Environment Configuration

搭建步骤

1. 安装并运行 WSL (Windows Subsystem for Linux)
2. 安装 Ubuntu 22.04 发行版本

```
wsl --install -d Ubuntu-22.04
```

3. 打开 Ubuntu 终端，更新软件包索引：

```
sudo apt update
sudo apt upgrade -y
```

4. 安装必要的开发工具和交叉编译工具链：

```
sudo apt-get install -y git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

5. 由于某些版本的 QEMU 可能不兼容，需要卸载当前版本并安装特定版本：

```
sudo apt-get remove  qemu-system-misc
sudo apt-get install qemu-system-misc=1:4.2-3ubuntu6
```

6. 确认交叉编译器版本：

```
riscv64-linux-gnu-gcc --version
```

7. 确认 QEMU 版本：

```
qemu-system-riscv64 --version
```

确认完毕后，到此实验环境配置完毕，准备开始实验

Github 代码仓库链接

https://github.com/WingWR/OS-Project-XV6

Lab Utilities

1. Boot xv6 (easy)

1.1 实验目的

获取实验室的xv6源代码，配置实验的环境

1.2 实验步骤

- 获取实验室的xv6源代码

```
$ git clone git://g.csail.mit.edu/xv6-labs-2020
$ cd xv6-labs-2020
$ git checkout util
```

- 构建并运行xv6

```
$ make qemu
```

1.3 实验中遇到的问题和解决办法

实验无问题

1.4 实验心得

配环境太折磨了

2. sleep (easy)

2.1 实验目的

实现xv6的程序sleep, sleep应该达到用户指定的计时数，一个计时数是由xv6内核定义的时间概念，即来自定时器芯片的两个中断之间的时间

2.2 实验步骤

- 在`user\user.h`中找到了`int sleep(int);`的声明

说明`sleep`接受一个`int`型整数的参数

- 编写文件`sleep.c`实现函数`int sleep(int)`

```
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char const *argv[])
{
  if (argc != 2) {
    fprintf(2, "usage: sleep <time>\n");
    exit(1);
  }

  sleep(atoi(argv[1]));
  exit(0);
}
```

2.3 实验中遇到的问题和解决办法

在编写代码时要先检测输入的数量，因为没有检测输入数量遇到错误。加上之后能正常运行

2.4 实验心得

加深了对 xv6 系统调用机制的理解

3. pingpong (easy)

3.1 实验目的

编写一个使用系统调用的程序来在两个进程之间“ping-pong”一个字节

3.2 实验步骤

- 在子进程中，分别关闭两个管道的读端口和写端口，此时子进程通过p2管道的读端口p2[0]来获取一比特信息(read())，如果得到，则打印: received ping。
- 在父进程中，分别关闭c2p，p2c两个管道的写端口和读端口，并将缓冲区buf中的内容写入p2c的写端口(write())。此时若能够从c2p的读端口读取一比特数据，则打印: received pong
- 编写文件`pingpong.c`,验证通信机制

```
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char const *argv[])
{
    ##define RD 0 //pipe的read端
    ##define WR 1 //pipe的write端

    char buf = 'P'; //传送字节
    int exit_status = 0;    //错误判断

    int p2c[2], c2p[2]; //创建双管道
    pipe(p2c);
    pipe(c2p);

    int pid = fork();
    if(pid == 0){   //子进程
        close(p2c[WR]);
        close(c2p[RD]);

        if (read(p2c[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received ping\n", getpid());
        }

        if (write(c2p[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child write() error!\n");
            exit_status = 1;
        }

        close(p2c[RD]);
        close(c2p[WR]);

        exit(exit_status);
    }
    else{
        close(p2c[RD]);
        close(c2p[WR]);

        if (write(p2c[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent write() error!\n");
            exit_status = 1;
        }
```

```
        if (read(c2p[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received pong\n", getpid());
        }

        close(p2c[WR]);
        close(c2p[RD]);

        exit(exit_status);
    }

}
```

3.3 实验中遇到的问题和解决办法

管道操作不知道如何实现，学习了相关的系统调用

a. fork系统调用：用于创建子进程；
- b. pipe系统调用：用于创建管道；
- c. read系统调用：用于读取管道：
- d. write系统调用：用于写管道；
- e. getpid系统调用：用于查询进程号；

3.4 实验心得

验证了验证Xv6的部分进程通讯机制

4. primes (moderate)/(hard)

4.1 实验目的

使用管道编写prime sieve(筛选素数)的并发版本

4.2 实验步骤

- 使用 pipe(p) 创建初始管道，父进程将 2~35 写入。
- 子进程执行 primes(p)，从管道中读取第一个数作为当前素数，并打印该素数
- 为剩余数创建一个新的管道 pipe(p2)；
- 把不能被该素数整除的数写入新管道；
- 然后再为新管道递归创建一个子进程，继续筛选。
- 编写文件`primes.c`

```
void primes(int lpipe[2])
{
  close(lpipe[WR]);
  int first;
  if (lpipe_first_data(lpipe, &first) == 0) {
    int p[2];
    pipe(p); // 当前的管道
    transmit_data(lpipe, p, first);
```

```
    if (fork() == 0) {
      primes(p);
    } else {
      close(p[RD]);
      wait(0);
    }
  }
  exit(0);
}
```

4.3 实验中遇到的问题和解决办法

如果子进程没有关闭写端或父进程没关读端，read 永远等不到 EOF。

- 在子进程中需要关闭写端，父进程中要关闭读端

4.4 实验心得

理解了 xv6 中的 pipe() 与 fork() 的协同工作机制

5. find（难度：Moderate）

5.1 实验目的

写一个简化版本的`find`程序,查找目录树中具有特定名称的所有文件

5.2 实验步骤

- 浏览学习user/ls.c中读目录的方法
- 在目录中递归寻找特定名字的文件
- 编写`find.c`

```
void find(char *path, const char *filename)
{
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;

  if ((fd = open(path, 0)) < 0) {
    fprintf(2, "find: cannot open %s\n", path);
    return;
  }

  if (fstat(fd, &st) < 0) {
    fprintf(2, "find: cannot fstat %s\n", path);
    close(fd);
    return;
  }

  //参数错误，find的第一个参数必须是目录
  if (st.type != T_DIR) {
    fprintf(2, "usage: find <DIRECTORY> <filename>\n");
```

