Introduction 介绍

This week we will be writing another driver for a PCI device. Our device this week is a more advanced specimen than the one we saw in Prac 4, capable of Direct Memory Access (DMA) as a Bus Master -- meaning it can read and write host memory directly. It also uses MSI-X interrupts.

本周我们将为 PCI 设备编写另一个驱动程序。本周我们的设备比我们在 Prac 4 中看到的设备更先进，能够作为总线主站进行直接内存访问 （DMA），这意味着它可以直接读取和写入主机内存。它还使用 MSI-X 中断。

For the userland interface to this driver, we will be using a device special file interface like the one you constructed in Prac 5.

对于此驱动程序的用户空间接口，我们将使用设备专用文件接口，就像您在 Prac 5 中构建的接口一样。

Boilerplate 样板

Since we've already been through the basic steps of creating a new device driver and attaching to a PCI device based on Vendor and Product ID, this week we will provide a boilerplate base patch to skip through those steps. Remember to remove any syscalls if you have added them before trying to apply this base patch.

由于我们已经完成了创建新设备驱动程序并根据供应商和产品 ID 附加到 PCI 设备的基本步骤，因此本周我们将提供一个样板基础补丁来跳过这些步骤。在尝试应用此基础补丁之前，如果您已添加任何系统调用，请记住删除它们。

Create a new branch for Prac 6 and download the boilerplate:

为 Prac 6 创建一个新分支并下载样板：

```
$ git checkout -b p6 openbsd-7.7
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/p6-base-patch-2025.patch
$ git am < p6-base-patch-2025.patch && rm p6-base-patch-2025.patch
```

You can use `git show --stat` to look at the files changed:

您可以使用 `git show --stat` 查看更改的文件：

```
$ git show --stat

commit ff5fd961000443771a07273b5a118fdf363b5a12 (HEAD -> p6)
Author: uqltinsl <uqltinsl@comp3301.eait.uq.edu.au>
Date:   Sun Aug 24 10:59:50 2025 +1000

    p6-base-patch-rebase

 sys/arch/amd64/amd64/conf.c |  3 +++
 sys/arch/amd64/conf/GENERIC |  1 +
 sys/dev/pci/files.pci       |  5 +++++
 sys/dev/pci/p6stats.c       | 73 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 sys/sys/conf.h              |  9 +++++++++
 5 files changed, 91 insertions(+)
```

As you can see, this boilerplate sets up the basic `autoconf(9)` pieces for our driver, and also adds a character device at major 102 for AMD64 for our driver to implement.

如您所见，此样板为我们的驱动程序设置了基本的 `autoconf（9）` 部分，并且还为 AMD64 添加了一个大 102 的字符设备供我们的驱动程序实现。

The file `sys/dev/pci/p6stats.c` is where we will be writing our code for the driver.

文件 `sys/dev/pci/p6stats.c` 是我们将为驱动程序编写代码的地方。

Base Patch Sanity Check 基础补丁健全性检查

To do a quick test the base patch applied correctly let's create a node and test. Firstly let's build our kernel.

为了快速测试正确应用基础补丁，让我们创建一个节点并进行测试。首先，让我们构建我们的内核。

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

Special Device Node

特殊设备节点

If you remember from the week 5 applied class 5 and prac 5 to interface with a device though open/close/read/write you need to create a special device file with the required major and minor number.

如果您还记得从第 5 周开始应用第 5 类和实践 5 通过打开/关闭/读取/写入与设备交互，您需要创建一个具有所需主要和次要编号的特殊设备文件。

After rebooting, you can create a device node for the new driver using `mknod`:

重新启动后，您可以使用 `mknod` 为新驱动程序创建设备节点：

```
$ doas mknod /dev/p6stats c 102 0
$ doas chmod 0666 /dev/p6stats
$ cat /dev/p6stats
cat: /dev/p5d: Operation Not supported by device
```

Here we have created out node, and tried to open our device, but since our open is returning `ENODEV` we are getting the Not Supported error. We have chmod it with 0666 to allow all users to perform operations on the file.

在这里，我们创建了 out 节点，并尝试打开我们的设备，但由于我们的 open 返回 `ENODEV`，我们收到了 Not Supported 错误。我们用 0666 对其进行了 chmod，以允许所有用户对文件执行作。

```
Octal:  0666
Binary: 110 110 110
rwx:    rw- rw- rw-
```

Device Interface 设备接口

The P6 devices have a register window referenced by the first BAR in the PCI configuration space header. Within that window the following registers are available:

P6 器件具有一个寄存器窗口，该窗口由 PCI 配置空间标头中的第一个 BAR 引用。在该窗口中，可以使用以下寄存器：

Offset	Size	Name	Description
0x00	64 bits	IBASE	The base linear (physical) address of the input buffer (R/W)
0x08	64 bits	ICOUNT	The number of 64-bit integers located at IBASE (R/W)
0x10	64 bits	OBASE	The base linear (physical) address of the output buffer (R/W)
0x18	64 bits	DBELL	Doorbell register to begin processing (write-only)

This looks very different to what we saw in Prac 4! For this device, since it can be used to compute statistics about an arbitrarily large number of integers, we don't write those integers directly into the BAR any more.

