Introduction 介绍

> If you are currently doing A1, then it is recommended that you attempt this prac after finishing A1. A1 does not rely on anything covered in this prac.
>
> 如果您目前正在学习 A1，那么建议您在完成 A1 后尝试此练习。A1 不依赖于本实践中涵盖的任何内容。

This week you learned about virtual memory, one of the biggest (and best) lies the OS tells to userland processes. In this practical we will start off by exploring some of the concepts from last week (locks and sleeping) and then have a go at manipulating process virtual memory in the kernel.

本周你了解了虚拟内存，这是作系统告诉用户空间进程的最大（也是最好的）谎言之一。在这个实践中，我们将从探索上周的一些概念（锁和睡眠）开始，然后尝试作进程 虚拟内存 在内核中。

We will be implementing two system calls: `sendnum` and `recvnum`, which together are used to transfer data between processes. One process calls `sendnum` with a number, and another calls `recvnum` to receive that number in turn.

我们将实现两个系统调用： `sendnum` 和 `recvnum`，它们一起用于在进程之间传输数据。一个进程使用一个数字调用 `sendnum`，另一个进程调用 `recvnum` 以依次接收该号码。

If you start a `recvnum` before the `sendnum`, the receiving process goes to sleep until `sendnum` is called. In the first part of the prac, the numbers will be given directly to the syscalls as arguments. In the second part, the `sendnum` syscall will instead take an address and the kernel will move the virtual memory allocation at that address from the sender to the receiver (without any copying).

如果在 `sendnum` 之前启动 `recvnum`，则接收进程将进入睡眠状态，直到调用 `sendnum`。在练习的第一部分中，数字将作为参数直接提供给系统调用。在第二部分中， `sendnum` 系统调用将采用一个地址，内核将该地址的虚拟内存分配从发送方移动到接收方（无需任何复制）。

Uninstalling Previous Syscalls

卸载以前的系统调用

> This step is very important if you're doing this prac after doing A1 (please don't do this prac while doing A1). You may optionally skip this step if you have not yet started A1 (though you should be starting A1 instead), but you must still complete this step after completing A1.
>
> 如果您在完成 A1 之后进行此练习，则此步骤非常重要（请不要在进行 A1 时进行此练习）。如果您尚未启动 A1，则可以选择跳过此步骤（尽管您应该改为启动 A1），但您仍必须在完成 A1 后完成此步骤。

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
```

Base Code Patch 基本代码补丁

Like previous practicals, this practical centres around creating new system calls. Since you've already been through that process before, we will be providing some boilerplate to help you get started more quickly and focus on the new parts of this prac.

与以前的实践一样，这种实践以创建新的系统调用为中心。由于您之前已经经历过该过程，因此我们将提供一些样板来帮助您更快地入门并专注于此实践的新部分。

The boilerplate patch for this prac is available at https://stluc.manta.uqcloud.net/comp3301/public/p3-base.patch.

该实践的样板补丁可在 https://stluc.manta.uqcloud.net/comp3301/public/p3-base.patch 获得。

You should apply this in a new branch named `p3` on top of the base tag:

您应该在基本标记顶部名为 `p3` 的新分支中应用它：

```
comp3301$ cd ~
comp3301$ ftp https://stluc.manta.uqcloud.net/comp3301/public/p3-base.patch
Trying 130.102.71.25...
Requesting https://stluc.manta.uqcloud.net/comp3301/public/p3-base.patch
100% |*****************************************************************************************|  8174
8174 bytes received in 0.00 seconds (16.18 MB/s)
comp3301$ cd /usr/src
comp3301$ git checkout -b p3 base
Switched to a new branch 'p3'
comp3301$ git am ~/p3-base.patch
Applying: COMP3301 2025 P3 Base Code Patch
```

This patch adds the two new system calls to `syscalls.master`, sets up a header file `sys/sendnum.h` defining them, adds them to `libc` and provides a skeleton source file `sys/kern/kern_sendnum.c`, just like you did in p2.

此补丁将两个新的系统调用添加到 `syscalls.master`，设置一个定义它们的头文件 `sys/sendnum.h`，将它们添加到 `libc` 并提供一个骨架源文件 `sys/kern/kern_sendnum.c`，就像您在 p2 中所做的那样。

The two system calls are defined initially as:

这两个系统调用最初定义为：

