MIT6.828/6.S081 xv6 实验报告

Lab1: Utility

1. Sleep

1) 实验目的

实现xv6的UNIX程序`sleep`：`sleep`应该暂停到用户指定的计时数。一个滴答(tick)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) 实验步骤

1. 在`user`下新建`sleep.c`，注意用`exit`退出：

```
// 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 time\n");
        exit(1);
    }
    else{
        sleep(atoi(argv[1]));
        exit(0);
    }
}
```

2. 将`sleep`程序添加到`Makefile`中的`UPROGS`中：

```
...
UPROGS=\
    ...
    $U/_sleep\
...
```

3) 问题与解决方法

1. 如果用`return 0`替代`exit(0)`，会出现错误：

```
usertrap(): unexpected scause 0x000000000000000d pid=4
            sepc=0x00000000000000f6 stval=0x0000000000003008
```

在操作系统中，一个用户程序的运行是在一个进程（process）的上下文中进行的。在用户程序的上下文中，一个进程可能同时包含多个函数调用，而且不仅仅是 `main` 函数。因此，如果用户程序使用 `return` 来退出，那么只会退出当前的函数调用，而不会退出整个进程，这可能会导致不可预料的结果。

当用户程序调用 `exit` 系统调用时，操作系统会执行一系列必要的清理工作，关闭文件描述符、释放内存等，并通知内核该进程的退出状态。这样可以确保进程的资源得到适当地释放，不会导致资源泄漏。

2. Pingpong

1) 实验目的

编写一个使用UNIX系统调用的程序来在两个进程之间“ping-pong”一个字节，请使用两个管道，每个方向一个。父进程应该向子进程发送一个字节;子进程应该打印“`<pid>: received ping`”，其中`<pid>`是进程ID，并在管道中写入字节发送给父进程，然后退出;父级应该从读取从子进程而来的字节，打印“`<pid>: received pong`”，然后退出。

2) 实验步骤

1. 新建`user/pingpong.c`用`pipe()`来创建两个管道：

```
int parent_to_child[2], child_to_parent[2]; // [0]为读端，[1]为写端
char msg;
if(pipe(parent_to_child) < 0 || pipe(child_to_parent) < 0){
    fprintf(2, "Failed to create pipes\n");
    exit(1);
}
```

2. 用`fork()`创建进程，`pid`为0的是子进程，否则是父进程。每个进程在收发消息前先关闭不用的端，之后关闭用完的端并用`exit()`退出：

```
int pid = fork();
if(pid == 0){
// 子进程
// 关闭不用的端
    close(child_to_parent[0]);
    close(parent_to_child[1]);
// 子进程接收父进程消息
    if(read(parent_to_child[0], &msg, 1) != 1){
        fprintf(2, "Read Error in child process\n");
        exit(1);
    }
    fprintf(1, "%d: received ping\n", getpid());
    // 子进程向父进程写
    if(write(child_to_parent[1], &msg, 1) != 1){
        fprintf(2, "Write Error in child process\n");
        exit(1);
    }
    //关闭用完的端
    close(child_to_parent[1]);
    close(parent_to_child[0]);
    exit(0);
}
```

```
else{
    // 父进程
    msg = 'A';
    // 关闭不用的端
    close(child_to_parent[1]);
    close(parent_to_child[0]);
    // 父进程向子进程写
    if(write(parent_to_child[1], &msg, 1) != 1){
        fprintf(2, "Write Error in parent process\n");
        exit(1);
    }
    // 父进程接收子进程消息
    if(read(child_to_parent[0], &msg, 1) != 1){
        fprintf(2, "Read Error in parent process\n");
        exit(1);
    }
    fprintf(1, "%d: received pong\n", getpid());
    //关闭用完的端
    close(child_to_parent[0]);
    close(parent_to_child[1]);
    exit(0);
}
exit(0);
```

3. 将程序添加到`Makefile`中的`UPROGS`中，与上一个实验相同。

3) 问题与解决方法

1. 管道一般用作单向通信，实际意思是`FIFO`，`linux`构建了一个循环队列，缓冲区设置两个指针，一个读指针，一个写指针，并保证读指针向前移动不能超过写指针，否则唤醒写进程并睡眠，直到读满需要的字节数。同理写指针向前也不能超过读指针，否则唤醒读进程并睡眠，直到写满要求的字节数。

   上述实现中，实际是有顺序的，即父进程将信息发送给子进程，子进程接收后再将其发送回去。如果二者各自发送一条数据，需要调用`wait()`，否则可能会导致两个进程同时输出，输出结果混在一起。

3. Primes

1) 实验目的

使用管道编写`prime sieve`(筛选素数)的并发版本。使用`pipe`和`fork`来设置管道。第一个进程将数字2到35输入管道。对于每个素数，创建一个进程，该进程通过一个管道从其左邻居读取数据，并通过另一个管道向其右邻居写入数据。由于xv6的文件描述符和进程数量有限，因此第一个进程可以在35处停止。

2) 实验步骤

1. 新建`user/primes.c`，根据算法示意图和伪代码不难写出：

```
int main()
{
    int p[2];
    pipe(p);
    int pid = fork();
```

```
    if(pid < 0){
        fprintf(2, "Fork failed\n");
        exit(1);
    }
    else if(pid > 0){
        // 最左侧进程
        close(p[0]);
        fprintf(1, "prime 2\n");
        for(int i = 3; i <= 35; i++){
            if(i % 2 != 0)
                write(p[1], &i, 4);
        }
        close(p[1]);
        wait(0);
    }
    else
        child(p);

    exit(0);
}
```

`child()` 用于递归地创建管道并向右传递：

```
void child(int p[]){
    close(p[1]);
    int prime;
    if(read(p[0], &prime, 4) > 0){
        fprintf(1, "prime %d\n", prime);
        int p1[2]; // 创建管道向右侧进程传递
        pipe(p1);
        if(fork() > 0){
            // 本进程
            close(p1[0]);
            int i;
            while(read(p[0], &i, 4) > 0){
                if(i % prime != 0)
                    write(p1[1], &i, 4);
            }
            close(p1[1]);
            wait(0);
        }
        else{
            // 右侧进程
            close(p[0]);
            child(p1);
        }
    }
}
```

