# Welcome 欢迎

In the last few practicals, we've added system calls to the OpenBSD operating system. You learnt how the kernel and userland agree on the system call number and how its arguments are transported between them.

在最近的几个实践中，我们向 OpenBSD 作系统添加了系统调用。您了解了内核和用户区如何就系统调用号达成一致，以及其参数如何在它们之间传输。

When we want to add support to the kernel for some specific hardware device, we don't usually want to make a new system call as the userland interface to that device. We would much rather use an existing interface to userland that fits our device if we can, so that software does not have to be written custom for each and every device.

当我们想为某些特定的硬件设备添加对内核的支持时，我们通常不希望进行新的系统调用作为该设备的用户空间接口。如果可以的话，我们宁愿使用适合我们设备的现有用户界面，这样就不必为每台设备定制软件。

In this practical, you will be implementing one such existing interface: the "device special file". Similarly to system calls, device special files have a magic number associated with them which is agreed upon by userland and the kernel. This number is called the device's "major" number, and rather than existing in header files or being built into `libc`, instead it's written into the filesystem in the `/dev` directory.

在本实践中，您将实现一个这样的现有接口：“设备特殊文件”。与系统调用类似，设备特殊文件有一个与之关联的幻数，该数字由用户区和内核商定。这个数字被称为设备的“主要”编号，它不是存在于头文件中或内置于 `libc` 中，而是写入文件系统中的 `/dev` 目录。

## Uninstalling Syscalls 卸载系统调用

> ⚠ Do not skip this step particularly if you have been working on an assignment or another prac. This ensures that your kernel and libc go back to the state they were when you installed OpenBSD. This will save a lot of headaches. It is best to avoid doing pracs whilst working on an assignment to avoid any undesirable issues.
>
> 不要跳过这一步，特别是如果你一直在做作业或其他练习。这可以确保你的内核和 libc 回到你安装 OpenBSD 时的状态。这将省去很多麻烦。最好避免在处理作业时进行练习，以避免任何不良问题。

Prac 2 only went through the steps for adding syscalls, but not removing syscalls. If you are currently running a kernel with other syscalls added, then you'll have to first uninstall those syscalls before starting this prac.

Prac 2 仅完成了添加系统调用的步骤，但没有删除系统调用。如果您当前正在运行一个添加了其他系统调用的内核，那么您必须先卸载这些系统调用，然后再开始此练习。

The steps for removing syscalls is different compared to adding syscalls:

与添加系统调用相比，删除系统调用的步骤不同：

```
comp3301$ cd /usr/src
comp3301$ git checkout main
comp3301$ cd /usr/src/include
comp3301$ doas make includes
comp3301$ cd /usr/src/lib/libc
comp3301$ make -j4
comp3301$ doas make install
comp3301$ cd /usr/src/libexec/ld.so
comp3301$ make
comp3301$ doas make install
comp3301$ cd /usr/src/sys/arch/amd64/compile/GENERIC.MP
comp3301$ make config
comp3301$ make -j4
comp3301$ doas make install
comp3301$ doas reboot
```

If removing a syscall fails, please see the backup guide. Worst case scenario, if you have a snapshot, then roll it back using the steps from the same guide. If you don't, however, you will need to reinstall the kernel from scratch.

如果删除系统调用失败，请参阅备份指南。 最坏的情况是，如果您有快照，请使用同一指南中的步骤将其回滚。但是，如果不这样做，则需要从头开始重新安装内核。

# Skeleton 骨架

## Magic Numbers 神奇数字

First, let's declare our device driver functions on the kernel side so that we can get it assigned a device major number. We'll name our driver `p5d` (for "P5 device"), and we'll make it a character device.

首先，让我们在内核端声明我们的设备驱动程序函数，以便我们可以为其分配一个设备主号。我们将驱动程序命名为 `p5d`（表示"P5 设备"），并将其设为字符设备。

Start a new branch based on the `openbsd-7.3` tag as usual:

像往常一样基于 `openbsd-7.3` 标签启动一个新分支：

```
$ git checkout -b p5 openbsd-7.7

$ git 结帐 -b p5 openBSD-7.7
```

Device major numbers are per-architecture in OpenBSD, and for this prac we'll just be adding our device to the AMD64 table, in `sys/arch/amd64/amd64/conf.c`.

在 OpenBSD 中，设备主要编号是每个架构的，对于这个实践，我们只会将我们的设备添加到 AMD64 表中，位于 `sys/arch/amd64/amd64/conf.c` 中。

Before we can add our entry, we need to define a "declaration" macro for our driver (these are macros of the form `cdev_X_init()` and `cdev_decl()`). These are found in `sys/sys/conf.h`, so open that file up now.

