

实验报告1

Xv6 操作系统实验报告：Unix 工具实现

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 0 - Xv6 and Unix Utilities

提交日期：2025年9月1日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在通过在 MIT 开发的教学操作系统 **xv6** 上实现一系列经典的 Unix 用户程序，帮助我们熟悉 xv6 的系统调用接口、用户程序开发流程以及基本的进程控制、管道通信和文件系统操作。xv6 是一个为教学设计的简化版类 Unix 操作系统，运行于 RISC-V 架构之上。

本次实验共需完成五个用户程序的编写与调试：

1. `sleep`：让进程休眠指定时间（以“tick”为单位）。
2. `pingpong`：使用管道实现父子进程间的“乒乓”通信。
3. `primes`：使用管道和多进程实现并发素数筛选法（埃拉托斯特尼筛选法的并发版本）。
4. `find`：在目录树中查找指定名称的文件。
5. `xargs`：从标准输入读取行，并将每行作为参数执行指定命令。

这些程序虽然功能简单，但涵盖了操作系统中进程管理、进程间通信（IPC）、文件系统遍历等核心概念，是理解操作系统行为的良好实践。

二、实验环境与准备

实验环境基于 Git 管理的 xv6 源码，使用 `util` 分支进行开发。

环境搭建步骤

1. 克隆 xv6 源码仓库：

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

```
git checkout util
```

2. 编译并启动 xv6:

```
make qemu
```

此命令会调用 RISC-V 交叉编译器编译内核和用户程序，并使用 QEMU 模拟器启动系统。成功启动后进入 xv6 shell，提示符为 \$。

3. 开发流程：

1. 在 user/ 目录下创建或编辑对应的 .c 文件。
2. 将程序名添加到 Makefile 的 UPROGS 变量中，以便 make qemu 时自动编译。
3. 使用 make grade 自动测试所有程序，或使用 make GRADEFLAGS=program_name grade 测试单个程序。
4. 提交前使用 make handin 打包并上传代码。

三、实验过程与原理分析

1. sleep 程序（简单）

实验过程

1. 创建 user/sleep.c，实现主函数 main。
2. 使用 argc 和 argv 获取命令行参数，若参数缺失则打印错误信息并退出。
3. 使用 atoi() 将字符串参数转换为整数。
4. 调用系统调用 sleep(n) 让当前进程休眠 n 个 tick。
5. 最后调用 exit() 结束进程。

原理分析

sleep 系统调用由内核提供（定义在 kernel/sysproc.c 中的 sys_sleep），其作用是将当前进程挂起，直到经过指定数量的时钟中断（tick）。用户程序通过 user/usys.S 中的汇编代码进入内核态，执行系统调用。进程休眠期间，CPU 资源被释放，调度器可调度其他进程运行，体现了操作系统的并发性和资源管理。

关键代码

```
// user/sleep.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(2, "Usage: sleep <ticks>\n");
        exit(1);
    }
    int ticks = atoi(argv[1]);
    sleep(ticks);
    exit(0);
}
```

2. pingpong 程序 (简单)

实验过程

1. 创建 `user/pingpong.c`。
2. 创建一个管道 `pipe(p)`，得到读写文件描述符 `p[0]` 和 `p[1]`。
3. 使用 `fork()` 创建子进程。
4. 父进程向管道写入一个字节（如 `1`），然后从管道读取子进程的响应。
5. 子进程从管道读取父进程发送的字节，打印“received ping”，再将该字节写回管道，打印“received pong”后退出。
|
6. 父进程读取响应后打印“received pong”并退出。

原理分析

管道 (Pipe) 是一种半双工的进程间通信机制，数据只能单向流动。本实验使用两个管道方向（通过两次 `pipe` 调用或一个双向管道模拟）实现双向通信。`fork()` 创建子进程，父子进程拥有独立的地址空间，但共享文件描述符表。`getpid()` 获取当前进程 ID，用于输出标识。
`read()` 和 `write()` 是系统调用，用于从管道读写数据。

关键代码

```
// user/pingpong.c
#include "kernel/types.h"
#include "kernel/stat.h"
```

```

#include "user/user.h"

int main() {
    int p1[2], p2[2];
    pipe(p1); pipe(p2);

    if (fork() == 0) {
        // 子进程
        close(p1[1]); close(p2[0]);
        char c;
        if (read(p1[0], &c, 1) != 1) {
            fprintf(2, "child: read failed\n");
            exit(1);
        }
        printf("%d: received ping\n", getpid());
        if (write(p2[1], &c, 1) != 1) {
            fprintf(2, "child: write failed\n");
            exit(1);
        }
        close(p1[0]); close(p2[1]);
        exit(0);
    } else {
        // 父进程
        close(p1[0]); close(p2[1]);
        char c = 'X';
        if (write(p1[1], &c, 1) != 1) {
            fprintf(2, "parent: write failed\n");
            exit(1);
        }
        if (read(p2[0], &c, 1) != 1) {
            fprintf(2, "parent: read failed\n");
            exit(1);
        }
        printf("%d: received pong\n", getpid());
        close(p1[1]); close(p2[0]);
        wait(0);
        exit(0);
    }
}

```

3. primes 程序 (中等/较难)

实验过程

1. 创建 user/primes.c。

2. 使用递归思想构建管道链：第一个进程生成 2~35 的整数，通过管道传给第一个素数处理进程。
3. 每个素数进程读取第一个数（必为素数），打印它，然后将后续所有不能被该素数整除的数传给下一个进程。
4. 新进程通过 `fork()` 动态创建，形成管道链。
5. 当输入结束（`read` 返回 0）时，进程退出。主进程等待所有子进程结束。

原理分析

该程序模拟了 **Doug McIlroy** 提出的“并发素数筛”思想：每个素数进程负责过滤其倍数，形成流水线。管道作为进程间的数据流通道，实现了“生产者-消费者”模型。必须及时关闭不需要的文件描述符，否则会耗尽系统资源（xv6 文件描述符有限）。`read` 返回 0 表示写端已关闭，可用于判断数据流结束。

关键代码

```
// user/primes.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void sieve(int left_fd) {
    int prime, n;
    if (read(left_fd, &prime, sizeof(prime)) != sizeof(prime)) {
        close(left_fd);
        exit(0);
    }
    printf("prime %d\n", prime);

    int p[2];
    if (pipe(p) < 0) {
        fprintf(2, "pipe failed\n");
        exit(1);
    }

    if (fork() == 0) {
        // 子进程：继续筛
        close(left_fd);
        close(p[1]);
        sieve(p[0]);
    } else {
        // 父进程：过滤非倍数
        close(p[0]);
        while (read(left_fd, &n, sizeof(n)) == sizeof(n)) {
```

```

        if (n % prime != 0) {
            write(p[1], &n, sizeof(n));
        }
    }
    close(left_fd);
    close(p[1]);
    wait(0);
    exit(0);
}
}

int main() {
    int p[2];
    pipe(p);

    if (fork() == 0) {
        // 子进程：启动筛法
        close(p[1]);
        sieve(p[0]);
    } else {
        // 父进程：生成 2~35
        close(p[0]);
        for (int i = 2; i <= 35; i++) {
            write(p[1], &i, sizeof(i));
        }
        close(p[1]);
        wait(0);
        exit(0);
    }
}

```

4. find 程序（中等）

实验过程

1. 创建 `user/find.c`。
2. 递归遍历目录：打开目录，读取每个条目（`struct dirent`）。
3. 跳过 `.` 和 `..`。
4. 若条目为文件且名称匹配，则输出完整路径。
5. 若条目为目录，则递归进入该目录查找。

原理分析

使用 `open()`、`read()` 和 `close()` 操作目录文件。 `struct stat` 提供文件元信息，`st_type` 区分文件类型 (`T_FILE` 或 `T_DIR`)。路径拼接使用字符串操作 (`sprintf`)。递归是处理树形结构（目录树）的自然方法。

关键代码

```
// user/find.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"

void find(char *path, char *target) {
    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 stat %s\n", path);
        close(fd);
        return;
    }

    if (st.type != T_DIR) {
        fprintf(2, "%s is not a directory\n", path);
        close(fd);
        return;
    }

    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
        fprintf(2, "find: path too long\n");
        close(fd);
        return;
    }
    strcpy(buf, path);
    p = buf + strlen(buf);
    *p++ = '/';

    while (read(fd, &de, sizeof(de)) == sizeof(de)) {
        if (de.inum == 0)
            continue;
```

```

if (!strcmp(de.name, ".") || !strcmp(de.name, ".."))
    continue;
memmove(p, de.name, DIRSIZ);
p[DIRSIZ] = 0;
if (stat(buf, &st) < 0) {
    printf("find: cannot stat %s\n", buf);
    continue;
}
if (st.type == T_FILE && !strcmp(de.name, target)) {
    printf("%s\n", buf);
} else if (st.type == T_DIR) {
    find(buf, target);
}
}
close(fd);
}

int main(int argc, char *argv[]) {
if (argc != 3) {
    fprintf(2, "Usage: find <dir> <name>\n");
    exit(1);
}
find(argv[1], argv[2]);
exit(0);
}

```

5. xargs 程序 (中等)

实验过程

1. 创建 `user/xargs.c`。
2. 从标准输入逐字符读取，遇到换行符时结束当前行。
3. 解析命令行参数，构造 `argv` 数组。
4. 使用 `fork()` 和 `exec()` 执行命令。
5. 父进程使用 `wait()` 等待子进程结束。

原理分析

`xargs` 是 Unix 中强大的工具，用于将输入流转换为命令行参数。`exec()` 系列函数用新程序替换当前进程的内存空间。参数解析需处理空格分隔的多个参数。使用 `wait()` 回收子进程，避免僵尸进程。

关键代码

```
// user/xargs.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(2, "Usage: xargs <command> [args...]\n");
        exit(1);
    }

    char line[1024];
    int i, j;
    for (i = 1; i < argc; i++) {
        line[i-1] = ' ';
        strcpy(line + i, argv[i]);
    }
    line[i] = '\0';

    char c;
    int pos = 0;
    char input[1024];

    while (read(0, &c, 1) == 1) {
        if (c == '\n') {
            input[pos] = '\0';
            pos = 0;

            if (fork() == 0) {
                // 子进程执行命令
                char *cmd_args[MAXARG];
                int argidx = 0;
                cmd_args[argidx++] = argv[1];
                for (j = 2; j < argc; j++) {
                    cmd_args[argidx++] = argv[j];
                }
                cmd_args[argidx++] = input;
                cmd_args[argidx] = 0;
                exec(argv[1], cmd_args);
                fprintf(2, "xargs: exec failed\n");
                exit(1);
            } else {
                // 父进程等待
                wait(0);
            }
        }
    }
}
```

```
    }
} else {
    input[pos++] = c;
}
}
exit(0);
}
```

四、实验结果

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.8s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.4s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.8s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.0s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.4s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (0.6s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.5s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (0.8s)
== Test time ==
time: OK
Score: 100/100
```

五、思考与总结

本次实验让我深刻理解了操作系统核心机制的实践应用。通过实现 `sleep`、`pingpong`、`primes`、`find` 和 `xargs` 这些经典 Unix 工具，我不仅熟悉了 xv6 的系统调用接口和用户程序开发流程，更直观体会到了进程控制、进程间通信（管道）、文件系统遍历等关键概念。`fork` 和 `pipe` 的组合展示了构建并发程序的强大能力，而 `exec` 和 `wait` 则体现了进程生命周期的管理。实验强调了资源管理的重要性——如及时关闭文件描述符，避免系统资源耗尽。同时，`find` 的递归设计和 `find` 与 `xargs` 的管道组合，生动诠释了 Unix 哲学中“小而专”的工具通过简单接口组合成强大功能的设计思想。这次实践极大地加深了我对操作系统底层工作原理的理解。

实验报告2

Xv6 操作系统实验报告：系统调用实现

课程名称：6.828 操作系统工程（MIT 6.S081）

实验名称：Lab 2 - System Calls

提交日期：2025年9月1日

作者：王奕博 学号：2353945

一、实验概述

本实验旨在通过在 MIT 开发的教学操作系统 xv6 中实现两个新的系统调用：`trace` 和 `sysinfo`，深入理解操作系统中用户态与内核态的切换机制、系统调用的注册与分发流程，以及内核对进程和内存资源的管理方式。xv6 是一个为教学设计的简化版类 Unix 操作系统，运行于 RISC-V 架构之上。

本次实验共需完成两个系统调用的编写与调试：

- **trace**：实现系统调用追踪功能，允许用户程序指定一个“掩码”（mask），用于控制对特定系统调用的追踪。当被追踪的系统调用即将返回时，内核应打印出当前进程 ID、系统调用名称和返回值。
- **sysinfo**：实现系统信息查询功能，该系统调用接收一个指向 `struct sysinfo` 的指针作为参数，内核需填充该结构体，其中 `freemem` 字段表示空闲内存字节数，`nproc` 字段表示当前非 UNUSED 状态的进程数量。

这两个系统调用虽然功能独立，但共同揭示了操作系统内核如何响应用户请求、管理内部状态并安全地与用户空间交互的核心机制，是理解操作系统内核工作原理的关键实践。

二、实验环境与准备

实验环境基于 Git 管理的 xv6 源码，使用 `syscall` 分支进行开发。

环境搭建步骤

- 克隆 xv6 源码仓库：

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

- 编译并启动 xv6：

```
make clean  
make qemu
```

此命令会调用 RISC-V 交叉编译器编译内核和用户程序，并使用 QEMU 模拟器启动系统。成功启动后进入 xv6 shell，提示符为 \$。

开发流程：

- 在 user/ 目录下创建或编辑对应的 .c 文件（如 trace.c 和 sysinfotest.c 已提供）。
- 将程序名添加到 Makefile 的 UPROGS 变量中，以便 make qemu 时自动编译。
- 使用 make grade 自动测试所有程序，或使用 make GRADEFLAGS=program_name grade 测试单个程序。
- 提交前使用 make handin 打包并上传代码。

三、实验过程与原理分析

1. trace 系统调用（中等）

实验过程

1. 在 user/user.h 中添加 trace 系统调用的函数原型：

```
// user/user.h  
int trace(int);
```

2. 在 user/usys.pl 中添加 trace 系统调用的入口，Perl 脚本会自动生成对应的汇编 stub：

```
# user/usys.pl  
entry("trace");
```

生成的汇编代码（user/usys.S）如下，其核心是将系统调用号存入 a7 寄存器并执行 ecall 指令：

```
.global trace  
trace:  
    li a7, SYS_trace
```

```
    ecall  
    ret
```

3. 在 `kernel/syscall.h` 中定义 `trace` 系统调用的编号:

```
#define SYS_trace 22
```

4. 扩展进程结构体 `struct proc`，在 `kernel/proc.h` 中添加 `tracemask` 字段用于存储追踪掩码:

```
// Per-process state  
struct proc {  
    struct spinlock lock;  
    // p->lock must be held when using these:  
    enum procstate state;          // Process state  
    struct proc *parent;           // Parent process  
    void *chan;                   // If non-zero, sleeping on chan  
    int killed;                   // If non-zero, have been killed  
    int xstate;                   // Exit status to be returned to parent's  
    wait  
    int pid;                      // Process ID  
    // these are private to the process, so p->lock need not be held.  
    uint64 kstack;                // Virtual address of kernel stack  
    uint64 sz;                     // Size of process memory (bytes)  
    pagetable_t pagetable;         // User page table  
    struct trapframe *trapframe;   // data page for trampoline.S  
    struct context context;        // swtch() here to run process  
    struct file *ofile[NFILE];    // Open files  
    struct inode *cwd;             // Current directory  
    char name[16];                // Process name (debugging)  
    uint64 tracemask;             // the sys calls this proc is tracing  
};
```

5. 修改 `kernel/proc.c` 中的 `allocproc()` 和 `fork()` 函数:

- `allocproc()` 中初始化新进程的 `tracemask` 为 0:

```
static struct proc*  
allocproc(void)  
{  
    struct proc *p;  
    char *sp;
```

```

    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == UNUSED) {
            goto found;
        } else {
            release(&p->lock);
        }
    }
    return 0;

found:
    p->pid = allocpid();
    p->state = USED;

    // Allocate a trapframe page.
    if((p->trapframe = (struct trapframe *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // An empty user page table.
    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // Set up new context to start executing at forkret.
    memset(&p->context, 0, sizeof(p->context));
    p->context.ra = (uint64)forkret;
    p->context.sp = p->kstack + PGSIZE;

    p->tracemask = 0; // 初始化 trace 掩码为 0

    return p;
}

```

- `fork()` 中将父进程的 `tracemask` 复制给子进程：

```

int
fork(void)
{
    int i, pid;
    struct proc *np;

```

```

struct proc *p = myproc();

// Allocate process.
if((np = allocproc()) == 0){
    return -1;
}

// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
}
np->sz = p->sz;

// copy saved user registers.
*(np->trapframe) = *(p->trapframe);

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;

// increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
        np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);

safestrcpy(np->name, p->name, sizeof(p->name));

pid = np->pid;

release(&np->lock);

acquire(&wait_lock);
np->parent = p;
release(&wait_lock);

acquire(&np->lock);
np->state = RUNNABLE;
np->tracemask = p->tracemask; // 继承父进程的 trace 掩码

release(&np->lock);

return pid;
}

```

6. 在 `kernel/sysproc.c` 中实现 `sys_trace()` 内核函数，使用 `argint()` 获取用户参数并设置 `tracemask`：

```
uint64
sys_trace(void) {
    int trace_sys_mask;
    if (argint(0, &trace_sys_mask) < 0)
        return -1;
    myproc()->tracemask = trace_sys_mask;
    return 0;
}
```

7. 在 `kernel/syscall.c` 中注册 `sys_trace` 函数指针和名称，并修改 `syscall()` 函数以支持追踪打印：

```
extern uint64 sys_trace(void);
static uint64 (*syscalls[])(void) = {
    [SYS_fork]    sys_fork,
    [SYS_exit]    sys_exit,
    [SYS_wait]    sys_wait,
    [SYS_pipe]    sys_pipe,
    [SYS_read]    sys_read,
    [SYS_kill]    sys_kill,
    [SYS_exec]    sys_exec,
    [SYS_fstat]   sys_fstat,
    [SYS_chdir]   sys_chdir,
    [SYS_dup]     sys_dup,
    [SYS_getpid]  sys_getpid,
    [SYS_sbrk]    sys_sbrk,
    [SYS_sleep]   sys_sleep,
    [SYS_uptime]  sys_uptime,
    [SYS_open]    sys_open,
    [SYS_write]   sys_write,
    [SYS_mknod]   sys_mknod,
    [SYS_unlink]  sys_unlink,
    [SYS_link]    sys_link,
    [SYS_mkdir]   sys_mkdir,
    [SYS_close]   sys_close,
    [SYS_trace]   sys_trace, // 添加 trace 系统调用
};

static char *sysnames[] = {
    "",
    "fork",
    "exit",
```

```

    "wait",
    "pipe",
    "read",
    "kill",
    "exec",
    "fstat",
    "chdir",
    "dup",
    "getpid",
    "sbrk",
    "sleep",
    "uptime",
    "open",
    "write",
    "mknod",
    "unlink",
    "link",
    "mkdir",
    "close",
    "trace" // 添加 trace 名称
};

void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num]();
        if (p->tracemask & (1 << num)) { // 检查是否需要追踪
            printf("%d: syscall %s -> %d\n", p->pid, sysnames[num], p->trapframe-
>a0);
        }
    } else {
        printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

- 将 `_trace` 添加到 Makefile 的 UPROGS 变量中，确保 `trace` 程序被编译。

原理分析

系统调用是用户程序请求内核服务的唯一途径。当用户调用 `trace(n)` 时，生成的汇编代码会将 `SYS_trace` 的编号 (22) 存入 `a7` 寄存器，并执行 `ecall` 指令触发异常，使 CPU 从用户态切换到内核态。内核的通用系统调用入口 `syscall()` 从 `a7` 读取编号，查表调用

`sys_trace()`。`sys_trace()` 使用 `argint()` 从用户栈获取参数 `n`，并将其按位或到当前进程的 `tracemask` 字段中。后续每次系统调用返回前，`syscall()` 函数都会检查 `tracemask` 是否包含当前调用的编号，若包含则调用 `printf` 输出追踪信息。`fork()` 继承 `tracemask` 确保了子进程也能被追踪，体现了进程状态的继承性。

2. sysinfo 系统调用（中等）

实验过程

- 在 `user/user.h` 中添加 `struct sysinfo` 的前向声明和 `sysinfo()` 函数原型：

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

- 在 `user/usys.pl` 中添加 `sysinfo` 系统调用的入口：

```
entry("sysinfo");
```

- 在 `kernel/syscall.h` 中定义 `sysinfo` 系统调用的编号：

```
#define SYS_sysinfo 23
```

- 在 `kernel/kalloc.c` 中实现 `kfreemem()` 函数，用于遍历空闲内存链表并统计总空闲字节数：

```
// Return the number of bytes of free memory
// should be multiple of PGSIZE
uint64
kfreemem(void) {
    struct run *r;
    uint64 free = 0;
    acquire(&kmem.lock); // 上锁，防止数据竞态
    r = kmem.freelist;
    while (r) {
        free += PGSIZE; // 每一页固定4096字节
        r = r->next; // 遍历单链表
    }
    release(&kmem.lock);
```

```
    return free;
}
```

5. 在 `kernel/proc.c` 中实现 `count_free_proc()` 函数，用于遍历进程表并统计非 `UNUSED` 状态的进程数：

```
// Count how many processes are not in the state of UNUSED
uint64
count_free_proc(void) {
    struct proc *p;
    uint64 count = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
        // 此处不一定需要加锁，因为该函数是只读不写
        // 但proc.c里其他类似的遍历时都加了锁，那我们也加上
        acquire(&p->lock);
        if(p->state != UNUSED) {
            count += 1;
        }
        release(&p->lock);
    }
    return count;
}
```

6. 在 `kernel/sysproc.c` 中实现 `sys_sysinfo()` 内核函数，整合数据并通过 `copyout()` 写回用户空间：

```
// collect system info
uint64
sys_sysinfo(void) {
    struct proc *my_proc = myproc();
    uint64 p;
    if(argaddr(0, &p) < 0) // 获取用户提供的buffer地址
        return -1;
    // construct in kernel first 在内核态先构造出这个sysinfo struct
    struct sysinfo s;
    s.freemem = kfreemem();
    s.nproc = count_free_proc();
    // copy to user space // 把这个struct复制到用户态地址里去
    if(copyout(my_proc->pagetable, p, (char *)&s, sizeof(s)) < 0)
        return -1;
    return 0;
}
```

7. 在 `kernel/syscall.c` 中注册 `sys_sysinfo` 的函数指针和名称：

```
extern uint64 sys_sysinfo(void);
static uint64 (*syscalls[])(void) = {
    [SYS_fork]    sys_fork,
    [SYS_exit]    sys_exit,
    [SYS_wait]    sys_wait,
    [SYS_pipe]    sys_pipe,
    [SYS_read]    sys_read,
    [SYS_kill]    sys_kill,
    [SYS_exec]    sys_exec,
    [SYS_fstat]   sys_fstat,
    [SYS_chdir]   sys_chdir,
    [SYS_dup]     sys_dup,
    [SYS_getpid]  sys_getpid,
    [SYS_sbrk]    sys_sbrk,
    [SYS_sleep]   sys_sleep,
    [SYS_uptime]  sys_uptime,
    [SYS_open]    sys_open,
    [SYS_write]   sys_write,
    [SYS_mknod]   sys_mknod,
    [SYS_unlink]  sys_unlink,
    [SYS_link]    sys_link,
    [SYS_mkdir]   sys_mkdir,
    [SYS_close]   sys_close,
    [SYS_trace]   sys_trace,
    [SYS_sysinfo] sys_sysinfo, // 添加 sysinfo 系统调用
};

static char *sysnames[] = {
    "",  

    "fork",  

    "exit",  

    "wait",  

    "pipe",  

    "read",  

    "kill",  

    "exec",  

    "fstat",  

    "chdir",  

    "dup",  

    "getpid",  

    "sbrk",  

    "sleep",  

    "uptime",  

    "open",  

    "write",  

    "mknod",
```

```
        "unlink",
        "link",
        "mkdir",
        "close",
        "trace",
        "sysinfo" // 添加 sysinfo 名称
    };
}
```

8. 将 `_sysinfotest` 添加到 Makefile 的 UPROGS 变量中，确保测试程序被编译。

原理分析

`sysinfo` 系统调用展示了内核如何安全地向用户空间暴露内部状态。用户程序传入一个指向 `struct sysinfo` 的指针，该指针指向用户空间的内存。内核不能直接写入用户空间地址，必须通过 `copyout()` 函数进行安全的跨地址空间数据拷贝。`kfreemem()` 通过遍历 `kmem.freelist` 链表，统计所有空闲物理页的总大小（每页 `PGSIZE=4096` 字节）。`count_free_proc()` 遍历全局 `proc[]` 数组，统计状态不为 `UNUSED` 的进程数量。这些信息被填充到内核栈上的 `struct sysinfo` 实例中，再通过 `copyout()` 复制到用户提供的地址。整个过程体现了内核对资源的集中管理和对用户空间的隔离保护。

四、实验结果

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.2s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.5s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (7.9s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (0.9s)
== Test time ==
time: OK
Score: 35/35
```

五、思考与总结

本次实验让我深入理解了操作系统内核中系统调用的实现机制与核心资源管理。通过实现 `trace` 和 `sysinfo` 两个系统调用，我不仅掌握了从用户接口到内核实现的完整链条，更直观体会到了用户态与内核态的切换、系统调用的分发、进程状态继承以及跨地址空间数据安全传输等关键概念。`trace` 系统调用的实现揭示了 `ecall` 指令、`a7` 寄存器、系统调用表和 `syscall()` 通用入口协同工作的细节，而 `fork()` 继承 `tracemask` 则体现了进程状态的传递性。`sysinfo` 系统调用则突显了内核对内存和进程资源的集中管理，并通过 `copyout()` 函数强调了用户空间与内核空间的隔离原则。整个实验过程让我认识到，系统调用不仅是功能接口，更是操作系统安全与稳定的核心防线。这次实践极大地加深了我对操作系统内核内部工作原理的理解，为后续实验打下了坚实基础。

实验报告3

Xv6 操作系统实验报告：页表（Page Tables）

课程名称：6.828 操作系统工程（MIT 6.S081）

实验名称：Lab 3 - Page Tables

提交日期：2025年9月2日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在深入理解 RISC-V 架构下的分页机制和虚拟内存管理，通过对 xv6 操作系统的页表（Page Table）进行修改，实现更高效、安全的用户空间与内核空间数据交互。xv6 使用三级页表结构管理虚拟内存，内核与用户进程共享同一内核页表，这导致内核在访问用户指针时必须通过软件遍历页表进行地址转换。

本次实验包含三个核心任务：

1. **打印页表**（简单）：实现 `vmprint` 函数，递归遍历并打印指定页表的所有有效页表项（PTE），用于调试和理解页表结构。
2. **每个进程一个内核页表**（较难）：为每个进程创建独立的内核页表副本，使内核在执行该进程时可直接使用其页表，为后续优化奠定基础。
3. **简化 `copyin` / `copyinstr`**（较难）：在每个进程的内核页表中添加对其用户地址空间的映射，从而允许内核函数 `copyin` 和 `copyinstr` 直接解引用用户指针，无需软件地址转换，大幅提升效率。

通过完成这些任务，我们将掌握页表的构建、遍历、切换以及虚拟内存映射的高级技巧，深刻理解操作系统内核如何管理内存空间。

二、实验环境与准备

实验环境基于 Git 管理的 xv6 源码，使用 `pgtbl` 分支进行开发。

环境搭建步骤

1. 克隆 xv6 源码仓库并切换到实验分支：

```
git clone git://g.csail.mit.edu/xv6-labs-2020
cd xv6-labs-2020
git fetch
```

```
git checkout pgtbl  
make clean
```

2. 编译并启动 xv6:

```
make qemu
```

此命令会编译内核和用户程序，并使用 QEMU 模拟器启动系统。成功启动后进入 xv6 shell。

3. 开发流程：

- 在 `kernel/` 目录下修改 `vm.c`、`proc.c`、`vmcopyin.c` 等文件。
- 将新函数的声明添加到 `kernel/defs.h`。
- 使用 `make grade` 运行所有测试，或使用 `make grade TESTNAME` 测试特定功能。
- 提交前使用 `make handin` 打包并上传代码。

三、实验过程与原理分析

1. 打印页表（简单）

实验过程

本任务要求实现一个函数 `vmprint`，用于递归打印给定页表的所有有效页表项（PTE），以可视化页表的三层结构。根据实验提示，可以参考 `freewalk` 函数的遍历逻辑。

首先，在 `kernel/defs.h` 中声明函数原型。然后在 `kernel/vm.c` 中实现 `vmprint` 及其递归辅助函数 `vmprint_helper`。`vmprint_helper` 接收页表和当前深度作为参数，遍历页表的 512 个 PTE。对于每个有效的 PTE（`PTE_V` 标志位被设置），打印其索引、PTE 值和对应的物理地址，并根据深度添加缩进（`...`）。关键判断是 `(pte & (PTE_R|PTE_W|PTE_X)) == 0`，这表示该 PTE 不是叶节点，而是指向一个下一级页表，此时递归调用 `vmprint_helper` 处理下一级。

最后，在 `exec.c` 的 `exec` 函数成功加载程序后，为第一个进程（`pid == 1`）调用 `vmprint(p->pagetable)`，以便在启动时输出其页表结构。

修改的代码

```
// kernel/defs.h  
void vmprint(pagetable_t pagetable);  
  
// kernel/vm.c  
void vmprint_helper(pagetable_t pagetable, int depth) {
```

```

static char* indent[] = {
    "",
    "..",
    "...",
    "...."
};

if (depth <= 0 || depth >= 4) {
    panic("vmprint_helper: depth not in {1, 2, 3}");
}