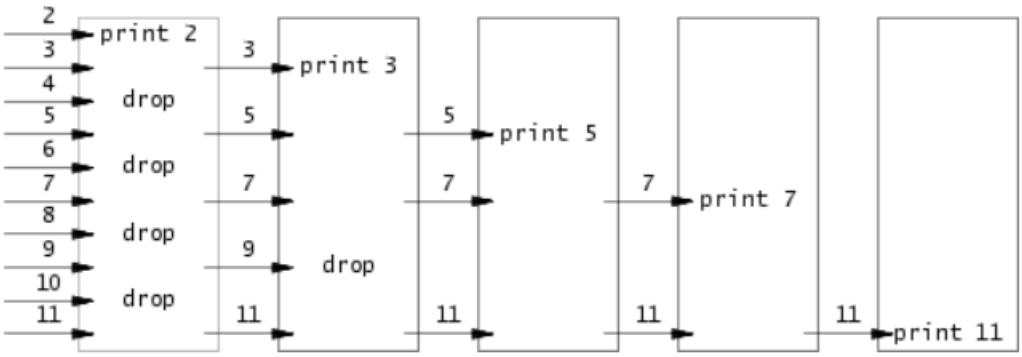
2. 将程序添加到 `Makefile` 中的 `UPROGS` 中。

3) 问题与解决方法

1. 算法详解：
    - 伪代码：

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
p = 从左邻居中获取一个数
print p
loop:
    n = 从左邻居中获取一个数
    if (n不能被p整除)
        将n发送给右邻居
```

    - 原理图：



    - 过程：将数字2、3、4、...、1000输入管道的左端：行中的第一个进程消除2的倍数，第二个进程消除3的倍数，第三个进程消除5的倍数，依此类推。

4. Find

1) 实验目的

编写一个简化版本的UNIX的`find`程序：查找目录树中具有特定名称的所有文件。使用递归允许`find`下降到子目录中，但不要在“`.`”和“`..`”目录中递归。

2) 实验步骤

1. 新建`user\find.c`，`main`函数接收两个参数：

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

2. 完成 `find()` 函数，对于第一个参数路径，需要对文件和路径两种情况分别处理：

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

    switch(st.type){
        case T_FILE:
            if(strcmp(fmtname(path), filename) == 0)
                fprintf(1, "%s\n", path);
            break;

        case T_DIR:
            if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
                fprintf(2, "find: path too long\n");
                break;
            }
            strcpy(buf, path);
            p = buf+strlen(buf);
            *p++ = '/';
            while(read(fd, &de, sizeof(de)) == sizeof(de)){
                if(de.inum == 0 || strcmp(de.name, ".") == 0 ||
strcmp(de.name, "..") == 0)
                    continue;
                memmove(p, de.name, DIRSIZ); // 将目录项的名称复制到 p 所指向的缓
冲区
                p[DIRSIZ] = 0;
                find(buf, filename);
            }
            break;
    }
    close(fd);
}
```

`fmtname()` 仿照 `ls.c` 中的写法，用于规范文件名格式，便于比较和输出：

```
char* fmtname(char *path)
{
  static char buf[DIRSIZ+1];
  char *p;
  // 寻找最后一个斜杠之后的第一个字符
```

```
  for(p=path+strlen(path); p >= path && *p != '/'; p--)
    ;
  p++;

  if(strlen(p) >= DIRSIZ)
    return p; // 文件名的长度大于等于 DIRSIZ，则直接返回文件名
  memmove(buf, p, strlen(p)+1); //考虑末尾的\0
  return buf;
}
```

3. 将程序添加到`Makefile`中的`UPROGS`中。

3) 问题与解决方法

1. C语言中，`char[]`字符串之间的比较要用`strcmp`，这一函数比较第一个`\0`之前的全部内容，如果出错大概率是因为在比较前的字符串操作中修改了`\0`的情况，因此我仿照`ls.c`中的写法，设计了`fmtname()`函数用于规范字符串格式以避免此问题。

5. Xargs

1) 实验目的

编写一个简化版UNIX的`xargs`程序：它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。

2) 实验步骤

1. 新建`user/xargs.c`，`main`函数接收的参数可能有很多，需要创建存储命令参数的数组`args[MAXARG]`和一个缓冲区`buf[1024]`，以及一些辅助变量：

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

char* args[MAXARG], buf[1024];
char c;
int pid, i, j = 0, status = 1;

for(i = 0; i < MAXARG - 1 && i < argc - 1; i++){
    args[i] = argv[i + 1];
}
args[i] = 0; // 最后一个参数为NULL，标记参数列表的结束，传递给 exec 不会出错
```

2. 函数主要逻辑循环：`i`初始化为`argc - 1`，标记当前行数。`args[i++]`存储的是当前行的起始位置。在内层循环中，通过`read`函数从标准输入读取一个字符。如果读取到文件末尾，程序退出。如果读取到换行符或空格，将缓冲区中的字符结束为字符串，将起始位置添加到`args`数组中，并根据情况退出内层循环。如果读取到普通字符，将字符添加到缓冲区中。

   退出循环后，用`fork()`创建一个子进程。子进程调用`exec()`来执行给定的命令，将之前构建的`args`数组传递给命令。父进程用`wait()`等待子进程完成。

```
while(status && i < MAXARG - 1){
    i = argc - 1; // 标记第几行
```

```
    args[i++] = &buf[j]; //第一行的起始位置
    while(1){ // 读一行输入
        if((status = read(0, &c, 1)) == 0)
            exit(0); // 从标准输入中读取，没有数据时退出
        if(c == '\n' || c == ' '){
            buf[j++] = 0;
            args[i++] = &buf[j];
            if(c == '\n')
                break;
        }
        else{
            buf[j++] = c;
        }
    }
    args[i] = 0;

    if((pid = fork()) < 0){
        fprintf(2, "xargs: cannot fork\n");
        exit(1);
    }
    else if(pid == 0)
        // 子进程
        exec(argv[1], args);
    else
        // 父进程
        wait(0);
}
```

3. 将程序添加到`Makefile`中的`UPROGS`中。

3) 问题与解决方法

1. `xargs`命令用法：

```
$ echo hello too | xargs echo bye
bye hello too
$
```

命令是`echo bye`，`echo`用于打印文本内容，额外的参数是`hello too`，管道运算符`|`用于将前一个命令的输出作为后一个命令的输入，这样就组成了命令`echo bye hello too`。因此，此命令输出`bye hello too`。

2. `xargs`与`find`、`grep`等结合：

```
$ find . b | xargs grep hello
```

命令是`grep hello`，`grep`用于在文件列表中中搜索模式字符串，第一个参数为要被搜索的模式串`hello`，第二个参数是由管道传来的`find . b`的结果。因此，整个命令将对“`.`”下面的目录中名为`b`的每个文件中搜索`hello`。

3. 从文件读取参数：

```
cat files.txt | xargs rm
```

文件 `files.txt` 包含多行文件名，上述命令将这些文件名逐行作为参数传递给 `rm` 命令来删除这些文件。

6. 测试结果

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.3s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.8s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.9s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (0.9s)
    (Old xv6.out.pingpong failure log removed)
== Test primes ==
$ make qemu-gdb
primes: OK (1.1s)
    (Old xv6.out.primes failure log removed)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.0s)
    (Old xv6.out.find_curdir failure log removed)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.1s)
    (Old xv6.out.find_recursive failure log removed)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.1s)
    (Old xv6.out.xargs failure log removed)
```

Lab2: System Calls

1. System call tracing

1) 实验目的

添加一个系统调用跟踪功能，创建一个新的 `trace` 系统调用来控制跟踪。它应该有一个参数，这个参数是一个整数“掩码”（`mask`），它的比特位指定要跟踪的系统调用。

2) 实验步骤

1. 研究 `user/trace.c` 核心代码，如下：

```
if (trace(atoi(argv[1])) < 0) {
    fprintf(2, "%s: trace failed\n", argv[0]);
    exit(1);
}
for(i = 2; i < argc && i < MAXARG; i++){
    nargv[i-2] = argv[i];
}
exec(nargv[0], nargv);
```

先调用 `trace()`，然后将 `argv` 第二个及以后的参数复制到 `nargv` 中。

2. 因此，在`kernel/proc.h`中新建一个`mask`用于保存`trace`的参数，并在`kernel/sysproc.c`中的`sys_trace()`实现对其的保存，在`fork()`时也要传递这一变量：

```
// kernel/proc.h
struct proc {
    ...
    int mask;          // Syscall trace argument container
}
```

```
// kernel/sysproc.c
uint
sys_trace(void)
{
  int pid;
  struct proc *p = myproc();
  if(argint(0, &pid) < 0)
    return -1;
  p->mask = pid;
  return 0;
}
```

```
// kernel/proc.c
int
fork(void)
{
    ...
    np->sz = p->sz;
    // copy trace mask
    np->mask = p->mask;
    // copy saved user registers.
    *(np->trapframe) = *(p->trapframe);
    ...
}
```

3. 在`kernel/syscall.h`和`kernel/syscall.c`保存新系统调用和系统调用号，用一个数组存储系统调用名称用以输出：

```
// kernel/syscall.h
#define SYS_trace  22
```

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

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

static char* syscall_names[22] = {
  "fork", "exit", "wait", "pipe", "read",
  "kill", "exec", "fstat", "chdir", "dup",
```

