

xv6 实验报告

目录

Lab1: Xv6 and Unix utilities	4
Boot xv6	4
一、 实验目的	4
二、 实验步骤	4
三、 实验中遇到的问题及解决方法	4
四、 实验心得	4
sleep	4
一、 实验目的	4
二、 实验步骤	4
三、 实验中遇到的问题及解决方法	5
四、 实验心得	5
Pingpong	5
一、 实验目的	5
二、 实验步骤	5
三、 实验中遇到的问题及解决方法	5
四、 实验心得	5
primes	6
一、 实验目的	6
二、 实验步骤	6
三、 实验中遇到的问题及解决方法	6
四、 实验心得	7
find	7
一、 实验目的	7
二、 实验步骤	7
三、 实验中遇到的问题及解决方法	7
四、 实验心得	8
xargs	8
一、 实验目的	8
二、 实验步骤	8
三、 实验中遇到的问题及解决方法	8
四、 实验心得	8
实验结果	8
Lab2:system calls	9
System call tracing	9
一、 实验目的	9
二、 实验步骤	9
三、 实验中遇到的问题及解决方法	9
四、 实验心得	9
Sysinfo	10
一、 实验目的	10

二、 实验步骤	10
三、 实验中遇到的问题及解决方法	10
四、 实验心得	11
实验结果	11
Lab3:page tables	12
Speed up system calls	12
一、 实验目的	12
二、 实验步骤	12
三、 实验中遇到的问题及解决方法	12
四、 实验心得	12
Print a page table	12
一、 实验目的	12
二、 实验步骤	13
三、 实验中遇到的问题及解决方法	13
四、 实验心得	13
Detecting which pages have been accessed	13
一、 实验目的	13
二、 实验步骤	13
三、 实验中遇到的问题及解决方法	14
四、 实验心得	14
实验结果	14
Lab4: traps	14
RISC-V assembly	14
一、 实验目的	14
二、 实验步骤	14
三、 实验中遇到的问题及解决方法	15
四、 实验心得	15
Backtrace	16
一、 实验目的	16
二、 实验步骤	16
三、 实验中遇到的问题及解决方法	16
四、 实验心得	16
Alarm	17
一、 实验目的	17
二、 实验步骤	17
三、 实验中遇到的问题及解决方法	17
四、 实验心得	18
实验结果	18
Lab5:Copy-on-Write Fork for xv6	19
一、 实验目的	19
二、 实验步骤	19
三、 实验中遇到的问题及解决方法	19
四、 实验心得	20
实验结果	20

Lab6:Multithreading	20
Uthread:switching between threads	20
一、 实验目的	20
二、 实验步骤	20
三、 实验中遇到的问题及解决方法	21
四、 实验心得	22
Using threads	22
一、 实验目的	22
二、 实验步骤	22
三、 实验中遇到的问题及解决方法	22
四、 实验心得	22
Barrier	23
一、 实验目的	23
二、 实验步骤	23
三、 实验中遇到的问题及解决方法	23
四、 实验心得	23
实验结果	24
Lab7:Networking	24
一、 实验目的	24
二、 实验步骤	24
三、 实验中遇到的问题及解决方法	26
四、 实验心得	26
实验结果	27
Lab8:Locks	27
Memory	27
一、 实验目的	27
二、 实验步骤	27
三、 实验中遇到的问题及解决方法	28
四、 实验心得	28
Buffer cache	29
一、 实验目的	29
二、 实验步骤	29
三、 实验中遇到的问题及解决方法	29
四、 实验心得	29
实验结果	30
Lab9:File System	30
Large files	30
一、 实验目的	30
二、 实验步骤	30
三、 实验中遇到的问题及解决方法	31
四、 实验心得	31
Symbolic links	31
一、 实验目的	31
二、 实验步骤	31

三、 实验中遇到的问题及解决方法	32
四、 实验心得	32
实验结果	33
Lab10:Mmap	33
一、 实验目的	33
二、 实验步骤	33
三、 实验中遇到的问题及解决方法	35
四、 实验心得	35
实验结果	36

Lab1: Xv6 and Unix utilities

本实验通过实现一些基本的 unix 程序来熟悉 xv6 及其系统调用

Boot xv6

一、实验目的

切换到 xv6-labs-2021 代码的 util 分支，使用 qemu 进入 xv6 系统

二、实验步骤

1. 下载 WSL
2. 将 xv6-labs-2021 代码克隆到本地，并进入相应目录，建立相应分支
3. 使用 make qemu 命令启动 xv6 操作系统

三、实验中遇到的问题及解决方法

无

四、实验心得

通过本次实验，学会了使用 git 对代码的管理，以及如何使用 QEMU 来启动 xv6 操作系统。

sleep

一、实验目的

为 xv6 系统实现 unix 的 sleep 程序，sleep 程序应实现暂停相应的滴答数（即时钟周期数），滴答数由用户指定

二、实验步骤

1. 在 user 目录下，创建一个名为 sleep.c 的文件
2. 在 sleep.c 中编写一个程序，该程序接受一个命令行参数，并使当前进程暂停相应的时间。如果用户传入的参数数量不等于 2（程序名和一个参数），程序应打印错误信息并退出。在确保只有一个时间参数被提供时，将参数转换为整数以确定暂停的时间。
3. 将程序以 \$U/_sleep\的形式，添加到 Makefile 的 UPROGS 中
4. 使用 make qemu 在 xv6 shell 中测试运行该程序
5. 使用 ./grade-lab-util sleep 进行单元测试

```

root@83a58abddb07: /xv6-labs-2021# ./grade-lab-util sleep
fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
sleep, no arguments: OK (3.9s)
== Test sleep, returns == sleep, returns: OK (2.3s)
== Test sleep, makes syscall == sleep, makes syscall: OK (2.2s)
root@83a58abddb07: /xv6-labs-2021#

```

三、实验中遇到的问题及解决方法

无

四、实验心得

通过实现 sleep 程序，我学会了如何为 xv6 系统增添新的程序，同时了解了如何处理命令行参数，以及系统时钟 ticks 的工作机制和其在进程暂停中的应用，使我对 xv6 的构建和运行流程有了更加直观的认识。

Pingpong

一、实验目的

使用 unix 系统编写一个 pingpong 程序，在一对管道上实现两个进程之间的通信

二、实验步骤

1. 在 user 目录下，创建一个名为 pingpong.c 的文件
2. 在 pingpong.c 中，编写程序。先创建两个管道分别用于父到子(ping)和子到父(pong)，管道包括读端和写端。接着通过 fork() 创建一个子进程，返回值在父进程中是子进程的 PID，在子进程中是 0。若为父进程，则先向父到子管道中写入 ping，然后等待子进程读写结束后，从子到读管道中读取 pong，并做出接受回应；若为子进程，则先从父到子管道中读取 ping，并做出接受回应，然后向子到父管道中写入 pong，子进程读写结束。最后 exit(0) 正常结束程序。
3. 将程序以 \$U/_pingpong 的形式，添加到 Makefile 的 UPROGS 中
4. 使用 make qemu 在 xv6 shell 中测试运行该程序

```

$ pingpong
4: received ping
3: received pong

```

5. 用 ./grade-lab-util pingpong 进行单元测试

```

root@83a58abddb07: /xv6-labs-2021# ./grade-lab-util pingpong
fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
make: 'kernel/kernel' is up to date.
== Test pingpong == fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
pingpong: OK (3.5s)

```

三、实验中遇到的问题及解决方法

无

四、实验心得

通过本次实验，使我对进程通信有了更加深刻的理解，了解了父子进程之间利用管道通信的工作原理，尤其是为保证进程同步，父进程需要等待子进程读写结束后再对管道进行读

取。也对今后写更复杂的并行程序有所帮助。

primes

一、实验目的

使用管道编写一个素数筛的并发版本。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，安排创建一个进程，该进程通过管道从其左侧邻居读取，并通过另一个管道写入其右侧邻居。

二、实验步骤

1. 在 `user` 目录下创建一个名为 `primes.c` 的文件
2. 在 `primes.c` 中编写程序。首先创建一个主进程，负责生成 2 到 35 之间的数字，创建初始管道，在父进程中，首先关闭初始管道的读端，将生成的数字输入到初始管道中，关闭写端并等待子进程结束；在子进程中，关闭初始管道的写端，开始筛选工作。筛选过程为子进程从左侧管道读取第一个数字，它是当前进程的素数并打印。接着创建右侧管道，在父进程中，首先关闭右侧管道的读端，然后从左侧管道中读取数字，第一个数为该进程的素数并打印，之后对于其他数字，若能够被该素数整除则不写入右侧管道，依次处理完毕后，关闭左侧管道的读端和右侧管道的写端并等待子进程结束；在子进程中，关闭右侧管道的写端，继续在右侧管道中递归筛选直至管道中没有元素。
3. 将程序以 `$U/_primes` 的形式，添加到 `Makefile` 的 `UPROGS` 中
4. 使用 `make qemu` 在 `xv6 shell` 中测试运行该程序，结果如下

```
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

5. 使用 `./grade-lab-util primes` 进行单元测试，结果如下