这看起来与我们在 Prac 4 中看到的非常不同！对于此设备，由于它可以用于计算有关任意数量整数的统计信息，因此我们不再将这些整数直接写入 BAR。

Instead, we now write a pointer to the array in main memory that we want the device to use, along with its size, and a pointer to some memory for the device to write the results into. The device will then use its Bus Master DMA capabilities to read and write our system DRAM directly by itself.

相反，我们现在写入一个指向我们希望设备使用的主内存中数组的指针及其大小，以及一个指向设备将结果写入的某个内存的指针。然后，该设备将使用其总线主 DMA 功能直接自行读取和写入我们的系统 DRAM。

As well as the two pointer registers and the register for the number of integers in the input, we have a 4th register called a "doorbell".

除了两个指针寄存器和输入中整数数的寄存器外，我们还有一个称为“门铃”的第 4 个寄存器。

A "doorbell" register is generally used with DMA-capable devices to send them a message when data is available or buffers are ready for them to access via DMA. In this case, we will be writing to it to tell the device that our input and output buffers are ready for it to begin computation. Doorbell registers are often written with an offset or index into the DMA region to indicate how much is now available. In this case, however, we will just be writing it with the value `1`. Our device will accept any write to the `DBELL` register to indicate when to start calculation.

“门铃”寄存器通常与支持 DMA 的设备一起使用，以便在数据可用或缓冲区准备好供它们通过 DMA 访问时向它们发送消息。在这种情况下，我们将写入它以告诉设备我们的输入和输出缓冲区已准备好开始计算。门铃寄存器通常带有偏移量或索引写入 DMA 区域，以指示现在可用的数量。但是，在这种情况下，我们将只用值 `1` 编写它。我们的设备将接受对 `DBELL` 寄存器的任何写入，以指示何时开始计算。

With this device, once it has finished the computation task, it will send the CPU an interrupt via MSI-X, to let it know that the task is complete and the output buffer is now valid.

使用此设备，一旦完成计算任务，它将通过 MSI-X 向 CPU 发送中断，让它知道任务已完成并且输出缓冲区现在有效。

So, our process for using this device will look roughly like:

因此，我们使用此设备的过程大致如下所示：

- Set up the input buffer, write its address into `IBASE` and the number of integers in it into `ICOUNT`

  设置输入缓冲区，将其地址写入 `IBASE`，将其中的整数数写入 `ICOUNT`
- Set up the output buffer, write its address into `OBASE`

  设置输出缓冲区，将其地址写入 `OBASE`
- Perform a store/write memory barrier

  执行存储/写入内存屏障
- Write to `DBELL` to trigger the device to start computation

  写入 `DBELL` 以触发设备开始计算
- Wait for the interrupt 等待中断
- Read from the output buffer

  从输出缓冲区读取

The format of the output buffer is as follows:

输出缓冲区的格式如下：

Offset	Size	Name	Description
0x00	64 bits	COUNT	Number of integers processed
0x08	64 bits	SUM	The sum of all the integers
0x10	64 bits	MEAN	The mean of all the integers, rounded down
0x18	64 bits	MEDIAN	The median of all the integers
0x20	8 bytes	—	(reserved for future use)

Next, we'll discuss the userland interface for our driver.

接下来，我们将讨论驱动程序的用户空间界面。

Userland Interface 用户空间界面

For this device, we're going to use a device special file interface to userland as we discussed earlier.

对于此设备，我们将使用设备专用文件接口来用户登录，正如我们之前所讨论的。

To be specific, since this device is built around a transactional operation (computing stats about a set of integers), we're going to make our device file support just a single `ioctl(2)` command.