```
    return;
  }

  if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
    fprintf(2, "find: path too long\n");
    return;
  }
  strcpy(buf, path);
  p = buf + strlen(buf);
  *p++ = '/'; //p指向最后一个'/'之后
  while (read(fd, &de, sizeof de) == sizeof de) {
    if (de.inum == 0)
      continue;
    memmove(p, de.name, DIRSIZ); //添加路径名称
    p[DIRSIZ] = 0;               //字符串结束标志
    if (stat(buf, &st) < 0) {
      fprintf(2, "find: cannot stat %s\n", buf);
      continue;
    }
    //不要在“.”和“..”目录中递归
    if (st.type == T_DIR && strcmp(p, ".") != 0 && strcmp(p, "..") != 0) {
      find(buf, filename);
    } else if (strcmp(filename, p) == 0)
      printf("%s\n", buf);
  }

  close(fd);
}
```

5.3 实验中遇到的问题和解决办法

未遇到问题

5.4 实验心得

理解了文件查找的过程

6. xargs (moderate)

6.1 实验目的

编写一个简化版UNIX的xargs程序，从标准输入中按行读取，并且为每一行执行一个命令，将行作为参数提供给命令

6.2 实验步骤

- 使用有限状态自动机（FSA）解析输入，识别参数边界；
- 使用数组 x_argv 构造命令参数，添加输入内容；
- 每行结束即调用 fork + exec 执行命令；
- 测试输入多行、空格间隔、末尾多空格等边界情况是否正常处理

6.3 实验中遇到的问题和解决办法

空格、换行混合情况处理不一致：引入状态机判断字符类型和状态转换，使输入处理更加鲁棒；

6.4 实验心得

本次实验让我深入理解了命令参数的构造和 xargs 的行为原理，通过状态机设计有效提高了解析复杂输入的能力。同时熟悉了 fork、exec、wait 等系统调用的协作流程，是一次非常有收获的系统编程练习。

Lab System Calls

1. System call tracing (moderate)

1.1 实验目的

创建一个新的`trace`系统调用来控制跟踪

1.2 实验步骤

- 修改 `Makefile`：
  在 UPROGS 变量中添加:

```
  $U_trace\
```

- 定义系统调用编号:
  在 `kernel/syscall.h` 文件中添加:

```
##define SYS_trace 22
```

- 添加系统调用处理函数:
  在 `kernel/syscall.c` 文件中添加:

```
uint64
sys_trace(void)
{
  // 获取系统调用的参数
  argint(0, &(myproc()->trace_mask));
  return 0;
}
```

- 注册系统调用:
  在 `kernel/syscall.c` 文件中添加:

```
extern uint64 sys_trace(void);

static uint64 (*syscalls[])(void) = {
  ...
  [SYS_trace]    sys_trace,
};

static char *syscalls_name[] = {
  ...
  [SYS_trace]    "trace",
};
```

- 在 `proc.h` 中添加字段:

```
struct proc {
  ...
  int tracemask;
};
```

- 在 `user/user.h` 添加声明:

```
int trace(int);
```

- 注册系统调用封装函数:
  打开 user/usys.pl，在尾部添加:

```
entry("trace");
```

1.3 实验中遇到的问题和解决办法

- 调用 trace 函数时找不到定义:
  在 user/usys.pl 中添加 entry("trace");，重新生成用户态系统调用接口。
- 发生`undefined reference to 'trace'`，链接时找不到对应实现。
  在内核代码中实现 `sys_trace` 函数，并在 `syscall.c` 中正确声明和注册。

1.4 实验心得

通过本次实验，深入理解了 xv6 系统调用的工作流程，包括用户态调用接口的生成、内核系统调用的注册与调度过程。

熟悉了内核数据结构 proc 的扩展和进程控制信息的保存方法。

2. Sysinfo (moderate)

2.1 实验目的

添加一个系统调用`sysinfo`，它收集有关正在运行的系统的信息

2.2 实验步骤

1. 在`Makefile`的`UPROGS`中添加`$U/_sysinfotest`
2. 在`kernel/kalloc.c`中添加一个函数用于获取空闲内存量

```
void
freebytes(uint64 *dst)
{
  *dst = 0;
  struct run *p = kmem.freelist; // 用于遍历

  acquire(&kmem.lock);
  while (p) {
    *dst += PGSIZE;
    p = p->next;
  }
  release(&kmem.lock);
}
```

3. 在`kernel/proc.c`中添加一个函数获取进程数

```
void
procnum(uint64 *dst)
{
  *dst = 0;
  struct proc *p;
  for (p = proc; p < &proc[NPROC]; p++) {
    if (p->state != UNUSED)
      (*dst)++;
  }
}
```

4. 实现`sys_sysinfo`，将数据写入结构体并传递到用户空间

```
uint64
sys_sysinfo(void)
{
  struct sysinfo info;
  freebytes(&info.freemem);
  procnum(&info.nproc);

  // 获取虚拟地址
  uint64 dstaddr;
  argaddr(0, &dstaddr);

  // 从内核空间拷贝数据到用户空间
  if (copyout(myproc()->pagetable, dstaddr, (char *)&info, sizeof info) < 0)
    return -1;

  return 0;
}
```

5. 在`user/user.h`中声明`sysinfo()`的原型

```
struct sysinfo;
int sysinfo(struct sysinfo *);
```

2.3 实验中遇到的问题和解决办法

1. 没有在 `sysproc.c` 文件中声明 `freebytes` 函数和 `procnum` 函数:

在 sysproc.c 中添加函数声明:

```
extern void freebytes(uint64 *dst);
extern void procnum(uint64 *dst);
```

2. 用户空间缺少 `sysinfo()` 函数的封装声明:

在 `usys.pl` 中添加 `entry("sysinfo");`，重新生成用户态系统调用接口。

在 `user/user.h` 中添加函数声明:

```
int sysinfo(struct sysinfo *);
```

3. 宏 `SYS_sysinfo` 未在头文件中定义或未生效:

在 `kernel/syscall.h` 中添加定义:

```
##define SYS_sysinfo 23
```

2.4 实验心得

本实验帮助我理解了如何在 xv6 中设计和实现一个新的系统调用流程，从用户态调用、参数传递、内核态实现再到数据回传。

Lab Page Tables

1. Print a page table (easy)

1.1 实验目的

定义一个名为 `vmprint()` 的函数。它应当接收一个 `pagetable_t` 作为参数，并以下面描述的格式打印该页表

```
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
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

1.2 实验步骤

1. 在`exec.c`中的`return argc`之前插入`if(p->pid==1) vmprint(p->pagetable);`

2. 在`kernel/vm.c`定义`vmprint()`函数

```
void
_vmprint(pagetable_t pagetable, int level){
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){
      for (int j = 0; j < level; j++){
        if (j) printf(" ");
        printf("..");
      }
      uint64 child = PTE2PA(pte);
      printf("%d: pte %p pa %p\n", i, pte, child);
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0){
        _vmprint((pagetable_t)child, level + 1);
      }
    }
  }
}

void
vmprint(pagetable_t pagetable){
  printf("page table %p\n", pagetable);
  _vmprint(pagetable, 1);
}
```

3. 在`kernel/defs.h`中声明`vmprint()`函数

```
int             copyin(pagetable_t, char *, uint64, uint64);
int             copyinstr(pagetable_t, char *, uint64, uint64);
void            vmprint(pagetable_t);
```

1.3 实验中遇到的问题和解决办法

1. 未判断递归条件导致无限递归:

判断是否为叶节点采用 (pte & (PTE_R|PTE_W|PTE_X)) == 0，确保只对非叶子页表进行递归打印。

2. 未声明函数导致无法运行:

在`kernel/defs.h`中声明`vmprint()`函数

```
int             copyin(pagetable_t, char *, uint64, uint64);
int             copyinstr(pagetable_t, char *, uint64, uint64);
void            vmprint(pagetable_t);
```

