Xv6 实验报告

2253878 方易

Lab1 Utilities

Boot xv6 (easy)

实验目的

启动 xv6 shell

实验步骤

源代码下载：在终端中使用 git 命令 git clone git://g.csail.mit.edu/xv6-labs-2021 ，克隆仓库源代码。

构建并运行 xv6：使用 make qemu 进行构建。

实验中遇到的问题和解决方法

对于实验运行环境的选择有很多，可以考虑在本机下载 linux 系统，或者使用虚拟机进行配置。本实验选择使用 Vmware 虚拟机应用，使用 Ubuntu 系统作为系统环境。

实验心得

本实验比较简单和基础，锻炼了我对于 linux 系统的操作，使我对其理解增加，并对 linux 系统更加感兴趣。

sleep (easy)

实验目的

实现 sleep 操作。

实验步骤

1. 阅读参考书第一章节，学习 fork wait 等函数，并参考其它源文件，熟悉文件格式。

2. 使用 vscode 打开文件夹并新建 sleep.c 文件。

3. 在主函数的实现中，通过 argc 和 argv 传递命令行的参数个数和内容。故应判断参数个数是否少于 2，若是，则打印错误信息；若不是，则将调用 sleep 实现功能。

4. 代码完成后，将 sleep.c 文件加入到 Makefile 的 UPROGS 中。其格式为 $U/_sleep\

5. 使用./grade-lab-util sleep 进行打分。

```
df@df-virtual-machine:~/OSlab/xv6-labs-2021$ ./grade-lab-util sleep
derbird 邮件/新闻 kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.8s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
```

实验中遇到的问题和解决方法

在代码编写过程当中，首先遇到了不知道如何编写的问题。通过观察学习 kernel 的相关代码，得以明白该操作的实现逻辑。

此外，通过学习得知命令行的参数可以通过主函数的 argc 和 argv 参数传入。

实验心得

在编写自己的代码之前，应当学习相应的前置知识，并且阅读已有的工程代码，学习和模仿其进行自己的代码实现。

pingpong (easy)

实验目的

编写一个程序，使用 UNIX 系统调用通过一对管道（每个方向一个）在两个进程之间

“乒乓”一个字节。父进程应向子进程发送一个字节；孩子应该打印“pid：收到 ping"，其中 pid 是它的进程 ID，将管道上的字节写入父进程，然后退出；父级应该从子级读取字节，打印“pid：收到 pong"，然后退出。

实验步骤

1. 学习相关库函数
2. 使用 pipe 创建管道。
3. 使用 fork 创建子管道。
4. 使用 read 从管道中读取数据，使用 write 向管道中写入数据。
5. 使用 getpid 查找调用进程的进程 ID。
6. 将程序添加到 Makefile 的 UPROGS 中，并评分。

```
df@df-virtual-machine:~/OSlab/xv6-labs-2021$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (1.1s)
```

实验中遇到的问题和解决方法

在代码编写过程中，开始不理解 fork 函数的用处和用法，后来重新阅读了 xv6 book，对其有了深入的认识之后，再阅读了 fork 的源代码，明白可以使用 fork 的返回值来判断当前进程。

实验心得

本实验是对于几个基本的系统函数以及进程知识的考察，通过这个任务，我学习了不同进程之间通信的方法，熟悉了管道运算符的使用，为实现操作系统打下基础。

primes (moderate)/(hard)

实验目的

学习使用 pipe 和 fork 来设置管道。第一个进程将数字 2 到 35 输入管道。对于每个素数创建一个进程，该进程通过一个管道从左边的邻居读取数据，并通过另一个管道向右边的邻居写入数据。由于 xv6 的文件描述符和进程数量有限，第一个进程可以在 35 处停止。

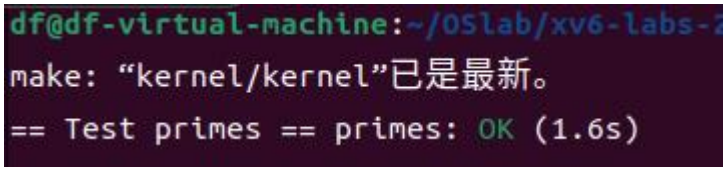
实验步骤

1. 创建父进程，父进程将数字 2 到 35 输入管道，此时不必创建其后所有进程，每一步迭代更新一对相对的父子进程（仅在需要时才创建管道中的进程）。

2. 对于 2-35 中的每个素数创建一个进程，进程之间需要进行数据传递：该进程通过一个管道从左边的父进程读取数据，并通过另一个管道向右边子进程写入数据。

3. 完成数据传递或更新时，需要及时关闭一个进程不需要的文件描述符

4.将程序添加到 Makefile 的 UPROGS 中，并评分。



实验中遇到的问题和解决方法

在代码编写过程中，我对于其编写逻辑和数据传递的逻辑并不理解，尤其是对于父子进程之间的数据传递没有头绪。后来考虑到应该用管道操作来实现这一部分，问题就迎刃而解了。另外，本部分还要考虑质数的判断逻辑，在这一部分可以选取复杂度较低的算法。

实验心得

本实验是对于几个父子进程之间的关系以及数据传递等知识点的考察，同时本实验还加深了我对 fork 操作以及管道操作的理解，让我对父子进程的理解更加深刻。除此之外，还学习到了快速判断素数的算法。最后，还要注意 xv6 资源的有限性。

find (moderate)

实验目的

学习并编写一个简单版本的 UNIX 查找程序：程序应当实现查找目录树中带有特定名称的所有文件。

实验步骤

1. 首先查看 user/ls.c 以了解如何读取目录。 user/ls.c 中包含一个 fmtname 函数，用于格式化文件的名称。它通过查找路径中最后一个 '/' 后的第一个字符来获取文件的名称部分。如果名称的长度大于等于 DIRSIZ，则直接返回名称。否则，将名称拷贝到一个静态字符数组 buf 中，并用空格填充剩余的空间，保证输出的名称长度为 DIRSIZ。

2. 使用递归进行查找

3.将程序添加到 Makefile 的 UPROGS 中，并评分。

```
df@df-virtual-machine:~/OSlab/xv6-labs-2021$ ./grade-lab-util find
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK
.6s)
== Test find, recursive == find, recursive: OK (1.6s)
```