```
root@83a58abddb07:~/xv6-labs-2021# ./grade-lab-util primes
fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
make: 'kernel/kernel' is up to date.
== Test primes == fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
primes: OK (3.2s)
```

三、实验中遇到的问题及解决方法

问题：在编写程序时，未能理解素数筛选的方法和过程，导致期初筛选未能如期进行。

解决方案：搜索学习后，采用 Eratosthenes 筛法，只有当一个数不能被当前素数整除时，才会被写入右侧管道，这样就成功筛选出了素数。

四、实验心得

通过本次实验，了解到了并发编程的基本概念，通过筛选素数这一应用，实现了并发和管道通信的有效结合，深入了解到了进程协作和管道的作用。

find

一、实验目的

编写一个简单版本的 UNIX find 程序，用于查找目录中具有特定名称的所有文件

二、实验步骤

1. 在 user 目录下，创建一个名为 find.c 的文件
2. 在 find.c 中编写程序。

在 main 中，首先检查输入参数的个数（包括程序名），如果参数个数不等于 3，则打印提示正确输入格式并退出；若输入格式正确，将第二个参数（路径）和第三个参数（要查找的文件名）传递给 find 函数。

在 find 函数中，首先尝试打开给定的路径，若无法打开则打印错误消息并返回；若可以打开，使用 fstat() 获取路径状态，若无法获取，关闭文件描述符，打印错误消息并返回。

循环读取目录项，如果目录项的 inum 为 0 即空目录项，则跳过。对于非空目录，将目录项名称复制到缓冲区中路径的末尾位置，添加字符串终止符，使缓冲区中的内容成为一个完整的文件或子目录路径。

根据路径类型，若新路径是文件，则直接跳出；若是目录，且目录名不是 "." 和 ".."，则递归地在该目录中查找目标文件。

3. 将程序以 \$U/_find\ 的形式，添加到 Makefile 的 UPROGS 中
4. 在 xv6 shell 中测试运行该程序

```
init: starting sh
$ echo > b
$ mkdir a
$ echo >a/b
$ find . b
./b
./a/b
$
```

5. 使用 ./grade-lab-util find 进行单元测试

```
root@83a58abddb07: ~/xv6-labs-2021# ./grade-lab-util find
fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
make: 'kernel/kernel' is up to date.
== Test find, in current directory == fatal: not a git repository (or any parent up to mount point /root)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
find, in current directory: OK (5.7s)
(Old xv6.out.find_curdir failure log removed)
== Test find, recursive == find, recursive: OK (2.1s)
(Old xv6.out.find_recursive failure log removed)
```

三、实验中遇到的问题及解决方法

问题：对递归结构的结束打印的位置不太明确，导致打印结构不完整

解决：重新审视代码，检查递归过程，并找到合适的递归结果打印位置

四、实验心得

通过本次实验，学习了如何使用递归结构遍历目录及其子目录进行文件的查找，更加熟悉了操作系统的文件结构及文件管理

xargs

一、实验目的

编写 UNIX `xargs` 程序的简单版本：从标准输入读取行并对每行运行一个命令，并将行作为参数提供给命令。

二、实验步骤

1. 在 `user` 目录下创建一个名为 `xargs.c` 的文件
2. 在 `xargs.c` 中编写程序
首先读取标准输入，将读取到的命令和参数存入一个字符串数组中。
将前一个指令的命令行参数存储到后一个指令的 `xargv` 中，并更新 `xargc`。
定义指针 `p` 指向缓冲区 `buf` 的起始位置。
遍历缓冲区字符：如果遇到换行符 `\n` 或空字符 `\0`，将其替换为终止符 `\0`，分割字符串。
调用 `fork` 创建子进程：在子进程中，将分割后的字符串指针存储到 `xargv` 中，并用 `exec` 函数执行命令。如果 `exec` 调用失败，输出错误信息并退出子进程。在父进程中，更新指针 `p` 指向下一个命令，并等待子进程完成。
3. 将程序以 `$U/_xargs` 的形式，添加到 `Makefile` 的 `UPROGS` 中
4. 在 `xv6 shell` 中测试运行该程序

```
$ sh < xargstest.sh
$ mkdir: a failed to create
$ $ $ $ mkdir: c failed to create
$ $ $ $ $ hello
hello
hello
```

5. 使用 `./grade-lab-util find` 进行单元测试

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (8.4s)
```

三、实验中遇到的问题及解决方法

问题：在执行 `find . b | xargs grep hello` 指令时，没有反应。

解决：由于 `find` 指令执行时较慢，耗时较长，导致 `find` 指令还未完成，`xargs` 指令已经执行结束，在 `xargs` 程序中增加 `sleep()` 以等待前一个指令执行结束再开始执行，从而能达到预期。

四、实验心得

通过本次实验，学习了如何实现 `xargs` 命令，从标准输入读取输入并将其作为参数传递给后续命令进行执行，学会了如何并行执行多个指令。

实验结果

使用 `./grade-lab-util` 进行测试结果如下


```

root@f90417feedb3: ~/xv6-labs-2021# ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (4.1s)
== Test sleep, returns == sleep, returns: OK (2.7s)
== Test sleep, makes syscall == sleep, makes syscall: OK (2.5s)
== Test pingpong == pingpong: OK (3.0s)
== Test primes == primes: OK (2.9s)
== Test find, in current directory == find, in current directory: OK (3.1s)
== Test find, recursive == find, recursive: OK (3.2s)
== Test xargs == xargs: OK (3.7s)
== Test time ==
time: OK
Score: 100/100

```

Lab2:system calls

System call tracing

一、实验目的

实现一个系统调用跟踪功能,通过创建新的系统调用,控制并跟踪特定的系统调用执行,包括进程 ID, 系统调用名和返回值, 以便于后续的调试工作。

二、实验步骤

1. 使用以下指令切换到 syscall 分支

```

git fetch
git checkout syscall
make clean

```

2. 根据 hints 提示修改相应文件

```

在 Makefile 中将 $U/_trace\添加到 UPROGS
在 user/user.h 中添加 int trace(int);声明系统调用原型
在 user/user.pl 中添加存根 entry("trace");
在 kernel/syscall.h 中添加系统调用号#define SYS_trace 22

```

3. 在 kernel/sysproc.c 中添加 sys_trace()函数,通过在 proc 结构中记住参数来实现新的系统调用。定义 sys_trace()函数, 将输入的掩码传递到 proc 的 mask 中; 并仿照其他系统调用, 增添 extern uint64 sys_trace(void);和[SYS_trace] sys_trace,; 在 syscall()中, 取得掩码及系统调用名并将其打印出来。

4. 在 kernel/proc.c 中, 修改 fork() 实现父进程复制跟踪掩码到子进程, np->mask=p->mask;

5. 使用指令进行代码测试

三、实验中遇到的问题及解决方法

问题: 在测试 grep hello REMADE 时未使用 trace 仍然打印了系统调用

解决: 经查资料, 释放进程时需要重置变量值, 在 freeproc()中, 增加 p->mask = 0;实现值重置, 问题得以解决。

四、实验心得

在此次实验中, 我深入理解了操作系统的系统调用机制, 通过实现 trace 系统调用, 学

会了在内核中添加自定义功能。我体验到内核编程的复杂性和严谨性，增强了调试和解决问题的能力。此外，我认识到系统调用跟踪对调试工作的巨大帮助，提升了我对操作系统内部机制的掌握，为今后的开发打下了坚实基础。

Sysinfo

一、实验目的

添加一个系统调用 `sysinfo`，用于收集有关正在运行的系统的信息。内核应填写此结构的各个字段：`freemem` 字段应设置为可用内存的字节数，`nproc` 字段应设置为状态不是 `UNUSED` 的进程数。

二、实验步骤

1. 根据 `hints` 中的提示修改相应文件
在 `Makefile` 中将 `$U/_sysinfotest\` 添加到 `UPROGS`
在 `user/user.h` 中添加 `int sysinfo(struct sysinfo*);` 声明系统调用原型
在 `user/user.pl` 中添加存根 `entry("sysinfo");`
在 `kernel/syscall.h` 中添加系统调用号 `#define SYS_sysinfo 23`
在 `kernel/syscall.c` 中添加 `extern uint64 sys_sysinfo(void);` 和 `[SYS_sysinfo] sys_sysinfo`
2. 参考 `kernel/sysfile.c` 中的 `sys_fstat()` 和 `kernel/file.c` 中的 `filestat()`，使用 `copyout()` 将 `sysinfo` 结构体复制回用户空间，在 `kernel/sysproc.c` 中编写 `sys_sysinfo()` 函数。
3. 使用 `sysinfotest` 命令测试代码正确性

三、实验中遇到的问题及解决方法

问题：`copyout()` 函数的使用

解决：使用 `argaddr(0, &addr)` 获取系统调用的第 0 个参数，成功，则将地址存储在 `addr` 指向的位置，并返回 0；失败则返回 -1。使用 `copyout(p->pagetable, addr, (char *)&info, sizeof(info))` 将状态信息从内核空间复制到用户空间。通过编写代码获取 `info.nproc` 和 `info.freemem`。

```
//kalloc.c
uint64
Acquire_freemem(void)
{
    struct run *r;
    uint64 freepage = 0;
    acquire(&kmem.lock);
    r = kmem.freelist;
    while (r)
    {
        r = r->next;
        freepage++;
    }
    release(&kmem.lock);
    return (freepage << 12);
}
```

```

//proc.c
uint64
acquire_nproc(void)
{
    struct proc *p;
    uint64 unused = 0;

    for(p = proc; p < &proc[NPROC]; p++)
    {
        if(p->state != UNUSED) {
            unused++;
        }
    }

    return unused;
}

```

在 kernel/sysproc.c 中导入并应用对应函数以获取相应值

四、实验心得

通过本次实验，我深入理解了系统调用的实现过程，尤其是在内核空间和用户空间之间进行数据传递。此外，还学会了如何通过遍历内核数据结构来获取系统的实时信息，例如内存状态和进程状态。

在解决问题的过程中，我也学会了如何有效地调试内核代码，并深刻认识到细节的重要性。每一个小的错误都可能导致整个系统的不稳定，只有通过仔细检查和逐步验证，才能确保系统的稳定运行。

实验结果

```

root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (3.2s)
== Test trace all grep == trace all grep: OK (1.7s)
== Test trace nothing == trace nothing: OK (1.6s)
== Test trace children == trace children: OK (12.0s)
== Test sysinfotest == sysinfotest: OK (3.5s)
== Test time ==
time: OK
Score: 35/35