```
int sendnum(int num);
int recvnum(int *num);
```

Implementation 1 实现 1

Time to implement `sys_sendnum()` and `sys_recvnum()`!

是时候实现 `sys_sendnum（）` 和 `sys_recvnum（）` 了!

> `sendnum` takes the number to be sent as its only argument. The number given to `sendnum` is kept by the kernel until a `recvnum` call is made to pick it up. Each `sendnum` call is matched with exactly one `recvnum` call (the number is not sent to multiple receivers). A call to `sendnum` returns immediately, whether a process is waiting in `recvnum` or not. If a process has already called `sendnum` and there is a number "waiting" that hasn't been read by a `recvnum` yet, then `sendnum` should return `-1` and set `errno` to `EBUSY`.
>
> `sendnum` 将要发送的数字作为其唯一参数。给 `sendnum` 的数字由内核保留，直到进行 `recvnum` 调用来获取它。每个 `sendnum` 调用恰好与一个 `recvnum` 调用匹配（该号码不会发送到多个接收方）。对 `sendnum` 的调用会立即返回，无论进程是否在 `recvnum` 中等待。如果进程已经调用了 `sendnum`，并且有一个数字“等待”，但 `recvnum` 尚未读取，则 `sendnum` 应返回 `-1` 并将 `errno` 设置为 `EBUSY`。

> `recvnum` takes a pointer to an integer, and writes the received number there. If a number is already waiting (a call to `sendnum` has been made already and not yet picked up) then it returns immediately, otherwise it will block (sleep) until a number is available. The sleep must be *interruptible* (i.e. you can kill or Ctrl + C a process blocked in `recvnum` and it exits normally).
>
> `recvnum` 接受指向整数的指针，并将接收到的数字写入其中。如果一个号码已经在等待（已经调用了 `sendnum` 但尚未被接听），那么它会立即返回，否则它将阻塞（睡眠），直到号码可用。睡眠必须*是可中断的*（即您可以杀死或 Ctrl + C 在 `recvnum` 中阻塞的进程，它会正常退出）。

The prototypes for the two system call handlers are ready for you to fill in, in `/usr/src/sys/kern/kern_sendnum.c`

两个系统调用处理程序的原型已准备好供您填写，在 `/usr/src/sys/kern/kern_sendnum.c`

Some advice: 一些建议：

- The processes calling `sendnum` and `recvnum` don't have to be related (not parent and child) and don't have to be the same program. You shouldn't assume anything about them.

  调用 `sendnum` 和 `recvnum` 的进程不必相关（不是父级和子级），也不必是同一个程序。你不应该对他们有任何假设。

- A simple solution like stashing the number to be send in a global variable in `sendnum()` and pull it out in `recvnum()` to give to `copyout(9)` will work, this doesn't have to be elaborate.

  一个简单的解决方案，例如将要发送的数字隐藏在 `sendnum（）` 中的全局变量中，然后将其提取到 `recvnum（）` 中以提供给 `copyout（9）` 将起作用，这不必很详细。

- Remember though: the two system calls are entered by different unrelated processes (threads), so if they manipulate shared data (e.g. a global variable), it needs to be protected by a lock. We recommend a simple `mutex(9)`.

  但请记住：这两个系统调用是由不同的不相关的进程（线程）输入的，因此如果它们作共享数据（例如全局变量），则需要用锁保护它。我们推荐一个简单的 `互斥锁（9）。`

- Interruptible sleep is achieved by using `PCATCH` (see `tsleep(9)`)

  可中断睡眠是通过使用 `PCATCH` 实现的（参见 `tsleep（9）`）

- Don't forget: a call to `tsleep(9)` or `msleep(9)` should always be in a loop.

  不要忘记：对 `tsleep（9）` 或 `msleep（9）` 的调用应该始终在循环中。

```c
enum sendnum_state { EMPTY, FULL };

static struct mutex sendnum_mtx = MUTEX_INITIALIZER(IPL_NONE);
static enum sendnum_state sendnum_state = EMPTY;
static int sendnum_num;

int
sys_sendnum(struct proc *p, void *v, register_t *retval)
{
    struct sys_sendnum_args *uap = v;
    int eno;

    mtx_enter(&sendnum_mtx);
    if (sendnum_state != EMPTY) {
        eno = EBUSY;
        goto out;
    }

    sendnum_num = SCARG(uap, num);
    sendnum_state = FULL;
    wakeup_one(&sendnum_state);

    eno = 0;