```
  "getpid", "sbrk", "sleep", "uptime", "open",
  "write", "mknod", "unlink", "link", "mkdir",
  "close", "trace",
};

void
syscall(void)
{
    ...
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num]();
        if(p->mask & (1 << num)){ // 检查第num位，判断当前系统调用是否被跟踪
          // 打印跟踪信息：PID、系统调用名称和返回值。
          printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num - 1],
p->trapframe->a0);
        }
    } else {
        printf("%d %s: unknown sys call %d\n",
                p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

4. 将系统调用的原型添加到`user/user.h`，存根添加到`user/usys.pl`：

```
// user/user.h
// system calls
...
int uptime(void);
int trace(int);
```

```
// usys.pl
...
entry("uptime");
entry("trace");
```

3) 问题与解决方法

理解系统调用的过程，从而理解如何新添加一个系统调用：

1. 用户态的系统调用函数被声明（没有实现）在`user/user.h`中，这些函数由汇编实现在`user/usys.S`中，其中有系统调用号，定义在`kernel/syscall.h`中。
2. 随后调用`ecall`指令，跳转到一个内核栈（存放内核服务）。同时，`ecall`会保存现场（如栈指针`sp`、程序计数器`pc`等），使结束系统调用的时候我们就可以顺利的恢复到当前状态。
3. `ecall`跳转到内核后，先进入`syscall()`函数，其会根据`a7`寄存器中的调用号，在`static uint64 (*syscalls[])(void)`中查找到名为`sys_xxx`的函数。
4. 在`kernel/sysproc.c`中，有系统服务的具体实现。
5. 调用完成后，在系统调用的返回值返回用户态时，会用`p->trapframe->a0 = syscalls[num]();`这句话赋值到`a0`寄存器上。

2. Sysinfo

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。系统调用采用一个参数：一个指向`struct sysinfo`的指针。这个结构的字段：`freemem`字段应该设置为空闲内存的字节数，`nproc`字段应该设置为`state`字段不为`UNUSED`的进程数。

2) 实验步骤

1. 在`kernel/kalloc.c`中添加一个函数用于获取空闲内存量：

```
// kernel/kalloc.c
void
freebytes(uint64 *size)
{
  *size = 0;
  struct run *p = kmem.freelist; // 指针p用于遍历内存链表
  acquire(&kmem.lock);
  while(p){
    *size += PGSIZE;
    p = p->next;
  }
  release(&kmem.lock);
}
```

2. 在`kernel/proc.c`中添加一个函数用于获取进程数：

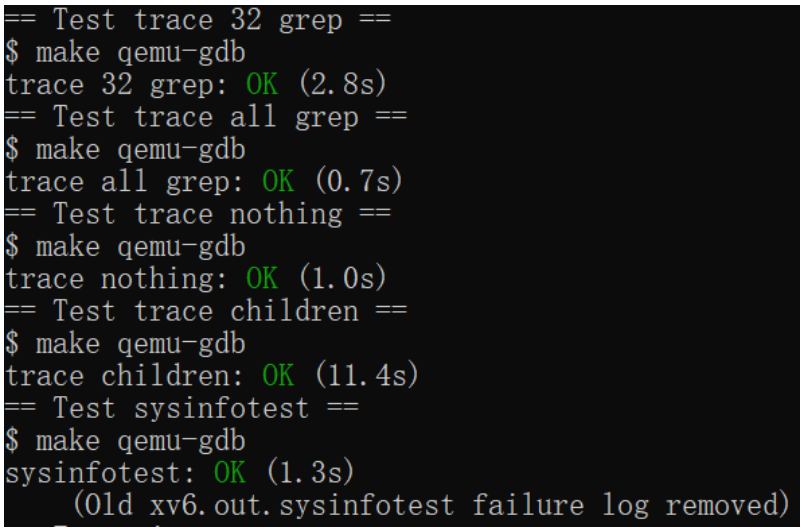
```
// kernel/proc.c
void
proc_num(uint64* num)
{
  *num = 0;
  struct proc *p; // 遍历进程表
  for (p = proc; p < &proc[NPROC]; p++) {
    if (p->state != UNUSED)
      ++*num;
  }
}
```

3. 用同样方式添加系统调用号等，并在`kernel/defs.h`中添加上述两个函数，`sys_sysinfo`实现如下，`sysinfo`需要将一个`struct sysinfo`复制回用户空间：

```
// kernel/sysproc.c
...
#include "sysinfo.h"
...
uint
sys_sysinfo(void)
{
  struct sysinfo info;
  freebytes(&info.freemem); // 获取空闲内存量
  proc_num(&info.nproc);    // 获取进程数
```

```
  // 将一个sysinfo复制回用户空间
  struct proc *p = myproc();
  uint64 addr;
  argaddr(0, &addr);
  if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;
  return 0;
}
```

3. 测试结果



Lab3: Page Tables

1. Speed up system calls

1) 实验目的

任务要求在xv6操作系统中，为`getpid()`系统调用实现一个优化，通过在用户空间和内核之间共享一个只读区域的数据来加速特定的系统调用，从而消除在执行这些系统调用时的内核切换。即，创建进程时，就直接把进程的`pid`放入共享空间中，然后用户查询`pid`时，就不必通过`ecall`跳转到内核了，省去了保存现场等开销。

2) 实验步骤

1. 在`kernel/memlayout.h`中定义地址`USYSCALL`和结构体`usyscall`：

```
// kernel/memlayout.h
#define USYSCALL (TRAPFRAME - PGSIZE)
struct usyscall {
  int pid;  // Process ID
};
```

2. 在`kernel/proc.h`中的`proc`结构体定义中新建一个`usyscall`结构体指针：

```
// kernel/proc.h
struct proc {
    ...
    struct usyscall *usc;
}
```

3. 在每个进程被创建时，用`mappages()`函数在地址`USYSCALL`处映射一个只读页。在这个页的起始位置，存储一个`usyscall`结构体），并将其初始化为存储当前进程的PID：

```
// kernel/proc.c
pagetable_t
proc_pagetable(struct proc *p)
{
  ...
  // 在地址USYSCALL处映射一个只读页
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usc), PTE_R | PTE_U) < 0){ // 设置读和用户可用两个标志
    uvmunmap(pagetable, TRAMPOLINE, 1, 0); // 若分配失败，则删除之前的页面映射
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }

  return pagetable;
}
```

4. 在`allocproc()`函数中为该页分配和初始化内存：

```
// kernel/proc.c
static struct proc*
allocproc(void){
    ...
    if((p->usc = (struct usyscall *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }
    p->usc->pid = p->pid;
    ...
}
```

5. 在`freeproc()`函数中释放该页：

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

3) 问题与解决方法

1. 启动时：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
panic: freewalk: leaf
```

`panic: freewalk: leaf`来自于`kernel\vm.c`中的`freewalk()`函数，表明仍有没有释放的页表项。最终发现是因为没有在`proc_freepagetable()`中释放新添加的页。

解决方法：

```
// kernel/proc.c
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

2.