```

Lab3:page tables

Speed up system calls

一、实验目的

某些操作系统（例如 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速某些系统调用。这消除了执行这些系统调用时内核交叉的需要。本实验为 xv6 中的 `getpid()` 系统调用实现此优化。

二、实验步骤

1. 切换到 lab3-pgtbl 实验环境（指令如下）

```
git fetch
git checkout pgtbl
make clean
```

2. 查看 `kernel/memlayout.h` 了解内存排列方式
3. 根据 hints 编写程序

在 `kernel/proc.c` 中的 `proc_pagetable()` 中添加页面映射关系，参考 TRAMPOLINE、TRAPFRAME 页面的映射方式，首先在 `proc.h` 中添加 USYSCALL 结构体：`struct usyscall *usyscall`; USYSCALL 允许用户 read 操作，所以 `mappages` 的最后参数使用 `PTE_R | PTE_U`，如果映射失败，先将 TRAMPOLINE 以及 TRAPFRAME 页面解除映射，再将整个进程的内存空间释放掉（此时的 `pagetable` 就是这个程序的程序页面）。

在 `allocproc()` 中为页面分配内存空间，参考 TRAPFRAME 页面的内存分配方式，将 `usyscall` 页面的 `pid` 定义为进程的 `pid`，如果分配失败，会调用 `freeproc()`，因此要比之前多释放掉一个 `p->usyscall` 参数，同时还会再调用 `proc_freepagetable`，需要多解除一个对 USYSCALL 页面的映射。

三、实验中遇到的问题及解决方法

问题：访问了没有权限的地址。

解决：`mappages` 的最后参数应使用 `PTE_R | PTE_U`，而非 `PTE_R | PTE_W`。

四、实验心得

在本次实验中，通过优化 xv6 操作系统中的 `getpid()` 系统调用，我们学到了如何在操作系统中实现用户空间和内核空间的高效通信。通过共享内存区域，我们减少了系统调用的开销，提高了操作系统的性能，在具体实现过程中，也深入理解了 xv6 的内存布局。

在实验过程中，我们遇到了一些挑战。例如，在实现页面映射关系时，如何正确设置 `mappages` 的参数，以确保 `usyscall` 结构体能够被用户空间读取。同时，在内存分配失败时，如何正确释放资源以避免内存泄漏。通过查阅资料和反复调试，我们成功地解决了这些问题。

通过本次实验，我深刻体会到操作系统内存管理的重要性，以及通过优化系统调用可以显著提高系统性能。这不仅加深了我对操作系统内核的理解，也为我今后的学习和工作奠定了坚实的基础。

Print a page table

一、实验目的

为了帮助我们更好地可视化 RISC-V 页表，并可能有助于未来的调试，此实验编写一个

打印页表内容的函数，当第一个程序启动时执行该函数。

二、实验步骤

1. 在 `kernel/vm.c` 中定义函数 `vmprint()`，其接收一个 `pagetable_t` 参数，并按照格式要求打印页表，在 `defs.c` 中声明 `void vmprint(pagetable_t pagetable);`，在 `vm.c` 中递归实现 `vmprintwalk()`。

2. 将 `if(p->pid == 1)vmprint(p->pagetable);` 添加到 `exec.c` 的 `exec()` 中的 `return argc;` 之前，实现打印第一个进程的页表。

3. 使用 `make grade` 测试程序。

三、实验中遇到的问题及解决方法

问题：页表项的解析

解决：表项的具体结构，包括各个字段的含义，如有效位（`valid bit`）、权限位（`read/write/execute bits`）、物理地址等。通过检查有效位，确保在递归调用之前，页表项是有效的，根据当前页表项是否指向下一级页表决定是否进行递归调用。

四、实验心得

通过本次实验，我对 RISC-V 页表的结构和工作机制有了更深入的理解。在编写 `vmprint()` 时，我不仅学会了如何递归遍历页表，还掌握了如何根据页表项的内容判断页表的层次结构以及对应的物理地址。这对于理解虚拟内存管理的具体实现细节非常有帮助。

在调试过程中，我遇到了一些挑战，例如如何正确处理页表项的解析和递归调用的边界条件。这些问题促使我深入研究 RISC-V 页表的硬件规范和 `xv6` 内核代码，从而提高了我的问题解决能力和代码调试技巧。

通过这次实验，我不仅学会了如何打印和分析页表，还体会到了虚拟内存管理在操作系统中的重要性。总的来说，这次实验大大增强了我对操作系统核心概念的理解，为我以后进一步学习和研究操作系统打下了坚实的基础。

Detecting which pages have been accessed

一、实验目的

一些垃圾收集器（一种自动内存管理形式）可以从有关哪些页面已被访问（读取或写入）的信息中受益。在这部分实验中，将向 `xv6` 添加一项新功能，该功能通过检查 RISC-V 页表中的访问位来检测并向用户空间报告此信息。每当 RISC-V 硬件页面遍历器解决 TLB 未命中时，它都会在 PTE 中标记这些位。

二、实验步骤

1. 阅读 `user/pgtbltest.c` 中的 `pgaccess_test()` 函数，了解 `pgaccess` 的系统调用方式

2. 在 `kernel/sysproc.c` 中实现 `sys_pageaccess()` 函数：

使用 `argaddr()` 和 `argint()` 从 `trapframe` 中按顺序获取参数

设置页数上限为 32，并当 `len < 0` 或 `len > 32` 时报错，返回 -1

在 `kernel/riscv.h` 中添加 `PTE_A` 的定义：`#define PTE_A (1L << 6)`；并参考 `vm.c` 中的 `walk()`

查找 PTE，检查访问位（`PTE_A`），如果被设置，则已访问过，将清楚访问位，即置为 0 通过 `copyout()` 将输出的位掩码从内核空间复制到用户空间。

3. 最后，在根目录下添加 `answers-pgtbl.txt` 和 `time.txt`，并向其中添加相应内容，通过 `make grade` 获得测试评分。

三、实验中遇到的问题及解决方法

问题：如何获得系统调用的参数

解决：根据提示及参考 `sysproc.c` 中的其他函数接收参数的方法，使用 `argaddr()` 和 `argint()` 函数，第一个参数表示参数的位置（即第 0 个、第 1 个等），第二个参数表示存储的地址。

四、实验心得

通过这次实验，我加深了对 `xv6` 操作系统的理解，特别是页面访问位机制和系统调用的实现。在实现 `PTE_A` 时，我学会了 `RISC-V` 架构中页表结构和页面访问位的重要性，理解了虚拟内存管理和硬件与软件的交互。

在实现 `sys_pageaccess()` 系统调用的过程中，我掌握了如何在内核中获取用户传递的参数，并通过内核函数进行操作。参考 `sysproc.c` 中的其他函数，我学会了使用 `argaddr()` 和 `argint()` 获取系统调用参数的方法。

总之，通过本次实验，不仅仅提高了我的变成能力，也让我更深层级地理解了操作系统的内存管理系统。

实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (3.6s)
== Test   pgtbltest: ugetpid ==
   pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
   pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (2.1s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests == (154.2s)
== Test   usertests: all tests ==
   usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

Lab4: traps

RISC-V assembly

一、实验目的

通过阅读 `xv6` 仓库中 `user/call.c` 文件，理解 `RISC-V` 汇编的基本概念

二、实验步骤

1. 使用以下命令，切换到 `lab4` 实验环境

```
git fetch
git checkout traps
git clean
```

2. 在 xv6 终端中输入运行 `make fs.img`, 编译 `user/call.c` 程序, 得到可读性较强的 `user/call.asm` 文件

3. 阅读生成的 `user/call.asm`, 并回答下列问题 (将答案写入 `answers-traps.txt` 中)

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

函数参数存储在寄存器 `a0-a7` 中, 如 `main` 函数的 `printf` 中的数 13 寄存在 `a2` 寄存器中

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.)

没有这样的代码。`g` 被内联到 `f(x)` 中, 然后 `f(x)` 有进一步被内联到 `main` 中

3. At what address is the function `printf` located?

`0x0000000000000630`, `main` 中使用 `pc` 相对寻址来计算得到这个地址

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

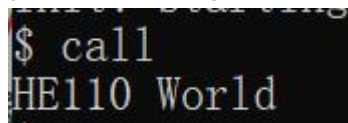
`0x0000000000000038`, `jalr` 指令的下一条汇编指令的地址

5. Run the following code.

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

What is the output? Here's an ASCII table that maps bytes to characters.

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?