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

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

// exec.c
// 在 exec 函数中, 成功加载程序后
proc_freepagetable(oldpagetable, oldsiz);
// 复制新的kernel page并刷新TLB
if (pagecopy(p->pagetable, p->kpagetable, 0, p->sz) != 0) {
    goto bad;
}
w_satp(MAKE_SATP(p->kpagetable));
sfence_vma();

if (p->pid == 1) { // 添加: 为第一个进程打印页表
    vmprint(p->pagetable);
}
return argc;

```

2. 每个进程一个内核页表 (较难)

实验过程

本任务的核心是为每个进程创建一个专属的内核页表，以实现更好的隔离性，并为后续优化 `copyin` 函数做准备。

- 扩展进程结构**: 在 `struct proc` 中添加 `kpagetable` 字段。
- 创建专属内核页表**: 实现 `ukvminit` 函数, 其逻辑与 `kvminit` 类似, 但返回一个新的页表。该函数分配一个物理页作为页表根节点, 并调用 `ukvmmmap` 将所有内核代码、数据、设备 (如 UART、PLIC) 和 trampoline 映射到这个新页表中。
- 分配进程时初始化**: 在 `allocproc` 中, 为新进程调用 `ukvminit` 创建 `kpagetable`。然后, 使用 `kvmpa` 获取该进程内核栈的物理地址, 并通过 `ukvmmmap` 将其虚拟地址映射到 `kpagetable` 中。
- 进程切换时加载页表**: 在 `scheduler` 中, 当切换到一个可运行进程时, 使用 `w_satp(MAKE_SATP(p->kpagetable))` 将 `satp` 寄存器设置为该进程的 `kpagetable`, 并调用 `sfence_vma()` 刷新 TLB。当进程执行完毕切换回来后, 必须立即调用 `kvminit hart()` 恢复全局内核页表, 否则后续代码将无法正确寻址。
- 清理资源**: 在 `freeproc` 中, 当进程销毁时, 需要释放其 `kpagetable`。我们实现了 `ufreewalk` 函数 (类似 `freewalk`) 来递归释放页表占用的物理页。注意, `kpagetable` 映射的物理内存 (如内核代码、设备) 是全局共享的, 因此 `ufreewalk` 只释放页表自身占用的物理页, 不释放这些共享的物理内存。

代码

```
// kernel/proc.h
struct proc {
    // ... 其他字段 ...
    pagetable_t pagetable;           // User page table
    pagetable_t kpagetable;          // the kernel table per process 专属内核页
    // ... 其他字段 ...
};

// kernel/vm.c
void ukvmmmap(pagetable_t kpagetable, uint64 va, uint64 pa, uint64 sz, int perm)
{
    if(mappages(kpagetable, va, sz, pa, perm) != 0)
        panic("ukvmmmap");
}

pagetable_t ukvminit() {
    pagetable_t kpagetable = (pagetable_t) kalloc();
    if (kpagetable == 0) {
        return kpagetable;
    }
    memset(kpagetable, 0, PGSIZE);
    ukvmmmap(kpagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
    ukvmmmap(kpagetable, VIRTI00, VIRTI00, PGSIZE, PTE_R | PTE_W);
    ukvmmmap(kpagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
}
```

```

    ukvmmmap(kpagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
    ukvmmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);
    ukvmmmap(kpagetable, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R
| PTE_W);
    ukvmmmap(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
    return kpagetable;
}

// kernel/proc.c
static struct proc* allocproc(void) {
    // ... 分配 trapframe 和 pagetable ...
    p->kpagetable = ukvminit();
    if(p->kpagetable == 0) {
        freeproc(p);
        release(&p->lock);
        return 0;
    }
    uint64 va = KSTACK((int) (p - proc));
    pte_t pa = kvmpa(va);
    memset((void *)pa, 0, PGSIZE);
    ukvmmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
    p->kstack = va;
    // ... 其他初始化 ...
}

void scheduler(void) {
    // ... 原有代码 ...
    if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        w_satp(MAKE_SATP(p->kpagetable)); // 切换到新进程的内核页表
        sfence_vma();
        swtch(&c->context, &p->context);
        // 进程执行完毕，切换回全局内核页表
        kvminithart();
        c->proc = 0;
        found = 1;
    }
    // ... 原有代码 ...
}

// kernel/vm.c
void ufreewalk(pagetable_t pagetable) {
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            uint64 child = PTE2PA(pte);

```

```

        ufreewalk((pagetable_t)child);
    }
    pagetable[i] = 0;
}
kfree((void*)pagetable);
}

// kernel/proc.c
static void freeproc(struct proc *p) {
    // ... 释放 trapframe 和 pagetable ...
    if (p->kpagetable) {
        ufreewalk(p->kpagetable); // 释放专属内核页表占用的内存
        p->kpagetable = 0;
    }
}

```

3. 简化 copyin / copyinstr (较难)

实验过程

本任务的目标是利用上一任务创建的 `kpagetable`，将进程的用户地址空间也映射进去。这样，当进程在内核态运行时，其 `kpagetable` 同时包含了用户和内核的映射，内核函数可以直接解引用用户指针，无需 `walk` 函数进行软件地址转换。

- 实现同步映射：**编写 `pagecopy` 函数，用于将一段用户虚拟地址的映射从 `pagetable` 复制到 `kpagetable`。该函数遍历指定范围内的虚拟地址，通过 `walk` 在 `pagetable` 中查找有效的 PTE，获取其物理地址和权限（移除 `PTE_U` 标志，因为内核页表项通常不需要用户标志），然后使用 `umappages`（一个允许重映射的 `mappages` 变体）将相同的映射添加到 `kpagetable` 中。
- 在关键点同步：**在任何修改用户页表（`pagetable`）的函数中，都必须调用 `pagecopy` 来同步更新 `kpagetable`。
 - fork**：子进程复制了父进程的 `pagetable`，因此也必须调用 `pagecopy` 将用户内存映射复制到其 `kpagetable`。
 - exec**：加载新程序会创建全新的用户页表，必须调用 `pagecopy` 进行同步。
 - sbrk**：扩展或收缩用户内存时，必须同步更新 `kpagetable`。增长时调用 `pagecopy`；收缩时调用 `uv munmap` 取消 `kpagetable` 中的映射。
 - userinit**：创建第一个进程时，初始化完 `pagetable` 后，立即调用 `pagecopy` 建立初始映射。
- 简化 copyin**：一旦 `kpagetable` 包含了用户空间映射，`copyin` 和 `copyinstr` 函数就可以直接使用用户虚拟地址进行内存操作，效率极大提升。

修改的代码

```
// kernel/vm.c
int umappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm) {
    uint64 a, last;
    pte_t *pte;
    a = PGROUNDDOWN(va);
    last = PGROUNDDOWN(va + size - 1);
    for(;;){
        if((pte = walk(pagetable, a, 1)) == 0)
            return -1;
        *pte = PA2PTE(pa) | perm | PTE_V;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}

int pagecopy(pagetable_t oldpage, pagetable_t newpage, uint64 begin, uint64 end)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    begin = PGROUNDUP(begin);
    for (i = begin; i < end; i += PGSIZE) {
        if ((pte = walk(oldpage, i, 0)) == 0)
            panic("pagecopy walk oldpage nullptr");
        if ((*pte & PTE_V) == 0)
            panic("pagecopy oldpage pte not valid");
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte) & (~PTE_U);
        if (umappages(newpage, i, PGSIZE, pa, flags) != 0) {
            goto err;
        }
    }
    return 0;
err:
    uvmunmap(newpage, 0, i / PGSIZE, 1);
    return -1;
}

// kernel/proc.c
int fork(void) {
```

```

// ... 复制父进程内存 ...
if (pagecopy(np->pagetable, np->kpagetable, 0, np->sz) != 0) {
    freeproc(np);
    release(&np->lock);
    return -1;
}
// ... 其他设置 ...
}

// kernel/exec.c
int exec(char *path, char **argv) {
    // ... 加载新程序 ...
    p->pagetable = pagetable;
    p->sz = sz;
    proc_freepagetable(oldpagetable, oldsiz);
    if (pagecopy(p->pagetable, p->kpagetable, 0, p->sz) != 0) {
        goto bad;
    }
    w_satp(MAKE_SATP(p->kpagetable));
    sfence_vma();
    // ... 返回 ...
}

// kernel/proc.c
int growproc(int n) {
    uint sz;
    struct proc *p = myproc();
    sz = p->sz;
    if(n > 0){
        if (sz + n > PLIC || (sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
            return -1;
        }
        if (pagecopy(p->pagetable, p->kpagetable, p->sz, sz) != 0) {
            return -1;
        }
    } else if(n < 0){
        sz = uvmdealloc(p->pagetable, sz, sz + n);
        if (sz != p->sz) {
            uvmunmap(p->kpagetable, PGROUNDUP(sz), (PGROUNDUP(p->sz) - PGROUNDUP(sz))
/ PGSIZE, 0);
        }
    }
    p->sz = sz;
    return 0;
}

// kernel/proc.c
void userinit(void) {

```

```
// ... 初始化进程 ...
uvminit(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
pagecopy(p->pagetable, p->kpagetable, 0, p->sz); // 同步初始映射
// ... 设置 trapframe ...
}
```

四、实验结果

所有实验任务均通过 `make grade` 测试：

```
$ make qemu-gdb
pte printout: OK (1.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.7s)
== Test usertests ==
$ make qemu-gdb
(104.3s)
== Test usertests: copyin ==
    usertests: copyin: OK
== Test usertests: copyinstr1 ==
    usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
    usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
    usertests: copyinstr3: OK
== Test usertests: sbrkmuch ==
    usertests: sbrkmuch: OK
== Test usertests: all tests ==
    usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
```

五、思考与总结

本次页表实验是迄今为止最具挑战性的一次，它要求我们深入操作系统的内存管理核心。实现 `vmprint` 让我清晰地看到了三级页表的树状结构。为每个进程创建专属内核页表，不仅加深了我

对 `satp` 寄存器、TLB 刷新（`sfence_vma`）和进程切换机制的理解，更让我体会到资源管理的复杂性——必须精确地分配和回收页表内存，避免内存泄漏或非法访问。

最令人兴奋的是简化 `copyin` 的部分。通过将用户空间映射到内核页表，我们巧妙地绕过了繁琐的软件地址转换，使内核能像访问自身内存一样访问用户数据。这生动地展示了操作系统设计中“空间换时间”的权衡。然而，实验手册也提醒我们，这种性能优化可能带来安全风险（如 Meltdown 攻击），这让我意识到现代操作系统在性能与安全之间所做的艰难抉择。这次实验极大地提升了我的系统级编程能力和对虚拟内存抽象的深刻理解。

实验报告4

Xv6 操作系统实验报告：陷阱与系统调用

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 4 - Traps

提交日期：2025年9月4日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在深入理解 xv6 操作系统中陷阱 (trap) 机制的实现，包括系统调用的处理、用户态与内核态的切换，以及用户级中断处理程序的实现。实验分为三个部分：

- RISC-V 汇编理解**：通过分析汇编代码，理解函数调用约定、寄存器使用和端序问题。
- 回溯跟踪 (Backtrace)**：实现内核中的栈回溯功能，用于调试和错误追踪。
- 报警处理 (Alarm)**：实现一个用户级的定时报警机制，周期性地调用用户指定的处理函数。

通过本实验，我们将掌握 xv6 中陷阱处理的全流程，包括陷阱帧 (trapframe) 的保存与恢复、用户态与内核态的切换，以及如何安全地在用户态处理程序中执行代码。

二、实验环境与准备

实验环境基于 xv6 的 traps 分支进行开发。

环境搭建步骤

- 切换到 traps 分支：

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

- 编译并启动 xv6：

```
make qemu
```

3. 开发流程：

- 修改内核文件（如 `kernel/trap.c`、`kernel/proc.h`）和用户程序（如 `user/alarmtest.c`）。
- 将新增系统调用添加到 `user/user.h`、`user/usys.pl`、`kernel/syscall.h` 和 `kernel/syscall.c`。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. RISC-V 汇编理解 (easy)

实验过程

1. 编译 `user/call.c` 生成汇编代码 `user/call.asm`：

```
make fs.img
```

2. 分析 `call.asm` 中的函数 `g`、`f` 和 `main`，回答以下问题：

问题解答

Q1: 哪些寄存器包含函数参数？例如，哪个寄存器在 `main` 调用 `printf` 时保存 13？

根据 RISC-V 调用约定，函数参数使用寄存器 `a0-a7` 传递。在 `main` 函数的汇编代码中：

```
24: 4635          li    a2,13
```

可以看到，值 13 被加载到 `a2` 寄存器中。

Q2: 在 `main` 的汇编代码中，对函数 `f` 的调用在哪里？对 `g` 的调用在哪里？

编译器对函数调用进行了内联优化。整个函数调用链 `f(g(8))+1` 被优化为常数 12：

```
26: 45b1          li    a1,12
```

值 12 被直接加载到 `a1` 寄存器中。

Q3: 函数 printf 位于什么地址?

从汇编代码可以看到:

```
00000000000000638 <printf>
```

printf 函数位于地址 0x638。

Q4: 在 main 中 jalr 到 printf 后, 寄存器 ra 中的值是什么?

jalr 指令会将下一条指令的地址保存到 ra 寄存器中:

```
30: 00000097      auipc    ra, 0x0
34: 608080e7      jalr     1544(ra) # 638 <printf>
38: 4501          li       a0, 0
```

jalr 指令位于 0x34, 下一条指令在 0x38, 因此 ra 的值为 0x38。

Q5: 运行以下代码, 输出是什么? 如果是大端序, 需要如何修改 i?

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

输出是 "HE110 World"。57616 的十六进制是 E110, 格式化为 %x 输出 E110。

i 的值为 0x00646c72, 在小端序系统中字节顺序为 72 6c 64 00, 对应 ASCII 字符 'r' 'l' 'd' '\0'。

如果 RISC-V 是大端序, 需要将 i 改为 0x726c6400, 57616 不需要改变。

Q6: 在以下代码中, 'y=' 后面会打印什么? 为什么?