out:
    mtx_leave(&sendnum_mtx);
    return (eno);
}

int
sys_recvnum(struct proc *p, void *v, register_t *retval)
{
    struct sys_recvnum_args *uap = v;
    int eno;

    mtx_enter(&sendnum_mtx);
    while (sendnum_state != FULL) {
        eno = msleep(&sendnum_state, &sendnum_mtx, PCATCH,
            "recvnum", 0);
        if (eno != 0)
            goto out;
    }

    eno = copyout(&sendnum_num, SCARG(uap, num), sizeof (int));
    if (eno != 0)
        goto out;
    sendnum_state = EMPTY;
    sendnum_num = 0;

    eno = 0;

out:
    mtx_leave(&sendnum_mtx);
    return (eno);
}
```

Compiling 编译

After implementing your code, you will need to recompile and install the kernel, includes, `ld.so` and `libc`:

实现代码后，您需要重新编译并安装内核、includes、 `ld.so` 和 `libc`：

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

A userland tool for testing your system calls is also included in `usr.bin/xnum`, which you can install:

`usr.bin/XNUMX` 中还包含用于测试系统调用的用户空间工具，您可以安装：

```
comp3301$ cd /usr/src/usr.bin/xnum
comp3301$ make obj
comp3301$ make
comp3301$ doas make install
```

Testing 1 测试 1

You can test your system call implementation by using the `xnum` utility in the boilerplate. Start two terminals (e.g., different SSH sessions, or panes in a `tmux`) and run:

您可以使用样板中的 `xnum` 实用程序来测试您的系统调用实现。启动两个终端（例如，不同的 SSH 会话或 `tmux` 中的窗格）并运行：

```
comp3301$ xnum -r    | comp3301$ xnum -s 1234
1234                 | comp3301$
comp3301$            |


comp3301$ XNUMX -r  |comp3301$ XNUMX -S 1234
1234  |comp3301$
comp3301$ |
```

You should see the `xnum -r` block until you run `xnum -s`, and then output whatever number you gave to `-s`. You should also be able to use `Ctrl + C` to interrupt the `xnum -r`.

您应该看到 `xnum -r` 块，直到运行 `xnum -s`，然后输出您给 `-s` 的任何数字。您还应该能够使用 `Ctrl + C` 打断 `xnum -r`。

What other testing can you think of? Are there other errors which the system calls are specified to return? How can you trigger them?

您还能想到哪些其他测试？系统调用是否指定返回其他错误？你怎么能触发它们？

Experiments 实验

Take some time now to do a few experiments, too. Try these out, so you know what these mistakes look like in future if you make them.

现在也花点时间做一些实验。尝试一下这些，这样你就知道如果你犯了这些错误，将来会是什么样子。

- What happens if you remove `PCATCH` from your `tsleep` or `msleep` call?

  如果从 `tsleep` 或 `msleep` 调用中删除 `PCATCH`，会发生什么情况?
- What happens if you don't use a lock like a mutex in this code? Can you make it misbehave without one?

  如果在此代码中不使用互斥锁等锁会发生什么? 你能让它在没有一个的情况下行为不端吗?
- What happens if you don't release your mutex at the end of a system call?

  如果在系统调用结束时不释放互斥锁会怎样?

You might want to make a disk snapshot of your VM before these experiments.

在进行这些试验之前，可能需要创建 VM 的磁盘快照。

Interfaces 接口

For the next part of this practical, we will make use of the kernel UVM (Unified Virtual Memory) framework.

在本实践的下一部分，我们将使用内核 UVM（统一虚拟内存）框架。

Each process on our OpenBSD system has its own independent virtual memory space, so most IPC primitives like our `sendnum`/`recvnum` system calls require copying data around. Typically, two copies are required (one from userland virtual memory in the sending process into a buffer in kernel memory, and then again from kernel memory out into the receiving process' userland virtual memory).

我们 OpenBSD 系统上的每个进程都有自己独立的虚拟内存空间，因此大多数 IPC 原语（例如我们的 `sendnum`/`recvnum` 系统调用）都需要复制数据。通常，需要两个副本（一个从发送进程中的用户区虚拟内存到内核内存中的缓冲区，然后再次从内核内存输出到接收进程的用户区虚拟内存中)。

Instead of copying, however, we can also "steal" memory from the sender process and inject it into the receiver. This involves creating separate page table entries for each process which are backed by the same physical page (or "page frame" in the textbook terminology). This, of course, means that the minimum amount of data we can move is one page (usually 4096 bytes on the x86 architecture).

然而，除了复制之外，我们还可以从发送方进程中“窃取”内存并将其注入接收方。这涉及为每个进程创建单独的页表条目，这些条目由相同的物理页面（或教科书术语中的“页框”）支持。当然，这意味着我们可以移动的最小数据量是一页（在 x86 架构上通常为 4096 字节)。

We will start by adjusting our system calls' arguments. The `sendnum` call will now take a pointer and length (in the sender's virtual address space), while `recvnum` fills out a matching pointer and length for the receiving process:

我们将从调整系统调用的参数开始。`sendnum` 调用现在将采用指针和长度（在发送方的虚拟地址空间中)，而 `recvnum` 为接收进程填写匹配的指针和长度:

```
int sendnum(void *addr, size_t len);
int recvnum(void **addr, size_t *len);
```

To simplify our implementation, let's also make one change to the semantics of how the two system calls rendezvous:

为了简化我们的实现，我们还对两个系统调用 rendezvous 的语义进行一项更改：

- The `sendnum` system call will now block until a matching `recvnum` call is made. If another `sendnum` call is currently blocked waiting, then `sendnum` returns `-1` and sets `errno` to `EBUSY`.

  `sendnum` 系统调用现在将阻塞，直到发出匹配的 `recvnum` 调用。如果当前阻止另一个 `sendnum` 调用等待，则 `sendnum` 返回 `-1` 并将 `errno` 设置为 `EBUSY`。

This change allows us to avoid unmapping the page from the sender process until the receiver is available to pick it up, which simplifies what we need to do in UVM, as you'll see in a moment.

此更改使我们能够避免从发送方进程中取消映射页面，直到接收方可以接收它，这简化了我们在 UVM 中需要执行的作，正如您稍后将看到的那样。

Make this change in the definition of these two system calls in your repository now: like you did in prac 2, edit `syscalls.master`, re-run `make syscalls` (in `sys/kern`), edit `sys/sendnum.h` , rebuild and reinstall the kernel, reinstall the includes, and rebuild and reinstall ld.so and libc.

现在在存储库中对这两个系统调用的定义进行此更改：就像您在实践 2 中所做的那样，编辑 `syscalls.master`，重新运行 `make syscalls`（在 `sys/kern` 中），编辑 `sys/sendnum.h`，重建并重新安装内核，重新安装 includes，重建并重新安装 ld.so 和 libc。

Next, we'll update `xnum.c` to use the new interface before diving into the actual implementation.

接下来，我们将更新 `xnum.c` 以使用新接口，然后再深入实际实现。

Updating Xnum 更新 Xnum

Let's update `xnum.c` so we understand how these new system calls will be used. Rather than an `int`, we need to give a pointer to `sendnum`, and that pointer needs to be page-aligned, with nothing else on the same page with it.

让我们更新 `xnum.c`，以便我们了解如何使用这些新的系统调用。我们需要给出一个指向 `sendnum` 的指针，而不是 `int`，并且该指针需要与页面对齐，没有其他任何东西与它在同一页面上。

We'll make our pointer point at a struct:

我们将指针指向一个结构体：

```
struct xnum_page {
    uint        xp_magic;
    int     xp_num;
};
```

This struct definition contains two fields: a "magic" number (so that we can check it to make sure everything is working properly) and the actual number itself that we wanted to communicate.

这个结构定义包含两个字段：一个"魔术"数字（这样我们就可以检查它以确保一切正常工作）和我们想要传达的实际数字本身。

You can choose what magic number you want to use, but we suggest making it something unique that won't be there by accident (so avoid zero or 1 or repeating patterns).

您可以选择要使用的幻数，但我们建议将其制作为独特的数字，不会偶然出现（因此请避免 0 或 1 或重复模式）。

To allocate an entire page just for our struct, we will use the `mmap(2)` system call. As well as mapping files into memory, `mmap` can also be used to allocate pages of "anonymous" virtual memory for our process. The basic idea looks like this:

为了仅为我们的结构分配整个页面，我们将使用 `mmap（2）` 系统调用。除了将文件映射到内存中外，`mmap` 还可用于为我们的进程分配"匿名"虚拟内存页。基本思想如下所示：