具体来说，由于该设备是围绕事务作（计算一组整数的统计信息）构建的，因此我们将使我们的设备文件仅支持单个 `ioctl（2）` 命令。

Let's define our userland interface, by creating a new header file: `sys/dev/pci/p6statsvar.h`:

让我们通过创建一个新的头文件来定义我们的用户空间界面：`sys/dev/pci/p6statsvar.h`：

```
#if !defined(_DEV_PCI_P6STATS_H)
#define _DEV_PCI_P6STATS_H

#include <sys/ioctl.h>
#include <sys/ioccom.h>
#include <sys/types.h>

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

struct p6stats_calc {
        uint64_t               *pc_inputs;
        uint64_t                pc_ninputs;
        struct p6stats_output  *pc_output;
};
#define P6STATS_IOC_CALC        _IOWR('6', 1, struct p6stats_calc)

#endif /* _DEV_PCI_P6STATS_H */
```

After creating the new header file, install it by running `make includes`:

创建新的头文件后，通过运行 `make includes` 进行安装：

```
$ cd /usr/src/include
$ doas make includes
$ ls -la /usr/include/dev/pci/p6statsvar.h
-r--r--r--  1 root  bin  560 Aug 15 16:27 /usr/include/dev/pci/p6statsvar.h
```

Here we've defined a `struct p6stats_output` which represents the output buffer format of the device itself. We'll re-use that for our interface to userland, which consists of the `ioctl(2)` command `P6STATS_IOC_CALC`. The `ioctl` takes a `struct p6stats_calc` as its argument.

在这里，我们定义了一个结构 `p6stats_output`，它表示设备本身的输出缓冲区格式。我们将在用户空间的接口中重复使用它，该接口由 `ioctl（2）` 命令 `P6STATS_IOC_CALC` 组成。`ioctl` 将结构 `p6stats_calc` 作为其参数。

To check your understanding of this userland interface, write a simple test program which will open `/dev/p6stats` and call `ioctl(2)` with `P6STATS_IOC_CALC`.

要检查您对此用户界面的理解，请编写一个简单的测试程序，该程序将打开 `/dev/p6stats` 并使用 `P6STATS_IOC_CALC` 调用 `ioctl（2）。`

You can provide a fixed array of integers to the `ioctl` for now, no need to deal with argument parsing. After performing the `ioctl` command, print out the fields of the output buffer to stdout.

您现在可以向 `ioctl` 提供一个固定的整数数组，无需处理参数解析。执行 `ioctl` 命令后，将输出缓冲区的字段打印到 stdout。

▼ 扩大

Our testing tool code: 我们的测试工具代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>
#include <stdint.h>

#include <dev/pci/p6statsvar.h>

int
main(int argc, char *argv[])
{
    int fd, rc;
    const char *fn = "/dev/p6stats";
    uint64_t arr[] = { 100, 105, 50, 70, 90, 10, 90, 50 };
    struct p6stats_output out;
    struct p6stats_calc arg;

    fd = open(fn, O_RDWR);
    if (fd < 0)
        err(1, "open");

    arg.pc_inputs = arr;
    arg.pc_ninputs = sizeof (arr) / sizeof (uint64_t);
    arg.pc_output = &out;
    rc = ioctl(fd, P6STATS_IOC_CALC, &arg);
    if (rc)
        err(1, "ioctl");