```
printf("x=%d y=%d", 3);
```

会打印一个随机值。因为 printf 期望两个参数, 但只提供了一个。第二个值 y 会从调用时 a2 寄存器中的任意值获取。

2. 回溯跟踪 (Backtrace, moderate)

实验过程

1. 在 `kernel/riscv.h` 中添加 `r_fp()` 函数:

```
// kernel/riscv.h
static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, $0" : "=r" (x));
    return x;
}
```

2. 在 `kernel/defs.h` 中添加函数声明:

```
// kernel/defs.h
void backtrace(void);
```

3. 在 `kernel/printf.c` 中实现 `backtrace()` 函数:

```
// kernel/printf.c
void
backtrace(void)
{
    printf("backtrace:\n");
    uint64 fp = r_fp();
    uint64 page_top = PGROUNDUP(fp);

    // 遍历栈帧直到栈底
    while (fp < page_top) {
        // 返回地址位于 fp-8
        uint64 ra = *(uint64*)(fp - 8);
        printf("%p\n", ra);

        // 上一个栈帧指针位于 fp-16
        uint64 prev_fp = *(uint64*)(fp - 16);
        if (prev_fp <= fp || prev_fp >= page_top) {
            break;
        }
        fp = prev_fp;
    }
}
```

4. 在 `sys_sleep` 中调用 `backtrace` 进行测试:

```
// kernel/sysproc.c
uint64
sys_sleep(void)
{
    // ...
    backtrace(); // 添加回溯
    // ...
}
```

- 在 `kernel/printf.c` 的 `panic` 函数中也添加 `backtrace` 调用：

```
// kernel/printf.c
void
panic(char *s)
{
    // ...
    backtrace();
    // ...
}
```

原理分析

在 RISC-V 中，每个栈帧的结构如下：

- fp-8: 返回地址 (return address)
- fp-16: 上一个栈帧的帧指针 (previous frame pointer)
- fp-...: 局部变量和保存的寄存器

通过当前帧指针 (s0 寄存器)，我们可以遍历整个调用栈。xv6 为每个内核栈分配一页内存，通过 PGROUNDDUP 可以找到栈顶地址，作为遍历的边界条件。

3. 报警处理 (Alarm, hard)

实验过程

第一步：添加系统调用

- 在 `user/user.h` 中添加函数声明：

```
// user/user.h
int sigalarm(int ticks, void (*handler)());
```

```
int sigreturn(void);
```

2. 在 `user/usys.pl` 中添加入口:

```
# user/usys.pl
entry("sigalarm");
entry("sigreturn");
```

3. 在 `kernel/syscall.h` 中添加系统调用号:

```
// kernel/syscall.h
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

4. 在 `kernel/syscall.c` 中添加函数声明和系统调用表项:

```
// kernel/syscall.c
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);

static uint64 (*syscalls[])(void) = {
    // ...
    [SYS_sigalarm]    sys_sigalarm,
    [SYS_sigreturn]   sys_sigreturn,
};
```

第二步：修改进程结构

1. 在 `kernel/proc.h` 的 `struct proc` 中添加字段:

```
// kernel/proc.h
struct proc {
    // ...
    int alarm_interval;           // 报警间隔时间
    void (*alarm_handler)();      // 报警处理函数
    int alarm_ticks;             // 距离下次报警的ticks数
    struct trapframe *alarm_trapframe; // 保存的陷阱帧
    int alarm_pending;           // 是否有未处理的报警
    // ...
};
```

2. 在 `kernel/proc.c` 的 `allocproc` 中初始化新字段：

```
// kernel/proc.c
static struct proc*
allocproc(void)
{
    // ...
    // 分配报警陷阱帧
    if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // 初始化报警字段
    p->alarm_interval = 0;
    p->alarm_handler = 0;
    p->alarm_ticks = 0;
    p->alarm_pending = 0;
    // ...
}
```

3. 在 `freeproc` 中释放资源：

```
// kernel/proc.c
static void
freeproc(struct proc *p)
{
    // ...
    if(p->alarm_trapframe)
        kfree((void*)p->alarm_trapframe);
    p->alarm_trapframe = 0;
    // ...
}
```

第三步：实现系统调用

1. 在 `kernel/sysproc.c` 中实现 `sys_sigalarm`：

```
// kernel/sysproc.c
uint64
sys_sigalarm(void)
{
    int interval;
```

```

uint64 handler;

if(argint(0, &interval) < 0 || argaddr(1, &handler) < 0) {
    return -1;
}

struct proc *p = myproc();
p->alarm_interval = interval;
p->alarm_handler = (void(*)())handler;
p->alarm_ticks = 0;

return 0;
}

```

2. 实现 sys_sigreturn :

```

// kernel/sysproc.c
uint64
sys_sigreturn(void)
{
    struct proc *p = myproc();

    if(p->alarm_pending) {
        // 恢复陷阱帧
        *p->trapframe = *p->alarm_trapframe;
        p->alarm_pending = 0;
    }

    return 0;
}

```

第四步：修改陷阱处理

在 kernel/trap.c 的 usertrap 函数中添加报警处理逻辑：

```

// kernel/trap.c
void
usertrap(void)
{
    // ...
    if(which_dev == 2) {
        // 时钟中断
        if(p->alarm_interval > 0) {
            p->alarm_ticks++;
        }
    }
}

```

```
// 检查是否应该触发报警
if(p->alarm_ticks >= p->alarm_interval && !p->alarm_pending) {
    // 保存当前陷阱帧
    *p->alarm_trapframe = *p->trapframe;

    // 设置陷阱帧的程序计数器为报警处理函数
    p->trapframe->epc = (uint64)p->alarm_handler;

    p->alarm_ticks = 0;
    p->alarm_pending = 1;
}
}
yield();
}
// ...
}
```

原理分析

报警机制的实现涉及以下几个关键点：

1. **定时检测**: 在时钟中断中检查是否达到报警间隔
2. **状态保存**: 在调用用户处理函数前保存完整的陷阱帧
3. **安全调用**: 通过修改陷阱帧的 epc 寄存器，使返回到用户态时执行处理函数
4. **状态恢复**: 在 sigreturn 系统调用中恢复之前保存的陷阱帧
5. **重入保护**: 通过 alarm_pending 标志防止处理函数重入

四、实验结果

```
-- Test running alarmtest ==
$ make qemu-gdb
(3.2s)
-- Test alarmtest: test0 ==
alarmtest: test0: OK
-- Test alarmtest: test1 ==
alarmtest: test1: OK
-- Test alarmtest: test2 ==
alarmtest: test2: OK
-- Test usertests ==
$ make qemu-gdb
usertests: OK (108.2s)
-- Test time ==
time: OK
Score: 85/85
```

五、思考与总结

通过本次实验，我深入理解了 xv6 操作系统的陷阱处理机制：

1. **陷阱处理流程**：掌握了用户态到内核态的完整切换过程，包括陷阱帧的保存与恢复、系统调用的分发和处理。
2. **栈结构与调试**：通过实现 backtrace 功能，深入理解了函数调用栈的结构和遍历方法，这对内核调试非常有价值。
3. **用户级中断处理**：实现了完整的报警机制，包括定时检测、状态保存与恢复、重入保护等关键功能。
4. **系统调用实现**：掌握了在 xv6 中添加新系统调用的完整流程，从用户接口到内核实现的各个环节。

实验中最具挑战性的部分是报警机制的状态管理。需要确保在处理报警函数时，能够完整保存和恢复执行状态，同时防止重入问题。通过使用 alarm_trapframe 保存完整状态和 alarm_pending 标志防止重入，最终实现了正确且安全的报警机制。

实验报告5

Xv6 操作系统实验报告：Lazy Page Allocation

课程名称：6.828 操作系统工程（MIT 6.S081）

实验名称：Lab 5 - Lazy Page Allocation

提交日期：2025年9月5日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在实现 xv6 操作系统中的**延迟页面分配（Lazy Page Allocation）**机制。传统的内存分配方式在调用 `sbrk()` 时会立即分配物理内存并建立页表映射，而延迟分配策略则仅在进程实际访问内存时才进行物理页的分配，从而提高内存使用效率并优化大内存分配的性能。

实验分为三个主要部分：

- 修改 `sbrk()` 系统调用：**取消立即分配物理内存的行为，仅增加进程的虚拟地址空间大小。
- 处理缺页中断：**在 `usertrap()` 和 `kerneltrap()` 中捕获页面错误异常（13号和15号），并动态分配物理页。
- 完善边界处理：**确保所有边界情况（如非法地址、内存不足、系统调用中的延迟分配等）都能正确处理。

通过本实验，我们将深入理解 xv6 的内存管理机制、缺页中断处理流程，以及如何在内核中实现高效的延迟分配策略。

二、实验环境与准备

实验环境基于 xv6 的 `lazy` 分支进行开发。

环境搭建步骤

- 切换到 `lazy` 分支：

```
git fetch  
git checkout lazy  
make clean
```

2. 编译并启动 xv6:

```
make qemu
```

3. 开发流程:

- 修改内核文件 (如 `kernel/sysproc.c`、`kernel/trap.c`、`kernel/vm.c`)。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. 修改 `sbrk()` 系统调用 (easy)

实验过程

修改 `kernel/sysproc.c` 中的 `sys_sbrk()` 函数，取消立即分配物理内存的行为:

```
// kernel/sysproc.c
uint64
sys_sbrk(void)
{
    int addr;
    int n;
    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;

    // 仅增加虚拟地址空间大小，不分配物理页
    // 原来的 growproc() 调用被注释掉
    myproc()->sz += n;

    return addr;
}
```

原理分析

修改后，`sbrk()` 不再调用 `growproc()`，因此不会立即分配物理页。当进程首次访问新分配的内存时，会触发缺页中断 (page fault)，内核在中断处理程序中再进行实际的物理页分配。

问题分析

运行 `echo hi` 命令后，会出现如下错误:

```
usertrap(): unexpected scause 0x000000000000000f pid=3
    sepc=0x0000000000001258 stval=0x0000000000004008
    va=0x0000000000004000 pte=0x0000000000000000
panic: uvmunmap: not mapped
```

这是因为 `sbrk()` 分配了虚拟地址空间但没有建立物理映射，当进程尝试访问这些地址时触发缺页中断（`scause=15`, 写错误），而内核尚未处理这种中断。

2. 处理缺页中断 (moderate)

实验过程

在 `usertrap()` 中处理用户态缺页中断

```
// kernel/trap.c
void
usertrap(void)
{
    // ...
} else if (r_scause() == 13 || r_scause() == 15) {
    // 13: Load page fault, 15: Store page fault
    uint64 va = r_stval();

    // 检查地址是否合法
    if(va >= p->sz || va < p->trapframe->sp) {
        p->killed = 1;
    } else {
        // 分配物理页并建立映射
        va = PGROUNDDOWN(va);
        uint64 pa = (uint64)kalloc();
        if(pa == 0) {
            p->killed = 1;
        } else {
            memset((void*)pa, 0, PGSIZE);
            if(mappages(p->pagetable, va, PGSIZE, pa, PTE_W|PTE_X|PTE_R|PTE_U) != 0)
{
                kfree((void*)pa);
                p->killed = 1;
            }
        }
    }
} else {
    // 其他异常处理
}
```

```
    }
    // ...
}
```

处理内核页表映射

由于在之前的实验 (pgtbl) 中为每个进程添加了独立的内核页表，需要在分配用户页表时同步映射内核页表：

```
// 在mappages后添加
if(mappages(p->kpagetable, va, PGSIZE, pa, PTE_W|PTE_X|PTE_R) != 0) {
    kfree((void*)pa);
    uvmunmap(p->pagetable, va, 1, 0);
    p->killed = 1;
}
```

在 kerneltrap() 中处理内核态缺页中断

```
// kernel/trap.c
void
kerneltrap()
{
    // ...
    uint64 scause = r_scause();
    if (scause == 13 || scause == 15) {
        uint64 va = r_stval();
        if(va >= myproc()->sz || va < myproc()->trapframe->sp) {
            myproc()->killed = 1;
        } else {
            va = PGROUNDDOWN(va);
            uint64 pa = (uint64)kalloc();
            if(pa != 0) {
                memset((void*)pa, 0, PGSIZE);
                if(mappages(myproc()->pagetable, va, PGSIZE, pa,
PTE_W|PTE_X|PTE_R|PTE_U) != 0) {
                    kfree((void*)pa);
                } else if(mappages(myproc()->kpagetable, va, PGSIZE, pa,
PTE_W|PTE_X|PTE_R) != 0) {
                    kfree((void*)pa);
                    uvmunmap(myproc()->pagetable, va, 1, 0);
                }
            }
        }
    }
}
```

```
// ...  
}
```

原理分析

- 缺页中断分为读错误 (13) 和写错误 (15)，均在 `usertrap()` 和 `kerneltrap()` 中被捕获。
- 需要检查地址是否在合法范围内（介于栈指针和进程大小之间）。
- 分配物理页后，需要在用户页表和内核页表中都建立映射。
- 内核态缺页中断需同样处理，以确保内核访问用户空间时也能正确触发延迟分配。

3. 完善边界处理 (moderate)

实验过程

修改 `uvmunmap()` 避免 panic

在 `kernel/vm.c` 中修改 `uvmunmap()`，使其在遇到未映射的页时跳过而非 panic：

```
// kernel/vm.c  
void  
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)  
{  
    // ...  
    for(uint64 i = 0; i < npages; i++){  
        if((pte = walk(pagetable, a, 0)) == 0)  
            continue; // 跳过未映射的页  
        if((*pte & PTE_V) == 0)  
            continue; // 跳过无效页  
        // ...  
    }  
}
```

处理 `copyout()` 中的延迟分配

在 `kernel/vm.c` 的 `copyout()` 中处理可能存在的延迟分配：

```
// kernel/vm.c  
int  
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)  
{
```

```

while(len > 0){
    va0 = PGROUNDDOWN(dstva);

    // 处理延迟分配
    pte_t *pte = walk(pagetable, va0, 0);
    if(pte == 0 || (*pte & PTE_V) == 0) {
        // 触发延迟分配
        if(lazyvalidate(myproc(), va0) != 0) {
            return -1;
        }
    }

    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
        return -1;
    // ...
}

return 0;
}

```

实现 `lazyvalidate()` 函数

将延迟分配逻辑封装为函数：

```

// kernel/vm.c
int lazyvalidate(struct proc* p, uint64 va) {
    if(va >= p->sz || va < p->trapframe->sp) {
        return -1; // 地址越界
    }

    va = PGROUNDDOWN(va);
    uint64 pa = (uint64)kalloc();
    if(pa == 0) {
        return -1; // 分配失败
    }

    memset((void*)pa, 0, PGSIZE);
    if(mappages(p->pagetable, va, PGSIZE, pa, PTE_W|PTE_X|PTE_R|PTE_U) != 0) {
        kfree((void*)pa);
        return -1;
    }

    if(mappages(p->kpagetable, va, PGSIZE, pa, PTE_W|PTE_X|PTE_R) != 0) {
        kfree((void*)pa);
        uvmunmap(p->pagetable, va, 1, 0);
        return -1;
    }
}

```

```
    }

    return 0;
}
```

处理 `copyin_new()` 的边界条件

修改 `kernel/vmcopyin.c` 中的 `copyin_new()`，修复边界检查：

```
// kernel/vmcopyin.c
int
copyin_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    struct proc *p = myproc();
    // 修复边界检查：使用 > 而不是 >=
    if (srcva > p->sz || srcva + len > p->sz || srcva + len < srcva)
        return -1;
    // ...
}
```

原理分析

- `uvmunmap()` 的修改避免了在释放未映射页时发生 panic。
- `copyout()` 和 `copyin_new()` 的修改确保了系统调用在处理延迟分配的内存时能正确工作。
- 边界检查防止了地址越界访问。
- 将延迟分配逻辑封装为 `lazyvalidate()` 函数提高了代码的可重用性和可维护性。

四、实验结果

```
== Test usertests: subdir ==
  usertests: subdir: OK
== Test usertests: fourfiles ==
  usertests: fourfiles: OK
== Test usertests: sharedfd ==
  usertests: sharedfd: OK
== Test usertests: exectest ==
  usertests: exectest: OK
== Test usertests: bigargtest ==
  usertests: bigargtest: OK
== Test usertests: bigwrite ==
  usertests: bigwrite: OK
== Test usertests: bsstest ==
  usertests: bsstest: OK
== Test usertests: sbrkbasic ==
  usertests: sbrkbasic: OK
== Test usertests: kernmem ==
  usertests: kernmem: OK
== Test usertests: validateetest ==
  usertests: validateetest: OK
== Test usertests: opentest ==
  usertests: opentest: OK
== Test usertests: writetest ==
  usertests: writetest: OK
== Test usertests: writebig ==
  usertests: writebig: OK
== Test usertests: createtest ==
  usertests: createtest: OK
```

```
== Test usertests: exitiput ==
    usertests: exitiput: OK
== Test usertests: iput ==
    usertests: iput: OK
== Test usertests: mem ==
    usertests: mem: OK
== Test usertests: pipe1 ==
    usertests: pipe1: OK
== Test usertests: preempt ==
    usertests: preempt: OK
== Test usertests: exitwait ==
    usertests: exitwait: OK
== Test usertests: rmdot ==
    usertests: rmdot: OK
== Test usertests: fourteen ==
    usertests: fourteen: OK
== Test usertests: bigfile ==
    usertests: bigfile: OK
== Test usertests: dirfile ==
    usertests: dirfile: OK
== Test usertests: iref ==
    usertests: iref: OK
== Test usertests: forktest ==
    usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119
```