```c
struct xnum_page *p;
p = mmap(
    NULL,                   /* we don't care what address it ends up at */
    sizeof(struct xnum_page),
    PROT_READ | PROT_WRITE,     /* no code being put there, don't need PROT_EXEC */
    MAP_PRIVATE | MAP_ANON,     /* give us "anonymous" private memory */
    -1,                     /* no file descriptor associated with it */
    0);
if (p == MAP_FAILED)
    err(1, "mmap");
p->xp_magic = ...;
p->xp_num = ...;
```

Then we can use `p` as the first argument to our modified `sendnum` system call:

然后我们可以使用 `p` 作为修改后的 `sendnum` 系统调用的第一个参数：

```c
if (sendnum(p, sizeof(struct xnum_page)))
    err(1, "sendnum");
```

On the receiving side, we don't need to use `mmap` or anything fancy: we just give a pointer to a pointer for it to place the final address in, and check the magic value before printing. It should look something like:

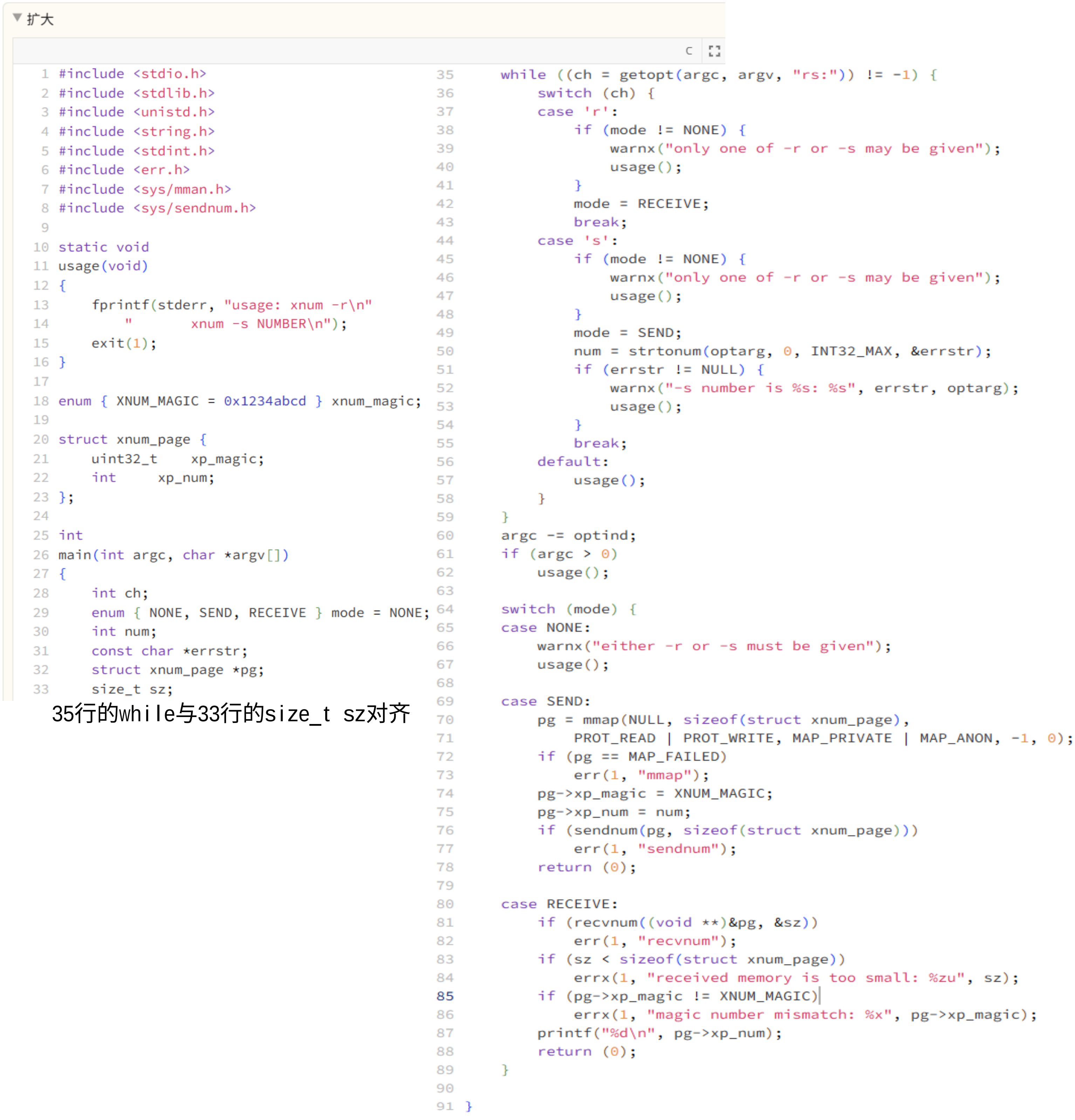
在接收端，我们不需要使用 `mmap` 或任何花哨的东西：我们只需给一个指针，让它放置最终地址，并在打印前检查魔术值。它应该看起来像：