    printf("count  = %llu\n", out.po_count);
    printf("sum    = %llu\n", out.po_sum);
    printf("mean   = %llu\n", out.po_mean);
    printf("median = %llu\n", out.po_median);
    return (0);
}
```

Mapping and Loading 映射和加载

Now it's time to start on our actual driver code.

现在是时候开始我们的实际驱动程序代码了。

First, in attach, add code to map the device BAR and stash enough information on the `softc` so that we can use the BAR later. This will look a lot like what we did in Prac 4: you'll want to call `pci_mapreg_map(9)`, and do some checks on the size.

首先，在附件中，添加代码来映射设备 BAR，并在 `softc` 上存储足够的信息，以便我们以后可以使用 BAR。这看起来很像我们在 Prac 4 中所做的：您需要调用 `pci_mapreg_map（9），` 并对大小进行一些检查。

We will also need to do some preparation in `attach` for when it comes time to write pointers to our input and output buffers in to the BAR. As noted on the table earlier, the pointers the device needs are pointers into *linear (physical)* memory.

我们还需要做一些 `附加` 准备工作，以便将指向输入和输出缓冲区的指针写入 BAR。如前面的表所述，设备需要的指针是 *指向线性（物理）* 内存的指针。

Regular kernel pointers (like we get from `malloc(9)` or `pool_get(9)`) are pointers into *kernel virtual address space*, which won't cut it for our purposes. If the device tries to write to these, it's going to go very badly.

常规内核指针（就像我们从 `malloc（9）` 或 `pool_get（9）` 中得到的那样）是指向*内核虚拟地址空间*的指针，这不会满足我们的目的。如果设备试图写入这些，它会非常糟糕。

In OpenBSD, when devices need to allocate or deal with memory buffers that devices can perform DMA against, they use the `bus_dma(9)` family of kernel functions.

在 OpenBSD 中，当设备需要分配或处理设备可以对其执行 DMA 的内存缓冲区时，它们使用 `bus_dma（9）` 系列内核函数。

Add two `bus_dmamap_t` members to your `softc`, one for the input buffer and one for the output buffer. Allocate them using `bus_dmamap_create(9)`.

向 软软件 添加两个 `bus_dmamap_t` 成员，一个用于输入缓冲区，一个用于输出缓冲区。使用 `bus_dmamap_create（9）` 分配它们。

In our device, we need to provide a single pointer for input and a single pointer for output. As a result, the `nsegments` argument we will be giving for both `bus_dmamap_create` calls will be `1`. Devices can (and usually do) support more than one pointer for performing large amounts of DMA, so that the memory does not have to be contiguous in linear address space (it can be broken up into multiple pieces).

在我们的设备中，我们需要为输入提供单个指针，为输出提供单个指针。因此，我们将为两个 `bus_dmamap_create` 调用给出的 `nsegments` 参数将为 `1`。设备可以（并且通常确实）支持多个指针来执行大量 DMA，因此内存不必在线性地址空间中连续（它可以分解成多个部分）。

Also give the flag `BUS_DMA_64BIT` in the `flags` argument to `bus_dmamap_create`. This flag isn't documented in the man page currently, but indicates to the framework that our device accepts full 64-bit linear addresses.

还要在 `flags` 参数中 `BUS_DMA_64BIT` 给 `bus_dmamap_create` 提供标志。此标志目前未记录在手册页中，但向框架指示我们的设备接受完整的 64 位线性地址。

Add code to set up the two `bus_dmamap_t` members to your `attach` function now. Don't forget to check return values for errors!

立即添加代码以将两个 `bus_dmamap_t` 成员设置为 `attach` 函数。不要忘记检查返回值是否有错误！

Implementation 实现

Open and close 打开和关闭

Next let's implement `open()` and `close()` callbacks for our device special file.

接下来，让我们为我们的设备特殊文件实现 `open（）` 和 `close（）` 回调。

We're going to allow for the possibility of multiple P6 devices on a machine, with the simple scheme of "device minor number = unit number". So, make a `p6stats_lookup` function which takes the `dev_t` and looks up the softc in `p6stats_cd.cd_devs`. We will call this at the top of all of device special file callbacks to retrieve our softc.