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: FAIL
    ...
         $ pgtbltest
         ugetpid_test starting
         usertrap(): unexpected scause 0x0000000000000005 pid=4
                     sepc=0x0000000000000476 stval=0x0000003fffffd000
         $ qemu-system-riscv64: terminating on signal 15 from pid 59047
(make)
    MISSING '^ugetpid_test: OK$'
```

`0x0000000000000005`表示一个`Load Access Fault`异常，这通常是由于用户程序在访问无效的虚拟地址（非法地址）导致的。

原因是在`allocproc()`时`proc_pagetable()`写在了`p->usc = (struct usyscall *)kalloc()`之前，`proc_pagetable()`中的`mappages()`找不到`(uint64)(p->usc)`对应的地址。

解决方法：将两句顺序调换即可解决此问题。

2. Print a page table

1) 实验目的

为了可视化RISC-V页表和后续调试，编写一个打印页表内容的函数：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。在`exec.c`中的`return argc`之前插入`if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表。

启动xv6时，它应该像这样打印输出来描述第一个进程刚刚完成`exec()` `init`时的页表：

```
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
```

第一行显示`vmprint`的参数。之后的每行对应一个PTE，包含树中指向页表页的PTE。每个PTE行都有一些“`..`”的缩进表明它在树中的深度。每个PTE行显示其在页表页中的PTE索引、PTE比特位以及从PTE提取的物理地址。不要打印无效的PTE。在上面的示例中，顶级页表页具有条目0和255的映射。条目0的下一级只映射了索引0，该索引0的下一级映射了条目0、1和2。

2) 实验步骤

1. 按照`freewalk()`函数的方法，用递归方式写函数`vmprint()`，`printf`调用中使用`%p`来打印像上面示例中的完成的64bit的十六进制PTE和地址：

```
// kernel/vm.c
// 递归打印页表内容，depth为页表层级
void
vmprint_recursive(pagetable_t pagetable, int depth)
{
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    // 页表项可用且不在最后一层时，递归调用
    if(pte & PTE_V){
      for (int j = 0; j < depth; j++){
        if (j) printf(" "); // 第一层打印开头无空格
        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_recursive((pagetable_t)child, depth+1);
    }
  }
}

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

2. 将`vmprint`的原型定义在`kernel/defs.h`中，以便在`exec.c`中调用：

```
// kernel/defs.h
// vm.c
...
void            vmprint(pagetable_t);
...
```

3. 在`exec.c`中的`return argc`之前插入`if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表：

```
// kernel/exec.c
int
exec(char *path, char **argv)
{
    ...
    if(p->pid==1) vmprint(p->pagetable);
    return argc; // this ends up in a0, the first argument to main(argc,
argv)
    ...
}
```

3) 问题与解决方法

1. `pte`和`pa`代表的含义：PTE（Page Table Entry）是页表项，PA（Physical Address）是物理地址。
2. 对`freewalk()`函数的理解：

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalk((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

在此函数中，先遍历整个页表（一张页表有512个页表项），当遇到页表项可用（`(pte & PTE_V)`）且不在最后一层（`(pte & (PTE_R|PTE_W|PTE_X)) == 0`）时，对其子级页表进行递归调用。

3. 判断页表项不在最后一层的原因：访问虚拟地址时，CPU会根据页表的映射关系找到对应的PTE，并通过PTE获取与虚拟地址相对应的物理地址。**然而，在页表的最后一级，它对应着实际存在于物理内存中的物理页面，而非逻辑上的存在。**当一个普通的用户进程访问某个虚拟地址时，操作系统需要确保该虚拟地址对应的物理页面在内存中可读可写可执行。因此，在页表的最后一级（最底层的页表），至少有一个权限位必须被设置为1，以确保该虚拟页的物理页面有实际的访问权限。

4. ```
   kernel/vm.c: In function ‘vmprint_recursive’:
   kernel/vm.c:298:14: error: suggest parentheses around comparison in operand of ‘&’ [-Werror=parentheses]
     298 |       if(pte & (PTE_R|PTE_W|PTE_X) == 0)
   ```

   解决方法：将`pte & (PTE_R|PTE_W|PTE_X) == 0`改为`(pte & (PTE_R|PTE_W|PTE_X)) == 0`。

3. Detecting which pages have been accessed

1) 实验目的

为xv6操作系统添加一个新功能，即通过检查RISC-V页表中的访问位，来检测和报告哪些页面已经被访问（读或写）。RISC-V硬件页表在解决TLB不命中时会标记这些访问位。

需要实现一个名为`pgaccess()`的系统调用，用于报告哪些页面已被访问。该系统调用接受三个参数。第一个参数是要检查的第一个用户页的起始虚拟地址。第二个参数是要检查的页数。最后一个参数是一个用户态地址，用于存储结果到位掩码（`bitmask`）。位掩码是一种数据结构，它每个页面使用一个位来表示，其中第一个页面对应最低有效位。

2) 实验步骤

1. 在`kernel/riscv.h`中定义`PTE_A`，即访问位，通过查阅RISC-V手册来确定它的值：

   ```
   // kernel/riskv.h
   #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_A (1L << 6)
   ```

2. 由于`sys_pgaccess`已经在`syscall.h`等处登记，所以直接从`kernel/sysproc.c`开始实现`sys_pgaccess()`，需要使用`argaddr()`和`argint()`来解析参数：

   ```
   uint64 start_addr;  // 要检查的第一个用户页的起始虚拟地址
   int num_pages;      // 要检查的页数
   uint64 user_addr;   // 用户态地址，用于存储结果到位掩码
   uint64 bitmask = 0; // 存储结果的位掩码

   argaddr(0, &start_addr);
   argint(1, &num_pages);
   argaddr(2, &user_addr);
   ```

3. 逐页检查，计算当前虚拟地址对应的页表项的地址，如果当前页的PTE存在且访问位被设置为1，将位掩码的对应位设置为1。同时，确保在检查完`PTE_A`位是否被设置后，将其清除。否则，将无法确定页面自上次调用`pgaccess()`以来是否被访问（即该位将永远被设置）：
   ```

```
struct proc *p = myproc();
for(int i = 0; i < num_pages; i++){
    // 计算当前虚拟地址对应的页表项的地址
    pte_t *pte = walk(p->pagetable, start_addr + PGSIZE * i, 0);
    if(pte != 0 && (*pte & PTE_A)){
        // 如果该虚拟页面的PTE存在且访问位被设置为1
        // 将位掩码的第i位设置为1
        bitmask |= (1UL << i);
        // 清除该PTE的访问位
        *pte &= ~PTE_A;
    }
}
```

4. 存储到`bitmask`完成后将用户态地址复制到用户空间：

```
// 存储到bitmask完成后将其复制到用户空间
copyout(p->pagetable, user_addr, (char *)bitmask, sizeof(bitmask));
```

5. 完整函数如下：

```
// kernel/sysproc.c
int
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 start_addr;  // 要检查的第一个用户页的起始虚拟地址
  int num_pages;      // 要检查的页数
  uint64 user_addr;   // 用户态地址，用于存储结果到位掩码
  uint64 bitmask = 0; // 存储结果的位掩码

  argaddr(0, &start_addr);
  argint(1, &num_pages);
  argaddr(2, &user_addr);

  struct proc *p = myproc();
  for(int i = 0; i < num_pages; i++){
    // 计算当前虚拟地址对应的页表项的地址
    pte_t *pte = walk(p->pagetable, start_addr + PGSIZE * i, 0);
    if(pte != 0 && (*pte & PTE_A)){
      // 如果该虚拟页面的PTE存在且访问位被设置为1
      // 将位掩码的第i位设置为1
      bitmask |= (1UL << i);
      // 清除该PTE的访问位
      *pte &= ~PTE_A;
    }
  }

  // 存储到bitmask完成后将其复制到用户空间
  copyout(p->pagetable, user_addr, (char *)bitmask, sizeof(bitmask));
  return 0;
}
```

3) 问题与解决方法

1.
```
kernel/sysproc.c: In function ‘sys_pgaccess’:
kernel/sysproc.c:96:18: error: implicit declaration of function ‘walk’ [-
Werror=implicit-function-declaration]
   96 |     pte_t *pte = walk(p->pagetable, start_addr + PGSIZE * i, 0);
      |                  ^~~~
kernel/sysproc.c:96:18: error: initialization of ‘pte_t *’ {aka ‘long
unsigned int *’} from ‘int’ makes pointer from integer without a cast [-
Werror=int-conversion]
cc1: all warnings being treated as errors
make: *** [Makefile:130: kernel/sysproc.o] Error 1
```

   解决方法：`kernel/vm.c`的`walk()`并没有在`kernel/defs.h`中声明，将其声明即可：

```
// kernel/defs.h
// vm.c
pte_t *         walk(pagetable_t, uint64, int);
```

2.
```
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ usertests
usertests starting
qemu-system-riscv64: terminating on signal 15 from pid 66231 (make)
```