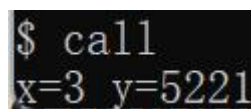
输出是 “HE110 World”

大端序将 `unsigned int i` 的值改为 `0x00726c64`

不需要, `57616` 的十六进制是 `110`, 无论端序 (十六进制和内存中的表示不是同一个概念)

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);
```



输出的是一个受调用前的代码影响的“随机”的值。因为 `printf` 尝试读的参数数量比提供的参数数量多。第二个参数 `3` 通过 `a1` 传递, 而第三个参数对应的寄存器 `a2` 在调用前不会被设置为任何具体的值, 而是会包含调用发生前的任何已经在里面的值。

三、实验中遇到的问题及解决方法

问题: HE110 World 输出的原因

解决: `57616 = 0xE110`, 即十六进制输出为 `E110`; `0x64/0x6c/0x72` 分别对应 `d/l/r`, 又因为是小端序即从低地址往高地址读 (从右往左), 对应 `rlid`

四、实验心得

在本次实验中, 深入探讨了 RISC-V 汇编语言的基本概念, 并通过分析 xv6 操作系统的源码加深了理解。在实验过程中, 首先熟悉了函数参数的传递机制, 了解到在 RISC-V 架构中,

函数的参数通过寄存器 **a0** 至 **a7** 传递。此外，还观察到编译器在某些情况下会内联函数调用，这种优化可以消除函数调用的开销，直接将函数的代码嵌入调用点，提升程序执行效率。

总之本次实验加深了我们对计算机体系结构和汇编语言的理解，并对实际编程中的内存管理和数据表示有了更清晰的认识。

Backtrace

一、实验目的

在 `kernel/printf.c` 中实现一个 `backtrace()` 函数，能够在在发生错误的位置上方堆栈上的函数调用列表。

二、实验步骤

1. 在 `defs.h` 中添加 `void backtrace(void);` 函数声明
2. 在 `sysproc.c` 的 `sys_sleep` 函数返回之前添加 `backtrace()` 函数调用
3. 在 `kernel/riscv.h` 中添加 `r_fp()` 函数，以便 `backtrace()` 能返回当前页的指针
4. 在 `kernel/printf.c` 中实现 `backtrace()` 函数
5. 在 `qemu` 环境中使用 `bttest` 命令测试，得到返回地址
6. 退出 `qemu`，在终端中运行 `addr2line -e kernel/kernel`，并粘贴 `bttest` 得到的返回地址，得到的结果所对应的位置为该函数的返回值位置
7. 在 `kernel/printf.c` 的 `panic()` 函数中添加 `backtrace()` 函数调用，程序崩溃时就可以将其调用关系打印出来

三、实验中遇到的问题及解决方法

问题：不理解函数调用栈相关概念及知识

解决：通过查阅资料得知，函数调用栈由高地址往地址增长，在 `xv6` 里，一页大小为 **4KB**，栈指针保存在 `sp` 寄存器里。对于栈帧，当前栈帧的地址保存在 `s0/fp` 寄存器中，当前栈帧的地址也叫栈帧的指针（`fp`），指向该栈帧的最高处；栈帧指针往下偏移 **8** 个字节是函数返回地址，往下偏移 **16** 个字节是上一个栈帧的栈帧指针。

四、实验心得

通过这次实验，我对函数调用栈有了更深入的理解，并且掌握了在操作系统内核中实现栈回溯（`backtrace`）的方法。在实现过程中，我了解到函数调用栈是函数调用过程中维护的一种数据结构，它保存了函数调用的返回地址、局部变量以及一些其他必要的信息。

在实验的过程中，我通过查阅相关资料，逐步解决了对函数调用栈理解不足的问题。尤其是通过分析和理解栈帧结构，成功地实现了 `backtrace()` 函数。

总的来说，这次实验不仅让我学会了如何在操作系统内核中实现栈回溯，还增强了我对操作系统底层机制的理解。通过不断实践和解决问题，我的编程能力和问题解决能力也得到

了提升。

Alarm

一、实验目的

向 xv6 添加一项功能，该功能会在进程使用 CPU 时间时定期向其发出警报。这可能会对计算受限的进程很有用，这些进程希望限制它们占用的 CPU 时间，或者对想要计算但也想要采取一些定期操作的进程很有用。更一般地说，将要实现一种原始形式的用户级中断/故障处理程序。

二、实验步骤

1. 在 Makefile 文件中的 UPROGS 中加入 \$U/_alarmtest\
2. 在 user/user.h 中添加如下声明
`int sigalarm(int ticks, void (*handler)());`
`int sigreturn(void);`
3. 在 user/usys.pl 中添加存根 `entry("sigalarm");`和 `entry("sigreturn")`
4. 在 kernel/syscall.h 中添加系统调用号
`#define SYS_sigalarm 22`
`#define SYS_sigreturn 23`
5. 在 kernel/syscall.c 中添加 `extern uint64 sys_sigalarm(void);`、`extern uint64 sys_sigreturn(void);`与 `[SYS_sigalarm] sys_sigalarm`、`[SYS_sigreturn] sys_sigreturn`
6. 在 kernel/proc.h 的 `proc()`添加属性，以让 `sys_sigalarm` 记录时钟间隔
`int ticks;`
`int ticks_cnt;`
`uint64 handler;`
`struct trapframe *alarm_trapframe;`
`int alarm_goingoff;`
在 kernel/proc.c 的 `allocproc()`中初始化这些属性
`p->ticks = 0;`
`p->handler = 0;`
`p->ticks_cnt = 0;`
`p->alarm_goingoff = 0;`
7. 在 kernel/trap.c 的 `usertrap()`中的 `if(which_dev == 2)`中处理中断，当 RISC-V 上的陷阱返回到用户空间时，决定用户空间代码恢复执行的指令地址的地方为 `p->trapframe->epc`，并将其修改为 `handler`
8. 在 kernel/sysproc.c 中实现 `sys_sigalarm()`和 `sys_sigreturn()`
9. 在 qemu 中运行 `alarmtest` 进行测试，通过 `test0`、`1`、`2` 后，运行 `usertests` 确保没有破坏内核中的其他部分
10. 在终端中输入 `./grade-lab-traps` 获得最终得分

三、实验中遇到的问题及解决方法

问题：如何实现信号处理机制

解决：要实现信号处理机制，核心在于设置定时器和处理函数，以及在信号处理函数执行完毕后恢复进程的执行状态。`sys_sigalarm` 系统调用主要用于设置定时器。当定时器到达设定的时间间隔时，系统会自动调用预先设定的处理函数。调用时需要提供两个参数：一个

是定时器的时间间隔，另一个是处理函数的地址。在实现中，需要将这些参数存储在当前进程的相应结构体中。这样，定时器一旦触发，系统就能调用指定的处理函数。`sys_sigreturn`系统调用当处理函数执行完毕后，系统需要恢复进程的原始执行状态。为此，`sys_sigreturn`会恢复信号处理函数触发之前的进程状态。实现过程中，可以利用内存拷贝函数将保存的旧状态恢复到当前状态。这确保了信号处理函数执行完毕后，进程能继续之前的操作。

四、实验心得

在本次实验中，我通过向 `xv6` 操作系统添加一个基于时间间隔的信号处理机制，深入理解了操作系统对进程控制的实现方式。整个过程使我更加熟悉了操作系统中的定时器和中断处理机制。具体来说，我学习了如何在操作系统内核中添加自定义的系统调用，以及如何利用这些调用设置用户级别的中断处理程序。通过编写代码和调试，我更深刻地认识到信号处理在多任务处理中的重要性，尤其是在需要定期执行某些操作的场景中。此外，通过完成这个项目，我增强了对内核结构的理解，如进程控制块、陷阱处理等。这个实验不仅让我在技术上得到了提高，也让我更加体会到了系统编程的复杂性和挑战性。

实验结果

```
root@f90417feedb3: /xv6-labs-2021# ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (4.5s)
== Test running alarmtest == (6.7s)
== Test  alarmtest: test0 ==
alarmtest: test0: OK
== Test  alarmtest: test1 ==
alarmtest: test1: OK
== Test  alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (168.4s)
== Test time ==
time: OK
Score: 85/85
```

Lab5:Copy-on-Write Fork for xv6

一、实验目的

实现写时拷贝（Copy-on-Write，简称 COW）的 `fork()`。推迟为子进程分配和复制物理内存页面，直到真正需要这些副本为止（如果有的话）。

二、实验步骤

1. 使用以下命令，切换到 lab5 实验环境
`git fetch`
`git checkout cow`
`make clean`
2. 在 `kernel/riscv.h` 中定义 `PTE_COW`，表示 RISC-V 中的 RSW(reserved for software)位。这是为了在页表项中标记哪些页是 COW 页：`#define PTE_COW(1L<<8)`
3. 在 `kernel/vm.c` 中修改 `uvmcopy()`，在复制父进程的内存到子进程的时候，不立刻复制数据，而是建立指向原物理页的映射，并将父子两端的页表项都设置为不可写
4. 在 `kernel/trap.c` 中的 `usertrap()` 中添加对 `page fault` 的检测，当 COW 页面上发生页面错误时，使用 `kalloc()` 分配一个新页面，将旧页面复制到新页面，并将新页面安装在 PTE 中并设置 `PTE_W`。同时 `copyout()` 由于是软件访问页表，不会触发缺页异常，所以需要手动添加同样的监测代码
5. 在 `kernel/vm.c` 中实现懒复制页的检测（`uvmcheckcowpage()`）和实复制（`uvmcowcopy()`）操作
6. 在 `kernel/kalloc.c` 中完成在支持懒复制的条件下的物理页生命周期管理。
`kalloc()`: 分配物理页，将其引用计数置为 1
`krefpage()`: 创建物理页的一个新引用，引用计数加 1
`kcopy_n_deref()`: 将物理页的一个引用实复制到一个新物理页上（引用计数为 1），返回得到的副本页；并将本物理页的引用计数减 1
`kfree()`: 释放物理页的一个引用，引用计数减 1；如果计数变为 0，则释放回收物理页

三、实验中遇到的问题及解决方法

问题 1：一开始没有考虑只读页的拷贝，导致单纯的非 COW 只读页被错误标记为 COW 页从而变成可写页

解决：进行判断 `if(*pte & PTE_W)`，不可写+`PTE_COW`，或者如果父进程页本身单纯只读非 COW，则子进程页同样只读且无 COW 标识

问题 2：只检查了第一个目标页的 COW 状态，对于跨越多页的 `copyout`，如果目标页中有多个 COW 页，只有刚好在地址范围开头的第一个页会被检查，导致共享页被误写

解决：`if(uvmcheckcowpage(dstva))`

`uvmcowcopy(dstva);`

在 `copyout()` 中对每一个目标页，在写入之前都对 COW 标志位进行检查。并实现 `uvmcheckcowpage` 函数

四、实验心得

通过这次实验，我深入理解了写时拷贝（Copy-on-Write, COW）在操作系统中的重要性，并成功实现了 COW fork 的功能。COW 机制通过推迟物理页的分配和复制，大大节省了内存资源，使得父子进程可以共享相同的物理页，直到其中一个进程试图修改这些页时才会触发真正的复制操作。在实现过程中，我遇到了如何正确标记 COW 页面的挑战，尤其是在处理只读页时需要特别小心。通过对页表项的精确判断，避免了将单纯的只读页错误地标记为 COW 页。实验让我更加体会到操作系统内存管理的复杂性和巧妙设计，也提高了我对页表操作和内存保护机制的理解。整个过程不仅加深了我对 xv6 内核的理解，也让我对写时拷贝这一概念有了更加全面的认识。

实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (10.5s)
== Test    simple ==
    simple: OK
== Test    three ==
    three: OK
== Test    file ==
    file: OK
== Test usertests == (143.8s)
== Test    usertests: copyin ==
    usertests: copyin: OK
== Test    usertests: copyout ==
    usertests: copyout: OK
== Test    usertests: all tests ==
    usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```

Lab6:Multithreading

本实验将帮助您熟悉多线程。您将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现屏障。

Uthread:switching between threads

一、实验目的

创建线程并保存/恢复寄存器以在线程之间切换

二、实验步骤

1. 使用以下命令，切换到 lab6 实验环境