实验中遇到的问题和解决方法

在代码编写过程中，ls.c 程序只能提供基本的文件和目录信息，并不包含递归遍历子目录的功能。因此如果我不对 find 函数进行递归遍历，它只能查找指定目录下的直接子文件和子目录，而无法继续向下递归地查找子目录中的文件，这也会导致我在测试时无法得到全面的查找结果。

实验心得

本实验是对于目录树递归遍历查找等知识点的考察，通过编写 find.c 程序，我深入理解了文件系统中目录和文件的关系，以及如何通过系统调用和文件系统接口来访问和操作文件。

xargs (moderate)

实验目的

编写一个 UNIX xargs 程序的简单版本：从标准输入中读取行，并为每一行运行一个命令，将行作为参数提供给命令。

实验步骤

1. 读取用户命令

2. 使用 fork 和 exec 在每一行输入上调用命令：在父进程中使用 wait 等待子进程完成命令。在子进程中修改 exec 的调用方法，在执行命令时将每个参数逐个提供给 exec 函数以符合实验要求，即不一次性向命令提供多个参数，而是每次提供一个参数。

3.将程序添加到 Makefile 的 UPROGS 中，并评分。

```
df@df-virtual-machine:~/OSlab/xv6-labs-2021$ ./grade-lab-util xargs
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (2.3s)
```

实验中遇到的问题和解决方法

在代码编写过程中，首先需要确定如何将用户的命令行中的各个参数传入并拆分，去除空格，存入适当的数据结构中。在这一部分容易出错，可以用一些提示语句进行当前信息的显示。

实验心得

本实验是对于命令行传参和处理等技术的考察。在这个实验当中，我学会了如何处理命令行中的字符串，同时实践了在子进程当中执行外部程序的方法。

Lab2 Utilities

System call tracing (moderate)

实验目的

在此作业中，将添加系统调用跟踪功能，该功能可以在调试后续实验时为您提供帮助。将创建一个新的 trace 将控制跟踪的系统调用。它应该采用一个参数，一个整数“掩码”，其位指定要跟踪哪些系统调用。如果掩码中设置了系统调用的编号，则必须修改 xv6 内核，以便在每个系统调用即将返回时打印一行。该行应包含进程 ID、系统调用名称和返回值。这 trace 系统调用应该启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

实验步骤

1. 将程序添加到 Makefile 的 UPROGS 中
2. 定义 trace 系统调用的原型, entry 和系统调用号，在 user/user.h 中添加 trace 系统调用原型；在 user/usys.pl 脚本中添加 trace 对应的 entry；在 kernel/syscall.h 中添加 trace 的系统调用号。
3. 编写 trace 的系统调用函数
4. 修改 fork 函数
5. 修改 syscall 函数
6. 添加 trace 调用
7. 实验结果

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
```

```
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
```

```
$ grep hello README
```

```
$ trace 2 usertests forkforkfork
usertests starting
8: syscall fork -> 9
test forkforkfork: 8: syscall fork -> 10
10: syscall fork -> 11
11: syscall fork -> 12
11: syscall fork -> 13
12: syscall fork -> 14
11: syscall fork -> 15
12: syscall fork -> 16
13: syscall fork -> 17
11: syscall fork -> 18
12: syscall fork -> 19
11: syscall fork -> 20
13: syscall fork -> 21
12: syscall fork -> 22
```

```
df@df-virtual-machine:~/OSlab/xv6-labs-2021$ ./grade-lab-syscall trace
make: “kernel/kernel”已是最新。
== Test trace 32 grep == trace 32 grep: OK (1.8s)
    (Old xv6.out.trace_32_grep failure log removed)
== Test trace all grep == trace all grep: OK (0.9s)
    (Old xv6.out.trace_all_grep failure log removed)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (24.1s)
    (Old xv6.out.trace_children failure log removed)
```

实验中遇到的问题和解决方法

在代码编写过程中，在系统调用的实现中，需要正确地传递参数和数据，包括用户态和内核态之间的切换。起初这让我很困惑，但是我通过仔细阅读系统调用相关的文档和源代码，找到了正确使用数据传递方法，如使用合适的系统调用参数读取函数等。

实验心得

从这个实验中，我学到了如何在 xv6 内核中添加新的系统调用，如何修改进程控制块以支持跟踪掩码，并且理解了如何在内核中实现系统调用的功能。此外，我还了解了如何在用户级程序中调用新增的系统调用，并在实验中验证系统调用的正确性。

Sysinfo (moderate)

实验目的

在此作业中，将添加一个系统调用， sysinfo ，收集有关正在运行的系统的信息。系统调用需要一个参数：一个指向 struct sysinfo （ kernel/sysinfo.h ）。内核应该填写这个结构体的字段： freemem 字段应设置为可用内存的字节数，并且 nproc 字段应设置为其进程的数量 state 不是 UNUSED 。

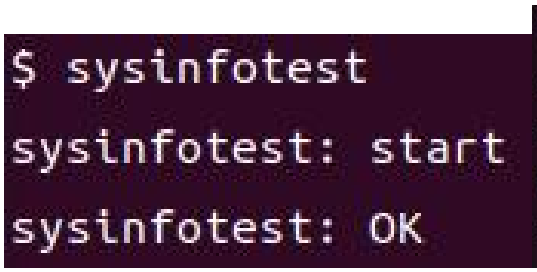
实验步骤

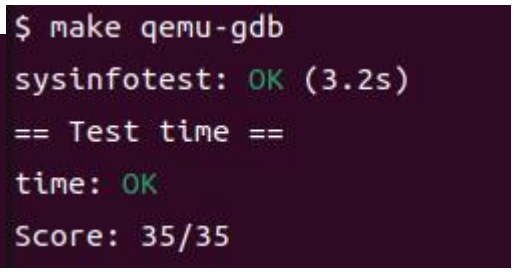
1. 将程序添加到 Makefile 的 UPROGS 中

2. 定义 sysinfo 系统调用的原型，需要预先声明 struct sysinfo

3. 编写函数，完成可用内存和进程数收集

4. 在 kernel/defs.h 中添加函数原型





实验中遇到的问题和解决方法

在代码编写过程中，需要考虑获取可用内存和进程的数量，因此如何设计这两个函数就是一个重要的课题。参考内核的函数可知，系统通过一个链表维护未使用的内存，故可通过这一点来获取内存数。

实验心得

本实验是对于系统运行信息收集的考察，通过这个实验，我了解了系统内核中相应源代码中的数据结构，以及对应的表示的信息。

Lab3 Page Tables

Speed up system calls (easy)

实验目的

一些操作系统（如 Linux）通过在用户空间和内核之间共享只读区域中的数据来加快某些系统调用的速度。这样，在执行这些系统调用时就不需要内核交叉了。本实验旨在学习如何在页表中插入映射，首先需要在 xv6 中的 getpid() 系统调用中实现这一优化。

通过在用户空间和内核之间共享一个只读区域中的数据，来加速特定的系统调用。

实验步骤

1. 在 kernel/proc.h 的 proc 结构体中添加指针来保存这个共享页面的地址。并在 proc.c 文件的 allocproc() 函数中，为每个新创建的进程分配一个只读页
2. 将 struct usyscall 结构放置在只读页的开头，并初始化其存储当前进程的 PID
3. 在 kernel/proc.c 的 freeproc() 函数中，释放之前分配的只读页
4. 运行并评分

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
```