   上述错误信息表明：有一条指令尝试访问一个非法地址`0x0000000000000001`，从而触发了`Illegal Instruction`异常，很容易发现是由于`sys_pgaccess()`中`copyout()`中的`(char *)bitmask`处将`uint64`类型的`bitmask`转换为了`char*`而导致的错误。

   解决方法：将`(char *)bitmask`改为`(char *)&bitmask`即可。

4. 测试结果

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (0.8s)
```

Lab4: Traps

1. RISC-V assembly

1) 实验目的

理解`RISC-V`汇编：阅读`call.asm`中函数`g`、`f`和`main`的代码并回答问题。

2) 实验步骤

1. **问题**：哪些寄存器保存函数的参数？例如，在`main`对`printf`的调用中，哪个寄存器保存13？

   回答：在`a0`-`a7`中存放参数，13存放在`a2`中 。代码`li  a2,13`将13加载到`a2`寄存器，作为`printf`函数的参数。

2. **问题**：`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(提示：编译器可能会将函数内联)

   回答：代码`li a1,12`可以看出，main进行了内联优化处理，直接计算出了`f(8)+1`的结果并储存。

3. **问题**：`printf`函数位于哪个地址?

   回答：`0x628`

4. **问题**：在`main`中`printf`的`jalr`之后的寄存器`ra`中有什么值?

   回答：代码：

   ```
   30: 00000097            auipc   ra,0x0
   34: 5f8080e7            jalr    1528(ra) # 628 <printf>
   ```

   第一行：`00000097H=00...0 0000 1001 0111B`，对比指令格式可知，`auipc`的操作码是`0010111`，`ra`寄存器代码是`00001`。这行代码将`0x0`左移12位加到PC（`0x30`）上并存入ra中，即`ra`中保存的是`0x30`。

   第二行：`5f8080e7H=01011111100000001000000011100111B`，操作码为`1100111`，`rs1`和`rd`均为`00001`，即都为`ra`，`ra`中保存`PC+4`，即`0x38`。

5. **问题**：运行以下代码：

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

   程序的输出是什么？这是将字节映射到字符的ASCII码表。

   输出取决于RISC-V小端存储的事实。如果RISC-V是大端存储，为了得到相同的输出，你会把`i`设置成什么？是否需要将`57616`更改为其他值?

   回答：

   57616=0xE110，0x00646c72小端存储为72-6c-64-00

   对照ASCII码表：72:r 6c:l 64:d 00:充当字符串结尾标识

   因此输出为：HE110 World

   若为大端存储，`i`应改为0x726c6400，不需改变57616

6. **问题**：在下面的代码中，“`y=`”之后将打印什么(注：答案不是一个特定的值）？为什么会发生这种情况?

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

   回答：需要两个参数，却只传入了一个，因此`y=`之后打印的结果为之前`a2`中保存的数据。

3) 问题与解决方法

1. 第五问中的`auipc`和`jalr`指令：
   - `auipc`(Add Upper Immediate to PC)：`auipc dest imm`，将高位立即数加到PC上。指令格式：

	start bit	end bit

	start bit	end bit
imm[31:12]	31	12
dest	11	7
opcode	6	0

该指令将20位的立即数左移12位之后（右侧补0）加上`PC`的值，将结果保存到`dest`位置。

- `jalr` (Jump And Link Register)：`jalr rd, offset(rs1)`跳转并链接寄存器。指令格式：

	start bit	end bit
offset[11:0]	31	20
rs1	19	15
funct3	14	12
rd	11	7
opcode	6	0

会将当前`PC+4`保存在`rd`中，然后跳转到指定的偏移地址`offset(rs1)`。

2. 汇编学习（以`g`函数为例）：

```
int g(int x) {
   0:   1141                    addi    sp,sp,-16
   2:   e422                    sd  s0,8(sp)
   4:   0800                    addi    s0,sp,16
  return x+3;
}
   6:   250d                    addiw   a0,a0,3
   8:   6422                    ld  s0,8(sp)
   a:   0141                    addi    sp,sp,16
   c:   8082                    ret
```

先是`g`函数的准备部分：

- `0: addi sp,sp,-16`：分配栈帧空间，将栈指针`sp`减去 16，为函数`g`的局部变量和保存上下文留出空间。
- `2: sd s0,8(sp)`：保存寄存器`s0`的值到当前栈帧中的偏移 8 的位置，即保存调用者的栈帧基址。
- `4: addi s0,sp,16`：设置`s0`寄存器为当前栈帧的基址，方便访问局部变量。

之后是`return x+3`部分：

- `6: addiw a0,a0,3`：将函数参数`x`的值加上 3，保存在`a0`寄存器中，然后返回。
- `8: ld s0,8(sp)`：从栈帧中恢复调用者的栈帧基址，存储在寄存器`s0`中。
- `a: addi sp,sp,16`：释放当前栈帧的空间，将栈指针`sp`加上 16，恢复到调用者的栈帧。
- `c: ret`：返回到调用者，恢复调用者的程序执行。

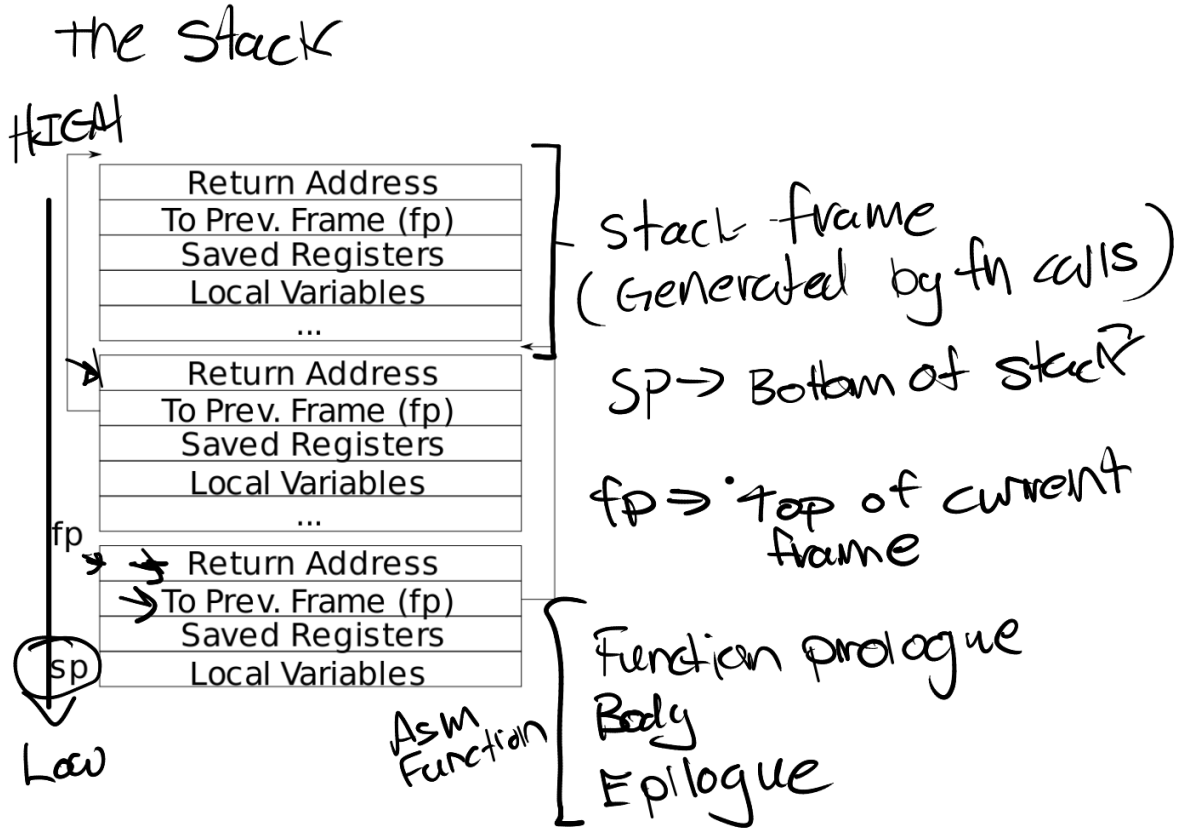
2. Backtrace

1) 实验目的

回溯(`Backtrace`)通常对于调试很有用：它是一个存放于栈上用于指示错误发生位置的函数调用列表。

在`kernel/printf.c`中实现名为`backtrace()`的函数。在`sys_sleep`中插入一个对此函数的调用，然后运行`bttest`，它将会调用`sys_sleep`。编译器向每一个栈帧中放置一个帧指针（frame pointer）保存调用者帧指针的地址。`backtrace`应当使用这些帧指针来遍历栈，并在每个栈帧中打印保存的返回地址。

栈帧布局如下笔记：



2) 实验步骤

1. 在`kernel/riscv.h`中添加内嵌汇编，用于获取当前的帧指针（`fp`）：

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

   这个函数读出了`s0`（`fp`的别名）这个寄存器的值，然后储存在`x`中并返回`x`。

2. 在`kernel/printf.c`中实现`backtrace()`，通过`PGROUNDDOWN(fp)`和`PGROUNDUP(fp)`来计算栈页面的顶部和底部地址：