```
git fetch
git checkout thread
make clean
```

2. 将以下汇编代码添加至 user/uthread_switch.S 中

```
sd ra, 0(a0)
sd sp, 8(a0)
sd s0, 16(a0)
sd s1, 24(a0)
sd s2, 32(a0)
sd s3, 40(a0)
sd s4, 48(a0)
sd s5, 56(a0)
sd s6, 64(a0)
sd s7, 72(a0)
sd s8, 80(a0)
sd s9, 88(a0)
sd s10, 96(a0)
sd s11, 104(a0)
ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
ld s1, 24(a1)
ld s2, 32(a1)
ld s3, 40(a1)
ld s4, 48(a1)
ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)
```

3. 在 user/uthread.c 中修改以下内容

- 定义 context 结构体（借鉴 proc.h）用于保存寄存器内容，并将其加入到 thread 结构体中（命名为 ctx）
- 在 thread_schedule() 中调用 thread_switch 进行上下文切换
thread_switch((uint64)&t->ctx, (uint64)&next_thread->ctx);
- 补全 thread_create(), 实现找到空闲位置，设置接下来要跑的线程栈以及运行其位置，然后切换线程

三、实验中遇到的问题及解决方法

问题：错误地认为栈是从低地址向高地址增长的，这样加入新元素后覆盖栈以外的其他内存区域，从而引发了程序崩溃

解决：将 sp 指针设置为指向 stack 的最高地址，确保栈增长方向正确

四、实验心得

在这个实验中，我对多线程的底层实现有了更深入的理解。通过手动实现线程的创建和切换，我不仅掌握了如何保存和恢复寄存器的内容，还学习了如何在用户级别进行线程的管理和调度。

我最初认为栈是从低地址向高地址增长的，结果在加入新元素后导致栈外的内存区域被覆盖，最终导致程序崩溃。在调试的过程中，通过查阅资料了解到栈实际上是从高地址向低地址增长的。为了修复这个问题，我将栈指针（sp）设置为指向栈的最高地址，从而确保栈的增长方向是正确的。

通过这个实验，我深刻认识到底层系统编程的复杂性，以及精确理解和操作硬件资源的重要性。这次的实践不仅提高了我对线程和栈管理的理解，也让我在调试复杂问题时更加耐心和细致。

Using threads

一、实验目的

使用哈希表探索线程和锁的并行编程。在具有多个内核的真实 Linux 或 MacOS 计算机（不是 xv6，不是 qemu）上完成此作业。此作业使用 UNIX pthread 线程库。

二、实验步骤

1. 在 notxv6/ph.c 中进行如下修改
 - 声明锁：pthread_mutex_t locks[NBUCKET];
 - 在 main 函数中初始化锁

```
for(int i=0;i<NBUCKET;i++) {
    pthread_mutex_init(&locks[i], NULL);
}
```
 - 在 put() 和 get() 中上锁（pthread_mutex_lock(&locks[i]);）与开锁（pthread_mutex_unlock(&locks[i]);）
2. 在计算机终端执行程序验证结果

三、实验中遇到的问题及解决方法

问题：锁的粒度问题

解决：通过降低锁的粒度，从而优化多线程效率。由于哈希表中，不同的 bucket 是互不影响的，一个 bucket 处于修改未完全的状态并不影响 put 和 get 对其他 bucket 的操作，所以实际上只需要确保两个线程不会同时操作同一个 bucket 即可，并不需要确保不会同时操作整个哈希表。所以可以将加锁的粒度，从整个哈希表一个锁降低到每个 bucket 一个锁。

四、实验心得

在本次实验中，我进一步加深了对多线程编程的理解，尤其是在并行编程中锁的使用和粒度控制的重要性。通过将锁的粒度降低到每个 bucket，我发现不同线程可以并行操作不同

的 bucket，从而提高了程序的并发性能。

这次实验不仅让我掌握了 `pthread` 库的基本用法，还让我意识到在实际开发中，合理的锁粒度设计对程序性能的影响至关重要。未来的开发中，我会更加注重锁的设计，力求在保证线程安全的前提下，最大限度地提高并行效率。

Barrier

一、实验目的

实现一个屏障：应用程序中所有参与线程都必须等待的点，直到所有其他参与线程也到达该点。使用 `pthread` 条件变量，这是一种类似于 xv6 的睡眠和唤醒的序列协调技术。应该在真实的计算机（不是 xv6，不是 qemu）上完成这个作业。

二、实验步骤

在 `notxv6/barrier` 中按照要求和提示补充完成 `barrier()` 函数，七步骤可分解如下

1. 获取互斥锁 `pthread_mutex_lock(&bstate.barrier_mutex);`，确保了共享变量的线程安全访问，防止竞争条件的发生
2. 增加线程计数：通过自增 (`++bstate.nthread`)，当前线程表明它已到达屏障
3. 检查屏障条件：如果到达屏障的线程数量 (`bstate.nthread`) 小于需要同步的总线程数 (`nthread`)，当前线程调用 `pthread_cond_wait`，线程在条件变量 `barrier_cond` 上等待，在等待期间，`barrier_mutex` 互斥锁会被释放，允许其他线程访问共享状态，一旦线程被唤醒（即条件变量被广播），它会自动重新获取互斥锁，然后继续执行。
4. 屏障释放：如果条件 `++bstate.nthread < nthread` 为假，即当前线程是最后一个到达屏障的线程。`nthread` 计数被重置为 0，为下一次同步做好准备。此外，`bstate.round++` 递增，用于记录屏障被通过的次数。`pthread_cond_broadcast` 唤醒所有在 `barrier_cond` 条件变量上等待的线程，这些线程会重新获取互斥锁并继续通过 `pthread_cond_wait` 之后的代码
5. 释放互斥锁 `pthread_mutex_unlock(&bstate.barrier_mutex);` 允许其他可能需要访问 `bstate` 的操作继续执行。

三、实验中遇到的问题及解决方法

暂无

四、实验心得

在本次实验中，我深入理解并实践了多线程同步中的屏障机制。通过实现一个基于 `pthread` 的屏障，我学会了如何使用条件变量和互斥锁来确保线程之间的协调与同步。

在代码实现过程中，我认识到在多线程环境下保证共享资源的安全访问是至关重要的。通过使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 对共享状态进行保护，避免了竞争条件的发生。`pthread_cond_wait` 和 `pthread_cond_broadcast` 的使用进一步加深了我对条件变量工作原理的理解，特别是在实现线程等待和唤醒的过程中。

此次实验也让我意识到线程同步中的细节处理是如何直接影响程序的正确性和效率的。例如，在屏障释放时，确保重置计数器和广播条件变量，才能让所有等待的线程安全地继续执行。通过这些操作，我掌握了如何设计一个健壮的屏障来协调多个线程。

总的来说，本实验不仅帮助我巩固了多线程编程的基础知识，还让我在实际操作中体验到了同步机制的复杂性和重要性。通过解决可能出现的竞争条件和死锁问题，我进一步提高了自己在多线程环境下编写可靠代码的能力。

实验结果

```
root@f90417feedb3: ~/xv6-labs-2021# ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (4.6s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (7.8s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (15.0s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (3.9s)
== Test time ==
time: OK
Score: 60/60
```

Lab7:Networking

一、实验目的

为网络接口卡 (NIC) 编写一个 `xv6` 设备驱动程序。包括创建以太网驱动程序，处理 ARP 请求和响应，实现 IP 数据包的发送和接收，处理 ICMP Echo 请求和响应，实现 UDP 数据包的发送和接收，以及实现一个简单的 DHCP 客户端以从 DHCP 服务器获取 IP 地址。通过这个实验，将深入理解网络协议栈的工作原理。

二、实验步骤

1. 使用以下命令，切换到 lab7 实验环境