在添加条目之前，我们需要为驱动程序定义一个"声明"宏（这些宏格式为 `cdev_X_init（）` 和 `cdev_decl（）`）。这些可以在 `sys/sys/conf.h` 中找到，所以现在打开该文件。

We want our driver to eventually handle `open`, `close`, `read`, `write` and `ioctl`. First let's define the `cdev_*_init` macro for our driver.

我们希望驱动程序最终能够处理 `open`、`close`、`read`、`write` 和 `ioctl`。首先，让我们为驱动程序定义 `cdev_*_init` 宏。

Look for `cdev_tape_init` in this file — it wants the same set of functions that we do. Copy-paste its definition and change the name to `cdev_p5d_init`:

在这个文件中寻找 `cdev_tape_init` ——它需要与我们相同的函数集。复制粘贴其定义并将名称更改为 `cdev_p5d_init`：

```
/* open, close, read, write, ioctl */
#define    cdev_p5d_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, seltrue_kqfilter }
```

Then, near the bottom of `conf.h`, look for all the `cdev_decl()` macro invocations. Add one for `p5d`:

然后，在 `conf.h` 的底部附近，查找所有 `cdev_decl（）` 宏调用。为 `p5d` 添加一个：

```
cdev_decl(p5d);


cdev_decl（p5d）;
```

Next, open up `sys/arch/amd64/amd64/conf.c` and look for the `cdevsw` table.

接下来，打开 `sys/arch/amd64/amd64/conf.c` 并查找 `cdevsw` 表。

Add one instance of our new device at the bottom by invoking `cdev_p5d_init()`:

通过调用 `cdev_p5d_init（）` 在底部添加一个新设备的实例：

```
...
    cdev_pppx_init(NPPPX,pppac),  /* 99: PPP Access Concentrator */
    cdev_ujoy_init(NUJOY,ujoy),   /* 100: USB joystick/gamecontroller */
    cdev_psp_init(NPSP,psp),      /* 101: PSP */
    cdev_p5d_init(1,p5d),         /* 102: COMP3301 prac 5 */
};
int    nchrdev = nitems(cdevsw);

int    mem_no = 2;    /* major device number of memory special file */
```

As we can see, adding at the bottom will assign us major number `101`. We'll keep this in mind for later (when we create the device node), but next we will implement our skeleton driver.

正如我们所看到的，在底部添加将为我们分配主要编号 `101`。我们将在以后（创建设备节点时）牢记这一点，但接下来我们将实现骨架驱动程序。

## Makefiles and Config Makefile 和配置

Next, we'll add our driver to the kernel `Makefile`s and `config` as a "pseudo-device". Let's plan to put our implementation in the file `sys/dev/p5d.c`, so we'll add that to `sys/conf/files`:

接下来，我们将驱动程序添加到内核 `Makefile` `中，` 并作为“伪设备”配置。让我们计划将我们的实现放在文件 `sys/dev/p5d.c` 中，因此我们将其添加到 `sys/conf/files` 中：

```
pseudo-device p5d
file dev/p5d.c                p5d
```

This time we've added a line above our `file` line, declaring our pseudo-device, and we've tagged the `file` entry as being part of that driver. Doing this allows `config` to turn our driver on or off in builds (e.g. for architectures that don't support it).

这一次，我们在 `文件` 行上方添加了一行，声明我们的伪设备，并将 `文件` 条目标记为该驱动程序的一部分。这样做允许 `config` 在构建中打开或关闭我们的驱动程序（例如，对于不支持它的架构）。

Next we'll add our device to the `GENERIC` configuration for AMD64, which lives in `sys/arch/amd64/conf/GENERIC`. Add at the bottom:

接下来，我们将把我们的设备添加到 AMD64 的 `GENERIC` 配置中，它位于 `sys/arch/amd64/conf/GENERIC` 中。在底部添加：

```
...
#viocon*    at virtio?  # Virtio console device
vmmci*      at virtio?  # VMM control interface
#viogpu*    at virtio?  # VirtIO GPU device
#wsdisplay0 at viogpu? console 1
#wsdisplay* at viogpu? mux - 1

pseudo-device p5d
```

Since we've only added the device to the AMD64 kernel config, it won't be built on other architectures (which is good, since we only added it to the devices table and gave it a major number for AMD64!).