```
// kernel/printf.c
```

```
void
backtrace(void)
{
  printf("backtrace:\n");
  uint64 cur = r_fp();
  uint64 top = PGROUNDUP(cur);
  uint64 bottom = PGROUNDDOWN(cur);
  while(cur < top && cur > bottom){
    // 返回地址存储在栈帧的前8个字节位置
    printf("%p\n", *((uint64 *)(cur - 8)));
    // 上一个调用函数的fp存储在前16个字节位置
    cur = *((uint64 *)(cur - 16));
  }
}
```

3. 在`panic()`和`sys_sleep()`中插入调用：

```
// kernel/printf.c
void
panic(char *s)
{
  ...
  printf("\n");
  backtrace();
  panicked = 1; // freeze uart output from other CPUs
  ...
}

// kernel/defs.h
//printf.c
...
void            backtrace(void);

// kernel/sysproc.c
uint64
sys_sleep(void)
{
  ...
  release(&tickslock);
  backtrace();
  return 0;
}
```

3) 问题与解决方法

1. 对于栈帧（`Stack Frame`）和其指针的理解：
   - 每调用一个函数时，函数为自己创建一个栈帧，只给自己使用
   - 栈总是向下扩展（从高位地址向地位地址使用），因此上述步骤中递减
   - 返回地址（`Return Address`）总是出现在栈帧的第一位
   - 用两个指针：`sp(Stack Pointer)`指向底部，`fp(Frame Pointer)`指向顶部

3. Alarm

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。这对于那些希望限制CPU时间消耗的受计算限制的进程，或者对于那些计算的同时执行某些周期性操作的进程可能很有用。这将实现用户级中断/故障处理程序的一种初级形式。

添加一个新的`sigalarm(interval, handler)`系统调用，如果一个程序调用了`sigalarm(n, fn)`，那么每当程序消耗了CPU时间达到`n`个"滴答"，内核应当使应用程序函数`fn`被调用。当`fn`返回时，应用应当在它离开的地方恢复执行。在XV6中，一个滴答是一段相当任意的时间单元，取决于硬件计时器生成中断的频率。如果一个程序调用了`sigalarm(0, 0)`，系统应当停止生成周期性的报警调用。

此外，还要实现一个`sigreturn()`系统调用，如果`handler`调用了这个系统调用，就应该停止执行`handler`这个函数，然后恢复正常的执行顺序。

test0: invoke handler

首先修改内核以跳转到用户空间中的报警处理程序，这将导致`test0`打印"`alarm!`"。不用担心输出"`alarm!`"之后会发生什么；如果程序在打印"`alarm!`"后崩溃，对于目前来说也是正常的。

即：先尝试去正确地跳转到**用户态**去执行函数，无所谓跳转之后地报错。

test1/test2(): resume interrupted code

要解决**test0**报错的问题，确保完成报警处理程序后返回到用户程序最初被计时器中断的指令执行。必须确保寄存器内容恢复到中断时的值，以便用户程序在报警后可以不受干扰地继续运行。最后，在每次报警计数器关闭后"重新配置"它，以便周期性地调用处理程序。

即：在执行完后返回到正确的位置。

2) 实验步骤

1. 首先完成**test0**，为了跟踪自上一次调用（或直到下一次调用）到进程的报警处理程序间经历了多少滴答，需要在`struct proc`中加入新字段，并在`allocproc()`中初始化和`freeproc()`中释放（设为0）：

```
// kernel/proc.h
struct proc {
  ...
  int alarm_ticks;              // 两次报警之间的滴答数
  int alarm_interval;           // 两次报警之间的间隔
  void (*alarm_handler)();      // 报警处理函数handler的地址
};
```

```
// kernel/proc.c
static struct proc*
allocproc(void)
{
  ...
  p->alarm_handler = 0;
  p->alarm_interval = 0;
  p->alarm_ticks = 0;
  return p;
}
```

```
static void
freeproc(struct proc *p)
{
  ...
  p->alarm_handler = 0;
  p->alarm_interval = 0;
  p->alarm_ticks = 0;
}
```

2. 在`sys_sigalarm()`中读取参数，`sys_sigreturn()`直接返回0即可：

```
// kernel/sysproc.c
uint64
sys_sigalarm(void)
{
  struct proc *p = myproc();
  if(argint(0, &p->alarm_interval) < 0 ||
    argaddr(1, (uint64 *)&p->alarm_handler) < 0)
    return -1;
  return 0;
}

uint64
sys_sigreturn(void)
{
  return 0;
}
```

3. 在`usertrap()`中实现：

```
// kernel/trap.c
  ...
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){ // 时钟中断编号为2
    if(++p->alarm_ticks == p->alarm_interval){
      // 到达规定时间时，直接修改epc
      p->trapframe->epc = (uint64)p->alarm_handler;
      p->alarm_ticks = 0; // 将计数器清零
    }
    yield();
  }
```

4. 修改`Makefile`、`kernel\syscall.c`、`kernel\syscall.h`、`user/usys.pl`、`user/user.h`，通过`test0`：

```
$ alarmtest
test0 start
.............................alarm!
test0 passed
test1 start
....alarm!
...alarm!
....alarm!
...alarm!
...alarm!
....alarm!
...alarm!
....alarm!
...alarm!
...alarm!

test1 failed: foo() executed fewer times than it was called
test2 start
..................................................alarm!
alarm!
test2 failed: alarm handler called more than once
```

5. 完成`test1`：报错的原因是当跳转到内核去响应陷入和系统调用时，寄存器的值是会改变的，先前的操作中没有备份寄存器，因此`epc`改变也无法正常执行。因此，需要在`struct proc`中加入一个`struct trapframe`进行备份工作：

```
// kernel/proc.h
struct proc{
    ...
    struct trapframe *alarm_trapframe; // 备份trapframe
}
```

相应地修改`allocproc()`和`freeproc()`：

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

  return p;
}

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

```
}
```

6. 在`usertrap()`中获取备份：

```
// kernel/trap.c
  ...
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){ // 时钟中断编号为2
    if(p->alarm_interval > 0){
      if(++p->alarm_ticks == p->alarm_interval){
        // 到达规定时间时，直接修改epc
        *p->alarm_trapframe = *p->trapframe; // 备份trapframe
        p->trapframe->epc = (uint64)p->alarm_handler;
        p->alarm_ticks = 0; // 将计数器清零
      }
    }
    yield();
  }
```

7. 在`sys_sigreturn()`中恢复：

```
// kernel/sysproc.c
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  *p->trapframe = *p->alarm_trapframe;
  return 0;
}
```

8. 此时测试结果如下：