1.4 实验心得

通过递归遍历页表，能够直观地查看每一级页表及其物理地址映射，极大帮助了对虚拟内存管理的认识

2. A kernel page table per process (hard)

2.1 实验目的

修改内核来让每一个进程在内核中执行时使用它自己的内核页表的副本。修改`struct proc`来为每一个进程维护一个内核页表，修改调度程序使得切换进程时也切换内核页表

2.2 实验步骤

1. 给`kernel/proc.h`里面的`struct proc`加上内核页表的字段:

```
pagetable_t kpagetable;      // 进程的内核页表
```

2. 在`kernel/vm.c`中添加新函数`vminit()`:

```
void
vminit(pagetable_t pagetable)
{
  // uart registers
  vmmap(pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
  // virtio mmio disk interface
  vmmap(pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  // // CLINT
  // vmmap(pagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  // PLIC
  vmmap(pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  // map kernel text executable and read-only.
  vmmap(pagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
  // map kernel data and the physical RAM we'll make use of.
  vmmap(pagetable, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);
  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  vmmap(pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
}
```

3. 将`procinit()`函数中分配内核栈的代码移到`allocproc()`中:

```
static struct proc*
allocproc(void)
{
    ...
  /** 创建内核页表 */
  p->kpagetable = proc_kpagetable(p);
  if(p->kpagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
```

```
  /** 理顺内核栈的映射关系 */
  char *pa = kalloc();
  if(pa == 0)
    panic("kalloc");
  uint64 va = KSTACK(0);
  vmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
  p->kstack = va;
    ...
}
```

4. 在 `scheduler()` 函数中添加切换页表寄存器 `satp` 的指令:

```
 ...
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;

        vminithart(p->kpagetable);
        ...
      }
    }
...
```

5. 释放内核页表，但不要释放最外围的物理内存页，封装为`proc_freekpagetable()`函数:

```
void
proc_freekpagetable(pagetable_t kpagetable, uint64 sz)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i=0; i<512; i++) {
    pte_t pte = kpagetable[i];
    if((pte & PTE_V) == 0)
      continue;

    /** 该PTE有效（内部节点和叶子节点） */
    if((pte & (PTE_R|PTE_W|PTE_X)) == 0) {
      uint64 child = PTE2PA(pte);
      proc_freekpagetable((pagetable_t)child, sz);
      kpagetable[i] = 0;
    }
  }
  kfree((void*)kpagetable);
}
```

6. 在 `freeproc()` 中添加释放内核栈的功能代码:

```
static void
freeproc(struct proc *p)
```

```
{
    ...
  p->xstate = 0;

  /** 在释放内核页表之前需要释放内核栈 */
  if(p->kstack)
    uvmunmap(p->kpagetable, p->kstack, 1, 1);
  p->kstack = 0;

  if(p->kpagetable)
    proc_freekpagetable(p->kpagetable, p->sz);
  p->kpagetable = 0;

  ...
  p->state = UNUSED;
}
```

7. 修改 `virtio_disk.c` 中的地址写入指令:

```
disk.desc[idx[0]].addr = (uint64) vmpa(myproc()->kpagetable, (uint64) &buf0);
```

2.3 实验中遇到的问题和解决办法

1. 访问 `myproc()->kpagetable` 编译时报错，提示`struct proc`不完整或字段不存在:

确认所有相关文件包含了 `proc.h`

2. 重复定义 `struct spinlock`，引起编译错误:

给头文件加上防止重复包含的宏定义`(##ifndef/##define/##endif)`

3. 调用 `proc_kpagetable` 和 `proc_freekpagetable` 函数时提示隐式声明和链接错误:

在对应的头文件中声明函数原型

2.4 实验心得

本实验通过为每个进程维护独立的内核页表，增强了对多进程虚拟内存管理的理解。内核页表的独立复制有效避免了多个进程共享单一内核页表时可能产生的安全和隔离问题。实现过程中深入体会了页表结构的层次性以及内存映射的细节。

3. Simplify copyin/copyinstr (hard)

3.1 实验目的

将定义在`kernel/vm.c`中的`copyin`的主题内容替换为对`copyin_new`的调用；对`copyinstr`和`copyinstr_new`执行相同的操作

3.2 实验步骤

1. 首先添加`u2kvmcopy()`函数:

```
int
u2kvmcopy(pagetable_t old, pagetable_t new, uint64 start, uint64 end)
{
```

```
  pte_t *pte;
  uint64 pa, i;
  uint flags;

  for(i=PGROUNDUP(start); i<end; i+=PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("u2kvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("u2kvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte) & (~PTE_U);

    if(mappages(new, i, PGSIZE, pa, flags) != 0)
      goto err;
  }
  return 0;

 err:
  uvmunmap(new, start, (i-start)/PGSIZE, 0);
  return -1;
}
```

2. 更改`fork()`函数:

```
...
if(u2kvmcopy(np->pagetable, np->kpagetable, 0, np->sz) < 0) {
    freeproc(np);
    release(&np->lock);
    return -1;
  }
...
```

3. 修改`exec()`函数:

```
...
uvmunmap(p->kpagetable, 0, PGROUNDUP(oldsz)/PGSIZE, 0);
/** 在替换原用户页表之后，将新用户页表塞进内核页表中 */
if(u2kvmcopy(p->pagetable, p->kpagetable, 0, p->sz) < 0)
  goto bad;
...
```

4. 修改`growproc()`函数:

```
int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if(PGROUNDUP(sz+n) >= PLIC)
      return -1;
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0)
```

```
      return -1;
    if(u2kvmcopy(p->pagetable, p->kpagetable, p->sz, sz) < 0)
      return -1;
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
    sz = vmdealloc(p->kpagetable, p->sz, sz, 0);
  }
  p->sz = sz;
  return 0;
}
```

5. 修改 `userinit()` 函数:

```
...
u2kvmcopy(p->pagetable, p->kpagetable, 0, p->sz);
...
```

3.3 实验中遇到的问题和解决办法

1. `u2kvmcopy()` 函数中调用 `walk` 返回空指针或访问未映射页面导致 `panic`:

确保拷贝的地址范围 `start` 到 `end` 对应的用户页表都已经被正确映射

2. 在 `fork()`、`exec()`、`growproc()` 等函数中调用 `u2kvmcopy()` 后系统崩溃

调用 `u2kvmcopy()` 后，及时更新进程 `sz` 字段

3.4 实验心得

本实验揭示了页表结构设计在操作系统中的核心地位，通过为每个进程配置独立内核页表，可实现更精细化的内存隔离，防止不同进程间通过内核地址空间越权访问

Traps

1. RISC-V assembly (easy)

1.1 实验目的

阅读 `call.asm` 中函数 `g`、`f` 和 `main` 的代码，回答一些问题。

1.2 实验步骤

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

- **Answer:** 在 `a0-a7` 中存放参数，`13` 存放在 `a2` 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.)