由于我们只将设备添加到 AMD64 内核配置中，因此它不会在其他架构上构建（这很好，因为我们只是将其添加到设备表中，并为其提供了 AMD64 的主要编号！

Now we're finally ready to fill out a basic skeleton of `p5d.c`.

现在我们终于准备好填写 `p5d.c` 的基本骨架了。

## Implementation 实现

Now let's fill in a basic skeleton of our `sys/dev/p5d.c`:

现在让我们填写 `sys/dev/p5d.c` 的基本骨架:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <sys/device.h>
#include <sys/vnode.h>
#include <sys/poll.h>

void
p5dattach(int n)
{
}

int
p5dopen(dev_t dev, int mode, int flags, struct proc *p)
{
    printf("hello p5d world\n");
    return (0);
}

int
p5dclose(dev_t dev, int flag, int mode, struct proc *p)
{
    return (0);
}

int
p5dwrite(dev_t dev, struct uio *uio, int flags)
{
    return (EOPNOTSUPP);
}

int
p5dread(dev_t dev, struct uio *uio, int flags)
{
    return (EOPNOTSUPP);
}

int
p5dioctl(dev_t dev, u_long cmd, caddr_t data, int flag, struct proc *p)
{
    return (ENXIO);
}
```

As you can see, this really is a skeleton: it does nothing useful except print a message (to `dmesg`) when the device is opened.

如您所见，这确实是一个骨架：除了在打开设备时打印一条消息（到 `dmesg`）之外，它没有任何用处。

We've implemented the functions for all of the system calls we declared that our device supports in `conf.h`: `open`, `close`, `read`, `write` and `ioctl`.

我们已经实现了我们在 `conf.h` 中声明的设备支持的所有系统调用的函数：`open`、`close`、`read`、`write` 和 `ioctl`。

Now let's build and install our kernel and try out our device.

现在让我们构建并安装我们的内核并试用我们的设备。

## Testing 测试

Build and install your kernel in the same way as in previous pracs:

以与之前练习相同的方式构建和安装内核：

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

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/p5d c 102 1
$ doas cat /dev/p5d
hello p5d world
cat: /dev/p5d: Operation not supported
```

(Note that you'll only see the `hello p5d world` output like that if you're on the serial console — otherwise you'll need to check in `dmesg`).

（请注意，只有在串行控制台上时，您才会看到这样的 `hello p5d 世界` 输出 - 否则您需要签入 `dmesg`）。

We've now successfully added a basic skeleton of a pseudo-device driver to the kernel. In the next part we'll be implementing some actual functionality.

我们现在已经成功地将伪设备驱动程序的基本骨架添加到内核中。在下一部分中，我们将实现一些实际功能。

# Sending Numbers 发送号码

## Design 设计

Next, we're going to implement the basic version of last week's prac in our device driver.

接下来，我们将在设备驱动程序中实现上周 prac 的基本版本。

Our device should: 我们的设备应该：

- Allow one process to `write()` an `int` to the device

  允许一个进程向设备`写入 （）` `int`
- Allow another unrelated process to `read()` that same `int` out again

  允许另一个不相关的进程再次`读取（）` 相同的 `int`
- When a process `write()`s a number to the device, it always returns immediately. If another number is already waiting, then the `write()` should return `EBUSY`

  当进程向设备`写入数字`时，它总是立即返回。如果另一个数字已经在等待，则 `write（）` 应返回 `EBUSY`
- When a process `read()`s from the device but there is no number waiting, it should block (sleep) until one is available

  当进程从设备读取 `（）` 但没有等待的数字时，它应该阻塞（睡眠）直到一个可用
- The `int` should be read/written as raw bytes (e.g. using `int foo; write(fd, &foo, sizeof(foo));` etc)

  `int` 应作为原始字节进行读取/写入（例如，使用 `int foo; write(fd, &foo, sizeof(foo));` etc）
- If a process attempts to perform a `read()` or `write()` with a size set to any other number than `sizeof(int)`, then it should return `EINVAL`

  如果进程尝试执行 `read（）` 或 `write（），` 并将大小设置为 `sizeof（int）` 以外的任何其他数字，则它应该返回 `EINVAL`

Rather than writing our own tool for testing this time, we can actually use some existing commands on our OpenBSD system to test reading and writing data to the device.

这次我们实际上可以使用 OpenBSD 系统上的一些现有命令来测试对设备的读取和写入数据，而不是编写我们自己的工具进行测试。

## Implementation 实现

Fill in your implementation in `sys/dev/p5d.c` now.

现在在 `sys/dev/p5d.c` 中填写您的实现。

It should look very much like your code from the previous prac (and feel free to copy-paste sections from it!). You'll need to fill out at least `p5dread()` and `p5dwrite()` for a minimal implementation.