```
test1 start
....alarm!
...alarm!
....alarm!
....alarm!
...alarm!
....alarm!
....alarm!
...alarm!
....alarm!
...alarm!
.test1 passed
test2 start
.....................................................
alarm!
test2 failed: alarm handler called more than once
```

可以通过`test1`但不能通过`test2`，这是因为当`handler`执行很慢时，已经超过了间隔但`handler`还没有执行完毕，此时再次修改`epc`则会导致出错。因此，需要为`struct proc`再添加一个字段，起到一个锁的作用，当其为1时说明正在执行`handler`：

```
// kernel/proc.h
struct proc{
    ...
    int alarm_lock;              // 为1时表示正在执行handler
}
```

同样地，也要修改`allocproc()`和`freeproc()`。

9. 修改`usertrap()`和`sys_sigreturn()`：

```
// kernel/trap.c
  ...
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){ // 时钟中断编号为2
    if(++p->alarm_ticks >= p->alarm_interval && !p->alarm_lock){
      // 到达规定时间时，直接修改epc
      *p->alarm_trapframe = *p->trapframe; // 备份trapframe
      p->trapframe->epc = (uint64)p->alarm_handler;
      p->alarm_ticks = 0; // 将计数器清零
      p->alarm_lock = 1;  // handler正在执行
    }
    yield();
  }
```

```
// kernel/sysproc.c
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  *p->trapframe = *p->alarm_trapframe;
  p->alarm_lock = 0;
  return 0;
}
```

10. 运行`alarmtest`，可以看到`test2`也通过：

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

3) 问题与解决方法

1. 如何判断到了要跳转的时间：每过一个滴答，硬件时钟就会强制一个中断，这个中断在`kernel/trap.c`中的`usertrap()`中处理。通过`usertrap()`的出现次数判断是否要跳转，如需要则在`kernel/trap.c`中修改`epc`（Exception Program Counter，执行系统调用后返回到用户空间继续执行的指令地址由其决定）为`handler`的。
2. 少写括号导致编译错误`assignment to 'struct trapframe *' from 'int' makes pointer from integer without a cast`：

   ```
   if(p->alarm_trapframe = (struct trapframe *)kalloc() == 0)
   ```

   解决方法：改为

   ```
   if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0)
   ```

3. 在`usertrap()`中没有先判断`if(p->alarm_interval > 0)`会导致某些`usertest`无法通过，因为某些测试设置的时钟间隔小于等于0，此时会引起错误。

4. 测试结果

```
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.9s)
== 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 (103.6s)
```

Lab5: Copy-on-Write Fork for xv6

1. Implement copy-on write

1) 实验目的

在xv6内核中实现写时复制（`copy-on-write，COW`），修改后的内核需要同时成功执行`cowtest`和`usertests`程序。

在没有`COW`的xv6中，`fork()`系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间，且这项工作经常造成大量浪费，例如：子进程中的`fork()`后跟`exec()`将导致子进程丢弃复制的内存，而其中的大部分可能都从未使用过。

如果父子进程都使用一个页面，并且其中一个或两个对该页面有写操作，则确实需要复制。

`copy-on-write (COW) fork()`的目标是推迟到子进程实际需要物理内存拷贝时，再进行分配和复制物理内存页面。其将父进程和子进程中的所有用户`PTE`标记为不可写，当任一进程试图写入时产生缺页错误。`usertrap()`检测到这一错误，将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关PTE指向新的页面，将PTE标记为可写。

COW下，给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

2) 实验步骤

1. 修改uvmcopy()将父进程的物理页映射到子进程，而不是分配新页。在子进程和父进程的PTE中清除PTE_W标志。为此，我们需要在新添加一个PTE的标志位（第8位是保留位），用于记录COW fork的页面：

```
// kernel/riscv.h
#define PTE_C (1L << 8) // 记录页面是否是COW fork的
```

修改uvmcopy()的标志位设置与虚拟地址映射，并设置引用计数器为后续操作使用：

```
// kernel/vm.c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  ...
    flags = PTE_FLAGS(*pte);

    if(flags & PTE_W){
      flags &= (~PTE_W); // 禁用写
      flags |= PTE_C;    // 设置COW
    }
    *pte = flags | PA2PTE(pa);
    // 删除原来实际分配内存的部分，并将虚拟地址i的映射从新物理地址mem改为父物理地址pa
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
      //kfree(mem);
      printf("uvmcopy failed!");
      goto err;
    }
    add_refcount((char *)pa); // 增加引用计数器
...
}
```

2. 修改usertrap()以识别页面错误。为此，需要实现为PTE_C标志位封装相关功能，包括判断是否为COW页与为COW页分配物理内存，并在defs.h中声明：

```
// kernel/vm.c
// 判断一个页面是否为COW
int
is_cow(pagetable_t pagetable, uint64 va)
{
  if(va >= MAXVA) return 0;
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0)  return 0;
  if((*pte & PTE_V) == 0) return 0;
  if((*pte & PTE_U) == 0) return 0;
  return (*pte & PTE_C ? 1 : 0);
}
```

为COW页分配新内存时，使用kalloc()分配一个新页面，并将旧页面复制到新页面，然后将新页面添加到PTE中并设置PTE_W：

```
// 为COW页分配物理内存
int
alloc_cow(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  if((pte = walk(pagetable, va, 0)) == 0) return -1;
  uint64 flags = PTE_FLAGS(*pte);
  uint64 pa = PTE2PA(*pte);

  char *new_page;
  if((new_page = kalloc()) == 0) return -1; // 申请内存
  uint64 new_va = PGROUNDDOWN(va);

  flags &= (~PTE_C); // 新页面不设置COW
  flags |= PTE_W;    // 新页面设置可写

  // 将父进程内容复制到子进程
  memmove(new_page, (char *)pa, PGSIZE);
  // 设置新的映射
  uvmunmap(pagetable, new_va, 1, 1);
  if(mappages(pagetable, new_va, PGSIZE, (uint64)new_page, flags) < 0){
    kfree(new_page);
    return -1;
  }

  return 0;
}
```

```
// kernel/defs.h
// vm.c
int             is_cow(pagetable_t, uint64);
int             alloc_cow(pagetable_t, uint64);
```

修改`usertrap()`，当`COW`页面出现页面错误时（`scause=15`代表缺页错误），为其分配新页面：

```
// kernel/trap.c
void
usertrap(void)
{
    ...
    // ok
  } else if(r_scause() == 15 && is_cow(p->pagetable, r_stval())){
    if(alloc_cow(p->pagetable, r_stval()) < 0)
      p->killed = 1;
  } else ...
}
```

3. 设置引用计数（`reference count`）：对于每个页帧，都有一个引用计数器来记录有多少个`COW`页正在使用该页，若没有`COW`页在使用这个页，则可以释放（类似于`close()`的实现）。用一个固定大小的整型数组作为引用计数器，并为其设置一个锁（在`kinit()`中初始化），为此，需要实现一个数据结构和对应的全局变量（`PHYSTOP`和`KERNBASE`分别是内存物理地址的起始和结束）：

```
// kernel/kalloc.c
struct ref_count {
  struct spinlock lock;
  int count[(PHYSTOP - KERNBASE) / PGSIZE];
} cnt;

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

修改`kalloc()`和`kfree()`：

```
// kernel/kalloc.c
void *
kalloc(void)
{
  ...
  if(r){
    kmem.freelist = r->next;
    acquire(&cnt.lock);
    cnt.count[((uint64)r - KERNBASE) / PGSIZE] = 1;
    release(&cnt.lock);
  }
  release(&kmem.lock);
  ...
}

void
kfree(void *pa)
{
  ...
  acquire(&cnt.lock);
  if(--cnt.count[((uint64)pa - KERNBASE) / PGSIZE] > 0){
    release(&cnt.lock);
  } else{ // 引用计数为0时才回收空间
    release(&cnt.lock);
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);
    r = (struct run*)pa;
    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
  }
}
```