```c
if (recvnum((void **)&p, &len))
    err(1, "recvnum");
if (len < sizeof(struct xnum_page))
    errx(1, "received memory too small");
if (p->xp_magic != ...)
    errx(1, "magic number invalid");
printf("%d\n", p->xp_num);
```

Make these modifications to your `xnum.c` now. Make sure it compiles at least, and then you can come back to any mistakes in it after you've implemented the system calls.

立即对您的 `xnum.c` 进行这些修改。确保它至少可以编译，然后您可以在实现系统调用后返回其中的任何错误。

Solution 溶液



```c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <string.h>
 5 #include <stdint.h>
 6 #include <err.h>
 7 #include <sys/mman.h>
 8 #include <sys/sendnum.h>
 9
10 static void
11 usage(void)
12 {
13     fprintf(stderr, "usage: xnum -r\n"
14         "       xnum -s NUMBER\n");
15     exit(1);
16 }
17
18 enum { XNUM_MAGIC = 0x1234abcd } xnum_magic;
19
20 struct xnum_page {
21     uint32_t    xp_magic;
22     int         xp_num;
23 };
24
25 int
26 main(int argc, char *argv[])
27 {
28     int ch;
29     enum { NONE, SEND, RECEIVE } mode = NONE;
30     int num;
31     const char *errstr;
32     struct xnum_page *pg;
33     size_t sz;
35         while ((ch = getopt(argc, argv, "rs:")) != -1) {
36             switch (ch) {
37             case 'r':
38                 if (mode != NONE) {
39                     warnx("only one of -r or -s may be given");
40                     usage();
41                 }
42                 mode = RECEIVE;
43                 break;
44             case 's':
45                 if (mode != NONE) {
46                     warnx("only one of -r or -s may be given");
47                     usage();
48                 }
49                 mode = SEND;
50                 num = strtonum(optarg, 0, INT32_MAX, &errstr);
51                 if (errstr != NULL) {
52                     warnx("-s number is %s: %s", errstr, optarg);
53                     usage();
54                 }
55                 break;
56             default:
57                 usage();
58             }
59         }
60         argc -= optind;
61         if (argc > 0)
62             usage();
63
64         switch (mode) {
65         case NONE:
66             warnx("either -r or -s must be given");
67             usage();
68
69         case SEND:
70             pg = mmap(NULL, sizeof(struct xnum_page),
71                 PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
72             if (pg == MAP_FAILED)
73                 err(1, "mmap");
74             pg->xp_magic = XNUM_MAGIC;
75             pg->xp_num = num;
76             if (sendnum(pg, sizeof(struct xnum_page)))
77                 err(1, "sendnum");
78             return (0);
79
80         case RECEIVE:
81             if (recvnum((void **)&pg, &sz))
82                 err(1, "recvnum");
83             if (sz < sizeof(struct xnum_page))
84                 errx(1, "received memory is too small: %zu", sz);
85             if (pg->xp_magic != XNUM_MAGIC)
86                 errx(1, "magic number mismatch: %x", pg->xp_magic);
87             printf("%d\n", pg->xp_num);
88             return (0);
89         }
90
91 }
```

35行的while与33行的size_t sz对齐

Implementation 2 实施 2

We recommend approaching the system call implementation in two parts:

我们建议分两部分进行系统调用实现：

- Get the rendezvous working correctly (both `sendnum` and `recvnum` block until the other call is made, and then both finish at the same time).

  让交会正常工作（`sendnum` 和 `recvnum` 块，直到进行另一个调用，然后同时完成）。
- Then, implement the actual transfer of pages.

  然后，实现页面的实际传输。

The rendezvous should look like an expanded version of what you did in the first part (but now both callers sleep, and you might need some additional global variables or flags).

会合应该看起来像您在第一部分中所做的事情的扩展版本（但现在两个调用者都处于睡眠状态，您可能需要一些额外的全局变量或标志）。