它应该看起来非常像您之前实践中的代码（并且可以随意从中复制粘贴部分！您至少需要填写 `p5dread（）` 和 `p5dwrite（）` 才能实现最小的实现。

Instead of `copyout(9)` or `copyin(9)` you'll need to use `uiomove(9)` (and don't forget to check its return value!).

您需要使用 `uiomove（9）` 而不是 `copyout（9）` 或 `copyin（9）（` 并且不要忘记检查其返回值！

To help get used to the idea of using a `softc` struct, try allocating one in your `p5dattach` function and storing the number there (rather than directly in a global variable). You can use `malloc(9)` in the kernel, but note that it has different arguments to userland.

为了帮助习惯使用 `softc` 结构体的想法，请尝试在 `p5dattach` 函数中分配一个结构体并将数字存储在那里（而不是直接存储在全局变量中）。你可以在内核中使用 `malloc（9），` 但请注意它对 userland 有不同的参数。

## Testing 测试

You can test your implementation by using a combination of the commands `printf(1)`, `od(1)` and `dd(1)`.

您可以使用命令 `printf（1）`、`od（1）` 和 `dd（1）` 的组合来测试您的实现。

For example, to write a number:

例如，要写一个数字：

```
$ printf '\x03\x00\x00\x00' | dd bs=4 of=/dev/p5d status=none
$ printf '\x04\x00\x00\x00' | dd bs=4 of=/dev/p5d status=none
dd: /dev/p5d: Device busy
```

And to read a number out:

读出一个数字：

```
$ dd bs=4 if=/dev/p5d count=1 status=none | od -I
0000000                3
0000004
```

The number appears in the second column (the first column output by `od(1)` is the offset).

该数字出现在第二列中（`od（1）` 输出的第一列是偏移量）。

Carry out similar basic tests to what you did for part 1 of the previous prac: make sure that sleeping works and is interruptible, test for `EBUSY`, and try changing the size of the read/write (using the `bs=` argument to `dd(1)`) to make sure `EINVAL` is returned.

执行与上一个练习的第 1 部分类似的基本测试：确保睡眠有效且可中断，测试 `EBUSY`，并尝试更改读/写的大小（使用 `bs=` 参数为 `dd（1）`）以确保返回 `EINVAL`。

You might also find it helpful to change the permissons on `/dev/p5d` to world-writeable so you can run these tools as a normal user:

您可能还会发现将 `/dev/p5d` 上的权限更改为 world-writeable 很有帮助，这样您就可以以普通用户身份运行这些工具：

```
$ doas chmod 0666 /dev/p5d
```

详情解决方案请查看send_numbers.c

# ioctl() ioctl （）

## Design 设计

Next we'll add an `ioctl(2)` command for our device. This will be a custom `ioctl`, used to read out the current status of the device (a flag for whether there is a number currently waiting).

接下来，我们将为我们的设备添加一个 `ioctl（2）` 命令。这将是一个自定义 `ioctl`，用于读出设备的当前状态（当前是否有号码等待的标志）。

Read the ioctl(2) manual page and the ioccom.h header for background on what an ioctl can do and how commands and data are encoded in the command argument. Encoding the direction(s) and length of the data in the command allows the ioctl syscall handler to perform most of the copying of memory in and out of the kernel on the drivers behalf.

阅读 ioctl（2） 手册页和 ioccom.h 头 ，了解 ioctl 可以做什么以及如何在命令参数中编码命令和数据的背景信息。对命令中数据的方向和长度进行编码，允许 ioctl 系统调用处理程序代表驱动程序执行内存的大部分复制。

First, let's create a header file `sys/sys/p5d.h` to define our `ioctl` in:

首先，让我们创建一个头文件 `sys/sys/p5d.h` 来定义我们的 `ioctl` in in：

```
#if !defined(_SYS_P5D_H)
#define _SYS_P5D_H

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

struct p5d_status_params {
    uint    psp_is_num_waiting;
};

#define     P5D_IOC_STATUS     _IOR('5', 1, struct p5d_status_params)

#endif /* _SYS_P5D_H */
```

We'll use a `struct` argument (even though an integer would have been fine) to demonstrate how more complex data can be passed through this method.

我们将使用 `结构参数` （即使整数也可以）来演示如何通过此方法传递更复杂的数据。

Many `ioctl` commands use structs like this even where an integer might suffice to enable future extensibility without forcing lots of changes in code that uses them.