实验中遇到的问题和解决方法

在代码编写过程中，对于映射的创建和管理十分重要。需要确保在 allocproc() 和 freeproc() 中正确创建和释放页表映射，还需要及时取消映射。否则会导致系统调用的失败。

实验心得

本实验是对于页表映射的考察，通过这次实验，我理解了内核性能优化的意义和实现。

具体而言，可以将数据放置在只读页中，实现调用的加速。

Print a page table (easy)

实验目的

深入理解 RISC-V 页表的结构和内容，并提供一个打印页表的函数 vmprint() 。通过这个实验，实现可视化页表的布局，了解页表的层次结构以及如何将虚拟地址映射到物理地址。

实验步骤

1. 在 kernel/defs.h 中定义 vmprint 的原型

2. 使用位操作和宏定义 (位掩码和位移等) 从页表项 (PTE) 中提取所需的信息，如 PTE 的有效位、权限位和物理地址。

3. 在 kernel/vm.c 中定义一个函数打印每个 PTE 的信息，并递归地调用 vmprint() 打印下一级的页表

4. 在 exec.c 的 exec() 函数中执行该函数

5. 运行并评分

```
hart 2 starting
hart 1 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
....0: pte 0x0000000021fda401 pa 0x0000000087f69000
......0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
......1: pte 0x0000000021fda00f pa 0x0000000087f68000
......2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
....511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
......509: pte 0x0000000021fdd813 pa 0x0000000087f76000
......510: pte 0x0000000021fddc07 pa 0x0000000087f77000
......511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

实验中遇到的问题和解决方法

在代码编写过程中，需要输出页表的结构，因此在格式化输出上需要注意语法。可以使用%p 来打印十六进制的地址。

实验心得

本实验是对于页表层次结构输出的考察，从根页表开始，逐级向下指向不同级别的页表页，最终到达最底层的页表页，其中包含了实际的物理页框映射信息。此外，这个实验加深了我对页表结构的理解，并且学会了如何在内核中操作位操作和宏定义，以及如何通过递归遍历页表来打印出整个页表的内容。

Detecting which pages have been accessed (hard)

实验目的

一些垃圾回收器（一种自动内存管理形式）可以从哪些页面已被访问（读取或写入）的信息中获益。在这部分实验中，您将为 xv6 添加一项 pgaccess()系统调用，通过检查 RISC-V 页表中的访问位来检测并向用户空间报告这些信息。每当 RISC-V 硬件走页器解决 TLB 未命中问题时，都会在 PTE 中标记这些位。本实验的目的是向 xv6 内核添加一个新特性，即通过检查 RISC-V 页表中的访问位，实现一个系统调用 pgaccess()，该系统调用可以报告哪些页面已被访问（读取或写入）。

实验步骤

1. 在 kernel/riscv.h 中定义一个 PTE_A，其为 Risc V 定义的 access bit。

2. 首先，在 kernel/sysproc.c 使用 argaddr() 和 argint() 来获取并解析参数，实现

pgacess

3. 使用 walk 函数找到正确的 PTE

4. 在内核中创建一个临时缓冲区，然后使用遍历页面的方法，检查每个页面的 PTE_A 访问位是否被设置。如果设置了，将相应的位设置为 1。否则，设置为 0。并在检查后清除。

5. 最后将内核中的位掩码缓冲区的内容复制到用户空间的指定位置

6. 运行并评分

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

实验中遇到的问题和解决方法

在代码编写过程中，需要找到正确的 PTE，这一点如果靠自己实现比较困难，好在可以使用系统提供的 walk 函数。另外，在 pageacess 函数中可以使用 argaddr 和 argint 进行参数的解析。

实验心得

本实验是对于内存管理机制的考察。在实现系统调用过程中，我深入了解了内核代码的组织结构和运行方式，以及如何将用户态的请求转换为内核态的操作，并了解了如何从用户空间传递参数到内核空间。

Lab4 Traps

RISC-V assembly (easy)

实验目的

了解一些 RISC-V 汇编很重要。在 xv6 repo 中有一个文件 user/call.c。make fs.img 会对其进行编译，并生成 user/call.asm 中程序的可读汇编版本。

实验步骤

1. Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

```
printf("%d %d\n", f(8)+1, 13);
24:   4635                    li      a2,13
26:   45b1                    li      a1,12
```

A2

2. Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

```
int f(int x) {
   e:   1141                    addi    sp,sp,-16
  10:   e422                    sd      s0,8(sp)
  12:   0800                    addi    s0,sp,16
  return g(x);
}
  14:   250d                    addiw   a0,a0,3
  16:   6422                    ld      s0,8(sp)
  18:   0141                    addi    sp,sp,16
  1a:   8082                    ret
```

3. At what address is the function printf located?