```
git fetch
git checkout net
make clean
```


2. 按照要求以及 hints 在 kernel/e1000.c 中完成 e1000_transmit() 函数和 e1000_recv() 函数，以便驱动程序可以发送和接收数据包

3. e1000_transmit() 函数用于传输数据包，其由驱动程序主动调用来传输数据包，实现过程如下

- 首先，获取 E1000 的锁，防止多进程同时发送数据出现 race，接着通过读取 E1000_TDT 控制寄存器，向 E1000 询问等待下一个数据包的索引（即下一个可用的 buffer 下标）
`acquire(&e1000_lock);`
`uint32 ind = regs[E1000_TDT];`
- 接着获取 buffer 的描述符，其中存储了关于该 buffer 的各种信息，检查是否溢出，如果该 buffer 中的数据还未传输完，则代表我们已经将环形 buffer 列表全部用完，缓冲区不足，返回错误
`struct tx_desc *desc = &tx_ring[ind];`
`if(!(desc->status & E1000_TXD_STAT_DD)) {`
 `release(&e1000_lock);`
 `return -1;`
`}`
- 否则如果该下标仍有之前发送完毕但未释放的 mbuf，则释放
`if(tx_mbufs[ind]) {`
 `mbuffree(tx_mbufs[ind]);`
 `tx_mbufs[ind] = 0;`
`}`
- 然后填写描述符。。m->head 指向内存中数据包的内容，m->len 是数据包的长度。设置必要的 cmd 标志，并保存指向 mbuf 的指针，以便稍后释放
`desc->addr = (uint64)m->head;`
`desc->length = m->len;`
`desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;`
`tx_mbufs[ind] = m;`
- 最后，通过对 E1000_TDT 加 1 再对 TX_RING_SIZE 取模来更新环形缓冲区
`regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;`
- 如果 e1000_transmit() 成功地将 mbuf 添加到环型缓冲区中，则返回 0。如果失败(例如，没有可用的描述符来传输 mbuf)，则返回 -1，以便调用方知道应该释放 mbuf。（第二步中有实现）

4. e1000_recv(void) 函数用于接收数据，当操作系统接受到数据包时，其将产生设备中断并在 devintr 中调用 e1000_intr，该函数调用 e1000_recv 完成对数据包的具体接受工作，其实现过程如下：

- 写一个循环处理，因为每次 recv 可能接收多个包
- 首先通过对 E1000_RDT 控制寄存器加 1 再对 TX_RING_SIZE 取模，来向 E1000 询问下一个等待接受数据包（如果有）所在的环索引
`uint32 ind = (regs[E1000_RDT] + 1) % RX_RING_SIZE;`
- 然后通过检查描述符 status 部分中的 E1000_RXD_STAT_DD 位来检查，如果需要接收的包都已经接收完毕，则退出
`struct rx_desc *desc = &rx_ring[ind];`
`if(!(desc->status & E1000_RXD_STAT_DD)) {`

- ```

 return;
 }

```
- 否则，将 mbuf 的 m->len 更新为描述符中报告的长度。使用 net\_rx() 将 mbuf 传递给上层网络栈，上层负责释放 mbuf  
`rx_mbufs[ind]->len = desc->length;`  
`net_rx(rx_mbufs[ind]);`
  - 然后使用 mbuf\_alloc() 分配一个新的 mbuf，以替换刚刚给 net\_rx() 的 mbuf，供给下一次轮到该下标时使用，将其数据指针(m->head)编程到描述符中，并将描述符的状态位清除为零  
`rx_mbufs[ind] = mbuf_alloc(0);`  
`desc->addr = (uint64)rx_mbufs[ind]->head;`  
`desc->status = 0;`
  - 最后，将 E1000\_RDT 寄存器更新为最后处理的环描述符的索引  
`regs[E1000_RDT] = ind;`

### 三、实验中遇到的问题及解决方法

问题：理解 CPU 与操作系统与外围设备交互的机制（传输与接收）

解决：查阅"E1000 Software Developer's Manual"，并查看 kernel/e1000\_dev.h 文件中的相关实现。从软件角度来看，E1000 的数据接收和传输主要依赖于一种叫做描述符(descriptor)的数据结构。描述符就像一个信息枢纽，包含了与网络数据包相关的所有关键信息，包括数据的存储位置（地址）、数据的大小（长度）、数据的完整性信息（校验和），以及描述符的当前状态和可能的错误信息。通过读取和修改描述符，硬件和软件能够共享数据，从而实现高效的通信。

### 四、实验心得

在这次实验中，我深入学习了网络协议栈的工作原理，并通过实际编写网络接口卡 (NIC) 驱动程序来巩固了这些知识。这次实验包括多个关键模块的实现，如以太网驱动程序、ARP 请求和响应、IP 数据包的发送和接收、ICMP Echo 请求和响应、UDP 数据包的发送和接收以及简单的 DHCP 客户端实现。

在编写 e1000\_transmit() 和 e1000\_recv() 函数时，我深入理解了 E1000 网卡的工作机制，特别是描述符的使用。描述符在硬件与软件之间的通信中起到了关键作用，通过描述符，硬件和软件能够高效地共享数据并协同工作。在实现过程中，我遇到了一些挑战，如理解 E1000 寄存器的操作和管理环形缓冲区的机制。通过查阅资料以及分析 xv6 内核代码，我最终掌握了这些机制并成功实现了数据包的传输和接收。

此外，这次实验还让我更深入地理解了 CPU 与外围设备的交互机制，特别是在操作系统层面如何管理硬件资源和处理数据传输。通过这次实验，我不仅提升了对网络协议和驱动程序开发的理解，还进一步加深了对操作系统内核工作原理的认识。这些知识和经验将对我未来的学习和工作起到重要的作用。

## 实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (5.7s)
== 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
== Test time ==
time: OK
Score: 100/100
```

## Lab8:Locks

多核机器上并行性差的一个常见症状是锁争用率高。提高并行性通常需要更改数据结构和锁定策略以减少争用。此实验通过重新设计代码以降低锁竞争，提高多核机器上系统的并行性。

### Memory allocator

#### 一、实验目的

通过拆分 `kmem` 中的空闲内存链表，降低 `kalloc` 实现中的 `kmem` 锁竞争。

#### 二、实验步骤

1. 使用以下命令，切换至 `lab8` 实验环境

```
git fetch
git checkout lock
git clean
```
2. 在 `kernel/kalloc.c` 中修改代码
  1. 为每个 CPU 分配独立的 `freelist`，并用独立的锁保护它

```
struct {
 struct spinlock lock;
 struct run *freelist;
} kmem[NCPU];
```
  2. 初始化所有锁

```
void kinit()
{
 char lockname[10];
```

```

for(int i = 0; i < NCPU; i++) {
 snprintf(lockname, 10, "kmem_CPU%d", i);
 initlock(&kmem[i].lock, lockname);
}
freerange(end, (void*)PHYSTOP);
}

```

3. 修改 `kfree()` 以便对这些锁进行释放，适应新的数据结构，使用 `push_off()` 和 `pop_off()` 来关闭和打开中断，以便 `cpuid()` 能够正确获得当前 CPU 编号

```

push_off();
int cpu_id = cpuid();
acquire(&kmem[cpu_id].lock);
r->next = kmem[cpu_id].freelist;
kmem[cpu_id].freelist = r;
release(&kmem[cpu_id].lock);
pop_off();

```

4. 修改 `kalloc()` 函数对申请空间中的锁进行处理，其具体实现逻辑如下

- 关闭中断并获取当前 CPU 的 id
- 获取当前 CPU 的内存池锁，并尝试从当前 CPU 的 `freelist` 中分配内存，如果有空闲则返回，如果 `freelist` 为空，则执行下一步，“内存偷取”操作
- 开始遍历所有的 CPU，跳过当前 CPU，获取其他 CPU 的内存锁并检查其 `freelist`。如果 `freelist` 为空则释放该 CPU 的内存锁并尝试下一个 CPU；若果其他 CPU 的 `freelist` 非空，偷取其最多 64 个空闲块，将其移动到当前 CPU 的 `freelist` 中，同时减少 `steal_left` 计数，如果 `steal_left` 达到 0，表示已经成功偷取了 64 页，跳出循环
- 在成功偷取到内存后，再次尝试从当前 CPU 的 `freelist` 中取出一个空闲块并返回。如果 `freelist` 非空，更新 `freelist`。
- 释放当前 CPU 的内存锁，并重新启用中断，返回空闲块。如果成功分配到内存，使用 `memset` 初始化分配的内存块为特定的值。

### 三、实验中遇到的问题及解决方法

**问题：**解决锁竞争，`kalloc` 原本的实现中，使用 `freelist` 链表，将空闲物理页本身直接用作链表项（这样可以不使用额外空间）连接成一个链表，同时用一个锁保护，这样所有的访问会产生对锁的竞争

**解决：**为每个 CPU 分配独立的 `freelist`，这样多个 CPU 并发分配物理页就不再会互相排斥了，提高了并行性。另外，由于在一个 CPU `freelist` 中空闲页不足的情况下，仍需要从其他 CPU 的 `freelist` 中“偷”内存页，所以一个 CPU 的 `freelist` 并不是只会被其对应 CPU 访问，还可能在“偷”内存页的时候被其他 CPU 访问，故仍然需要使用单独的锁来保护每个 CPU 的 `freelist`。

### 四、实验心得

在本次实验中，我通过优化内存分配器中的锁使用策略，成功减少了多核系统中的锁竞争，提高了并行性。学习了如何通过为每个 CPU 独立分配空闲列表来降低锁竞争，提高多处理器的性能。

通过这次实验，我深刻体会到了在多核系统中，设计合理的数据结构和锁策略对提升系

统性能的重要性。在实际实现过程中，我也遇到了一些挑战，例如在“偷取”其他 CPU 的内存页时如何确保线程安全，以及如何处理内存不足的情况。这些问题的解决让我对并发编程有了更深入的理解。

此外，实验还让我意识到，优化锁使用不仅仅是减少锁的数量或范围，而是需要从整体上考虑系统的并发性和数据访问的频率。最终，通过这次实验，我掌握了一种有效的优化方法，并且增强了我在多核系统上进行高效并发编程的能力。

## Buffer cache

### 一、实验目的

Buffer cache 用于保存磁盘的一部分数据，减少磁盘操作的时间消耗，但这也导致所有进程都会共享这个数据结构，多个进程密集使用文件系统，它们可能会争用 `bcache.lock`，本实验通过设计和修改解决该问题