许多 `ioctl` 命令使用这样的结构，即使整数可能足以启用未来的可扩展性，而不会强制对使用它们的代码进行大量更改。

This `ioctl` command is defined above using the `_IOR` macro to take a `struct p5d_status_params *` argument. You would call this `ioctl` with code that looks like this:

上面定义了此 `ioctl` 命令，使用 `_IOR` 宏来获取 `struct p5d_status_params *` 参数。您可以使用如下所示的代码调用此 `ioctl`：

```
int fd;
struct p5d_status_params p;

fd = open("/dev/p5d", O_RDWR);
if (fd < 0)
  err(1, "open");

if (ioctl(fd, P5D_IOC_STATUS, &p))
  err(1, "ioctl");

printf("%d", p.psp_is_num_waiting);
/* etc */
```

(We'll come back to this example code when testing in a moment)

（稍后我们将在测试时回到这个示例代码）

Once you've created the header file, re-run `make includes` to install it on your live system:

创建头文件后，重新运行 `make includes` 以将其安装到您的实时系统上：

```
$ cd /usr/src/include
$ doas make -s includes


$ cd /usr/src/include
$ doas make -s 包括
```

Then we're ready to start on the implementation.

然后我们就可以开始实施了。

## Implementation 实现

Now let's implement the `ioctl` handler on the kernel side.

现在让我们在内核端实现 `ioctl` 处理程序。

Edit the function `p5dioctl()` in your `sys/dev/p5d.c`. The `cmd` argument is the magic number of the `ioctl` (e.g. `P5D_IOC_STATUS`). The `data` argument is a pointer to kernel memory containing the `ioctl` argument, which you can cast to a `struct p5d_status_params *`. The function should return an `errno`.

编辑 `sys/dev/p5d.c` 中的函数 `p5dioctl ()` 。 `cmd` 参数是 `ioctl` 的幻数（例如 `P5D_IOC_STATUS`）。`data` 参数是指向包含 `ioctl` 参数的内核内存的指针，您可以将其强制转换为 `结构 p5d_status_params *`。该函数应返回 `errno`。

Remember: `ioctls` defined with the `_IOR()` family of macros do not need `copyin()` or `copyout()` in their handlers in the kernel — the `ioctl` system call handler will do that for you.

请记住：使用 `_IOR ()` 系列宏定义的 `ioctl` 在内核中的处理程序中不需要 `copyin ()` 或 `copyout ()` — `ioctl` 系统调用处理程序将为您完成此作。

Set the `psp_is_num_waiting` field to `1` if there's a number waiting, otherwise set it to `0`. Don't forget to take any locks as needed.

如果有数字等待，则将 `psp_is_num_waiting` 字段设置为 `1`，否则将其设置为 `0`。不要忘记根据需要带上任何锁。

## Testing 测试

To test our `ioctl`, we'll need to write some C code.

为了测试我们的 `ioctl`，我们需要编写一些 C 代码。

Take the `xnum` tool from prac 4 part 1 and adjust it: add a new mode if the `-t` option is given which "tests" for whether a number is waiting and outputs `yes` or `no`.

从实践 4 第 1 部分中获取 `XNUMX` 个工具并对其进行调整：如果给出了 `-t` 选项，则添加一个新模式，该选项"测试"数字是否正在等待并输出 `是或 否` 。

Also adjust the existing `-s` and `-r` code to use the new interface (open the device and use `read(2)` or `write(2)`.

还要调整现有的 `-s` 和 `-r` 代码以使用新接口（打开设备并使用 `read (2)` 或 `write (2)` ）。

Run through some tests to make sure it all works.

运行一些测试以确保一切正常。

```
$ xnum -t
no
$ xnum -s 12345
$ xnum -s 124
xnum: write: Device busy
$ xnum -t
yes
$ xnum -r
12345
$ xnum -t
no
```

## Example Solution 示例解决方案

▼ 扩大

Our solution for the ioctl handler:

我们的 ioctl 处理程序解决方案：

```
int
p5dioctl(dev_t dev, u_long cmd, caddr_t data, int flag, struct proc *p)
{
    struct p5d_status_params *status;
    int eno = ENXIO;

    switch (cmd) {
    case P5D_IOC_STATUS:
        status = (struct p5d_status_params *)data;
        mtx_enter(&sc->sc_mtx);
        status->psp_is_num_waiting = (sc->sc_flags & SEND_WAITING) != 0;
        mtx_leave(&sc->sc_mtx);
        eno = 0;
        break;
    }

    return (eno);
}
```

And for the userland testing tool:

对于用户区测试工具：

详情参见testing_tools.c