- **Answer:** 在C代码中，`main` 调用 `f`，`f` 调用 `g`。而在生成的汇编中，`main` 函数进行了内联优化处理

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

- **Answer:** 在 `0x630`

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

- **Answer:** 执行此行代码后，将跳转到 `printf` 函数执行，并将 `PC+4=0X34+0X4=0X38` 保存到 `ra` 中

5. Run the following code.

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

What is the output?

- **Answer:**输出为：`HE110 World`

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?

- **Answer:** 若为大端存储，`i` 应改为 `0x726c6400`，不需改变 `57616`

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);`
- **Answer:** 原本需要两个参数，却只传入了一个，因此 `y=` 后面打印的结果取决于之前 `a2` 中保存的数据

1.3 实验中遇到的问题和解决办法

未遇到问题

1.4 实验心得

通过本次实验，我对 RISC-V 汇编函数调用规范 和 参数传递方式 有了更直观的认识，从底层角度理解了 C 语言与汇编之间的联系，并让我更加熟悉了寄存器使用规范和编译器优化对汇编代码的影响

2. Backtrace(moderate)

2.1 实验目的

在 `kernel/printf.c` 中实现名为 `backtrace()` 的函数

2.2 实验步骤

1. 在 `kernel/printf.c` 中实现名为 `backtrace()` 的函数

```
void
backtrace(void) {
  printf("backtrace:\n");
  // 读取当前帧指针
  uint64 fp = r_fp();
  while (PGROUNDUP(fp) - PGROUNDDOWN(fp) == PGSIZE) {
    // 返回地址保存在-8偏移的位置
    uint64 ret_addr = *(uint64*)(fp - 8);
    printf("%p\n", ret_addr);
    // 前一个帧指针保存在-16偏移的位置
    fp = *(uint64*)(fp - 16);
  }
}
```

2. 在`kernel/defs.h`中添加`backtrace`的原型

3. 在`kernel/riscv.h`中添加

```
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

2.3 实验中遇到的问题和解决办法

1. `r_fp()`未声明:

在`kernel/riscv.h`中定义`r_fp()`

2. 未成功调用`backtrace()`:

在`kernel/sysproc.c`中调用`backtrace()`

2.4 实验心得

深入理解了`RISC-V`函数调用栈帧结构 及其回溯原理。在实现`backtrace()`的过程中，我学习了如何利用帧指针`fp`在内核态中遍历调用栈，并逐层打印出返回地址，从而直观展示了函数的调用路径。

3. Alarm(Hard)

3.1 实验目的

向`xv6`添加一个特性，在进程使用CPU的时间内，XV6定期向进程发出警报

3.2 实验步骤

1. 在`user.h`中添加函数声明:

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

2. 在`sysproc.c`中定义函数:

```
uint64
sys_sigalarm(void)
{
    int ticks;
    uint64 handler;
    struct proc *p = myproc();
    if(argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
    return -1;
    p->alarminterval = ticks;
    p->alarmhandler = (void (*)())handler;
    p->alarmticks = 0;
    return 0;
}

uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  p->sigreturned = 1;
  *(p->trapframe) = p->alarmtrapframe;
  usertrapret();
  return 0;
}
```

3. 初始化警告字段

4. 修改`usertrap()`函数

```
if(which_dev == 2)
  {
    p->alarmticks += 1;
    if ((p->alarmticks >= p->alarminterval) && (p->alarminterval > 0))
    {
      p->alarmticks = 0;
      if (p->sigreturned == 1)
      {
        p->alarmtrapframe = *(p->trapframe);
        p->trapframe->epc = (uint64)p->alarmhandler;
        p->sigreturned = 0;
        usertrapret();
      }
    }
    yield();
  }
```

3.3 实验中遇到的问题和解决办法

1. 未声明`alarmtest`:

在`MakeFile`中添加`$U_alarmtest\`

2. 未声明`sigalarm`和`sigreturn`:

在`user.h`中添加`sigalarm`和`sigreturn`

3.4 实验心得

最后这个Alarm实验部分还是难度非常大，用了很久时间才完成，很多地方也并没有完全理解。但是通过完成实验的过程，
了解了时钟中断是怎么让内核定时介入用户进程的，同时还学会了保存和恢复进程状态，必须保存`trapframe`，并在`sigreturn()`中恢复

Lab Lazy Page Allocation

1. Eliminate allocation from sbrk() (easy)

1.1 实验目的

删除`sbrk(n)`系统调用中的页面分配代码，新的`sbrk(n)`应该只将进程的大小增加`n`，然后返回旧的大小

1.2 实验步骤

- 修改`sbrk()`函数:

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;

  addr = myproc()->sz;
  // lazy allocation
  myproc()->sz += n;

  return addr;
}
```

1.3 实验中遇到的问题和解决办法

未遇到什么问题

1.4 实验心得

理解了`Lazy Allocation`的概念，为后面做准备

2. Lazy allocation (moderate)

2.1 实验目的

修改`trap.c`中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行

2.2 实验步骤

1. 修改`usertrap()`函数，如果是`13`或`15`就进行下一步的处理:

```
else if(cause == 13 || cause == 15) {
    // 处理页面错误
    uint64 fault_va = r_stval();  // 产生页面错误的虚拟地址
    char* pa;                     // 分配的物理地址
    if(PGROUNDUP(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz &&
      (pa = kalloc()) != 0) {
        memset(pa, 0, PGSIZE);
        if(mappages(p->pagetable, PGROUNDDOWN(fault_va), PGSIZE, (uint64)pa,
PTE_R | PTE_W | PTE_X | PTE_U) != 0) {
          kfree(pa);
          p->killed = 1;
        }
    } else {
      // printf("usertrap(): out of memory!\n");
      p->killed = 1;
    }
  }
```

2. 修改`uvmunmap()`防止系统发生`panic`

```
for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      continue;
```

2.3 实验中遇到的问题和解决办法

暂时未遇见问题

2.4 实验心得

1. 对`usertrap()`中缺页异常的判断和处理逻辑：当出现页面错误时，需要根据`stval`找到出错的虚拟地址，并在合法范围内为其分配新的物理页并映射
2. 修改`uvmunmap()`时需要注意跳过未映射的页，否则会因为访问空指针而`panic`

3. Lazytests and Usertests (moderate)

3.1 实验目的

修改 `xv6` 内核代码，使得它能够通过 `lazytests` 和 `usertests`，即保证惰性内存分配机制在各种复杂情况下都能正常运行

3.2 实验步骤

1. 考虑`sbrk`中参数可以为负数的问题，进行修改:

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;
  struct proc *p = myproc();
  if(argint(0, &n) < 0)
    return -1;
  addr = p->sz;
  if(n < 0){
    if(p->sz + n < 0){ // 一个进程不能释放比自己大的空间
      return -1;
    }
    if(growproc(n) < 0){
      // 注意这里是实际调用 growproc 去释放空间的。
      printf("growproc err\n");
      return -1;
    }
  }else{
    myproc()->sz += n;
  }
  // if(growproc(n) < 0)
  //   return -1;
  return addr;
}
```

2. 如果一个进程出现缺页错误的地址以前并没有被分配过。那么我们就不去分配这个页，而是直接把进程`kill`:

- 用函数判断虚拟地址是否合法:

```
int is_lazy_addr(uint64 va){
  struct proc *p = myproc();
  if(va < PGROUNDDOWN(p->trapframe->sp)
  && va >= PGROUNDDOWN(p->trapframe->sp) - PGSIZE
  ){
    // 防止 guard page，这个之后会提到
    return 0;
  }
  if(va > MAXVA){
    return 0;
  }
  pte_t* pte = walk(p->pagetable, va, 0);
```

```
  if(pte && (*pte & PTE_V)){
    return 0;
  }

  if(va >= p->sz){
    return 0;
  }

  return 1;
}
```

3. 正确的处理 `fork()` 中从父进程到子进程的内存拷贝:

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      continue;
    if((*pte & PTE_V) == 0)
      continue;
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

4. 如果分配物理页的时候，没有足够内存了，应该把当前进程 kill

```
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;

  if(va >= MAXVA)
    return 0;
```

```
  if(is_lazy_addr(va)){ // 注意这里，如果是懒分配的会先分配物理地址。
    lazy_alloc(va);
  }
  pte = walk(pagetable, va, 0);

  if(pte == 0)
    return 0;
  if((*pte & PTE_V) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte);
  return pa;
}
```

5. 正确处理发生在用户栈下面地址的缺页错误

3.3 实验中遇到的问题和解决办法

- 自定义的函数`is_lazy_addr(uint64 va)`和`int lazy_alloc(uint64 va)`无法在其他文件中调用:

将函数声明在`defs.h`中，可以全局调用

3.4 实验心得

通过本次实验，对`Lazy Allocation`的机制和内核内存管理有了更深入的理解。在实现过程中，需要考虑各种情况，例如`sbrk`参数为负数时释放内存、`fork`时的内存拷贝、缺页异常的处理以及内核在访问未分配页时的应对

Lab Copy on-write

1. Implement copy-on write(hard)

1.1 实验目的

在`xv6`内核中实现`copy-on-write fork`

1.2 实验步骤

1. 在`kernel/riscv.h`中加入宏定义

```
##define PTE_V (1L << 0) // valid
##define PTE_R (1L << 1)
##define PTE_W (1L << 2)
##define PTE_X (1L << 3)
##define PTE_U (1L << 4) // 1 -> user can access
##define PTE_C (1L << 8)
```

2. 定义引用计数的全局变量`ref`

```
struct ref_stru {
  struct spinlock lock;
  int cnt[PHYSTOP / PGSIZE];  // 引用计数
} ref;
```

3. 在`kinit`中初始化`ref`的自旋锁

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  initlock(&ref.lock, "ref");
  freerange(end, (void*)PHYSTOP);
}
```

4. 修改`kalloc`和`kfree`函数，在`kalloc`中初始化内存引用计数为1，在`kfree`函数中对内存引用计数减1，如果引用计数为0时才真正删除

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // 只有当引用计数为0了才回收空间
  // 否则只是将引用计数减1
  acquire(&ref.lock);
  if(--ref.cnt[(uint64)pa / PGSIZE] == 0) {
    release(&ref.lock);

    r = (struct run*)pa;

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
  } else {
    release(&ref.lock);
  }
}

// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;
```

```
  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    acquire(&ref.lock);
    ref.cnt[(uint64)r / PGSIZE] = 1;  // 将引用计数初始化为1
    release(&ref.lock);
  }
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

5. 修改`freerange`

```
void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
    // 在kfree中将会对cnt[]减1，这里要先设为1，否则就会减成负数
    ref.cnt[(uint64)p / PGSIZE] = 1;
    kfree(p);
  }
}
```

6. 修改`uvmcopy`，不为子进程分配内存，而是使父子进程共享内存

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);

    // 仅对可写页面设置COW标记
    if(flags & PTE_W) {
      // 禁用写并设置COW Fork标记
      flags = (flags | PTE_F) & ~PTE_W;
      *pte = PA2PTE(pa) | flags;
    }

    if(mappages(new, i, PGSIZE, pa, flags) != 0) {
```

```
      uvmunmap(new, 0, i / PGSIZE, 1);
      return -1;
    }
    // 增加内存的引用计数
    kaddrefcnt((char*)pa);
  }
  return 0;
}
```

7. 修改`usertrap`，处理页面错误

```
else if((which_dev = devintr()) != 0){
  // ok
} else if(r_scause() == 13 || r_scause() == 15) {
  uint64 fault_va = r_stval();  // 获取出错的虚拟地址
  if(fault_va >= p->sz
    || cowpage(p->pagetable, fault_va) != 0
    || cowalloc(p->pagetable, PGROUNDDOWN(fault_va)) == 0)
    p->killed = 1;
} else {
  ...
}
```

1.3 实验中遇到的问题和解决办法

1. 引用计数结构体无法识别:

使用外部声明`extern struct ref_stru ref;`

2. 起初将 `COW` 标志与 `PTE_W`一同保留，导致无法触发写时异常， `COW` 无法生效:

在设置 `COW` 页时，只保留 `PTE_R`，去掉 `PTE_W`

3. 某次调试时，意外地把内核内存也标记为 `COW`，导致无法进入 `shell`:

加入地址检查逻辑

```
if ((uint64)pa < (uint64)end || pa >= PHYSTOP)
```

1.4 实验心得

理解了操作系统中虚拟内存管理和进程复制机制的底层实现，`Copy-on-Write` 技术通过延迟复制内存页，提高了 `fork()` 的性能，是现代操作系统中非常常见的优化策略。

调试过程让我对 C 语言中的内存模型和编译链接原理有了更深入体会。

Lab Multithreading

1. Uthread: switching between threads (moderate)

1.1 实验目的

提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划

1.2 实验步骤

1. 给进程加入上下文属性:

```
struct Context{
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct Context ctx;
};
```

2. 完成`thread_switch`函数

```
.text

/*
* save the old thread's registers,
* restore the new thread's registers.
*/

.globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    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)
    ret    /* return to ra */
```

3. 修改`thread_scheduler`，添加线程切换语句:

```
if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)&t->context, (uint64)&current_thread->context);
  } else
    next_thread = 0;
```

4. 在`thread_create`中对`thread`结构体做初始化

```
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.ra = (uint64)func;                    // 设定函数返回地址
  t->context.sp = (uint64)t->stack + STACK_SIZE;   // 设定栈指针
}
```

1.3 实验中遇到的问题和解决办法

未遇到特别问题

1.4 实验心得

本实验主要实现了用户级线程的切换机制，核心在于保存和恢复线程的寄存器上下文，通过 `thread_switch` 完成线程切换。在实现过程中，最大的收获是加深了对寄存器保存/恢复和线程栈初始化的理解，对线程调度、上下文切换有了更深刻的认识

2. Using threads (moderate)

2.1 实验目的

通过使用线程和锁机制，解决多线程访问哈希表的并发安全问题并提升程序的并行性能。

2.2 实验步骤

- 修改 `put()` 和 `get()` 函数，给操作上锁:

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;

  pthread_mutex_lock(&bkt_lock[i]);
  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&bkt_lock[i]);
```

```
}

static struct entry*
get(int key)
{
  int i = key % NBUCKET;

  pthread_mutex_lock(&bkt_lock[i]);
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key) break;
  }
  pthread_mutex_unlock(&bkt_lock[i]);
  return e;
}
```