我们将允许在一台机器上使用多个 P6 设备的可能性，使用“设备次要编号 = 单元号”的简单方案。因此，创建一个 `p6stats_lookup` 函数，该函数获取 `dev_t` 并在 `p6stats_cd.cd_devs` 中查找 softc。我们将在所有设备特殊文件回调的顶部调用它来检索我们的 softc。

We don't have any special actions to take in `open()` and `close()`, so we just need to validate that the `dev_t` maps to an actual device which has been attached, and then return `0`.

我们在 `open（）` 和 `close（）` 中没有任何特殊作，因此我们只需要验证 `dev_t` 是否映射到已附加的实际设备，然后返回 `0`。

Both of these functions should return `ENXIO` if the `dev_t` is invalid, or if the device in question failed to attach properly. You might need to add a field in your `softc` which you set at the end of `attach` if everything went ok, to use for this purpose.

如果 `dev_t` 无效，或者相关设备无法正确连接，这两个函数都应返回 `ENXIO`。如果一切正常，您可能需要在 `softc` 中添加一个字段，该字段在 连接 结束时设置，以用于此目的。

ioctl

Once you've implemented `p6statsopen()` and `p6statsclose()`, let's move on to `p6statsioctl()`.

实现 `p6statsopen()` 和 `p6statsclose()` 后，让我们继续 `p6statsioctl()`。

The only `ioctl(2)` command we're going to support is `P6STATS_IOC_CALC`, so return `ENXIO` early if anything else is given in `cmd`.

我们唯一要支持的 `ioctl(2)` 命令是 `P6STATS_IOC_CALC`，因此如果在 `cmd` 中给出了其他任何内容，请尽早返回 `ENXIO`。

Then cast our `data` argument to a `struct p6stats_calc *` so we can read its fields easily.

然后将我们的 `数据` 参数转换为 `结构体 * p6stats_calc` 以便我们可以轻松读取其字段。

For our first approach to this, we're going to use `bus_dmamap_load_uio(9)` to directly generate a linear address space pointer to the userland memory we were given, without copying it.

对于我们的第一种方法，我们将使用 `bus_dmamap_load_uio(9)` 直接生成指向我们获得的用户空间内存的线性地址空间指针，而无需复制它。

To use that function, we will need to construct a `struct uio`. This struct represents an I/O request from userland: in `read()` or `write()` we would get one for free from the system call layer, but in `ioctl()` we have to generate one ourselves.

要使用该函数，我们需要构造一个 `结构 uio`。这个结构体代表来自用户空间的 I/O 请求：在 `read()` 或 `write()` 中，我们会从系统调用层免费获得一个，但在 `ioctl()` 中，我们必须自己生成一个。

You will need to construct a single `struct iovec` to be pointed at by the `struct uio`. Fill out its `iov_base` using the `pc_inputs` member of the `p6stats_calc`, and calculate `iov_len` based on `pc_ninputs`. Then fill out the following fields in the `struct uio`:

需要构造一个结构体 `iovec` 以供 `结构体 uio` 指向。使用 `p6stats_calc` 的 `pc_inputs` 成员填写其 `iov_base`，并根据 `pc_ninputs` 计算 `iov_len`。然后在 `结构体 uio` 中填写以下字段：

- `uio_iov`: pointer to the `iovec`

  `uio_iov`：指向 `iovec` 的指针

- `uio_iovcnt`: `1` (since we only have one iovec)

  `uio_iovcnt`：`1`（因为我们只有一个 iovec）

- `uio_resid`: same as `iov_len`

  `uio_resid`：与 `iov_len` 相同

- `uio_rw`: `UIO_WRITE` for the inputs, `UIO_READ` for the outputs (remember: read/write is from userland's perspective, not the device)

  `uio_rw`：输入 `UIO_WRITE`，`输出 UIO_READ`（记住：读/写是从用户的角度，而不是设备的角度）

- `uio_segflg`: `UIO_USERSPACE`

  `uio_segflg`：`UIO_USERSPACE`

- `uio_procp`: the `struct proc *p` argument to `p6statsioctl()`

  `uio_procp`：`p6statsioctl()` 的 `struct proc *p` 参数

Once you've done that, the `struct uio` should be ready to use with `bus_dmamap_load_uio(9)`. Add the call to that function now for your input buffer. Remember that this first transaction is a DMA "write" from the host's perspective (so you'll be giving `BUS_DMA_WRITE`).

完成此作后，`struct uio` 应该可以与 `bus_dmamap_load_uio（9）` 一起使用。现在为您的输入缓冲区添加对该函数的调用。请记住，从主机的角度来看，第一个事务是 DMA“写入”(因此您将给出 `BUS_DMA_WRITE`)。

Then construct your `iovec` and `uio` for the output buffer in a similar fashion. Remember that it should be set to `UIO_READ`, and give `BUS_DMA_READ` to `bus_dmamap_load_uio()`.