For handling the actual transfer of page mappings, you should use the function `uvm_share`. There are other ways to do it, but `uvm_share` is one of the least perilous. UVM is deep in what is sometimes colloquially called the "animal brain" of the kernel, so documentation is sparse and unreliable. Use `cscope` or a similar tool to find the definition of `uvm_share` and have a look at it. What arguments does it need? Have a quick read of either its implementation (and any comments above it), or a place that currently calls it. Are any locks needed before calling it (e.g. the UVM `vm_map_lock()`), or does it take them itself?

要处理页面映射的实际传输，应使用函数 `uvm_share`。还有其他方法可以做到这一点，但 `uvm_share` 是最不危险的方法之一。UVM 深入于有时俗称的内核"动物大脑"中，因此文档稀疏且不可靠。使用 `cscope` 或类似工具查找 `uvm_share` 的定义并查看它。它需要什么论据？快速阅读它的实现（以及它上面的任何注释），或者当前调用它的地方。在调用它之前是否需要任何锁（例如 UVM `vm_map_lock（）`），或者它自己需要它们吗？

In the contact you should have seen where the `vm_map` for a process resides. You'll need that for two of the most important arguments. Also amongst the information it needs is a pointer in the destination address space that's not currently in use. To get this, we recommend looking at the implementation of the `mquery(2)` system call in `sys_mquery`. It has a convenient two-line incantation you can use to get a pointer to an unused part of a UVM map.

在联系人中，您应该已经看到了流程的 `vm_map` 所在的位置。您将需要它来讨论两个最重要的论点。它需要的信息中还有目标地址空间中当前未使用的指针。为此，我们建议查看 `sys_mquery` 中 `mquery（2）` 系统调用的实现。它有一个方便的两行咒语，您可以使用它来获取指向 UVM 地图中未使用部分的指针。

After calling `uvm_share`, you should also remove the mapping from the original sender process (so it can't be changed after the fact). You can figure out how this is done by looking at the implementation of the `munmap(2)` system call in `sys_munmap`. Don't forget to take note of locking requirements and ordering when copying from these system calls.

调用 `uvm_share` 后，还应从原始发送方进程中删除映射（因此事后无法更改）。您可以通过查看 `sys_munmap` 中 `munmap（2）` 系统调用的实现来了解这是如何完成的。从这些系统调用复制时，不要忘记注意锁定要求和排序。

Testing 2 测试 2

You can test your system call implementations as in the first part of this prac, by running the updated `xnum` tool. It should behave as it did before (but now the sender blocks until the transaction is completed).

您可以通过运行更新的 `xnum` 工具来测试您的系统调用实现，如本练习的第一部分所示。它的行为应该像以前一样（但现在发送者会阻止交易完成）。

You should also consider adding some additional `printf` calls to `xnum` to print out the pointers being sent and received. Are the pointers the same on the sender and receiver? Do they change from run to run? Why or why not?

您还应该考虑向 `xnum` 添加一些额外的 `printf` 调用以打印出正在发送和接收的指针。发送方和接收方的指针是否相同？它们会因跑步而变化吗？为什么或为什么不？

Then try providing some invalid arguments to `sendnum`. What happens?

然后尝试向 `sendnum` 提供一些无效的参数。会发生什么？

Experiments 实验

Let's also try some more in-depth experiments:

我们也尝试一些更深入的实验：

- Make the address of the mapping in the destination process predictable (by e.g. replacing your call to `uvm_map_hint` with a fixed value). Does this work? What kinds of addresses can you choose? Are some invalid?

  使目标进程中的映射地址可预测（例如，通过将对 `uvm_map_hint` 的调用替换为固定值）。这有效吗？您可以选择哪些类型的地址？有些是无效的吗？

- Take out the code which removes the mapping from the sending process. Then, after `sendnum` returns, alter something in the data. Can you observe this alteration in the receiver? (You might need to loop for a little while)

  取出从发送过程中删除映射的代码。然后，在 `sendnum` 返回后，更改数据中的某些内容。你能观察到接收器的这种变化吗？（您可能需要循环一段时间）

Think also about how you might modify this design to accommodate `sendnum` returning immediately (like the original design in the first part). What would you use instead of or as well as `uvm_share`? Where and how would you keep the page while waiting for the receiver?

还要考虑如何修改此设计以适应立即返回的 `sendnum`（就像第一部分中的原始设计一样）。你会用什么来代替或以及 `uvm_share`？在等待接收者时，您将在哪里以及如何保存页面？