2.3 实验中遇到的问题和解决办法

未遇到特别问题

2.4 实验心得

通过本次实验，我对多线程编程中的并发问题有了更直观的认识。最初由于多个线程同时访问和修改哈希表，导致键值丢失，体现了线程安全的重要性。通过在`put()`和`get()`中引入锁机制，保证了数据一致性，成功消除了丢失的键。

3. Barrier(moderate)

3.1 实验目的

实现基于条件变量的线程同步屏障，确保所有线程在同一轮次到达屏障后再共同继续执行。

3.2 实验步骤

- 完成`barrier()`函数:

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  pthread_mutex_lock(&bstate.barrier_mutex);
  bstate.nthread++;
  if(bstate.nthread < nthread){
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }else{
    bstate.nthread = 0;
    bstate.round++;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
```

```
}
```

3.3 实验中遇到的问题和解决办法

- 未考虑栈顶导致地址越界:

更正栈顶指针:

```
t->ctx.sp = (uint64)t->stack + STACK_SIZE ;  // 设定栈指针
```

3.4 实验心得

通过本次实验，我掌握了利用条件变量和互斥锁实现多线程同步屏障的方法，加深了对线程协作机制的理解。

Lab Locks

1. Memory allocator (moderate)

1.1 实验目的

实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取

1.2 实验步骤

1. 将`kmem`定义为一个数组，包含`NCPU`个元素，每个CPU对应一个

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];
```

2. 修改`kinit`，为所有锁初始化以`kmem`开头的名称

```
void
kinit()
{
  char lockname[8];
  for(int i = 0;i < NCPU; i++) {
    snprintf(lockname, sizeof(lockname), "kmem_%d", i);
    initlock(&kmem[i].lock, lockname);
  }
  freerange(end, (void*)PHYSTOP);
}
```

3. 改`kfree`，使用`cpuid()`和它返回的结果时必须关中断

```
void
kfree(void *pa)
{
  struct run *r;
```

```
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  push_off();  // 关中断
  int id = cpuid();
  acquire(&kmem[id].lock);
  r->next = kmem[id].freelist;
  kmem[id].freelist = r;
  release(&kmem[id].lock);
  pop_off();  //开中断
}
```

4. 修改`kalloc`，使得在当前CPU的空闲列表没有可分配内存时窃取其他内存的

```
void *
kalloc(void)
{
  struct run *r;

  push_off();// 关中断
  int id = cpuid();
  acquire(&kmem[id].lock);
  r = kmem[id].freelist;
  if(r)
    kmem[id].freelist = r->next;
  else {
    int antid;  // another id
    // 遍历所有CPU的空闲列表
    for(antid = 0; antid < NCPU; ++antid) {
      if(antid == id)
        continue;
      acquire(&kmem[antid].lock);
      r = kmem[antid].freelist;
      if(r) {
        kmem[antid].freelist = r->next;
        release(&kmem[antid].lock);
        break;
      }
      release(&kmem[antid].lock);
    }
  }
  release(&kmem[id].lock);
  pop_off();  //开中断

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

1.3 实验中遇到的问题和解决办法

1. 获取`cpuid()`前未关闭中断

添加`push_off()`和`pop_off()`，确保CPU一致性

2. 误将`kmem`作为指针处理，导致使用.而非->访问其成员

统一改为kmem[i].lock格式

1.4 实验心得

对xv6中内存分配器的设计有了更深入的理解。单一全局锁虽然简单，但不适合多核环境

按CPU划分空闲链表并配合窃取机制，能提升并发性能

2. Buffer cache (hard)

2.1 实验目的

修改`bget`和`brelse`，以便`bcache`中不同块的并发查找和释放不太可能在锁上发生冲突

2.2 实验步骤

1. 定义哈希桶结构，并在`bcache`中删除全局缓冲区链表，改为使用素数个散列桶

```
##define NBUCKET 13
##define HASH(id) (id % NBUCKET)

struct hashbuf {
  struct buf head;       // 头节点
  struct spinlock lock;  // 锁
};

struct {
  struct buf buf[NBUF];
  struct hashbuf buckets[NBUCKET];  // 散列桶
} bcache;
```

2. 在`binit()`中进行初始化

```
void
binit(void) {
  struct buf* b;
  char lockname[16];

  for(int i = 0; i < NBUCKET; ++i) {
    // 初始化散列桶的自旋锁
    snprintf(lockname, sizeof(lockname), "bcache_%d", i);
    initlock(&bcache.buckets[i].lock, lockname);

    // 初始化散列桶的头节点
    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }
```

```
  // Create linked list of buffers
  for(b = bcache.buf; b < bcache.buf + NBUF; b++) {
    // 利用头插法初始化缓冲区列表,全部放到散列桶0上
    b->next = bcache.buckets[0].head.next;
    b->prev = &bcache.buckets[0].head;
    initsleeplock(&b->lock, "buffer");
    bcache.buckets[0].head.next->prev = b;
    bcache.buckets[0].head.next = b;
  }
}
```

3. 更改`brelse`，不再获取全局锁

```
void
brelse(struct buf* b) {
  if(!holdingsleep(&b->lock))
    panic("brelse");

  int bid = HASH(b->blockno);

  releasesleep(&b->lock);

  acquire(&bcache.buckets[bid].lock);
  b->refcnt--;

  // 更新时间戳
  // 由于LRU改为使用时间戳判定，不再需要头插法
  acquire(&tickslock);
  b->timestamp = ticks;
  release(&tickslock);

  release(&bcache.buckets[bid].lock);
}
```

4. 更改`bget`，当没有找到指定的缓冲区时进行分配，分配方式是优先从当前列表遍历，找到一个没有引用且`timestamp`最小的缓冲区，如果没有就申请下一个桶的锁，并遍历该桶，找到后将该缓冲区从原来的桶移动到当前桶中

```
static struct buf*
bget(uint dev, uint blockno) {
  struct buf* b;

  int bid = HASH(blockno);
  acquire(&bcache.buckets[bid].lock);

  // Is the block already cached?
  for(b = bcache.buckets[bid].head.next; b != &bcache.buckets[bid].head; b = b-
>next) {
    if(b->dev == dev && b->blockno == blockno) {
      b->refcnt++;

      // 记录使用时间戳
      acquire(&tickslock);
```

```
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  b = 0;
  struct buf* tmp;