然后以类似的方式为输出缓冲区构造 `iovec` 和 `uio`。请记住，它应该设置为 `UIO_READ`，并给 `bus_dmamap_load_uio()` 提供 `BUS_DMA_READ`。

Now we have one last step before we start writing our pointers into the BAR: calling `bus_dmamap_sync(9)` to construct appropriate memory barriers for the transfer.

现在，在开始将指针写入 BAR 之前，我们还有最后一步：调用 `bus_dmamap_sync（9）` 为传输构建适当的内存屏障。

You'll need to do a `BUS_DMASYNC_PREWRITE` on the input mapping, and a `BUS_DMASYNC_PREREAD` on the output mapping.

您需要对输入映射进行 `BUS_DMASYNC_PREWRITE`，并对输出映射进行 `BUS_DMASYNC_PREREAD`。

Then add code to write pointers into the BAR. You can access the pointer values at `->dm_segs[0].ds_addr` on each `bus_dmamap_t`. Write to `IBASE`, `ICOUNT` and `OBASE`.

然后添加代码以将指针写入 BAR。您可以在每个 `bus_dmamap_t` 上访问 `->dm_segs[0].ds_addr` 处的指针值。写入 `IBASE`、 `ICOUNT` 和 `OBASE`。

Then do a `bus_space_barrier(9)` with `BUS_SPACE_BARRIER_WRITE` before finally writing the value `1` to `DBELL`.

然后用 `BUS_SPACE_BARRIER_WRITE` 做一个 `bus_space_barrier（9），` 最后将值 `1` 写入 `DBELL`。

Leave some space next for our code to wait for an interrupt: we'll come back to that in the next step. After the interrupt has happened, we need to do our final `bus_dmamap_sync` calls (`BUS_DMASYNC_POSTWRITE` and `BUS_DMASYNC_POSTREAD`) and then unload the `bus_dmamap_t`s so they're ready for re-use.

接下来为我们的代码留出一些空间来等待中断：我们将在下一步中回到这一点。中断发生后，我们需要进行最后的 `bus_dmamap_sync` 调用（`BUS_DMASYNC_POSTWRITE` 和 `BUS_DMASYNC_POSTREAD`），然后卸载 `bus_dmamap_t`，以便它们准备好重复使用。

Interrupt 中断

In the last section, we left some space in our `ioctl()` handler for us to wait for the interrupt to occur. Let's implement that now.

在上一节中，我们在 `ioctl（）` 处理程序中留了一些空间，供我们等待中断发生。现在让我们实现它。

First, we will need to map and activate the interrupt in our `attach` function. Add some code there to call `pci_intr_map_msix(9)` and `pci_intr_establish(9)`.

首先，我们需要在 `attach` 函数中映射并激活中断。在那里添加一些代码来调用 `pci_intr_map_msix（9）` 和 `pci_intr_establish（9）。`

You will need to give the interrupt vector number to `pci_intr_map_msix` -- you want interrupt vector `0` (the first and only interrupt) on this device.

您需要将中断向量编号提供给 `pci_intr_map_msix` -- 您需要此设备上的中断向量 `0`（第一个也是唯一一个中断)。

You will also need a function to be called when the interrupt occurs. Define a `static int p6stats_intr(void *arg)` for this.

您还需要在中断发生时调用一个函数。为此定义一个 `static int p6stats_intr(void *arg)` 。

And lastly, you will also need to set an interrupt priority level or IPL, in the arguments to `pci_intr_establish`.

最后，您还需要在参数中设置中断优先级或 IPL，以 `pci_intr_establish`。

IPLs in OpenBSD control which kinds of device interrupts can interrupt other devices' interrupts. While their values are unique per-CPU-architecture, the names of the IPL macros are shared. You can see them all defined for AMD64 in `sys/arch/amd64/include/intrdefs.h` and in the man page `spl(9)`.

OpenBSD 中的 IPL 控制哪些类型的设备中断可以中断其他设备的中断。虽然它们的值在每个 CPU 架构上是唯一的，但 IPL 宏的名称是共享的。您可以在手册页 `spl（9）` 中 `sys/arch/amd64/include/intrdefs.h` 看到它们为 AMD64 定义的所有内容。

For our device we will be using `IPL_BIO`, the "block I/O" priority level, which is the lowest priority generally used for hardware device interrupts. You should give this to `pci_intr_establish` combined with `IPL_MPSAFE`, which specifies that our interrupt handler does not need the big kernel lock.

对于我们的设备，我们将使用 `IPL_BIO`，即“块 I/O”优先级，这是通常用于硬件设备中断的最低优先级。您应该将其与 `IPL_MPSAFE` 结合使用 `pci_intr_establish`，这指定我们的中断处理程序不需要大内核锁。