```
void
printf(const char *fmt, ...)
{
 638:   711d                    addi    sp,sp,-96
 63a:   ec06                    sd      ra,24(sp)
 63c:   e822                    sd      s0,16(sp)
 63e:   1000                    addi    s0,sp,32
 640:   e40c                    sd      a1,8(s0)
 642:   e810                    sd      a2,16(s0)
 644:   ec14                    sd      a3,24(s0)
 646:   f018                    sd      a4,32(s0)
 648:   f41c                    sd      a5,40(s0)
 64a:   03043823                sd      a6,48(s0)
 64e:   03143c23                sd      a7,56(s0)
  va_list ap;
```

4. What value is in the register ra just after the jalr to printf in main?

```
 30:   00000097                auipc   ra,0x0
 34:   608080e7                jalr    1544(ra) # 638 <printf>
```

0x38

5. Run the following code.

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

What is the output?The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

结果：He110 World。如果在大端序，i 的值应该为 0x00726c64 才能保证与小端序输出的内容相同。57616 不需要改变。

6. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen? printf("x=%d y=%d", 3);

无法确定，少一个参数，但它会从某个位置读取一个值，而该值未知，不可信

实验中遇到的问题和解决方法

本次实验比较特殊，在其中遇到的问题，比如系统的大小端问题，可以从 CSDN 上获取已有的知识。

实验心得

本实验是对于汇编语言的考察。对于编译器来说，很多的操作并不会和源代码相同，而是会由编译器进行优化以增加效率。另外，本实验的各种计算机底层知识让我对编译原理有了一定的认识。

Backtrace (moderate)

实验目的

实现一个回溯（backtrace）功能，用于在操作系统内核发生错误时，输出调用堆栈上的函数调用列表。这有助于调试和定位错误发生的位置。

实验步骤

1. 在 kernel/defs.h 中添加 backtrace 函数的原型

2. GCC 编译器将当前正在执行的函数的帧指针（frame pointer）存储到寄存器 s0 中，遍历栈帧需要一个停止条件。有用的信息是：每个内核栈由一整个页（4k）组成，所有的栈帧都在同一个页上面。你可以使用 PGROUNDDOWN(fp) 来定位帧指针所在的页面，从而确定循环停止的条件。

3. 在 kernel/printf.c 中实现一个名为 backtrace 的函数,通过遍历调用堆栈中的帧指针来输出保存在每个栈帧中的返回地址。

4. 在 sys_sleep 函数中调用 backtrace 函数。

```
$ bttest
backtrace:
0x00000000800021b4
0x0000000080002086
0x0000000080001cf4
```

5. 使用 addr2line 工具将这些地址转换为函数名和文件行号，以确定错误发生的位置

```
df@df-virtual-machine:~/Code/OS-Xv6-Lab-2023$ addr2line -e kernel/kernel 0x0000000800021b4
/home/df/Code/OS-Xv6-Lab-2023/kernel/sysproc.c:63
```

实验中遇到的问题和解决方法

在代码编写过程中，在 backtrace 函数中，我需要遍历整个调用堆栈，但是起初我不清楚这个循环需要一个什么样的终止条件。后来通过学习我发现，使用 PGROUNDDOWN 和 PGROUNDUP 宏可以帮助我计算栈页的顶部和底部地址，从而确定循环终止的条件。

实验心得

本实验是对于程序执行过程中函数调用和返回的考察。在实现过程中，帧指针在调用堆栈中的作用是关键。帧指针是一个在每个堆栈帧中保存调用者帧指针的位置，它帮助我们在调用链中向上移动。通过正确使用帧指针，我能够遍历每个堆栈帧并访问其中保存的返回地址，从而实现了回溯功能。

Alarm (hard)

实验目的

本次实验将向 xv6 内核添加一个新的功能，即周期性地为进程设置定时提醒。这个功能类似于用户级的中断/异常处理程序，能够让进程在消耗一定的 CPU 时间后执行指定的函数，然后恢复执行。通过实现这个功能，我们可以为计算密集型进程限制 CPU 时间，或者为需要周期性执行某些操作的进程提供支持。

实验步骤

1. 在 Makefile 中添加$U/_alarmtest\

2. 更新 user/usys.pl

3. 在 syscall.h 中声明 sigalarm 和 sigreturn 的用户态库函数.在 syscall.c 中添加对应的系统调用处理函数

4. 在 sysproc.c 中实现 sys_sigalarm 和 sys_sigreturn 的内核处理逻辑。

5. 每次时钟中断发生时，硬件时钟会产生一个中断，这将在 usertrap() 函数中进行处理（位于 kernel/trap.c）。因此接下来需要修改 usertrap 函数，使得硬件时钟每滴答一次都会强制中断一次。

6. 完成第一步骤的工作后，我们可能遇到 alarmtest 在打印 "alarm!"后在 test0 或 test1 中崩溃，或者是 alarmtest（最终）打印 "test1 失败"，或者是 alarmtest 退出时没有打印 "test1 通过"。因此，这一步涉及确保在处理完警报处理函数后，控制能够返回到被定时中断中断的用户程序指令处，同时保证寄存器的内容被恢复，以便用户程序能够在警报处理之后继续执行。最后，你需要在每次警报触发后 "重新激活" 定时器计数器，以便定时器定期触发处理函数的调用。

7. 用户警报处理程序在完成后必须调用 sigreturn 系统调用。请看 alarmtest.c 中的 periodic 示例。这意味着您可以在 usertrap 和 sys_sigreturn 中添加代码，使用户进程在处理完警报后恢复正常。

8. 我们可以编写一个符合要求的 sys_sigreturn 函数：

9. 函数的目的是在信号处理函数完成后，从用户态返回到中断代码。它模拟了一个中断返回过程，恢复了被中断的用户态代码的执行状态。