  // Recycle the least recently used (LRU) unused buffer.
  // 从当前散列桶开始查找
  for(int i = bid, cycle = 0; cycle != NBUCKET; i = (i + 1) % NBUCKET) {
    ++cycle;
    // 如果遍历到当前散列桶，则不重新获取锁
    if(i != bid) {
      if(!holding(&bcache.buckets[i].lock))
        acquire(&bcache.buckets[i].lock);
      else
        continue;
    }

    for(tmp = bcache.buckets[i].head.next; tmp != &bcache.buckets[i].head; tmp =
tmp->next)
      // 使用时间戳进行LRU算法，而不是根据结点在链表中的位置
      if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp))
        b = tmp;

    if(b) {
      // 如果是从其他散列桶窃取的，则将其以头插法插入到当前桶
      if(i != bid) {
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.buckets[i].lock);

        b->next = bcache.buckets[bid].head.next;
        b->prev = &bcache.buckets[bid].head;
        bcache.buckets[bid].head.next->prev = b;
        bcache.buckets[bid].head.next = b;
      }

      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;

      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
```

```
    } else {
      // 在当前散列桶中未找到，则直接释放锁
      if(i != bid)
        release(&bcache.buckets[i].lock);
    }
  }

  panic("bget: no buffers");
}
```

2.3 实验中遇到的问题和解决办法

1. `bget` 中重新分配持有会两个锁，如果桶a持有自己的锁，再申请桶b的锁，与此同时如果桶b持有自己的锁，再申请桶a的锁就会造成死锁:

使用了 `if(!holding(&bcache.bucket[i].lock))` 来进行检查。此外，代码优先从自己的桶中获取缓冲区

2. 在 `release` 桶的锁并重新 `acquire` 的这段时间，另一个CPU可能也以相同的参数调用了 `bget`，也发现没有该缓冲区并想要执行分配。导致 `usertests` 中的 `manywrites` 测试报错

先释放了散列桶的锁之后再重新获取，之所以这样做是为了让所有代码都保证申请锁的顺序

```
// 1. 第一次查找，没有找到 block 对应的 buffer
release(&bcache.buckets[bid].lock);
// 2. 获取全局锁（此时有分配全局 buffer 的唯一权）
acquire(&bcache.lock);
// 3. 再次获取桶锁
acquire(&bcache.buckets[bid].lock);
// 4. 再次查找桶中是否已经有人插入了该 block
for (b = bcache.buckets[bid].head; b; b = b->next) {
  if (b->dev == dev && b->blockno == blockno) {
    b->refcnt++;
    release(&bcache.lock);
    release(&bcache.buckets[bid].lock);
    return b;
  }
}
```

2.4 实验心得

`xv6` 的 `buffer cache` 是用一个全局的链表和锁管理所有缓冲区，这在单核环境下简单有效，但在多核并发环境下会成为瓶颈，也容易引发数据竞争。

理解了操作系统中缓存一致性和并发控制的重要性

Lab File System

1. Large files (moderate)

1.1 实验目的

增加xv6文件的最大大小

1.2 实验步骤

1. 在`fs.h`中添加宏定义

```
// 文件系统相关
##define NDIRECT 11
##define NINDIRECT (BSIZE / sizeof(uint))
##define NDINDIRECT ((BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint)))
##define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
##define NADDR_PER_BLOCK (BSIZE / sizeof(uint))  // 一个块中的地址数量
```

2. 修改`inode`结构体中`addrs`元素数量

```
// fs.h
struct dinode {
  ...
  uint addrs[NDIRECT + 2];   // Data block addresses
};

// file.h
struct inode {
  ...
  uint addrs[NDIRECT + 2];
};
```

3. 修改`bmap`支持二级索引

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp, *bp2;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
```

```
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  bn -= NINDIRECT;

  if(bn < NBI_INDIRECT){
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    uint idx_b1 = bn / NINDIRECT;
    if((addr = a[idx_b1]) == 0){ // 一个一级块负责 256 个二级块，这里检测对应一级块是否存
在
      a[idx_b1] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    bp2 = bread(ip->dev, addr); // bp2 为二级块的缓存
    a = (uint *)bp2->data;
    uint idx_b2 = bn % NINDIRECT;
    if((addr = a[idx_b2]) == 0){
      a[idx_b2] = addr = balloc(ip->dev);
      log_write(bp2);
    }
    brelse(bp2);
    return addr;
  }

  panic("bmap: out of range");
}
```

4. 修改`itrunc`释放所有块

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  if(ip->addrs[NDIRECT]){
```

```
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }

  if(ip->addrs[NDIRECT + 1]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for (i = 0; i < NINDIRECT; i++){
      if(a[i]){
        struct buf* bp2 = bread(ip->dev, a[i]);
        uint *a2 = bp2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(a2[j])
            bfree(ip->dev, a2[j]);
        }

        brelse(bp2);
        bfree(ip->dev, a[i]);
        // 和 a[i] 对应的是 bp2
        // a[i] 是块号，bp2 是实际的块缓存
      }
    }
    brelse(bp); // 释放缓存
    bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放磁盘块
    ip->addrs[NDIRECT + 1] = 0; // 不是 + 1
  }
  ip->size = 0;
  iupdate(ip);
}
```

1.3 实验中遇到的问题和解决办法

1. `uint *a2 = bp2->data;` 导致类型不匹配警告:

使用强制类型转换 `uint *a2 = (uint *)bp2->data;`，消除类型不匹配

2. 在计算一级和二级索引时，注意避免索引越界:

修改 `NDIRECT`、`NINDIRECT` 和 `NDINDIRECT` 宏定义，避免索引计算错误

1.4 实验心得

理解了多级间接索引在文件系统中扩展文件大小的实现机制。修改 `bmap` 函数支持二级索引，使得文件可以存储更多数据块，极大提升了文件系统的容量

2. Symbolic links (moderate)

2.1 实验目的

实现`symlink(char *target, char *path)`系统调用，该调用在引用由`target`命名的文件的路径处创建一个新的符号链接

2.2 实验步骤

1. 添加`O_NOFOLLOW`的标志位，打开软连接本身

```
##define O_RDONLY  0x000
##define O_WRONLY  0x001
##define O_RDWR    0x002
##define O_CREATE  0x200
##define O_TRUNC   0x400
##define O_NOFOLLOW 0x004
```

2. 注册系统调用`sys_symlink()`

```
uint64 sys_symlink(void) {
  char target[MAXPATH], path[MAXPATH];
  struct inode *ip;
  int n;

  if ((n = argstr(0, target, MAXPATH)) < 0
    || argstr(1, path, MAXPATH) < 0) {
    return -1;
  }

  begin_op();
  // create the symlink's inode
  if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
    end_op();
    return -1;
  }
  // write the target path to the inode
  if(writei(ip, 0, (uint64)target, 0, n) != n) {
    iunlockput(ip);
    end_op();
    return -1;
  }

  iunlockput(ip);
  end_op();
  return 0;
}
```

3. 修改`sys_open()`函数

```
...
if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
    if((ip = follow_symlink(ip)) == 0) {
      end_op();
      return -1;
    }
  }
...
```

2.3 实验中遇到的问题和解决办法

1. 软链接之间存在循环引用，导致系统崩溃

在 `follow_symlink()` 函数中加入最大递归深度限制

2.4 实验心得

通过本次实验，理解了文件系统中软链接的实现原理

Lab Mmap

mmap (hard)

1 实验目的

实现一个 `UNIX` 操作系统中常见系统调用 `mmap()` 和 `munmap()` 的子集。此系统调用会把文件映射到用户空间的内存，这样用户可以直接通过内存来修改和访问文件

2 实验步骤

1. 在 `Makefile` 中添加 `$U/_mmaptest`
2. 添加有关 `mmap` 和 `munmap` 系统调用的定义声明
3. 在 `kernel/proc.h` 中定义 `vm_area` 结构体