### 二、实验步骤

在 `kernel/bio.c` 中修改代码

1. 定义 `buckets` 数量

2. 修改定义 `bcache`

3. 修改 `binit()` 函数，初始化每个桶的锁和双向链表。每个桶都有一个自己的锁，用于在多线程环境中保护该桶的数据。每个桶也有一个双向链表，用于存储该桶中的所有缓冲区

4. 在 `bget()` 函数中，首先计算块号的哈希值，然后定位到相应的桶。接着，在该桶中查找是否已有对应的块缓冲区存在。如果找到了匹配的缓冲区，就直接返回。如果未找到，则需要从其他桶中寻找一个未使用的缓冲区，将其重新分配到当前桶，并设置为所需的块。此外，将 `&bcache.lock` 表达改为使用 `&bcache.hashlock[index]` 进行同步操作，避免了锁机制表达的重复

5. 在 `breles` 函数中，首先减少缓冲区的引用计数。如果引用计数降为 0，表示没有进程正在使用这个缓冲区。将缓冲区的 `lastuse_tick` 更新为当前的 `ticks`，以记录缓冲区的最近使用时间，这样可以取代将缓冲区移动到链表头部的逻辑，从而表示这个缓冲区是最近最少使用的。

### 三、实验中遇到的问题及解决方法

问题：buffer cache 的锁竞争问题，如果只使用 `bcache.lock` 一个锁，会造成很多竞争

解决：根据数据块的 `blocknumber` 将其保存进一个哈希表中，而哈希表的每个 `bucket` 都有一个相应的锁来保护，这样竞争只会发生在两个进程同时访问同一个 `bucket` 内的 `block`。

### 四、实验心得

通过本次实验，我深入理解了操作系统中缓冲区高速缓存（Buffer Cache）的重要性及其在提高磁盘 I/O 性能方面的作用。在实验过程中，我遇到了锁竞争的问题，即多个进程同时访问共享数据结构时，容易造成系统性能的下降。这使我认识到，在多线程环境下，优化锁的使用策略是至关重要的。

总的来说，这次实验让我不仅在理论上更加理解了 Buffer Cache 的设计原理，也在实践中积累了优化多线程环境下锁使用的经验。这将为我的未来操作系统开发和优化工作奠定坚实的基础。

## 实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kallocetest == (103.1s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch == kallocetest: sbrkmuch: OK (9.2s)
== Test running bcachetest == (10.7s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests == usertests: OK (154.0s)
== Test time ==
time: OK
Score: 70/70
```

## Lab9:File System

向 xv6 文件系统添加大型文件和符号链接

### Large files

#### 一、实验目的

增加 xv6 文件的最大大小，并将一个直接数据块号替换成一个两层简介数据块号，即指向一个包含间接数据块号的数据块

#### 二、实验步骤

1. 使用以下命令，切换至 lab9 实验环境

```
git fetch
```

```
git checkout fs
```

```
make clean
```

2. 修改 kernel/fs.h 中的宏定义

```
#define NDIRECT 11 //直接指向数据块的地址数量 12 -> 11
```

```
#define NINDIRECT (BSIZE / sizeof(uint)) //一个间接块可以包含的地址数量
```

```
#define NDINDIRECT NINDIRECT * NINDIRECT //一个二级间接块包含的地址数量
```

```
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT) //一个文件包含的最大数据块数量
```

3. 修改 dinode 和 inode 结构：这一步在 dinode 和 inode 结构中增加了一个地址字段，用于存储二级间接块的地址。这样做的目的是为了能够访问更多的数据块。(inode 在 kernel/file.h 中)即将 uint addr[NDIRECT+1]变为 uint addr[NDIRECT+2]

4. 修改 kernel/fs.c 中的 bmap(获取 inode 中第 bn 个块的快号)和 itrunc(释放该 inode 所使用的所有数据块)，使其能够处理二级间接块，仿照第一级的写法，只是在查出一级块

号后，需要将一级块中的数据读入，然后再次查询

### 三、实验中遇到的问题及解决方法

暂无

### 四、实验心得

在本次实验中，我通过对 **xv6** 文件系统的理解与修改，成功地实现了对大型文件的支持。这一过程让我更加深入地理解了文件系统的设计与实现，特别是关于数据块管理的机制。通过增加二级间接块，我扩展了文件系统能够管理的数据块数量，使得 **xv6** 能够处理更大的文件。这一改动虽然看似简单，但实际上涉及到了文件系统内部多个模块的协调工作，特别是在数据块分配与释放的逻辑处理上，需要仔细考虑如何正确地管理间接块和二级间接块的引用。

在编写和调试代码的过程中，我也进一步体会到了系统编程的挑战性，尤其是在对已有系统进行改动时，必须十分小心，确保所有相关的功能模块都能正确运行。总的来说，这次实验不仅让我掌握了如何扩展文件系统的功能，也提升了我对系统设计和实现的整体把握能力。这将对我未来的操作系统开发工作带来很大的帮助。

## Symbolic links

### 一、实验目的

向 **xv6** 添加符号链接。符号链接（或软链接）通过路径名引用链接文件；打开符号链接时，内核会按照链接找到引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备

### 二、实验步骤

1. 为 **symlink** 创建一个新的系统调用号，同 **lab2**。将程序以 **\$U/\_symlinktest\** 的形式添加到 **Makefile** 的 **UPROGS** 中。向 **kernel/stat.h** 添加新的文件类型(**T\_SYMLINK**)以表示符号链接。在 **kernel/fcntl.h** 中添加一个新标志(**O\_NOFOLLOW**)，该标志可用于 **open** 系统调用。

```
//user/usys.pl
entry("symlink");
//kernel/syscall.h
#define SYS_symlink 22
//kernel/syscall.c
extern uint64 sys_symlink(void);
[SYS_symlink] sys_symlink,
//kernel/stat.h
#define T_SYMLINK 4
//kernel/fcntl.h
#define O_NOFOLLOW 0x800
```

2. 在 **kernel/sysfile.c** 中实现 **sys\_symlink**

- 首先，函数获取两个参数:目标路径 **target** 和符号链接的路径 **path**，分别存储在

target 和 path 数组中。

- 然后，开始一个新的文件系统操作，保证文件系统的一致性。
- 接着，函数调用 `create()` 函数来创建一个新的符号链接。`create()` 函数的参数包括路径文件类型(这里是 `T_SYMLINK`，表示符号链接)、主设备号和次设备号。`create()` 函数返回一个指向新创建的 `inode` 的指针。
- 如果创建失败，函数会结束文件系统操作，并返回错误。
- 如果创建成功，函数会调用 `writel()` 函数将目标路径写入到新创建的符号链接的数据块中。
- `writel()` 函数的参数包括 `inode` 指针、写入的数据、偏移量和写入的数据长度。
- 如果写入失败，函数会结束文件系统操作，并返回错误。
- 如果写入成功，函数会调用 `inodeunlockput()` 函数解锁并释放 `inode`，然后结束文件系统操作。
- 最后，函数返回 0，表示操作成功。

3. 在 `kernel/sysfile.c` 中修改 `sys_open()` 函数，使其在遇到符号链接的时候，可以递归跟随符号链接，直到跟随到非符号链接的 `inode` 为止。设置最大搜索深度为 10，如果达到 10 次，则说明文件打开失败。

### 三、实验中遇到的问题及解决方法

问题：对于文件系统的理解

解决：通过查阅资料重新学习，整个文件系统包括了 7 层：文件描述符、路径名、目录、`inode`、日志、缓冲区、磁盘，具体如下：

1. **文件描述符**：这是最高层，也是用户程序直接交互的层次。文件描述符是一个非负整数，用于标识一个已打开的文件。用户程序通过系统调用(如 `read`、`write`、`close` 等)和文件描述符进行交互。

2. **路径名**：这一层处理路径名到 `inode` 的转换。当用户程序打开一个文件时，它会提供一个路径名，文件系统需要将这个路径名解析为对应的 `inode`。

3. **目录**：这一层处理目录的操作。目录是一种特殊的文件，它包含了一组文件名到 `inode` 的映射。当解析路径名时，文件系统需要查找目录来找到对应的 `inode`。

4. **inode**：这一层处理 `inode` 的操作。每个文件都有一个对应的 `inode`，其中包含了文件的元数据，如文件大小、文件类型、文件所有者和权限等，以及指向文件数据块的指针。

5. **日志**：这一层处理文件系统的日志操作。为了保证文件系统的一致性，`xv6` 使用了日志文件系统。当进行修改文件系统的操作时，会先将操作记录到日志中，然后再实际执行操作。如果在执行操作的过程中发生了错误，可以使用日志来恢复文件系统的一致性。

6. **缓冲区**：这一层处理磁盘块的缓存。为了提高性能，文件系统会将经常使用的磁盘块缓存到内存中。当需要读写磁盘块时，首先会查找缓冲区，如果缓冲区中有对应的磁盘块，就可以直接使用，否则需要从磁盘中读取。

7. **磁盘**：这是最底层，处理实际的磁盘操作。当需要读写磁盘块时，如果缓冲区中没有对应的磁盘块，就需要进行磁盘操作。

### 四、实验心得

通过此次实验，我深入了解了符号链接在文件系统实现原理以及操作系统如何处理路径解析。在实现符号链接功能的过程中，我需要在现有的文件系统结构中增加对新文件类型的支持，这不仅包括符号链接的创建，还涉及到路径解析的修改，以便在打开文件时正确处理符号链接。

此外，实验让我体会到了递归处理符号链接的复杂性，特别是在处理可能出现的循环引



用问题时，需要设置合理的递归深度以避免死循环。这使我更加理解文件系统设计中细节的把控，以及如何在保证系统功能的同时，提高系统的健壮性和可靠性。

总的来说，这次实验不仅让我掌握了符号链接的实现技术，也让我更深刻地理解了文件系统的内部机制。通过这些知识的积累，我对操作系统的整体架构有了更为全面的认识，这将有助于我未来在更复杂系统中的开发工作。

## 实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (193.4s)
(Old xv6.out.bigfile failure log removed)
== Test running symlinktest == (2.2s)
== Test symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (446.7s)
== Test time ==
time: OK
Score: 100/100
```

