准备就绪
未准备好读取
P6Stats 设备准备输入
p6stats 忙
```

The program should still perform normal read and/or write operations if `-r` or `-w` is specified. If no valid operation is selected, it should provide a meaningful error message to the user.

如果指定了 `-r` 或 `-w`，程序仍应执行正常的读取和/或写入作。如果未选择有效作，则应向用户提供有意义的错误消息。

Hints 提示

Create a kqueue and populate a `struct kevent` array with the filters you want to check.

创建一个 kqueue 并使用要检查的过滤器填充 `struct kevent` 数组。

Use `kevent()` twice: first to register the events, then again with a zero-timeout to query readiness. Examine the `filter` field of any returned events to determine whether the device is ready for reading or writing.

使用 `kevent ()` 两次：首先注册事件，然后再次使用零超时来查询就绪情况。检查任何返回事件的 `筛选器` 字段，以确定设备是否已准备好读取或写入。

If `kevent()` returns no events, that means the file descriptor is not ready, so print an appropriate "not ready" message. Finally, remember to close both the kqueue and the device file descriptor before exiting.

如果 `kevent ()` 没有返回任何事件，则表示文件描述符尚未就绪，因此请打印适当的"未就绪"消息。最后，请记住在退出之前关闭 kqueue 和设备文件描述符。

Solution 溶液

Here is our solution for the user program:

这是我们针对用户程序的解决方案：

▶ 扩大

And our solution for the kernel driver:

而我们的内核驱动程序解决方案：

代码见Kevent In Userland1.c和Kevent In Userland2.c

▶ 扩大