```
struct vm_area {
  int used;           // 是否已被使用
  uint64 addr;        // 起始地址
  int len;            // 长度
  int prot;           // 权限
  int flags;          // 标志位
  int vfd;            // 对应的文件描述符
  struct file* vfile; // 对应文件
  int offset;         // 文件偏移，本实验中一直为0
};
```

4. 在`allocproc`中初始化`vma`数组
5. 完成`mmap()`函数的实现

```
uint64
sys_mmap(void) {
  uint64 addr;
```

```
  int length;
  int prot;
  int flags;
  int vfd;
  struct file* vfile;
  int offset;
  uint64 err = 0xffffffffffffffff;

  // 获取系统调用参数
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
    argint(3, &flags) < 0 || argfd(4, &vfd, &vfile) < 0 || argint(5, &offset) <
0)
    return err;

  if(addr != 0 || offset != 0 || length < 0)
    return err;

  if(vfile->writable == 0 && (prot & PROT_WRITE) != 0 && flags == MAP_SHARED)
    return err;

  struct proc* p = myproc();
  // 没有足够的虚拟地址空间
  if(p->sz + length > MAXVA)
    return err;

  // 遍历查找未使用的VMA结构体
  for(int i = 0; i < NVMA; ++i) {
    if(p->vma[i].used == 0) {
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].len = length;
      p->vma[i].flags = flags;
      p->vma[i].prot = prot;
      p->vma[i].vfile = vfile;
      p->vma[i].vfd = vfd;
      p->vma[i].offset = offset;

      // 增加文件的引用计数
      filedup(vfile);

      p->sz += length;
      return p->vma[i].addr;
    }
  }

  return err;
}
```

6. 完成`munmap()`函数的实现

```
uint64
sys_munmap(void) {
  uint64 addr;
  int length;
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0)
```

```
    return -1;

  int i;
  struct proc* p = myproc();
  for(i = 0; i < NVMA; ++i) {
    if(p->vma[i].used && p->vma[i].len >= length) {
      // 根据提示，munmap的地址范围只能是
      // 1. 起始位置
      if(p->vma[i].addr == addr) {
        p->vma[i].addr += length;
        p->vma[i].len -= length;
        break;
      }
      // 2. 结束位置
      if(addr + length == p->vma[i].addr + p->vma[i].len) {
        p->vma[i].len -= length;
        break;
      }
    }
  }
  if(i == NVMA)
    return -1;

  // 将MAP_SHARED页面写回文件系统
  if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
    filewrite(p->vma[i].vfile, addr, length);
  }

  // 判断此页面是否存在映射
  uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

  // 当前VMA中全部映射都被取消
  if(p->vma[i].len == 0) {
    fileclose(p->vma[i].vfile);
    p->vma[i].used = 0;
  }

  return 0;
}
```

7. 修改`exit()`，将进程的已映射区域取消映射

```
...
for(int i = 0; i < NVMA; ++i) {
    if(p->vma[i].used) {
      if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
        filewrite(p->vma[i].vfile, p->vma[i].addr, p->vma[i].len);
      }
      fileclose(p->vma[i].vfile);
      uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].len / PGSIZE, 1);
      p->vma[i].used = 0;
    }
  }
...
```

8. 修改 `fork()`，复制父进程的 `VMA` 并增加文件引用计数

```
...
for(i = 0; i < NVMA; ++i) {
    if(p->vma[i].used) {
      memmove(&np->vma[i], &p->vma[i], sizeof(p->vma[i]));
      filedup(p->vma[i].vfile);
    }
  }
...
```

3 实验中遇到的问题和解决办法

1. 未在 `fork()` 函数中复制 `VMA`，会导致子进程没有对应的映射信息，从而访问非法地址

在 `fork()` 中遍历父进程的 `vma` 数组并使用 `memmove()` 拷贝至子进程，同时对映射文件调用 `filedup()` 增加引用计数

2. 在定义 `struct file` 时，`sleeplock` 被报为不完整类型

在 `file.h` 文件头部加入 `#include "sleeplock.h"`

3. 在解除映射时没有正确调用 `uvmunmap()`，导致错误

调用 `uvmunmap(p->pagetable, addr, length / PGSIZE, 1)` 正确解除映射

4 实验心得

通过本次实验，对 `mmap()` 与 `munmap()` 系统调用的底层实现有了深入理解。实验过程中，我体会到了操作系统如何将文件映射到用户空间，并通过页表实现按需访问

Lab Networking

Your Job (hard)

1 实验目的

在 `kernel/e1000.c` 中完成 `e1000_transmit()` 和 `e1000_recv()`，以便驱动程序可以发送和接收数据包

2 实验步骤

1. 完成 `e1000_transmit()` 函数，用来发送

```
int
e1000_transmit(struct mbuf *m)
{
  acquire(&e1000_lock);
  // 查询ring里下一个packet的下标
  int idx = regs[E1000_TDT];

  if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
    release(&e1000_lock);
    return -1;
```

```
  }

  // 释放上一个包的内存
  if (tx_mbufs[idx])
    mbuffree(tx_mbufs[idx]);

  tx_mbufs[idx] = m;
  tx_ring[idx].length = m->len;
  tx_ring[idx].addr = (uint64) m->head;
  tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
  regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;

  release(&e1000_lock);
  return 0;
}
```

2. 完成`e1000_recv()`函数，用来接收

```
static void
e1000_recv(void)
{
  while (1) {
    // 把所有到达的packet向上层递交
    int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    if ((rx_ring[idx].status & E1000_RXD_STAT_DD) == 0) {
      return;
    }
    rx_mbufs[idx]->len = rx_ring[idx].length;
    // 向上层network stack传输
    net_rx(rx_mbufs[idx]);

    rx_mbufs[idx] = mbufalloc(0);
    rx_ring[idx].status = 0;
    rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;
    regs[E1000_RDT] = idx;
  }
}
```

3. 在`MakeFile`中加入`nettests`

```
$U/_nettests\
```

3 实验中遇到的问题和解决办法

1. 程序能够发送数据包，但收不到响应，网络通信卡在等待阶段。

在 `e1000_recv` 中及时调用 `mbufalloc()` 分配新的缓冲区，重置描述符状态

设置发送命令时包含 E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP 标志，确保硬件完成后通知驱动

2. 描述符中 `addr` 字段为何用物理地址？为什么要强制转换指针

驱动需要把内存缓冲区的虚拟地址转换成物理地址，传递给网卡DMA控制器。因为网卡直接访问物理内存，不能识别虚拟地址

4 实验心得

本次实验深入理解了以太网驱动的发送与接收流程，特别是环形缓冲区的管理和硬件寄存器的操作细节