Remember to stash the `void *` interrupt handle pointer returned by `pci_intr_establish` on the `softc` so we can tear it down later if needed.

请记住将 `pci_intr_establish` 返回的 `void *` interrupt 句柄指针隐藏在 `softc` 上，以便我们稍后在需要时将其拆除。

p6stats_intr

Next, let's implement `p6stats_intr()` itself.

接下来，让我们实现 `p6stats_intr（）` 本身。

Since our `p6stats_intr` function is going to be called in interrupt context, it cannot allocate memory or sleep in any way, and if it is going to touch anything in our device `softc`, it's going to need to take a lock first.

由于我们的 `p6stats_intr` 函数将在中断上下文中被调用，因此它不能以任何方式分配内存或睡眠，如果它要触及我们设备 `softc` 中的任何内容，它将需要先锁定。

In P4 we used a `rwlock(9)`, which is a kind of sleeping lock. We cannot use a sleeping lock to synchronise with an interrupt handler, since the interrupt handler cannot sleep.

在 P4 中，我们使用了一个 `rwlock（9），` 这是一种休眠锁。我们不能使用休眠锁与中断处理程序同步，因为中断处理程序不能休眠。

Instead, use a `mutex(9)` for this prac. Note that `mtx_init(9)` needs to know the IPL of the interrupt handler it's locking against -- use `IPL_BIO` since that's what we've set it up to run at.

相反，请为此练习使用 `mutex（9）。` 请注意，`mtx_init（9）` 需要知道它所锁定的中断处理程序的 IPL -- 使用 `IPL_BIO`，因为这是我们设置为运行它的位置。

Add a "state" integer member to the `softc` which contains enumerated values for the current state of operation. We suggest using the names "READY", "BUSY" and "DONE" to describe the 3 states:

将“state”整数成员添加到 `softc`，其中包含当前作状态的枚举值。我们建议使用名称“READY”、“BUSY”和“DONE”来描述 3 种状态：

- `READY`: there is no operation in progress

  `READY：` 没有正在进行的作

- `BUSY`: the `ioctl()` routine has set up a calculation and started it, it is now in progress

  `BUSY`： `ioctl（）` 例程已设置计算并启动，现在正在进行中

- `DONE`: the interrupt has fired and the results of the calculation are waiting for `ioctl()` to pick them up and then change state back to `READY`

  `DONE：` 中断已触发，计算结果正在等待 `ioctl（）` 拾取它们，然后将状态更改回 `READY`

Set this state member as appropriate in your `p6stats_intr` code (after taking the mutex) and in `p6statsioctl()`. You'll need to modify `p6statsioctl()` to take and hold the mutex, as well.

在 `p6stats_intr` 代码（采用互斥锁之后）和 `p6statsioctl()` 中适当地设置此状态成员。您还需要修改 `p6statsioctl()` 以获取并保留互斥锁。

Then we need to make `p6statsioctl` sleep until the calculation is finished. Use `msleep(9)` for this since we have a mutex for synchronisation. Don't forget the `PCATCH` flag, and add the matching `wakeup(9)` call in `p6stats_intr`.

然后我们需要让 `p6statsioctl` 进入睡眠状态，直到计算完成。为此使用 `msleep(9)`，因为我们有一个用于同步的互斥锁。不要忘记 `PCATCH` 标志，并在 `p6stats_intr` 中添加匹配的 `wakeup(9)` 调用。

Finally, we will need to add some code near the top of `p6statsioctl` which checks to see if the device is in the `READY` state before attempting to set up the new calculation.

最后，我们需要在 `p6statsioctl` 的顶部附近添加一些代码，在尝试设置新计算之前检查设备是否处于 `READY` 状态。

Make this code sleep on the same wait channel until state reaches `READY`. Add a matching `wakeup(9)` at the end of `p6statsioctl` where you change it back to this value after doing `bus_dmamap_sync` and `bus_dmamap_unload`.

使此代码在同一等待通道上休眠，直到状态达到 `READY`。在 `p6statsioctl` 的末尾添加一个匹配的 `wakeup(9)`，在执行 `bus_dmamap_sync` 和 `bus_dmamap_unload` 后将其更改回此值。

Now we are ready to test it out!

现在我们准备好测试它了！

Testing 测试

Build and install the new kernel as per usual.

像往常一样构建和安装新内核。

If you have followed the Boilerplate instructions you should have already created a node, but as a quick sanity check let's see if it's there.

如果您按照样板说明进行作，您应该已经创建了一个节点，但作为快速健全性检查，让我们看看它是否存在。