```
test0 start
..............alarm!
test0 passed
test1 start
.alarm!
.alarm!
.alarm!
..alarm!
.alarm!
.alarm!
..alarm!
.alarm!
..alarm!
.alarm!
test1 passed
test2 start
................................alarm!
test2 passed
```

实验中遇到的问题和解决方法

在调用处理函数后，需要确保能够正确地恢复进程的执行状态，包括保存和恢复寄存器状态等。一开始我并没有注意到，中断的时候已经在 proc->saved_trapframe 中保存了中断帧信息，从而对于如何回复中断没有头绪。后来我才发现在中断发生时保存了中断帧，其中包含了被中断的用户态代码执行时的寄存器状态，因此我们需要将它恢复回 proc->trapframe。

实验心得

本实验是对于定时中断的考察。在实现过程中，帧指针在调用堆栈中的作用是关键。除此之外，通过这一次的测试程序，我还明白，在修改内核操作的时候，应当确保不影响系统稳定性，即在实现定时中断处理功能时，要确保不会影响系统的稳定性和正常运行，确保中断处理程序能够及时返回，避免影响其他中断和系统调度。

Lab5 Copy-on-Write Fork for xv6

Implement copy-on write(hard)

实验目的

实验的主要目的是在 xv6 操作系统中实现写时复制（Copy-on-Write，COW）的 fork 功能。传统的 fork() 系统调用会复制父进程的整个用户空间内存到子进程，而 COW fork() 则通过延迟分配和复制物理内存页面，只在需要时才进行复制，从而提高性能和节省资源。通过这个实验，你将了解如何使用写时复制技术优化进程的 fork 操作。

实验步骤

1. 查看 kernel/riscv.h ，了解一些对于页表标志（page table flags）有用的宏和定义。记录一个 PTE 是否是 COW 映射是有帮助的。

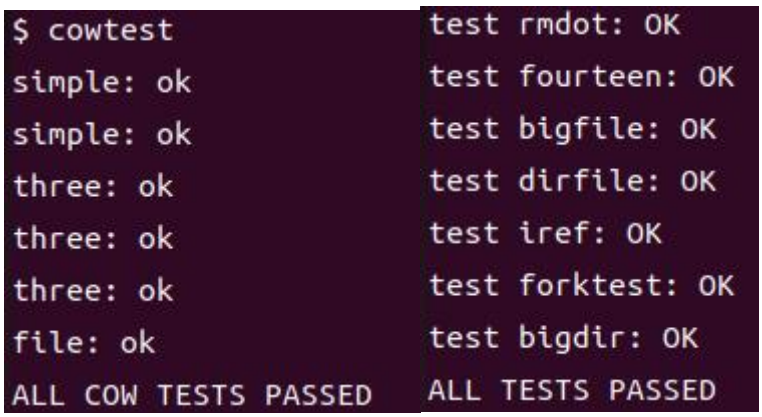
2. 修改 uvmcopy() 函数，将父进程的物理页面映射到子进程，而不是分配新的页面。同时，清除父子进程的 PTE 中的 PTE_W 标志，使得页面变为只读状态。

3. 在 vm.c 文件中修改 uvmcopy 函数。

4. 修改 usertrap() 函数，使其能够识别页面故障（page fault）

5. 确保当物理页面的最后一个 PTE 引用消失时释放它。可以为每个物理页面维护一个“引用计数”，表示指向该页面的用户页表数量。

6. 在 trap.c 中定义 cowhandler 函数进行检查。只有存在的、被标记为 COW 且属于用户级别的，才可以被分配内存。如果页是只读的，若在此时尝试对其进行写入，就会返回 -1，最终被 kill 。



实验中遇到的问题和解决方法

本实验难度较大，遇到的问题很多。比如对于获取寄存器值的方法，可以通过查阅 tutorial 中的资料进行理解和学习。阅读书目之后我们可以从中发现：在 RISC 架构中存在一个监控程序陷阱值（stval）寄存器。如果在指令获取、加载或存储过程中出现断点、地址对齐、访问故障或页面故障异常时，stval 被写入一个非零值，那么 stval 将包含发生故障的虚拟地址。

实验心得

在实现写时复制时，需要管理页面的分配和释放，以及处理页面复制。我通过学习如何正确地为页面分配和释放内存，使得其在需要时进行复制，以避免资源泄漏和内存溢出。这是提高计算机性能的重要的操作和思想，通过这样的学习我获益匪浅。

Lab6 Multi-threading

Uthread: switching between threads (moderate)

实验目的

设计并实现一个用户级线程系统的上下文切换机制。补充完成一个用户级线程的创建和切换上下文的代码。需要创建线程、保存/恢复寄存器以在线程之间切换，并且确保解决方案通过测试。

实验步骤

1. 在 uthread.c 中创建适当的数据结构来表示线程。每个线程被定义成一个结构体，包括保存线程状态（寄存器）的信息：

2. 在 user/uthread.c 中修改 thread_create()

3. user/uthread_switch.S 文件中的 thread_switch 实现线程切换的汇编代码。函数中需要实现保存调用者保存的寄存器，切换到下一个线程，然后恢复下一个线程的寄存器状态。在实现上，可以仿照 swtch.S 的写法。

4. 修改 usertrap() 函数，使其能够识别页面故障（page fault）

5. 在 user/uthread.c 中添加 thread_schedule 函数，调用 thread_switch 来实现线程的切换。需要传递一些参数给 thread_switch，以便它知道要切换到哪个线程

6. 使用 make qemu 来运行 xv6，然后运行 uthread 程序，观察线程的输出。

```
== Test uthread == uthread: OK (2.9s)
```

实验中遇到的问题和解决方法

在创建线程时，需要为每个线程分配独立的堆栈空间。在 RISC-V 架构中，寄存器 ra

是返回地址寄存器（Return Address Register），它存储了函数调用后的返回地址。而寄存器 sp 则是栈指针寄存器（Stack Pointer Register），它存储了当前栈的顶部地址。

实验心得

本实验是对线程的概念以及线程切换的底层实现机制的考察。让我更好地理解了寄存器的功能特性。在创建线程时，正确分配线程的堆栈空间是关键。通过理解 RISC-V 架构中寄存器的功能特性，我能够选择适当的寄存器来存储必要的信息，比如函数指针和栈顶指针。这样，当线程被调度执行时，它能够正确跳转到函数起始位置，并且在自己的独立栈上执行，避免与其他线程的干扰。

Using threads (moderate)

实验目的

本实验旨在通过使用线程和锁实现并行编程，以及在多线程环境下处理哈希表。学习如何使用线程库创建和管理线程，以及如何通过加锁来实现一个线程安全的哈希表，使用锁来保护共享资源，以确保多线程环境下的正确性和性能。

实验步骤

1. 使用提供的代码文件构建实验程序，通过命令行构建和运行实验程序，可以通过传递参数指定线程数，测试单线程和多线程情况下的性能。

2. 运行实验程序时，观察输出并理解性能和正确性问题。特别是，在多线程情况下，我们会注意到出现了缺少的键（missing keys）问题，即部分键没有被正确添加到哈希表中

3. 在多线程环境中，出现缺少的键问题可能是由于竞争条件引起的。尝试分析多线程情况下的序列事件，找出可能导致缺少键的情况。

4. 可以在代码中添加适当的锁以保护共享资源，以防止多线程竞争引起的问题。

5. 在某些情况下，线程之间可能不会在哈希表的内存区域上发生重叠访问，此时可以尝试优化性能。考虑是否可以为每个哈希桶（bucket）设置一个锁，以便在不需要保护重叠区域时实现更好的并行性能。

6. 测试