五、思考与总结

通过本次实验，我深入掌握了 xv6 延迟页面分配机制的核心原理与实现细节：该机制通过推迟物理页分配直至实际访问，显著提升内存使用效率，尤其在大内存或稀疏访问场景下性能优势明显。实验过程中，我熟练掌握了缺页中断（RISC-V 中13号读错误、15号写错误）的处理流程，理解了用户态与内核态协同分配物理页的机制，并因先前实验引入的独立内核页表，进一步学习了用户页分配时同步更新内核页表的必要性与实现方法，从而深化了对 xv6 内存管理架构的理解。同时，我系统处理了地址越界、内存不足、系统调用中延迟访问等边界情况，修改了 `uvmunmap`、`copyout` 等关键函数以兼容延迟分配逻辑。调试阶段借助 `printf` 与 `vmprint` 工具深入观察页表结构，提升了系统级调试能力。实验主要难点在于内核页表同步遗漏、边界条件疏忽及系统调用路径中未预分配页的处理，最终通过同步更新内核页表、全面检查用户空间访问

路径、封装 `Lazyvalidate()` 函数等方案逐一攻克。本次实验不仅巩固了操作系统内存管理的核心知识，更锻炼了系统编程与复杂问题调试能力，为后续学习高级内存管理技术奠定了坚实基础。

实验报告6

Xv6 操作系统实验报告：写时复制（Copy-on-Write）

课程名称：6.828 操作系统工程（MIT 6.S081）

实验名称：Lab 6 - Copy-on-Write

提交日期：2025年9月5日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在实现 xv6 操作系统中的写时复制（Copy-on-Write, COW）机制。传统的 `fork()` 系统调用会完整复制父进程的用户空间内存到子进程，这在内存使用上效率低下，尤其是当子进程很快执行 `exec()` 时，复制的内存往往被立即丢弃。COW 机制通过延迟内存复制到实际需要写入时才进行，显著提高了内存利用率和系统性能。

实验的主要任务包括：

- 修改 `uvmcopy()`：使子进程与父进程共享物理页面，而非立即复制。
- 实现引用计数：跟踪每个物理页的被引用次数，确保仅在最后一个引用消失时释放页面。
- 处理页错误：在写入 COW 页面时分配新页面并复制内容。
- 修改 `copyout()`：处理内核态下的 COW 页面写入。

通过本实验，我们将深入理解虚拟内存管理、页表机制和陷阱处理的全流程。

二、实验环境与准备

实验环境基于 xv6 的 `cow` 分支进行开发。

环境搭建步骤

- 切换到 `cow` 分支：

```
git fetch  
git checkout cow  
make clean
```

2. 编译并启动 xv6:

```
make qemu
```

3. 开发流程:

- 修改内核文件 (如 `kernel/kalloc.c`、`kernel/vm.c`、`kernel/trap.c`)。
- 使用 `cowtest` 和 `usertests` 进行测试。
- 使用 `make grade` 验证实现正确性。

三、实验过程与原理分析

1. 实现引用计数机制

为了跟踪物理页的被引用情况，我们在 `kernel/kalloc.c` 中定义了一个全局引用计数数组 `cowcount`，并修改了 `kalloc()` 和 `kfree()` 函数。

代码实现

```
// kernel/kalloc.c

#define PA2INDEX(pa) (((uint64)pa) / PGSIZE)

// 物理页引用计数数组
int cowcount[PHYSTOP / PGSIZE];

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

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

    acquire(&kmem.lock);
    int remain = --cowcount[PA2INDEX(pa)];
    release(&kmem.lock);

    if (remain > 0) {
        return; // 仍有引用，不真正释放
    }
}
```

```

// 原始释放逻辑
memset(pa, 1, PGSIZE);
r = (struct run*)pa;
acquire(&kmem.lock);
r->next = kmem.freelist;
kmem.freelist = r;
release(&kmem.lock);
}

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

    acquire(&kmem.lock);
    r = kmem.freelist;
    if (r)
        kmem.freelist = r->next;
    release(&kmem.lock);

    if (r) {
        memset((char *)r, 5, PGSIZE);
        int idx = PA2INDEX(r);
        if (cowcount[idx] != 0) {
            panic("kalloc: cowcount[idx] != 0");
        }
        cowcount[idx] = 1; // 新页引用计数为1
    }
    return (void*)r;
}

// 辅助函数: 调整引用计数
void adjustref(uint64 pa, int num) {
    if (pa >= PHYSTOP) {
        panic("adjustref: pa too big");
    }
    acquire(&kmem.lock);
    cowcount[PA2INDEX(pa)] += num;
    release(&kmem.lock);
}

```

原理分析

为了跟踪物理页的被引用情况，在 kernel/kalloc.c 中定义了全局引用计数数组 cowcount，并修改了 kalloc() 和 kfree() 函数。引用计数数组以物理页号索引，记录每个物理页的被引用数。在 kfree() 中，仅当引用计数降为0时才真正释放页面；在 kalloc() 中，新分配的页面引用计数初始化

为1。同时提供了 `adjustref()` 辅助函数来安全地增加或减少引用计数，该函数通过加锁操作保证原子性。

2. 修改 `uvmcopy()` 实现页面共享

在 `fork()` 时，不再复制物理页，而是让子进程共享父进程的页面，并标记为只读。

代码实现

```
// kernel/vm.c

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);
        *pte &= ~PTE_W; // 清除写权限，标记为COW页
        flags = PTE_FLAGS(*pte);
        if (mappages(new, i, PGSIZE, pa, flags) != 0) {
            goto err;
        }
        adjustref(pa, 1); // 增加引用计数
    }
    return 0;

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

原理分析

在 `fork()` 时，不再复制物理页，而是让子进程共享父进程的页面，并标记为只读。具体实现中，`uvmcopy()` 函数遍历父进程的页表，将子进程的页表项指向相同的物理页，同时清除写权限位 (`PTE_W`) 以标记为COW页。每共享一页，就通过 `adjustref()` 函数增加该物理页的引用计数。这种设计确保了父子进程初始时共享所有物理页面，只有在实际写入时才进行复制。

3. 实现写时复制处理

当进程写入COW页面时，触发页错误 (scause=15)，在 `usertrap()` 中处理并分配新页面。

代码实现

```
// kernel/vm.c

// COW分配函数：为当前进程复制共享页
int
cowalloc(pagetable_t pagetable, uint64 va)
{
    if (va >= MAXVA) {
        return -1;
    }

    pte_t *pte = walk(pagetable, va, 0);
    if (pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0) {
        return -1;
    }

    uint64 pa_old = PTE2PA(*pte);
    uint64 pa_new = (uint64)kalloc();
    if (pa_new == 0) {
        return -1;
    }

    memmove((void *)pa_new, (const void *)pa_old, PGSIZE);
    kfree((void *)pa_old); // 减少原页的引用计数

    *pte = PA2PTE(pa_new) | PTE_FLAGS(*pte) | PTE_W; // 新页可写
    return 0;
}
```

在 `kernel/trap.c` 中处理页错误：

```
// kernel/trap.c

void
usertrap(void)
{
    // ...
    if (r_scause() == 15) { // COW页写入错误
```

```

if (cowalloc(p->pagetable, r_stval()) < 0) {
    p->killed = 1;
}
} else if (r_scause() == 8) {
    // 系统调用
    // ...
} else {
    // 其他错误
}
// ...
}

```

原理分析

当进程写入COW页面时，会触发页错误 (scause=15)，在 usertrap() 中处理并分配新页面。cowalloc() 函数负责处理具体的复制工作：它首先检查虚拟地址有效性，然后分配新物理页，复制原页内容，更新页表项指向新页并设置写权限，最后通过对原页调用 kfree() 减少引用计数。这种机制确保了只有在实际需要写入时才进行页面复制，最大限度地节省了内存空间。

4. 修改 copyout() 处理内核态COW页面

copyout() 在内核态执行，不会触发页错误，因此需显式检查COW页并处理。

代码实现

```

// kernel/vm.c

int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;

    while (len > 0) {
        va0 = PGROUNDDOWN(dstva);
        pte_t *pte = walk(pagetable, va0, 0);
        if (pte == 0 || (*pte & PTE_U) == 0 || (*pte & PTE_V) == 0) {
            return -1;
        }
        if ((*pte & PTE_W) == 0) { // COW页
            if (cowalloc(pagetable, va0) < 0) {
                return -1;
            }
        }
        pa0 = PTE2PA(*pte);

```

```
// 复制数据...
// ...
}
return 0;
}
```

原理分析

由于 copyout() 在内核态执行，不会触发页错误，因此需要显式检查COW页并处理。在复制数据前，copyout() 会检查目标页是否为COW页（通过检查写权限位），如果是则调用 cowalloc() 进行页面复制。这种处理确保了内核态和用户态对COW页面的一致性处理。

四、实验结果

```
-- Test running cowtest ==
$ make qemu-gdb
(5.6s)
-- Test simple ==
simple: OK
-- Test three ==
three: OK
-- Test file ==
file: OK
-- Test usertests ==
$ make qemu-gdb
(46.2s)
-- 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
```

五、思考与总结

通过本次实验，我深入理解了写时复制机制的原理和实现方式。COW 机制通过延迟内存复制到实际需要时进行，显著提高了内存使用效率，特别是在 fork() 后接 exec() 的常见场景中优势明显。引用计数机制是实现COW的关键，需要精心设计以确保原子性和正确性。页错误处理机制使得操作系统能够捕获对COW页的写入操作，并在此时进行实际的页面复制。

实验中最具挑战性的部分是确保引用计数管理的原子性和正确处理内核态下的COW页面。通过加锁保护引用计数操作和修改 copyout() 函数，最终实现了正确且高效的COW机制。本次实验不仅加深了对虚拟内存和页表机制的理解，也提高了系统编程和调试能力，为后续操作系统开发打下了坚实基础。

从系统设计角度，COW 机制体现了"延迟优化"的思想，通过增加一定程度的复杂性来换取性能提升，这种权衡在系统设计中十分常见。实现过程中还需要考虑各种边界情况，如内存不足时的处理、多个进程同时访问COW页的同步问题等，这些都是构建健壮操作系统必须面对的挑战。

实验报告7

Xv6 操作系统实验报告：Multithreading

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 7 - Multithreading

提交日期：2025年9月6日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在深入理解多线程编程的核心机制，包括用户级线程的上下文切换、多线程环境下的数据同步与互斥，以及线程间的屏障同步。实验分为三个主要部分：

1. **Uthread: 用户级线程切换**: 实现一个用户级线程包，包括线程的创建、调度和上下文切换机制。
2. **Using threads**: 在多线程环境下优化哈希表的性能，通过加锁机制解决数据竞争问题，并利用分桶锁提升并行效率。
3. **Barrier**: 实现一个屏障同步原语，确保所有线程在指定点等待，直到所有线程都到达该点后再继续执行。

通过本实验，我们将掌握用户级线程的上下文保存与恢复、多线程编程中锁的使用与性能优化，以及条件变量在同步中的应用。

二、实验环境与准备

实验环境基于 xv6 的 `thread` 分支进行开发。

环境搭建步骤

1. 切换到 `thread` 分支：

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

2. 编译并启动 xv6：

```
make qemu
```

3. 开发流程：

- 修改用户级线程包文件（如 `user/uthread.c` 和 `user/uthread_switch.S`）。
- 修改哈希表测试文件 `notxv6/ph.c`。
- 修改屏障同步文件 `notxv6/barrier.c`。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. Uthread: 用户级线程切换 (moderate)

实验过程

实现线程上下文切换汇编代码

将内核中的上下文切换汇编代码移植到用户级线程包中：

```
# user/uthread_switch.S
.text

.globl thread_switch
thread_switch:
    /* Save old thread's registers */
    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)

    /* Restore new thread's registers */
```

```
    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
```

定义线程上下文结构体

在 `user/uthread.c` 中定义线程上下文结构体和线程结构体：

```
// Saved registers for user-level thread switching
struct thread_context {
    uint64 ra;
    uint64 sp;
    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 {
    struct thread_context context;
    char stack[STACK_SIZE];
    int state;
};
```

实现线程初始化辅助函数

```

void clear_thread(struct thread *t, void (*func)()) {
    memset(t->stack, 0, STACK_SIZE);
    memset(&t->context, 0, sizeof(struct thread_context));
    t->state = RUNNABLE;
    t->context.sp = (uint64)((char *)&t->stack + STACK_SIZE); // 栈顶作为初始sp
    t->context.ra = (uint64)func; // 设置函数入口地址
}

```

修改线程创建和调度函数

```

void thread_create(void (*func)()) {
    struct thread *t;
    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    clear_thread(t, func); // 初始化新线程
}

void thread_schedule(void) {
    // ... 线程选择逻辑 ...
    if (current_thread != next_thread) {
        next_thread->state = RUNNING;
        struct thread *prev = current_thread;
        current_thread = next_thread;
        thread_switch((uint64)&prev->context, (uint64)&next_thread->context);
    }
    // ...
}

```

原理分析

用户级线程的上下文切换机制是整个多线程包的核心基础。其工作原理在于通过保存和恢复线程的执行上下文来实现线程间的切换。在 RISC-V 架构中，我们需要特别关注被调用者保存寄存器（callee-saved registers），因为这些寄存器在函数调用过程中保持不变，由被调用函数负责保存和恢复。具体来说，我们需要保存的寄存器包括返回地址 ra、栈指针 sp 以及 s0-s11 这些在函数调用中保持稳定的寄存器。

线程初始化过程需要为每个线程建立独立的执行环境。关键步骤包括设置线程的初始栈指针和返回地址。栈指针必须指向线程栈的最高地址，因为栈在内存中是从高地址向低地址增长的。返回地址则设置为用户提供的线程函数地址，这样当线程第一次被调度执行时，就会从该函数开始运行。

线程调度器采用协作式多任务机制，这意味着线程必须主动调用 `yield` 函数来放弃 CPU 控制权，而不是像抢占式调度那样由系统强制中断线程执行。这种设计简化了同步要求，但要求程序员显式地插入 `yield` 调用来保证公平性。

上下文切换函数 `thread_switch` 通过两个参数工作：第一个参数指向当前线程的上下文存储位置，第二个参数指向要切换到的线程的上下文。该函数首先将当前寄存器的值保存到内存中，然后从新线程的上下文存储区域加载寄存器值，最后通过 `ret` 指令跳转到新线程的返回地址继续执行。

问题分析

初始实现没有输出，因为缺少上下文切换机制。添加上述代码后，线程能够正确切换并执行，输出三个线程的交替执行结果。

2. Using threads (moderate)

实验过程

分析哈希表竞争条件

多线程环境下，哈希表的插入操作可能出现竞争条件。例如，两个线程同时向同一桶插入元素时，可能导致一个线程的插入被覆盖。

实现分桶锁机制