```
$ ls -la /dev/p6stats
crw-rw-rw-  1 root  wheel  102, 0 Aug 30 11:26 /dev/p6stats
```

Note that we need major 102 and minor 0 for the first p6stats device on the system. So check that those numbers are correct in the output, and that we have the correct permissions `crw-rw-rw`. If any of this is incorrect we can just delete it and recreate our node.

请注意，我们需要系统上的第一个 p6stats 设备的主要 102 和次要的 0。因此，请检查输出中的这些数字是否正确，并且我们是否具有正确的权限 `crw-rw-rw`。如果其中任何一个不正确，我们可以将其删除并重新创建我们的节点。

```
$ doas mknod /dev/p6stats c 102 0
$ doas chmod 0666 /dev/p6stats
$ ls -la /dev/p6stats
crw-rw-rw-  1 root  wheel  102, 0 Aug 30 11:26 /dev/p6stats
```

If this shows correctly, we can try out our simple test program and make sure it works.

如果显示正确，我们可以尝试我们的简单测试程序并确保它有效。

Next we are going to break it.

接下来我们将打破它。

代码见文件p6stats.c

Bouncing 弹跳

In the last section, you should have observed that `bus_dmamap_load_uio(9)` returned `EFBIG` when you added enough integers or allocated them in particular ways. Why?

在上一节中，您应该已经注意到，当您添加足够的整数或以特定方式分配它们时，`bus_dmamap_load_uio（9）` 返回了 `EFBIG`。为什么?

Even if a buffer of integers looks like it was one contiguous long array to userland, as soon as the buffer crosses a page boundary, it may no longer be one big array in physical memory!

即使整数缓冲区看起来像是一个连续的长数组，一旦缓冲区越过页面边界，它可能就不再是物理内存中的一个大数组了！

Physical memory after boot is often quite fragmented, so when userland asks for 4 pages, the kernel will almost always give it pages that are not together in a neat row.

启动后的物理内存通常非常碎片化，因此当用户空间要求 4 页时，内核几乎总是会给它不整齐排列的页面。

Since the P6 device expects us to give it a single pointer and length in physical memory to read from, and we told `bus_dmamap_create` that that's what we need, if it can't fulfill this it fails with `EFBIG`.

由于 P6 设备希望我们在物理内存中给它一个指针和长度来读取，并且我们告诉 `bus_dmamap_create` 这就是我们需要的，如果它不能满足这一点，它就会失败 `EFBIG`。

So what do we do?

那么我们该怎么办?

Bounce buffers 退回缓冲区

One solution is *bounce buffering*, where we work around this by allocating a physically contiguous buffer that can be used by the hardware, and copying the data from the program into the buffer before handing the physical address of the buffer to the hardware to use.

一种解决方案是*反弹缓冲*，我们通过分配一个可供硬件使用的物理连续缓冲区，并在将缓冲区的物理地址交给硬件之前将数据从程序复制到缓冲区中来解决这个问题。

The function `bus_dmamem_alloc(9)` can be used in the kernel to pre-allocate contiguous pages that are in range of DMA for a given device. We can use it to allocate our large pre-defined bounce buffer and copy our data in and out of that instead when it can no longer be mapped straight from the userland pages.

函数 `bus_dmamem_alloc（9）` 可以在内核中用于为给定设备预先分配 DMA 范围内的连续页面。我们可以使用它来分配我们大型预定义的退回缓冲区，并在无法再直接从用户空间页面映射数据时将数据复制到该缓冲区中。

`bus_dmamem_alloc` outputs a set of `bus_dma_segment_t` structs, like we saw in the `bus_dmamap_t`, but these are different! The output segments from `bus_dmamem_alloc` can **only** be used with `bus_dmamem_map`, `bus_dmamem_free` and `bus_dmamap_load_raw`. They must not be used as addresses directly for DMA. Conceptually it's a whole different struct, which you should consider opaque and not touch the contents of.

`bus_dmamem_alloc` 输出一组 `bus_dma_segment_t` 结构，就像我们在 `bus_dmamap_t` 中看到的那样，但这些结构是不同的！ `bus_dmamem_alloc` 的输出段**只能**用于 `bus_dmamem_map`、`bus_dmamem_free` 和 `bus_dmamap_load_raw`。它们不得直接用作 DMA 的地址。从概念上讲，这是一个完全不同的结构，您应该将其视为不透明的，而不是触及其内容。

To get a kernel virtual pointer to the memory that `bus_dmamem_alloc` allocates, we can give its output to `bus_dmamem_map`. That will give us a `caddr_t` which we can cast to any pointer type to read or write to the memory from kernel code.

要获取指向 `bus_dmamem_alloc` 分配的内存的内核虚拟指针，我们可以将其输出提供给 `bus_dmamem_map`。这将为我们提供一个 `caddr_t`，我们可以将其转换为任何指针类型，以从内核代码读取或写入内存。

When it's ready for DMA, we load the memory into a `bus_dmamap_t` like before, using `bus_dmamap_load_raw` this time rather than `bus_dmamap_load_uio`. Then we continue just as before.

当它准备好进行 DMA 时，我们像以前一样将内存加载到 `bus_dmamap_t` 中，这次使用 `bus_dmamap_load_raw` 而不是 `bus_dmamap_load_uio`。然后我们像以前一样继续。

Change your code from the previous section to handle the `EFBIG` error by allocating a bounce buffer and copying the integers into it, using that buffer for DMA instead.

更改上一节中的代码，通过分配退回缓冲区并将整数复制到其中，改用该缓冲区进行 DMA 来处理 `EFBIG` 错误。

Then try out your failing test cases from before. Does it fix them? How big can you go before `bus_dmamem_alloc` fails as well?

然后尝试之前失败的测试用例。它能解决它们吗？在 `bus_dmamem_alloc` 倒闭之前，你能做多大？

More Fixes? 更多修复？

At this point we've gone about as far as we can go without changing the device interface.

在这一点上，我们已经在不更改设备界面的情况下尽了最大努力。

If we wanted the P6 device to be able to handle even larger data buffers, we would have to change it so that it can take more than one pointer.

如果我们希望 P6 设备能够处理更大的数据缓冲区，我们必须对其进行更改，以便它可以接受多个指针。

When a device takes more than one `(pointer,length)` tuple and treats them as if they were one big buffer concatenated, we call it *scatter DMA* (when the device is writing) or *gather DMA* (when it is reading). This is a very commonly used feature with DMA-supporting devices.

当设备采用多个 `（pointer，length）` 元组并将它们视为一个连接的大型缓冲区时，我们将其称为*分散 DMA*（当设备写入时）或*收集 DMA*（当它读取时）。这是支持 DMA 的设备非常常用的功能。

In your next assignment (A2) you will see an example of a device which makes extensive use of scatter-gather DMA.

在您的下一个作业 （A2） 中，您将看到一个广泛使用散射聚集 DMA 的设备示例。