```
ph_safe: OK (11.2s)
== Test ph_fast == make[1]: 进入目录“/home/df/Code/OS-Xv6-Lab-2023”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/df/Code/OS-Xv6-Lab-2023”
ph_fast: OK (26.0s)
```

实验中遇到的问题和解决方法

确保在使用锁时遵循正确的获取和释放顺序，以避免死锁情况的发生。同时，在最后尽管已经释放了锁，但是创建了锁会占用一定的资源和内存，这影响了程序的性能。因此要尽量减少锁的使用。

实验心得

当我参与使用线程进行并行编程的实验时，我深刻地了解到了多线程编程的挑战和重要性。本次实验涉及了使用线程和锁来实现一个哈希表，并探索了在多线程环境下的表现。我明白了锁在多线程环境中的关键作用。使用锁可以确保在访问共享资源时的线程安全性，防止竞争条件和数据不一致的问题。

Barrier(moderate)

实验目的

本实验旨在通过实现一个线程屏障（barrier），即每个线程都要在 barrier 处等待，

直到所有线程到达 barrier 之后才能继续运行，加深对多线程编程中同步和互斥机制的理解。在多线程应用中，线程屏障可以用来确保多个线程在达到某一点后都等待，直到所有其他参与的线程也达到该点。通过使用 pthread 条件变量，我们将学习如何实现线程屏障，解决竞争条件和同步问题。

实验步骤

1. 理解出现断言失败的原因。断言错误表明在某个线程提前离开屏障，导致当前轮次的线程数量增加，影响后续轮次的正确性。

2. 为了实现 barrier，需要用到 UNIX 提供的条件变量以及 wait/broadcast 机制。在 barrier.c 中的 barrier()函数中添加逻辑，确保线程在达到屏障后等待其他线程。使用 pthread_cond_wait()来等待条件满足，pthread_cond_broadcast()来唤醒等待的线程。

3. 使用 bstate.round 来记录当前轮次，确保每次所有线程都达到屏障后增加轮次。避免在上一轮仍在使用 bstate.nthread 时，另一个线程增加了该值。

4. 测试

```
barrier: OK (11.2s)
```

实验中遇到的问题和解决方法

由于多线程环境中存在竞争条件，所以需要使用互斥锁来保护共享资源的访问。在更新 bstate.nthread 等共享变量时，我发现这同样需要使用互斥锁进行保护，否则多个线程可能会同时修改变量的值，从而导致不确定的行为和错误的结果。

实验心得

本次实验是对多线程的考察。通过本次实验，我深入了解了多线程编程中的同步机制，特别是条件变量和互斥锁的使用。我学会了如何设计和实现屏障同步，以保证多个线程在特定点同步等待和唤醒，从而实现了程序的并发控制。

Lab7 Networking

Your Job (hard)

实验目的

编写一个在 xv6 操作系统中用于网络接口卡（network interface card, NIC）的设备驱动程序。通过这个实验，你将学习如何初始化并操作一个虚拟的网络设备，以及如何处理网络通信，从而深入理解操作系统中设备驱动程序的工作原理。

实验步骤

1. 使用 E1000 的网络设备来处理网络通信。这个虚拟设备模拟了一个真实的硬件连接到真实的以太网局域网（LAN）。在 xv6 中，E1000 看起来就像连接到真实 LAN 的硬件设备。

2. 在 kernel/e1000.c 文件中完成 e1000_transmit()和 e1000_recv()函数。

3. 测试驱动程序

4. 在实现驱动程序的过程中，可能会遇到一些问题，例如数据包的正确传输和接收，以及寄存器的正确配置等。如果遇到问题，可以通过查阅 E1000 软件开发手册和源代码来找到解决方法。

5. 测试