```
// notxv6/ph.c
pthread_mutex_t locks[NBUCKET]; // 每个桶一个锁

void init_locks() {
    for (int i = 0; i < NBUCKET; i++) {
        pthread_mutex_init(&locks[i], NULL);
    }
}

static void put(int key, int value) {
    int i = key % NBUCKET;
    pthread_mutex_lock(&locks[i]); // 获取桶锁
    // 插入逻辑
    struct entry *e;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }
    if (e) {
        e->value = value;
    } else {
```

```

        insert(key, value, &table[i], table[i]);
    }
    pthread_mutex_unlock(&locks[i]); // 释放桶锁
}

static struct entry* get(int key) {
    int i = key % NBUCKET;
    pthread_mutex_lock(&locks[i]); // 获取桶锁
    struct entry *e;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }
    pthread_mutex_unlock(&locks[i]); // 释放桶锁
    return e;
}

```

原理分析

多线程环境下的哈希表操作面临着严峻的数据竞争和一致性挑战。当多个线程同时访问和修改哈希表时，如果没有适当的同步机制，就会导致数据丢失、内存泄漏甚至程序崩溃等严重问题。本实验通过实现分桶锁机制来解决这些问题。

哈希表的竞争条件主要发生在插入操作过程中。考虑两个线程同时向同一个哈希桶插入不同键值对的情景：两个线程都可能遍历到链表的末尾，然后尝试修改最后一个节点的 next 指针。由于修改操作不是原子性的，后执行修改的线程会覆盖前一个线程的修改结果，导致一个插入操作丢失。这种问题在单线程环境中不会出现，但在多线程环境下却是常见问题。

需要注意的是，虽然分桶锁提升了写操作的并行性，但对读操作也同样需要加锁以确保一致性。即使读操作不会修改数据，在没有锁保护的情况下，也可能读到中间状态或不一致的数据结构。

性能优化

- 分桶锁相比全局锁显著提升了并行性能。
- 测试通过 `ph_safe` (数据正确性) 和 `ph_fast` (性能提升要求)。

3. Barrier (moderate)

实验过程

实现屏障同步机制

```

// notxv6/barrier.c
struct barrier {
    pthread_mutex_t barrier_mutex;

```

```

pthread_cond_t barrier_cond;
int nthread;
int round;
int flag;
} bstate;

static __thread int thread_flag = 0; // 线程本地标志

static void barrier() {
    pthread_mutex_lock(&bstate.barrier_mutex);
    thread_flag = !thread_flag; // 反转本地标志

    // 等待所有线程退出上一轮
    while (thread_flag == bstate.flag) {
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }

    int arrived = ++bstate.nthread;
    if (arrived == nthread) {
        // 最后一个到达的线程
        bstate.round++;
        bstate.flag = !bstate.flag; // 反转全局标志
        bstate.nthread = 0;
        pthread_cond_broadcast(&bstate.barrier_cond);
    } else {
        // 等待其他线程
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
}

```

原理分析

屏障同步是多线程编程中的一种重要协调机制，它要求所有参与线程在继续执行前必须到达某个共同的同步点。这种机制在迭代算法、并行计算和模拟等场景中非常有用，其中每个计算步骤需要所有线程都完成当前阶段后才能进入下一阶段。

本实验实现的屏障同步机制基于条件变量和互斥锁的组合，这是 POSIX 线程库中实现同步的经典模式。互斥锁用于保护共享状态（如到达线程计数器和轮次标志）的原子访问，而条件变量则用于线程间的等待和通知。

屏障实现的核心挑战在于处理多轮屏障调用之间的状态管理。每个线程在进入屏障时都会增加计数器，但如果快速循环的线程在下一轮中过早增加计数器，就会干扰仍在上一轮中等待的线程。为了解决这个问题，实验采用了标志位反转技术：每个轮次都有一个全局标志位，而每个线程有本地标志位。线程通过比较本地标志位与全局标志位来判断自己是否处于当前轮次。

边界处理

- 确保线程在快速循环时不会干扰前一轮的状态。
 - 测试通过多种线程数的场景。
-

四、实验结果

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.7s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/wyb/xv6-labs-2020-thread'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/wyb/xv6-labs-2020-thread'
ph_safe: OK (7.2s)
== Test ph_fast == make[1]: Entering directory '/home/wyb/xv6-labs-2020-thread'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/wyb/xv6-labs-2020-thread'
ph_fast: OK (17.9s)
== Test barrier == make[1]: Entering directory '/home/wyb/xv6-labs-2020-thread'
make[1]: 'barrier' is up to date.
make[1]: Leaving directory '/home/wyb/xv6-labs-2020-thread'
barrier: OK (2.5s)
== Test time ==
time: OK
Score: 60/60
```

五、思考与总结

通过本次实验，我深入掌握了多线程编程的核心机制和实现原理。用户级线程的上下文切换机制揭示了线程调度的基础原理，通过保存和恢复寄存器状态实现执行现场的切换，这一过程需要精心设计栈结构和函数调用约定。多线程环境下的数据同步挑战通过分桶锁方案得到解决，这种细粒度锁设计在保持数据一致性的同时最大化了并行性能，体现了锁粒度对并发性能的重要影响。屏障同步的实现展示了条件变量和互斥锁的强大组合，通过标志位管理和线程本地存储技术解决了多轮同步中的状态干扰问题。

实验过程中的主要难点在于理解用户级线程的初始执行环境设置，特别是栈指针和返回地址的正确初始化；分桶锁的实现需要确保所有访问路径都得到适当的保护，同时避免死锁情况；屏障同步则需要处理多轮调用之间的状态管理和线程间协调。通过仔细分析每个组件的职责边界和交互协议，这些挑战得以逐一克服。

本次实验不仅巩固了多线程编程的基础知识，更培养了系统级编程和调试能力。从上下文切换的底层机制到高级同步原语的应用，我深入理解了操作系统如何管理并发执行流，以及如何设计并

发数据结构以保证正确性和性能。这些经验为后续学习高级并发模型、分布式系统和性能优化奠定了坚实基础，也增强了解决复杂系统问题的信心和能力。

实验报告8

Xv6 操作系统实验报告：Locks

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 8 - Locks

提交日期：2025年9月6日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在通过重新设计代码以提高多核环境下的并行性，重点解决高锁竞争问题。实验分为两个主要部分：

- Memory Allocator**：重新设计物理内存分配器，将全局空闲列表拆分为每个CPU核心一个空闲列表，减少锁竞争，并在需要时实现核心间的内存“窃取”机制。
- Buffer Cache**：优化磁盘块缓存机制，将全局缓存锁替换为基于哈希桶的细粒度锁，使用时间戳实现LRU替换算法，提高并发访问性能。

通过本实验，我们将掌握多核环境下锁机制的设计与优化，理解如何通过数据结构和锁策略的改进减少锁竞争，提升系统整体性能。

二、实验环境与准备

实验环境基于 xv6 的 `lock` 分支进行开发。

环境搭建步骤

- 切换到 `lock` 分支：

```
git fetch  
git checkout lock  
make clean
```

- 编译并启动 xv6：

```
make qemu
```

3. 开发流程：

- 修改内存分配器文件 `kernel/kalloc.c`。
- 修改缓冲区缓存文件 `kernel/bio.c`。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. Memory Allocator (moderate)

实验过程

修改数据结构为每CPU空闲列表

将原来的全局空闲列表改为每个CPU核心一个空闲列表，每个列表有自己的锁：

```
// kernel/kalloc.c
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU];

void kinit() {
    for (int i = 0; i < NCPU; i++) {
        char name[9] = {0};
        sprintf(name, 8, "kmem-%d", i);
        initlock(&kmem[i].lock, name);
    }
    freerange(end, (void*)PHYSTOP);
}
```

实现内存释放函数

在释放内存时，需要关闭中断以确保当前CPU核心不会变化，然后将内存页添加到当前CPU的空闲列表中：

```
// kernel/kalloc.c
void kfree(void *pa) {
```

```

struct run *r;

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

push_off();
int cpu = cpuid();
memset(pa, 1, PGSIZE);
r = (struct run*)pa;
acquire(&kmem[cpu].lock);
r->next = kmem[cpu].freelist;
kmem[cpu].freelist = r;
release(&kmem[cpu].lock);
pop_off();
}

```

实现内存分配函数

在分配内存时，首先尝试从当前CPU的空闲列表中获取内存页。如果当前CPU的空闲列表为空，则从其他CPU的空闲列表中“窃取”内存页：

```

// kernel/kalloc.c
void* kalloc(void) {
    struct run *r;
    push_off();
    int cpu = cpuid();
    acquire(&kmem[cpu].lock);
    r = kmem[cpu].freelist;
    if (r) {
        kmem[cpu].freelist = r->next;
    }
    release(&kmem[cpu].lock);

    if (r == 0) {
        r = ksteal(cpu);
    }
    if(r)
        memset((char*)r, 5, PGSIZE);
    pop_off();
    return (void*)r;
}

```

实现内存窃取函数

当当前CPU的空闲列表为空时，从其他CPU的空闲列表中窃取内存页。采用轮询方式从下一个

CPU开始尝试窃取：

```
// kernel/kalloc.c
void* ksteal(int cpu) {
    struct run *r;
    for (int i = 1; i < NCPU; i++) {
        int next_cpu = (cpu + i) % NCPU;
        acquire(&kmem[next_cpu].lock);
        r = kmem[next_cpu].freelist;
        if (r) {
            kmem[next_cpu].freelist = r->next;
        }
        release(&kmem[next_cpu].lock);
        if (r) {
            break;
        }
    }
    return r;
}
```

原理分析

内存分配器的优化核心在于减少多核环境下的锁竞争。原始设计中，所有CPU核心共享一个全局空闲列表和一个全局锁，这导致在高并发场景下，多个核心频繁竞争同一把锁，严重降低了系统性能。

通过将全局空闲列表拆分为每个CPU核心一个空闲列表，每个列表有自己的锁，我们实现了分配和释放操作的并行化。每个CPU核心主要操作自己的空闲列表，只有在自己的列表为空时才需要访问其他核心的列表。这种设计显著减少了锁竞争，提高了系统整体性能。

内存窃取机制确保了系统的公平性和效率。当某个CPU核心的空闲列表为空时，它会以轮询方式尝试从其他核心的列表中窃取内存页。这种设计避免了某些核心过度占用内存资源而其他核心饥饿的情况。虽然一次只窃取一页可能不是最高效的方式，但对于本实验的目标来说已经足够。

在实现过程中，需要注意中断的控制。在确定当前CPU核心编号和操作空闲列表时，需要暂时关闭中断，防止进程在执行过程中被调度到其他核心，导致数据不一致。

测试结果

修改后的内存分配器通过了 `kallocstress` 和 `usertests sbrkmuch` 测试，锁竞争显著减少，性能得到提升。

2. Buffer Cache (hard)

实验过程

修改数据结构为哈希桶

将原来的全局缓冲区缓存改为基于哈希桶的结构，每个桶有自己的锁和缓冲区数组：

```
// kernel/bio.c
#define BUCKETSIZE 13
#define BUFFERSIZE 5

extern uint ticks;

struct {
    struct spinlock lock;
    struct buf buf[BUFFERSIZE];
} bcachebucket[BUCKETSIZE];

int hash(uint blockno) {
    return blockno % BUCKETSIZE;
}
```

初始化哈希桶锁

在初始化函数中为每个哈希桶初始化锁：

```
// kernel/bio.c
void binit(void) {
    for (int i = 0; i < BUCKETSIZE; i++) {
        initlock(&bcachebucket[i].lock, "bcachebucket");
        for (int j = 0; j < BUFFERSIZE; j++) {
            initsleeplock(&bcachebucket[i].buf[j].lock, "buffer");
        }
    }
}
```

实现缓冲区获取函数

在获取缓冲区时，首先计算哈希桶编号，然后在对应的桶中查找缓冲区。如果找不到，则在当前桶中寻找最近最少使用的缓冲区进行替换：

```
// kernel/bio.c
static struct buf* bget(uint dev, uint blockno) {
    struct buf *b;
    int bucket = hash(blockno);
```

```

acquire(&bcachebucket[bucket].lock);

for (int i = 0; i < BUFFERSIZE; i++) {
    b = &bcachebucket[bucket].buf[i];
    if (b->dev == dev && b->blockno == blockno) {
        b->refcnt++;
        b->lastuse = ticks;
        release(&bcachebucket[bucket].lock);
        acquiresleep(&b->lock);
        return b;
    }
}

uint least = 0xffffffff;
int least_idx = -1;
for (int i = 0; i < BUFFERSIZE; i++) {
    b = &bcachebucket[bucket].buf[i];
    if(b->refcnt == 0 && b->lastuse < least) {
        least = b->lastuse;
        least_idx = i;
    }
}

if (least_idx == -1) {
    panic("bget: no unused buffer for recycle");
}

b = &bcachebucket[bucket].buf[least_idx];
b->dev = dev;
b->blockno = blockno;
b->lastuse = ticks;
b->valid = 0;
b->refcnt = 1;
release(&bcachebucket[bucket].lock);
acquiresleep(&b->lock);
return b;
}

```

实现缓冲区释放函数

在释放缓冲区时，只需要减少引用计数，不需要移动缓冲区：

```

// kernel/bio.c
void brelse(struct buf *b) {
    if(!holdingsleep(&b->lock))
        panic("brelse");
}

```

```
int bucket = hash(b->blockno);
acquire(&bcachebucket[bucket].lock);
b->refcnt--;
release(&bcachebucket[bucket].lock);
releasesleep(&b->lock);
}
```

修改缓冲区引脚函数

修改缓冲区引脚和取消引脚函数，使用哈希桶锁：

```
// kernel/bio.c
void bpin(struct buf *b) {
    int bucket = hash(b->blockno);
    acquire(&bcachebucket[bucket].lock);
    b->refcnt++;
    release(&bcachebucket[bucket].lock);
}

void bunpin(struct buf *b) {
    int bucket = hash(b->blockno);
    acquire(&bcachebucket[bucket].lock);
    b->refcnt--;
    release(&bcachebucket[bucket].lock);
}
```

原理分析

缓冲区缓存的优化目标是减少多进程访问磁盘块缓存时的锁竞争。原始设计中，所有缓冲区由一个全局锁保护，这导致在高并发文件系统访问场景下，多个进程频繁竞争同一把锁，降低了系统性能。

通过将缓冲区缓存组织为哈希桶结构，每个桶有自己的锁，我们实现了不同块号缓冲区的并行访问。只有哈希到同一个桶的缓冲区访问才会竞争同一把锁，这显著减少了锁竞争。

LRU（最近最少使用）替换算法的实现基于时间戳机制。每个缓冲区都有一个最后使用时间戳，当需要替换缓冲区时，选择同一桶中最近最少使用的缓冲区。这种设计避免了维护全局LRU列表的开销，简化了实现。