## Lab10:Mmap

### 一、实验目的

`mmap` 和 `munmap` 系统调用允许 UNIX 程序对其地址空间施加详细控制。它们可用于在进程之间共享内存、将文件映射到进程地址空间，以及作为用户级页面错误方案的一部分。在本实验中，需要完成向 `xv6` 添加 `mmap` 和 `munmap`，重点关注内存映射文件。

### 二、实验步骤

1. 使用以下命令，切换至 `lab10` 实验环境

```
git fetch
git checkout mmap
make clean
```
2. 将程序以 `$U/_mmaptest\` 的形式添加到 `Makefile` 的 `UPROGS` 中
3. 同 `lab2`，如下修改相应文件添加系统调用

```
// kernel/syscall.c
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
[SYS_mmap] sys_mmap,
[SYS_munmap] sys_munmap,
// kernel/syscall.h
#define SYS_mmap 22
#define SYS_munmap 23
// user/usys.pl
```

```

entry("mmap");
entry("munmap");
// user/user.h
void* mmap(void *, int, int, int, int, uint);
int munmap(void *, int);

```

4. 在 kernel/proc.h 定义 VMA(虚拟内存区域)对应的结构体, 记录 mmap 创建的虚拟内存范围的地址、长度、权限、文件。由于 xv6 内核中没有内存分配器, 因此可以声明一个固定大小的 VMA 数组, 并根据需要从该数组进行分配。大小为 16, 并将 structvma va[NVMA]; 添加到结构体 proc 中。

5. 修改 usertrap, 在延迟加载或内存映射文件的场景中。首先, 判断触发页故障的虚拟地址 (va) 是否有效。如果虚拟地址超出了进程的地址空间, 或与进程的堆栈区域重叠, 则进程应被标记为待终止。接着, 在进程的虚拟内存区域 (VMA) 中寻找包含该虚拟地址的区域。如果找到对应的 VMA, 则为该地址分配一块新的物理内存, 并将该内存清零。随后, 将 VMA 关联的文件中的数据通过 readi 读入到新分配的物理内存中。根据 VMA 的权限设置页表项的权限, 例如, 如果 VMA 允许写操作, 页表项也应设置为可写权限。最后, 使用 mappages 将虚拟地址映射到新分配的物理内存中。如果映射失败 (例如内存不足), 则应释放新分配的内存, 并将进程标记为待终止。

6. 在 kernel/sysfile.c 中实现 sys\_mmap()函数, 具体实现步骤如下

- **获取系统调用参数:** 通过调用 argaddr、argint 和 argfd, 获取映射的起始地址、长度、权限、标志、文件描述符和偏移量。
- **检查权限:** 验证文件的写权限是否与映射请求的权限一致。如果文件不具备写权限, 而映射要求写权限且是共享映射, 则返回错误。
- **检查映射长度:** 确定映射的长度是否会导致进程的虚拟内存空间超过允许的最大限制 ( $p \rightarrow sz > MAXVA - length$ )。如果超出, 则返回错误。
- **查找未使用的虚拟内存区域:** 遍历进程的虚拟内存区域(VMA)数组, 找到一个未被占用的区域用于映射。
- **设置 VMA 信息:** 在找到的空闲 VMA 中, 设置映射的相关信息, 包括起始地址、长度、文件、权限、标志、文件描述符和偏移量。
- **增加文件引用计数:** 调用 filedup()函数为文件增加引用计数。由于 VMA 现在引用了该文件, 所以需要增加文件的引用计数以防止文件在仍有引用的情况下被关闭。
- **更新进程的虚拟内存空间:** 调整进程的虚拟内存空间大小, 并返回映射的起始地址。

7. 在 kernel/sysfile.c 中实现 sys\_munmap()函数, 具体实现步骤如下

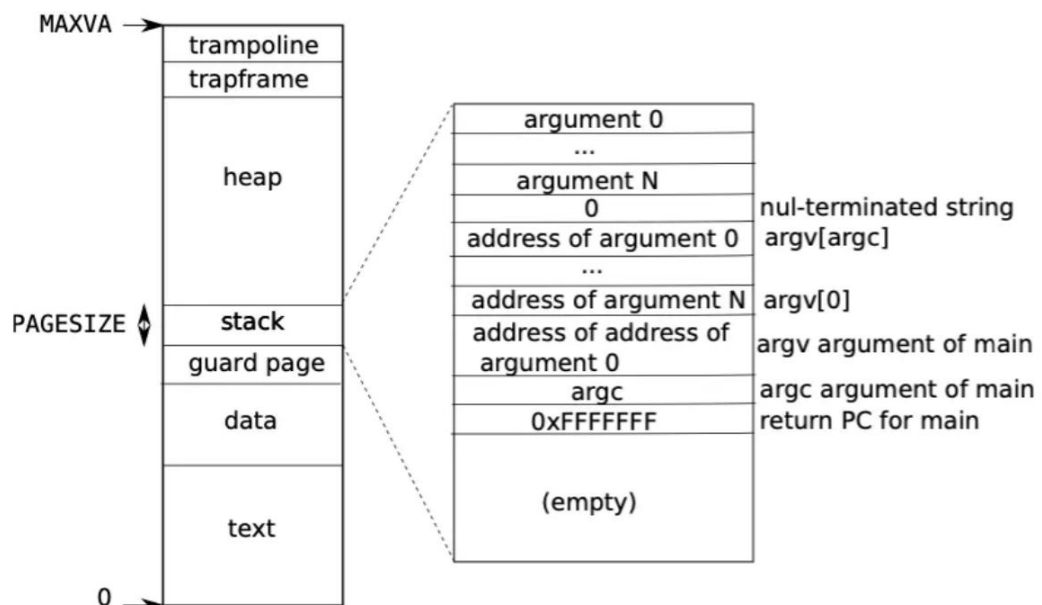
- **获取系统调用参数:** 获取需要取消映射的起始地址和长度参数。
- **调整地址和长度:** 将起始地址向下取整到页边界, 将长度向上取整到页边界, 以确保操作符合页为单位的内存管理。
- **遍历 VMA 数组:** 遍历进程的虚拟内存区域 (VMA) 数组, 找到包含需要取消映射的地址范围的区域。如果未找到符合条件的 VMA, 则直接返回。
- **处理 VMA:** 如果找到对应的 VMA, 检查其起始地址是否与需要取消映射的地址一致。如果一致, 执行以下操作:
  - 更新 VMA 的起始地址和长度, 移除与取消映射部分对应的区域。
  - 如果 VMA 的映射类型为共享映射, 调用 filewrite() 函数, 将已修改的页面 (脏页) 写回文件, 以确保数据一致性。
  - 调用 uvmunmap() 函数, 解除页表的映射, 并释放物理内存。
- **处理 VMA 的有效性:** 如果 VMA 的长度变为 0, 关闭文件, 并将 VMA 标记为无效。

- 返回结果：函数最后返回 0，表示操作成功。
8. 修改 kernel/vm.c 中的 uvmcopy()和 uvmunmap()函数，避免因不合法而 panic
  9. 在 kernel/proc.c 中修改 exit()函数，将进程已映射区域删除映射，功能与 munmap 类似
  10. 在 kernel/proc.c 中修改 fork()函数，添加对父进程文件映射的复制，确保子对象具有与父对象相同的映射区域

### 三、实验中遇到的问题及解决方法

问题：用户空间的地址分配结构

解决：通过查阅资料，了解用户的地址空间分配，heap 的范围一直从 stack 到 trapframe。由于进程本身所使用的内存空间是从低地址往高地址生长的（sbrk 调用）。为了尽量使得 map 的文件使用的地址空间不要和进程所使用的地址空间产生冲突，选择将 mmap 映射进来的文件 map 到尽可能高的位置，也就是刚好在 trapframe 下面。并且若有多个 mmap 的文件，则向下生长。



### 四、实验心得

在完成本实验的过程中，我对 mmap 和 munmap 系统调用的实现有了深入的理解，尤其是在将文件映射到进程的虚拟地址空间方面。这次实验让我意识到虚拟内存管理在操作系统设计中的重要性。通过 mmap，我们能够对进程的虚拟地址空间进行精细控制，既可以在进程间共享内存，又可以高效地将文件映射到内存中使用。

在实现这些系统调用的过程中，我学会了如何处理延迟加载和懒分配。延迟加载的核心是在发生缺页错误时，再动态地分配物理内存并读取文件数据，而不是在映射时就立即分配和加载。

此外，我在实验中也遇到了一些挑战，特别是在用户空间的地址分配上。由于进程内存空间从低地址向高地址增长，为了避免与进程内存区域发生冲突，我将文件映射的起始地址选择在靠近高地址的位置，即在 trapframe 下面，并使多个 mmap 映射的文件地址向下生长。这种策略有效避免了地址冲突，保证了文件映射的安全性和稳定性。

总的来说，通过这次实验，我不仅掌握了 mmap 和 munmap 的实现原理和技术细节，还对虚拟内存管理有了更深的理解。这为我在操作系统的其他领域进一步探索和研究打下了

坚实的基础。

## 实验结果

```
root@f90417feedb3:~/xv6-labs-2021# ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (4.1s)
== 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
== Test usertests == usertests: OK (128.3s)
== Test time ==
time: OK
Score: 140/140
```

我的仓库地址: [Maxrayyy/xv6-labs-2021](https://github.com/Maxrayyy/xv6-labs-2021): 大二下 OS 课设 (github.com)