```
== Test running nettests ==
$ make qemu-gdb
(3.9s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
```

实验中遇到的问题和解决方法

在初始化和操作 E1000 设备时，寄存器的配置可能会出现问题。因此我会需要按照 E1000 软件开发手册中的指导，正确配置寄存器，以确保设备的正常工作。其中，为了实现驱动程序通过内存映射的控制寄存器与 E1000 交互，以检测接收到的数据包何时可用，并通知 E1000 驱动程序已在某些 TX 描述符中填入要发送的数据包，通过教程指导我了解到全局变量 regs 保存着指向 E1000 第一个控制寄存器的指针，因此我考虑驱动程序可以通过数组索引 regs 来访问其他寄存器，其中就可以使用 E1000_RDT 和 E1000_TDT 这两个索引。

实验心得

通过这次实验，我深入了解了设备驱动程序的编写和调试过程，特别是在网络设备上的应用。我学会了如何初始化和操作虚拟网络设备，以及如何处理数据包的发送和接收。

Lab8 Locks

Memory allocator (moderate)

实验目的

通过修改内存分配器的设计，以减少锁竞争，从而提高多核系统中的性能。具体来说，需要实现每个 CPU 都有一个独立的自由列表（free list），每个列表都有自己的锁。这样可以让不同 CPU 上的内存分配和释放操作可以并行进行，从而减少锁的争用。还需要实现当一个 CPU 的自由列表为空时，能够从其他 CPU 的自由列表中获取部分内存。

实验步骤

1. 从 kalloctest 的输出来看，问题主要集中在 kmem 和 proc 这两个锁上，出现了大量的锁竞争和争用情况。

2. 在实验代码中找到内存分配器相关的部分，这可能涉及 kalloc.c 和 kalloc.h。修改内存分配器的设计，使得每个 CPU 都有独立的自由列表，每个自由列表都有一个锁。

3. 实现代码来维护每个 CPU 的自由列表，确保在不同 CPU 上的操作可以并行执行，从而减少锁的竞争。

4. 运行 kalloctest 测试，查看锁的竞争情况是否减少，确认你的实现是否成功降低了锁的争用。

5. 运行 usertests sbrkmuch 测试，确保内存分配器仍然能够正确分配所有内存。

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

6. 运行 usertests 测试，确保所有的用户测试都通过。

实验中遇到的问题和解决方法

虽然目标是减少锁的争用，但是在借用页面的问题上，究竟以多少页为限制是一个比较好的选择，这也许需要进一步的实际验证来决定，在本次实验中，我选择了 1024，1024 页面的选择可能是因为这个数量可以覆盖一定的范围，同时避免了一次性偷取过多导致的长时间的锁竞争。然而实际上可以根据具体的硬件架构、应用负载和性能需求进行调整。

实验心得

当我开始进行实验时，我对于操作系统内核的设计和锁竞争问题并没有太多的了解。然而，通过完成这个实验，我逐渐理解了操作系统内核的基本结构以及如何解决多核心环境下的锁竞争问题。在多核心环境下，锁的使用是解决并发访问问题的关键。在实验中，我学会了如何使用 acquire 和 release 函数来保护共享资源，以避免竞争条件和数据不一致问题。

Buffer cache (hard)

实验目的

优化 xv6 操作系统中的缓冲区缓存（buffer cache），减少多个进程之间对缓冲区缓存锁的争用，从而提高系统的性能和并发能力。通过设计和实现一种更加高效的缓冲区管理机制，使得不同进程可以更有效地使用和管理缓冲区，减少锁竞争和性能瓶颈。

实验步骤

1. 运行 bcachetest 测试程序，观察锁竞争情况和测试结果。测试结果表明，在多个进程之间，对 bcache.lock 锁的竞争比较激烈，导致多个进程在试图获取该锁时需要进行较多次的 test-and-set 操作和 acquire()操作。

2. 根据分析结果，我们需要重新设计和实现缓冲区管理机制。全局结构体通过将缓冲区数

组和分桶数组结合在一起，实现了对缓冲区的高效管理、分配和释放。它允许对每个分桶进行独立的加锁，从而减少了不同分桶之间的竞争，并且通过哈希函数将缓冲区分散存储在不同的分桶中，提高了访问效率。

3. 对每个缓冲区进行初始化

4. 实现了缓冲区的获取逻辑

5. 修改释放缓冲区逻辑

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

6. 运行 usertests 测试，确保所有的用户测试都通过。

实验中遇到的问题和解决方法

在修改缓冲区管理代码时，可能会遇到并发情况下的数据一致性问题，如资源分配冲突、竞争条件等。起初我没有修改增减缓冲区计数的函数，使得其可能被其他线程干扰导致出错，后来我增加了锁，根据缓冲区的块号计算哈希索引 v，获取对应的分桶结构体指针 bucket 和分桶的自旋锁，以确保在操作缓冲区引用计数时不会被其他线程干扰。

实验心得

在开始实验之前，我深入理解了 Buffer Cache 的结构，包括缓存的大小、缓存块的管理方式以及数据结构等。这为我后续的编码和设计提供了重要的指导。实验要求的功能涵盖了从缓存的获取、写入到缓存的释放等多个方面。我将实验任务分成不同的阶段，逐步实现和测试每个功能。这样有助于确保每个功能都能够独立正常运行。在多线程环境下，缓存的并发访问可能导致竞态条件等问题。我使用了适当的同步机制，如自旋锁和信号量，以确保缓存的安全性和一致性。

Lab9 File system

Large files (moderate)

实验目的

本次实验的目的是扩展 xv6 文件系统，使其支持更大的文件大小。目前 xv6 文件大小受限于 268 个块，或 268 * BSIZE 字节（在 xv6 中，BSIZE 为 1024）。这个限制是因为 xv6 inode 包含 12 个“直接”块号和一个“单间接”块号，它引用一个可以容纳多达 256 个块号的块，总共为 12+256=268 个块。

实验步骤

1. 打开 kernel/fs.h 文件，查找 struct dinode 结构的定义。

2. 查找 NDIRECT 和 NINDIRECT 的定义。这些常量表示直接块和单间接块的数量

3. 修改 addrs[] 数组的大小以支持双间接块。

4. 打开 kernel/fs.c 文件，找到 bmap() 函数的定义。这个函数负责将文件的逻辑块号映射到磁盘块号。我们需要考虑如何将逻辑块号映射到直接块、单间接块和双间接块之间的关系，然后修改这个函数以支持双间接块

5. 确保 itrunc 函数释放文件的所有块，包括双向块

6. 在 kernel/file.c 文件中，找到 filewrite() 函数的实现。确保在写文件时正确调用了 bmap() 函数。

7. 运行 bigfile 测试，确保它可以成功创建文件，并报告正确的文件大小。这个测试可能会花费一些时间。

8. 运行 usertests 测试套件，确保所有的测试都通过。这可以验证你的修改是否没有引入错误，并且 xv6 正常运行。

实验中遇到的问题和解决方法

实验中需要弄清各种数据块号之间的关系，一开始的时候，我对此毫无头绪。之后通过查阅 xv6 的数据明白其管理的方式。

实验心得

阅读并理解原始代码时，我最初的难点是理解 xv6 文件系统的数据结构，包括 inode 结构、块地址数组等。通过阅读代码注释、文档并且查阅相关资料，我逐步理解了这些概念。并且参照上面的 inode 结构图，准确修改了文件系统宏定义，且在 bmap 上实现顺序映射逻辑块号到磁盘块号，在 itrunc 上实现逆序地释放块。通过完成本次实验，我更加地深入理解了 xv6 文件系统的内部结构和工作原理。

Symbolic links (moderate)

实验目的

本次实验的主要目的是在 xv6 操作系统中实现符号链接（软链接）的功能。符号链接是一种通过路径名来引用另一个文件的方式，与硬链接不同，符号链接可以跨越不同的磁盘设备。通过实现这个系统调用，我们将深入理解路径名查找的工作原理。

实验步骤

1. 添加系统调用号： 添加有关 symlink 系统调用的定义声明

2. 在 kernel/stat.h 中添加一个新的文件类型 T_SYMLINK，用于表示符号链接。这将帮助区分普通文件和符号链接

3. 在 kernel/fcntl.h 中添加一个新的打开标志 O_NOFOLLOW，该标志可以与 open 系统调用一起使用。

4. 在 kernel/sysfile.c 中实现 sys_symlink 函数。sys_symlink 函数即用来生成符号链接。符号链接相当于一个特殊的独立的文件，其中存储的数据即目标文件的路径。

5. 实现 symlink(target, path) 系统调用，该调用在指定路径 path 创建一个新的符号链接，该链接引用 target 指定的文件。

6. 修改 open 系统调用，以处理路径引用到符号链接的情况。

7. 改 kernel/sysfile 的 sys_open() 函数：sys_open() 函数用于打开文件的，对于符号链接一般情况下需要打开的是其链接的目标文件，因此需要对符号链接文件进行额外处理。

8. 如果链接的文件本身也是符号链接，你需要递归地跟随链接，直到达到一个非链接的文件为止。如果链接形成循环，你需要返回一个错误代码。

9. 在 Makefile 中添加对测试文件 symlinktest.c 的编译.

实验中遇到的问题和解决方法

对于循环连接问题，需要设置一个措施，当识别成为循环链接，应该返回一个错误代码。可以通过在链接深度达到某个阈值 (例如 10) 时返回错误代码来近似处理循环链接的情况。对于链接的深度，可以在 kernel/fs.h 中定义了 NSYMLINK ，表示最大的符号链接深度，超过该深度将不会继续跟踪而是返回错误。

实验心得

在本次实验中，我成功地向 xv6 操作系统添加了符号链接（软链接）的支持。符号链接是一种特殊类型的文件，可以跨越磁盘设备引用其他文件。此外，为了防止符号链接文件溯源的过程中陷入死循环，我们还额外考虑了循环的深度限制和成环检测，这提高了我们的效率，并且保障了程序的安全性。

Lab10 mmap(hard)

实验目的

本次实验旨在向 xv6 操作系统添加 mmap 和 munmap 系统调用，实现对进程地址空间的详细控制。通过实现这两个系统调用，我们可以实现内存映射文件的功能，包括共享内存、将文件映射到进程地址空间等。这有助于理解虚拟内存管理和页面错误处理的机制。

实验步骤

1. 在 Makefile 中添加 $U/_mmaptest

2. 添加系统调用： 添加有关 mmap 和 munmap 系统调用的定义声明。

3. 定义 vm_area 结构体及其数组，在 kernel/proc.h 中定义 struct vm_area 结构体，在 struct proc 结构体中定义相关字段

4. 在 kernel/fcntl.h 中，已经定义了 PROT_READ 和 PROT_WRITE 权限。你需要为 mmap 系统调用实现 MAP_SHARED 和 MAP_PRIVATE 标志位

5. 类似于之前的惰性页分配实验，通过在页面错误处理代码中（或者被 usertrap 调用）懒加载页表。确保 mmap 不会分配物理内存或读取文件。

6. 定义一个与 Lecture 15 中虚拟内存区域(VMA)相对应的结构，用于记录由 mmap 创建的虚拟内存范围的地址、长度、权限和文件等信息。在 xv6 内核中，可以声明一个固定大小的 VMA 数组，并根据需要从数组中分配。

7. 找到进程地址空间中未使用的区域，用于映射文件，并将 VMA 添加到进程的映射区域表中。VMA 应包含指向要映射文件的 struct file 的指针。确保在 mmap 中增加文件的引用计数，以防止文件在关闭时被删除。

8. 由于在 sys_mmap() 中对文件映射的内存采用的是 Lazy allocation，因此需要对访问

文件映射内存产生的 page fault 进行处理。当在 mmap 映射的区域发生页面错误时，分配一个物理内存页，从相关文件中读取 4096 字节到该页面，并将其映射到用户地址空间。你可以使用 readi 函数从文件中读取数据。

9. 找到要取消映射的地址范围的 VMA，并取消映射指定的页面。如果 munmap 移除了前一个 mmap 的所有页面，则应减少相应的 struct file 的引用计数。如果一个页面被修改且文件是 MAP_SHARED 映射，则将修改的页面写回文件。

10.设置脏页标志位

11.上述内容完成了基本的 mmap 和 munmap 系统调用的部分，最后需要对 exit 和 fork 两个系统调用进行修改。

12.测试