在实现过程中，需要注意以下几点：

1. 哈希函数的选择：使用质数大小的哈希桶可以减少哈希冲突的概率。
2. 原子操作：查找缓冲区和分配新缓冲区的操作必须是原子的，需要持有桶锁。

3. 死锁避免：在持有多个锁时，需要确保锁的获取顺序一致，避免死锁。

虽然本实现没有实现跨桶的缓冲区窃取机制，但在测试中表现良好，满足了实验要求。

测试结果

修改后的缓冲区缓存通过了 `bcachetest` 和 `usertests` 测试，锁竞争显著减少，性能得到提升。

四、实验结果

```
-- Test running kalloc test ==
$ make qemu-gdb
(42.2s)
== Test kalloc test: test1 ==
    kalloc test: test1: OK
== Test kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch ==
$ make qemu-gdb
kalloc test: sbrkmuch: OK (3.9s)
== Test running bcachetest ==
$ make qemu-gdb
(3.4s)
== Test bcachetest: test0 ==
    bcachetest: test0: OK
== Test bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (51.0s)
== Test time ==
time: OK
Score: 70/70
```

五、思考与总结

通过本次实验，我深入掌握了多核环境下锁机制的设计与优化技术。内存分配器的优化展示了如何通过数据分区和细粒度锁减少锁竞争，每个CPU核心独立管理自己的空闲列表，只有在必要时

才进行跨核心内存窃取，这种设计在保持数据一致性的同时显著提高了并行性能。缓冲区缓存的优化则演示了如何通过哈希分桶和时间戳机制实现高效的并发访问，基于哈希桶的锁设计允许不同桶的缓冲区并行访问，而时间戳机制实现了简化的LRU替换算法，避免了全局锁的竞争。

实验过程中的主要难点在于理解多核环境下的数据同步需求，以及如何设计锁机制来平衡性能与正确性。内存分配器的实现需要确保在窃取内存时的原子操作，避免数据竞争。缓冲区缓存的实现需要处理哈希冲突和缓冲区替换的边界情况，确保系统的稳定性和性能。

本次实验不仅巩固了锁机制和并发编程的基础知识，更培养了系统级性能优化的能力。从全局锁到细粒度锁的转变，我深入理解了锁粒度对并发性能的重要影响，以及如何通过数据结构和算法设计减少锁竞争。这些经验为后续学习高级并发模型、分布式系统和性能优化奠定了坚实基础，也增强了解决复杂系统问题的信心和能力。

实验报告9

Xv6 操作系统实验报告：File System

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 9 - File System

提交日期：2025年9月6日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在深入理解 xv6 文件系统的核心机制，包括 inode 结构、数据块映射、以及符号链接的实现。实验分为两个主要部分：

- Large files**：通过修改 inode 的数据块映射机制，将单层间接块扩展为双层间接块，显著提升文件的最大容量。
- Symbolic links**：实现符号链接系统调用，支持通过路径名引用文件，并处理链接递归和循环问题。

通过本实验，我们将掌握文件系统中 inode 的结构与数据块管理、符号链接的实现原理，以及系统调用与文件系统操作的交互机制。

二、实验环境与准备

实验环境基于 xv6 的 `fs` 分支进行开发。

环境搭建步骤

- 切换到 `fs` 分支：

```
git fetch  
git checkout fs  
make clean
```

- 编译并启动 xv6：

```
make qemu
```

3. 开发流程：

- 修改文件系统相关代码（如 `kernel/fs.h`、`kernel/fs.c`、`kernel/sysfile.c`）。
- 添加符号链接系统调用及相关支持。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. Large files (moderate)

实验过程

修改 inode 结构定义

首先修改 `kernel/fs.h` 中的宏定义和结构体，减少直接块数量，增加双层间接块支持：

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NININDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NININDIRECT)

struct dinode {
    short type;
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT + 2]; // 11 direct + 1 indirect + 1 double-indirect
};
```

同时修改 `kernel/file.h` 中的内存 inode 结构：

```
struct inode {
    uint dev;
    uint inum;
    int ref;
    struct sleeplock lock;
    int valid;
```

```
    short type;
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT + 2];
};


```

实现双层间接块映射

在 `kernel/fs.c` 的 `bmap` 函数中添加双层间接块的处理逻辑：

```
static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a, *b;
    struct buf *inbp, *ininbp;

    if (bn < NDIRECT) {
        // 直接块处理
        if ((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = alloc(ip->dev);
        return addr;
    }
    bn -= NDIRECT;

    if (bn < NINDIRECT) {
        // 单层间接块处理
        if ((addr = ip->addrs[NDIRECT]) == 0)
            ip->addrs[NDIRECT] = addr = alloc(ip->dev);
        inbp = bread(ip->dev, addr);
        a = (uint*)inbp->data;
        if ((addr = a[bn]) == 0) {
            a[bn] = addr = alloc(ip->dev);
            log_write(inbp);
        }
        brelse(inbp);
        return addr;
    }
    bn -= NINDIRECT;

    if (bn < NININDIRECT) {
        // 双层间接块处理
        if ((addr = ip->addrs[NDIRECT + 1]) == 0)
            ip->addrs[NDIRECT + 1] = addr = alloc(ip->dev);
        inbp = bread(ip->dev, addr);
    }
}
```

```

a = (uint*)inbp->data;
if ((addr = a[bn / NINDIRECT]) == 0) {
    a[bn / NINDIRECT] = addr = balloc(ip->dev);
    log_write(inbp);
}
brelse(inbp);

ininbp = bread(ip->dev, addr);
b = (uint*)ininbp->data;
if ((addr = b[bn % NINDIRECT]) == 0) {
    b[bn % NINDIRECT] = addr = balloc(ip->dev);
    log_write(ininbp);
}
brelse(ininbp);
return addr;
}

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

```

修改文件截断函数

在 `itrunc` 函数中添加释放双层间接块的逻辑：

```

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

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

    // 释放单层间接块
    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;
}

// 释放双层间接块
if (ip->addrs[NDIRECT + 1]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) {
            inbp = bread(ip->dev, a[j]);
            b = (uint*)inbp->data;
            for (k = 0; k < NINDIRECT; k++) {
                if (b[k])
                    bfree(ip->dev, b[k]);
            }
            brelse(inbp);
            bfree(ip->dev, a[j]);
        }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
}

ip->size = 0;
iupdate(ip);
}

```

原理分析

文件系统的 inode 是管理文件数据块的核心数据结构。原始 xv6 的 inode 采用 12 个直接块和 1 个单层间接块的设计，最多支持 268 个数据块。通过减少一个直接块，增加一个双层间接块，我们显著扩展了文件的最大容量。

双层间接块通过两级索引实现：第一级索引指向一个包含 256 个第二级索引块的块，每个第二级索引块又指向 256 个数据块。这样，一个双层间接块可以管理 $256 \times 256 = 65,536$ 个数据块，加上原有的 11 个直接块和 256 个单层间接块，总容量达到 65,803 个数据块。

在 `bmap` 函数中，我们通过逻辑块号的三段式处理实现映射：直接块处理前 11 个块，单层间接块处理接下来的 256 个块，双层间接块处理剩余的块。每级映射都按需分配块，确保空间的高效使用。

`itrunc` 函数负责释放文件的所有数据块。对于双层间接块，需要遍历两级索引结构，逐级释放所有数据块和索引块，避免内存泄漏。

问题分析

初始实现可能因块号计算错误导致 panic 或数据损坏。通过仔细计算偏移量并验证各级索引的正确性，确保映射逻辑的准确性。测试时使用 `bigfile` 创建最大文件，验证其正确写入 65,803 个块。

2. Symbolic links (moderate)

实验过程

添加系统调用和符号链接类型

首先在 `kernel/fcntl.h` 中添加 `O_NOFOLLOW` 标志：

```
#define O_NOFOLLOW 0x800
```

在 `kernel/stat.h` 中添加符号链接文件类型：

```
#define T_SYMLINK 4
```

实现符号链接系统调用

在 `kernel/sysfile.c` 中实现 `sys_symlink`：

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

    if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
        return -1;

    begin_op();

    // 检查路径是否已存在
    if ((ip = namei(path)) != 0) {
        iunlockput(ip);
        end_op();
        return -1;
    }
}
```

```

// 创建符号链接 inode
ip = create(path, T_SYMLINK, 0, 0);
if (ip == 0) {
    end_op();
    return -1;
}

// 将目标路径写入 inode 数据块
if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip);
    end_op();
    return -1;
}

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

```

修改打开系统调用以支持符号链接

在 `sys_open` 中添加符号链接处理逻辑：

```

uint64
sys_open(void)
{
    // 省略参数解析和创建处理

    if (omode & O_CREATE) {
        // 原有创建逻辑
    } else {
        // 处理符号链接
        int depth = 0;
        while (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
            char target[MAXPATH];
            if (readi(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
                iunlockput(ip);
                end_op();
                return -1;
            }
            iunlockput(ip);
            if ((ip = namei(target)) == 0) {
                end_op();
                return -1;
            }
            ilock(ip);
        }
    }
}

```

```
    if (++depth > 10) {
        iunlockput(ip);
        end_op();
        return -1;
    }
}

// 后续处理
}
```

原理分析

符号链接是一种特殊类型的文件，其内容不是数据而是另一个文件的路径名。当打开符号链接时，系统会自动解析路径指向的实际文件。

实现符号链接的关键在于：

1. 创建特殊的符号链接 inode 类型 (T_SYMLINK) 。
2. 将目标路径存储在 inode 的数据块中。
3. 在打开文件时递归解析符号链接，直到找到实际文件或检测到循环。

在 `sys_symlink` 中，我们创建符号链接 inode 并将目标路径写入其数据块。`sys_open` 中添加的解析逻辑会递归跟随符号链接，同时通过深度计数器防止无限循环。

符号链接与硬链接的主要区别在于：硬链接直接指向 inode，而符号链接通过路径名间接引用文件。这使得符号链接可以跨文件系统使用，但也增加了解析开销。

边界处理

- 处理不存在的目标路径时返回错误。
- 限制递归深度为 10 层，防止循环链接导致的无限递归。
- 使用 `O_NOFOLLOW` 标志打开符号链接本身而非目标文件。

四、实验结果

```
make[1]: Leaving directory '/home/wyb/xv6-labs-2020-fs'
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (89.6s)
== Test running symlinktest ==
$ make qemu-gdb
(0.4s)
== Test symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (136.6s)
== Test time ==
time: OK
Score: 100/100
```

五、思考与总结

通过本次实验，我深入掌握了 xv6 文件系统的核心机制和实现原理。Large files 部分通过修改 inode 结构和使用双层间接块，显著提升了文件的最大容量，理解了数据块映射的多级索引原理。Symbolic links 部分实现了符号链接系统调用，掌握了路径解析、递归处理和循环检测的技术。

实验过程中的主要难点在于理解 inode 结构与数据块映射的细节，以及符号链接递归解析的正确性保证。通过仔细分析代码逻辑和多次测试验证，确保了实现的正确性和鲁棒性。

本次实验不仅巩固了文件系统的基础知识，更培养了系统级编程和调试能力。从 inode 管理到系统调用实现，我深入理解了操作系统如何管理文件和路径，以及如何设计高效可靠的文件系统操作。这些经验为后续学习高级文件系统特性、分布式文件系统等奠定了坚实基础，也增强了解决复杂系统问题的信心和能力。

实验报告10

Xv6 操作系统实验报告：mmap

课程名称：6.828 操作系统工程（MIT 6.S081）

实验名称：Lab 10 - mmap

提交日期：2025年9月6日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在实现 UNIX 系统中的 `mmap` 和 `munmap` 系统调用，允许进程将文件映射到其地址空间，从而通过内存读写操作直接访问文件内容。实验涉及虚拟内存管理、缺页中断处理、文件系统操作和进程管理等多个核心模块，要求实现懒加载机制以提高大文件映射的效率，并支持共享映射和私有映射两种模式。

实验分为以下几个主要部分：

- 数据结构设计**：为每个进程维护一个虚拟内存区域（VMA）表，记录映射的文件、地址范围、权限和标志等信息。
- mmap 系统调用**：实现文件映射功能，在进程地址空间中预留区域但不立即分配物理页。
- 缺页中断处理**：在访问映射区域时触发缺页中断，动态加载文件内容到物理内存。
- munmap 系统调用**：解除映射关系，释放物理页，并根据需要将修改写回文件。
- 进程管理扩展**：在 `exit` 和 `fork` 中正确处理映射区域的释放和继承。

通过本实验，我们将深入理解内存映射文件的实现原理，掌握懒加载技术的应用，以及多模块协同工作的系统编程技巧。

二、实验环境与准备

实验环境基于 xv6 的 `mmap` 分支进行开发。

环境搭建步骤

- 切换到 `mmap` 分支：

```
git fetch  
git checkout mmap  
make clean
```

2. 编译并启动 xv6:

```
make qemu
```

3. 开发流程:

- 修改 `kernel/proc.h` 添加 VMA 结构定义。
- 实现 `kernel/sysfile.c` 中的 `sys_mmap` 和 `sys_munmap`。
- 修改 `kernel/trap.c` 处理缺页中断。
- 修改 `kernel/proc.c` 处理进程退出和复制时的映射清理和继承。
- 使用 `make grade` 进行测试，确保所有功能正确。

三、实验过程与原理分析

1. 数据结构设计

实验过程

在 `kernel/proc.h` 中定义 VMA (Virtual Memory Area) 结构体和进程的 VMA 数组：

```
// Number of virtual memory area per process  
#define NVMA 16  
  
// Virtual Memory Area  
struct vma {  
    int valid;           // 是否有效  
    uint64 addr;        // 映射的起始虚拟地址  
    int length;         // 映射长度  
    int prot;           // 权限保护位 (PROT_READ, PROT_WRITE, PROT_EXEC)  
    int flags;          // 映射标志 (MAP_SHARED, MAP_PRIVATE)  
    int fd;              // 文件描述符  
    int offset;         // 文件偏移量  
    struct file* f;     // 文件结构指针  
};
```

```
// Per-process state
struct proc {
    // ... 其他字段 ...
    struct vma vmas[NVMA]; // 虚拟内存区域数组
};
```

原理分析

VMA 结构用于记录进程地址空间中每个映射区域的元信息。每个进程最多支持 16 个映射区域，可通过修改 NVMA 调整上限。`valid` 字段表示该 VMA 是否有效，避免无效区域的检查开销。`addr` 和 `length` 定义了映射的虚拟地址范围，`prot` 和 `flags` 控制访问权限和共享行为，`fd`、`offset` 和 `f` 关联映射的文件及其位置。

2. mmap 系统调用实现

实验过程

在 `kernel/sysfile.c` 中实现 `sys_mmap`：

```
uint64 sys_mmap(void) {
    uint64 failure = (uint64)((char *) -1);
    struct proc* p = myproc();
    uint64 addr;
    int length, prot, flags, fd, offset;
    struct file* f;

    // 解析参数
    if (argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
        argint(3, &flags) < 0 || argfd(4, &fd, &f) < 0 || argint(5, &offset) < 0)
        return failure;

    // 安全检查
    length = PGROUNDUP(length);
    if (MAXVA - length < p->sz)
        return failure;
    if (!f->readable && (prot & PROT_READ))
        return failure;
    if (!f->writable && (prot & PROT_WRITE) && (flags == MAP_SHARED))
        return failure;

    // 查找空闲 VMA 槽位并填充
    for (int i = 0; i < NVMA; i++) {
        struct vma* vma = &p->vmas[i];
        if (vma->valid == 0) {
```

```

        vma->valid = 1;
        vma->addr = p->sz;
        p->sz += length; // 扩展进程虚拟地址空间
        vma->length = length;
        vma->prot = prot;
        vma->flags = flags;
        vma->fd = fd;
        vma->f = f;
        filedup(f); // 增加文件引用计数
        vma->offset = offset;
        return vma->addr;
    }
}

return failure; // 无可用 VMA 槽位
}

```

原理分析

`mmap` 系统调用将文件映射到进程地址空间，采用懒加载策略：仅预留虚拟地址范围，不立即分配物理页或读取文件内容。参数检查确保映射请求合法，包括地址空间不足、文件权限不匹配等情况。找到空闲 VMA 槽位后，记录映射信息并扩展 `p->sz`，返回映射区域的起始地址。通过 `filedup` 增加文件引用计数，确保文件在映射期间不会被关闭。

3. 缺页中断处理

实验过程

在 `kernel/trap.c` 的 `usertrap` 函数中处理缺页中断：

```

void usertrap(void) {
    // ... 其他中断处理 ...

} else if (r_scause() == 13 || r_scause() == 15) { // 读或写造成的缺页中断
    uint64 va = r_stval();
    struct proc* p = myproc();
    if (va > MAXVA || va >= p->sz) {
        p->killed = 1;
    } else {
        int found = 0;
        for (int i = 0; i < NVMA; i++) {
            struct vma* vma = &p->vmas[i];
            if (vma->valid && va >= vma->addr && va < vma->addr + vma->length) {
                // 计算缺页地址对应的文件偏移
            }
        }
    }
}

```

```

va = PGROUNDDOWN(va);
uint64 pa = (uint64)kalloc();
if (pa == 0) break;
memset((void *)pa, 0, PGSIZE);

// 读取文件内容到物理页
ilock(vma->f->ip);
if (readi(vma->f->ip, 0, pa, vma->offset + (va - vma->addr), PGSIZE) <
0) {
    iunlock(vma->f->ip);
    kfree((void*)pa);
    break;
}
iunlock(vma->f->ip);

// 设置页表权限
int perm = PTE_U;
if (vma->prot & PROT_READ) perm |= PTE_R;
if (vma->prot & PROT_WRITE) perm |= PTE_W;
if (vma->prot & PROT_EXEC) perm |= PTE_X;

// 映射到用户页表
if (mappages(p->pagetable, va, PGSIZE, pa, perm) < 0) {
    kfree((void*)pa);
    break;
}
found = 1;
break;
}
}
if (!found) p->killed = 1;
}
}
}

```

原理分析

当进程访问未加载的映射区域时，触发缺页中断（scause 13 或 15）。中断处理程序首先检查故障地址是否合法，然后在进程的 VMA 表中查找对应的映射区域。找到后，分配物理页，从文件读取相应内容（使用 `readi`），并根据 VMA 的权限设置页表项。懒加载机制避免了一次性加载大文件的开销，支持映射超过物理内存的文件。

4. munmap 系统调用实现

实验过程

在 `kernel/sysfile.c` 中实现 `sys_munmap` :

```
uint64 sys_munmap(void) {
    uint64 addr;
    int length;
    if (argaddr(0, &addr) < 0 || argint(1, &length) < 0)
        return -1;
    struct proc *p = myproc();
    struct vma* vma = 0;
    int idx = -1;

    // 查找对应的 VMA
    for (int i = 0; i < NVMA; i++) {
        if (p->vmas[i].valid && addr >= p->vmas[i].addr &&
            addr < p->vmas[i].addr + p->vmas[i].length) {
            idx = i;
            vma = &p->vmas[i];
            break;
        }
    }
    if (idx == -1) return -1;

    addr = PGROUNDDOWN(addr);
    length = PGROUNDDUP(length);

    // 共享映射需写回文件
    if (vma->flags & MAP_SHARED) {
        if (filewrite(vma->f, addr, length) < 0) {
            return -1;
        }
    }

    // 解除映射并释放物理页
    uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

    // 更新 VMA
    if (addr == vma->addr && length == vma->length) {
        // 完全解除映射
        fileclose(vma->f);
        vma->valid = 0;
    } else if (addr == vma->addr) {
        // 解除头部映射
        vma->addr += length;
        vma->length -= length;
        vma->offset += length;
    }
}
```

```

} else if (addr + length == vma->addr + vma->length) {
    // 解除尾部映射
    vma->length -= length;
} else {
    panic("munmap: middle unmap not supported");
}
return 0;
}

```

原理分析

`munmap` 用于解除部分或全部映射区域。共享映射 (MAP_SHARED) 的修改需写回文件 (通过 `filewrite`)。调用 `uv munmap` 释放物理页并清除页表项。根据解除映射的区域更新 VMA：完全解除时释放文件引用并标记 VMA 无效；部分解除时调整 VMA 的起始地址、长度和文件偏移。实验假设解除操作仅针对映射区域的头部或尾部，不支持中间挖洞。

5. 进程退出和复制的映射处理

实验过程

在 `kernel/proc.c` 中修改 `exit` 和 `fork` 函数：

```

// exit: 释放所有映射区域
void exit(int status) {
    // ... 其他清理代码 ...
    for (int i = 0; i < NVMA; i++) {
        if (p->vmas[i].valid) {
            if (p->vmas[i].fFlags & MAP_SHARED) {
                filewrite(p->vmas[i].f, p->vmas[i].addr, p->vmas[i].length);
            }
            fileclose(p->vmas[i].f);
            uv munmap(p->pagetable, p->vmas[i].addr, p->vmas[i].length / PGSIZE, 1);
            p->vmas[i].valid = 0;
        }
    }
    // ...
}

// fork: 复制父进程的 VMA 数组
int fork(void) {
    // ... 其他复制代码 ...
    for (int i = 0; i < NVMA; i++) {
        if (p->vmas[i].valid) {
            memmove(&np->vmas[i], &p->vmas[i], sizeof(struct vma));
        }
    }
}

```

```
    filedup(p->vmas[i].f); // 增加文件引用计数
} else {
    np->vmas[i].valid = 0;
}
}
// ...
}
```

原理分析

进程退出时需释放所有映射区域：共享映射的修改写回文件，关闭文件引用，解除页表映射并标记 VMA 无效。fork 时子进程继承父进程的 VMA 数组，通过 `memmove` 复制结构体，并调用 `filedup` 增加文件引用计数。子进程的映射区域独立于父进程，缺页时分配新的物理页，不共享物理内存（可选挑战项目支持共享）。

四、实验结果

```
== Test running mmaptest ==
$ make qemu-gdb
(2.0s)
== 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 ==
$ make qemu-gdb
usertests: OK (53.5s)
== Test time ==
time: OK
Score: 140/140
```

五、思考与总结

通过本次实验，我深入理解了内存映射文件的实现机制和懒加载技术的优势。mmap 系统调用通过将文件映射到进程地址空间，提供了高效的文件访问方式，避免了频繁的 read/write 系统调用开销。实验涉及多个核心模块的协作，包括虚拟内存管理、文件系统、缺页中断处理和进程管理，增强了系统编程和调试能力。

实验过程中的主要难点在于缺页中断的处理和映射区域的动态管理。缺页中断需准确定位故障地址对应的 VMA，正确读取文件内容并设置页表权限。munmap 需处理部分解除映射的边界情况，并确保共享映射的修改正确写回文件。通过仔细设计数据结构和算法，这些挑战得以解决。

本实验还展示了懒加载技术的强大之处：支持映射超过物理内存的大文件，提高了内存利用率和响应速度。未来的优化方向包括共享物理页减少内存占用、利用脏位避免不必要的写回、以及支持中间挖洞的解除映射操作。

总之，本次实验不仅巩固了操作系统核心概念，还提升了解决复杂系统问题的能力，为后续学习和研究奠定了坚实基础。

实验报告11

Xv6 操作系统实验报告：Networking

课程名称：6.828 操作系统工程 (MIT 6.S081)

实验名称：Lab 11 - Networking

提交日期：2025年9月6日

作者：王奕博

学号：2353945

一、实验概述

本实验旨在深入理解操作系统中的设备驱动编程，特别是网络接口卡（NIC）驱动程序的实现。实验基于 xv6 操作系统，通过为 E1000 网卡编写驱动程序，掌握 DMA（直接内存访问）机制、描述符环（Descriptor Ring）的管理，以及网络数据包的发送和接收流程。实验分为两个主要部分：

- E1000 数据包发送**：实现 `e1000_transmit()` 函数，将网络数据包放入发送环中，由网卡通过 DMA 机制发送。
- E1000 数据包接收**：实现 `e1000_recv()` 函数，从接收环中取出已到达的数据包，并交付给上层网络协议栈处理。

通过本实验，我们将掌握网络设备驱动的基本工作原理，包括描述符环的管理、DMA 的使用、硬件寄存器的访问，以及驱动与上层网络栈的交互机制。

二、实验环境与准备

实验环境基于 xv6 的 `net` 分支进行开发。

环境搭建步骤

- 切换到 `net` 分支：

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

- 编译并启动 xv6：

```
make qemu
```

3. 开发流程：

- 修改 E1000 驱动文件 `kernel/e1000.c`。
- 使用 `make server` 在主机上启动服务器。
- 在 xv6 中运行 `nettests` 进行测试。
- 使用 `make grade` 验证所有功能正确。

三、实验过程与原理分析

1. E1000 数据包发送 (moderate)

实验过程

实现数据包发送函数

在 `kernel/e1000.c` 中实现 `e1000_transmit()` 函数，其功能是将一个网络数据包 (mbuf) 放入发送环中，由网卡进行发送：

```
int
e1000_transmit(struct mbuf *m)
{
    acquire(&e1000_lock);
    // 获取当前发送环的尾部索引
    uint32 idx = regs[E1000_TDT];

    // 检查描述符状态：如果上一个包还未发送完成，则返回错误
    if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
        release(&e1000_lock);
        return -1;
    }

    // 释放之前已发送完成的 mbuf (如果存在)
    if (tx_mbufs[idx] != NULL) {
        mbuffree(tx_mbufs[idx]);
        tx_mbufs[idx] = NULL;
    }

    // 设置当前描述符的地址和长度
    tx_ring[idx].addr = (uint64)m->head;
```

```

tx_ring[idx].length = m->len;

// 设置命令标志: EOP (End of Packet) 和 RS (Report Status)
tx_ring[idx].cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;

// 保存 mbuf 指针供后续释放
tx_mbufs[idx] = m;

// 更新尾部索引, 环大小为 TX_RING_SIZE
regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;

release(&e1000_lock);
return 0;
}

```

原理分析

数据包发送是网络驱动中最基础的功能之一，其核心在于管理发送描述符环（TX Ring）。每个描述符包含一个数据包在内存中的地址、长度、命令和状态信息。网卡通过 DMA 机制直接从内存中读取数据包内容，并通过物理网络接口发送出去。

发送过程始于上层网络栈调用 `e1000_transmit()` 函数，传入一个待发送的 `mbuf` 结构。驱动程序首先获取当前发送环的尾部索引（TDT），该索引表示网卡期望的下一个可用描述符位置。如果该描述符的状态位 `E1000_TXD_STAT_DD` 未被设置，说明网卡尚未完成前一个数据包的发送，此时驱动程序返回错误，通知上层稍后重试。

如果描述符可用，驱动程序首先释放之前占用的 `mbuf`（如果有），然后将新的 `mbuf` 信息填充到描述符中：数据包地址设置为 `m->head`，长度设置为 `m->len`，命令字段包含 `EOP`（表示这是一个完整的数据包）和 `RS`（要求网卡在发送完成后回写状态）。最后，驱动程序更新 TDT 寄存器，通知网卡有新的数据包待发送。

需要注意的是，驱动程序必须保存 `mbuf` 的指针，以便在数据包发送完成后释放其内存。网卡在发送完成后会设置描述符的 `DD` 状态位，驱动程序在下次使用该描述符时检查这一状态并释放对应的 `mbuf`。

2. E1000 数据包接收 (moderate)

实验过程

实现数据包接收函数

在 `kernel/e1000.c` 中实现 `e1000_recv()` 函数，其功能是从接收环中取出已到达的数据包，并交付给上层网络协议栈：

```

static void
e1000_recv(void)
{
    while (1) {
        // 获取当前接收环的尾部索引，并计算下一个待检查的描述符索引
        uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;

        // 检查描述符状态：如果数据包未就绪，则退出循环
        if ((rx_ring[idx].status & E1000_RXD_STAT_DD) == 0) {
            break;
        }

        // 更新 mbuf 的长度为实际接收到的数据长度
        rx_mbufs[idx]->len = rx_ring[idx].length;

        // 将数据包交付给上层网络栈
        net_rx(rx_mbufs[idx]);

        // 分配新的 mbuf 并重新初始化描述符
        rx_mbufs[idx] = mbufalloc(0);
        rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;
        rx_ring[idx].status = 0; // 清除状态位

        // 更新尾部索引，通知网卡当前处理位置
        regs[E1000_RDT] = idx;
    }
}

```

原理分析

数据包接收是网络驱动的另一核心功能，其关键在于管理接收描述符环（RX Ring）。每个接收描述符包含一个空闲缓冲区的地址，网卡通过 DMA 机制将接收到的数据包直接写入该缓冲区，并更新描述符的状态和长度信息。

接收过程由网卡的中断触发，中断处理函数调用 `e1000_recv()` 函数。驱动程序通过 RDT 寄存器获取当前接收环的尾部索引，并检查下一个描述符的 DD 状态位。如果该位被设置，说明网卡已向该描述符对应的缓冲区写入了一个新数据包。

驱动程序首先更新 mbuf 的长度字段为实际接收到的数据长度，然后调用 `net_rx()` 函数将数据包交付给上层网络协议栈处理。之后，驱动程序必须为该描述符分配一个新的 mbuf 作为接收缓冲区，并重新初始化描述符的地址和状态字段。最后，驱动程序更新 RDT 寄存器，通知网卡该描述符已重新就绪，可用于接收后续数据包。

需要注意的是，接收环是循环使用的，驱动程序必须确保始终有足够的空闲缓冲区供网卡使用。否则，可能会导致数据包丢失或网卡停止工作。

四、实验结果

```
== 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
== Test time ==
time: OK
Score: 100/100
```

所有测试均通过 `make grade`

五、思考与总结

通过本次实验，我深入掌握了网络设备驱动程序的实现原理和工作机制。E1000 网卡的发送和接收功能基于描述符环和 DMA 技术，实现了高效的数据传输。发送过程中，驱动程序通过管理发送描述符环，将数据包交给网卡处理，并在发送完成后释放内存；接收过程中，驱动程序从接收描述符环中取出已到达的数据包，交付给上层网络栈，并重新初始化描述符以供下次使用。

实验过程中的主要难点在于理解描述符环的循环使用机制，以及正确管理 mbuf 内存的分配和释放。通过仔细阅读 E1000 开发手册和 xv6 提供的初始化代码，我逐步理解了硬件寄存器的访问方式、描述符字段的含义，以及驱动与硬件的交互协议。

本次实验不仅巩固了设备驱动编程的基础知识，更培养了底层硬件交互和调试能力。从描述符环的管理到 DMA 机制的应用，我深入理解了操作系统如何与网络设备协同工作，以及如何设计高效的数据传输机制。这些经验为后续学习高性能网络、分布式系统和内核开发奠定了坚实基础，也增强了解决复杂系统问题的信心和能力。