```
$ make qemu-gdb
(6.5s)
== Test   mmaptest: mmap f ==
  mmaptest: mmap f: OK
== Test   mmaptest: mmap private ==
  mmaptest: mmap private: OK
== Test   mmaptest: mmap read-only ==
  mmaptest: mmap read-only: OK
== Test   mmaptest: mmap read/write ==
  mmaptest: mmap read/write: OK
== Test   mmaptest: mmap dirty ==
  mmaptest: mmap dirty: OK
== Test   mmaptest: not-mapped unmap ==
  mmaptest: not-mapped unmap: OK
== Test   mmaptest: two files ==
  mmaptest: two files: OK
== Test   mmaptest: fork_test ==
  mmaptest: fork_test: OK
```

实验中遇到的问题和解决方法

在实现 munmap 时，我需要确保正确地找到要取消映射的页面，并进行相应的处理。在教程指导上说，如果页面被修改，且文件是 MAP_SHARED 映射，你需要将修改的数据写回文件。

实验心得

在本次实验中，我了解了虚拟内存、页表操作和文件系统，实现了 sys_mmap 的系统调用。此外，还了解了脏页的识别和处理。

总的来说，这是一次非常全面和深入以及生动的实验过程。实话说，过程中的各种程序实现非常难，但是也同时给予了我一次对于操作系统内核的非常深入的学习和理解。