实现计数器的自增：

```
// kernel/kalloc.c
void
add_refcount(void* pa)
{
  acquire(&cnt.lock);
  ++cnt.count[((uint64)pa - KERNBASE) / PGSIZE];
  release(&cnt.lock);
}
```

4. 最后，修改 `copyout()`，这是因为有些系统调用也会去往 `COW` 页上写数据。因为 `PTE_W` 没有设置，就会引发缺页错误，所以要修改 `copyout()`：

```
// kernel/vm.c
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  ...
  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    if(is_cow(pagetable, va0))
      if(alloc_cow(pagetable, va0) < 0) return -1;
    ...
  }
}
```

3) 问题与解决方法

1. 修改 `copyout()` 时，`alloc_cow()` 应当在 `walkaddr()` 之前，确保 `walkaddr()` 查找到的物理地址是新分配的物理地址，这样写入的是子进程独享的页帧；否则，`walkaddr()` 查找到的物理地址是父进程的共享页帧，往这里写入会导致错误：

```
== Test   file ==
  file: FAIL
    ...
        three: ok
        file: erreerorrr:o rr:o rrreeaadd:  frae afda iillefda
        ied
        led
        $ qemu-system-riscv64: terminating on signal 15 from pid 52195 (make)
    MISSING '^file: ok$'
```

2. 测试结果

```
== Test running cowtest == (4.6s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests == (100.6s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

Lab6: Multithreading

1. Uthread: switching between threads

1) 实验目的

实现用户态线程：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。

包括以下三个函数：

1. `thread_switch()`：与内核中的`swtch()`完全相同，只需要保存/还原被调用方保存的寄存器（`callee-save register`）。
2. `thread_create()`：用于创建新的用户线程，`ra`寄存器决定跳转位置，`sp`寄存器决定恢复出的被调用者保存寄存器的位置。
3. `thread_schedule()`：类似于内核中的`schedule()`，在当前进程`yield()`后，找到一个`RUNNABLE`的进程并执行，需要在`thread_schedule()`中添加对`thread_switch()`的调用。

2) 实验步骤

1. 为了实现上下文切换的功能，首先要为`struct thread`加入一个上下文的属性，并仿照`swtch()`实现对应的数据结构：

```
// user/uthread.c
struct context {
  /*   0 */ uint64 ra;
  /*   8 */ uint64 sp;
  // callee
  /*  16 */ uint64 s0;
  /*  24 */ uint64 s1;
  /*  32 */ uint64 s2;
  /*  40 */ uint64 s3;
  /*  48 */ uint64 s4;
  /*  56 */ uint64 s5;
  /*  64 */ uint64 s6;
  /*  72 */ uint64 s7;
  /*  80 */ uint64 s8;
  /*  88 */ uint64 s9;
  /*  96 */ uint64 s10;
  /* 104 */ uint64 s11;
};

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

2. 同样地，仿照`kernel/swtch.S`，写出`user/uthread_switch.S`的内容，完成`thread_switch()`：

```
    .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. 对于`thread_create()`，初始化`thread`的`ra`和`sp`（栈底），其他寄存器一开始未使用，故无需考虑：

```
void
thread_create(void (*func)())
{
  ...
  // YOUR CODE HERE
  t->ctx.ra = (uint64)func;
  t->ctx.sp = (uint64)&t->stack + STACK_SIZE - 1;
}
```

4. 在`thread_schedule()`中，第一个循环找到第一个`RUNNABLE`的线程，并为`next_thread`赋值，因此，之后要交换`next_thread`和`current_thread`，为此只需补充一句：

```
void
thread_schedule(void)
{
  ...
  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->ctx, (uint64)&current_thread->ctx);
   } else
     next_thread = 0;
}
```

2. Using threads

1) 实验目的

`notxv6/ph.c` 包含一个简单的哈希表，`ph` 运行两个基准程序。首先，它通过调用 `put()` 将许多键添加到哈希表中，并以每秒为单位打印 `puts` 的接收速率。之后它使用 `get()` 从哈希表中获取键。它打印由于 `puts` 而应该在哈希表中但丢失的键的数量，并以每秒为单位打印 `gets` 的接收数量。

如果单个线程使用（即 `ph` 参数为1），该哈希表是正确的（`0 keys missing`）；但是多个线程使用时（即 `ph` 参数大于1），该哈希表是不正确的，会导致线程丢失键值对。

为什么两个线程都丢失了键，而不是一个线程?

为了避免这一情况，在 `put` 和 `get` 中插入 `lock` 和 `unlock` 语句，以便在两个线程中丢失的键数始终为0。相关的 `pthread` 调用包括：

- `pthread_mutex_t lock;            // declare a lock`
- `pthread_mutex_init(&lock, NULL); // initialize the lock`
- `pthread_mutex_lock(&lock);       // acquire lock`
- `pthread_mutex_unlock(&lock);     // release lock`

2) 实验步骤

1. 定义一个数组，为每个散列桶定义一个锁：

```
pthread_mutex_t lock[NBUCKET];
```

2. 在 `put` 和 `get` 中插入 `lock` 和 `unlock` 语句：

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&lock[i]);
  // is the key already present?
  ...
  pthread_mutex_unlock(&lock[i]);
}

static struct entry*
get(int key)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&lock[i]);
  ...
  pthread_mutex_unlock(&lock[i]);
```

```
  return e;
}
```

3) 问题与解决方法

1. 研究`insert()`得出为什么会出现上述错误（即问题一的回答）：

   在哈希表中，如果哈希函数将多个不同的键映射到了同一位置上（同一个散列桶中），会以链表的形式存储，在查找时遍历此链表以寻找正确的键值对，`insert()`用于向链表中插入元素，`e->next = n;*p = e;`中，将`e`插入`*p`之前，即插入是通过头插法实现的。

   由此可以得出出现错误的原因：如果有两个线程`t1`和`t2`同时调用`put()`，且插入同一个散列桶中，两个线程同时调用`insert()`，当`t1`的`insert()`还未返回，`t2`便开始时，后者会覆盖前者插入的元素；或者`t1`插入过程中发生了线程调度，切换到`t2`进行插入，此时`t1`会覆盖掉`t2`插入的元素。

3. Barrier

1) 实验目的

实现一个同步屏障（`Barrier`）：用程序中的一个点，所有参与的线程在此点上必须等待，直到所有其他参与线程也达到该点。

除了在`ph`作业中看到的`lock`原语外，还需要以下新的`pthread`原语：

- `pthread_cond_wait(&cond, &mutex); // 在cond上进入睡眠，释放锁mutex，在醒来时重新获取`
- `pthread_cond_broadcast(&cond);    // 唤醒睡在cond的所有线程`

问题：

- 处理一系列的`barrier`调用，我们称每一连串的调用为一轮（`round`）。`bstate.round`记录当前轮数。每次当所有线程都到达屏障时，都应增加`bstate.round`。
- 处理这样的情况：一个线程在其他线程退出`barrier`之前进入了下一轮循环。特别是，在前后两轮中重复使用`bstate.nthread`变量。确保在前一轮仍在使用`bstate.nthread`时，离开`barrier`并循环运行的线程不会增加`bstate.nthread`。

2) 实验步骤

1. 理解一下`barrier`的原理和代码，不难写出`barrier()`的实现：

```
// notxv6/barrier.c
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{
    // 最后一个线程到达
    bstate.round++;
```

```
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

当非最后一个线程时，先调用`pthread_cond_wait()`令其等待，当最后一个线程到达时，增加`round`并用`pthread_cond_broadcast()`唤醒所有等待中的线程，在整个过程中上锁。

3) 问题与解决方法

1. `barrier`的数据结构如下：

```
struct barrier {
  pthread_mutex_t barrier_mutex;
  pthread_cond_t barrier_cond;
  int nthread;      // Number of threads that have reached this round of the
barrier
  int round;     // Barrier round
} bstate;
```

- `pthread_mutex_t barrier_mutex`：这是一个互斥锁，用于保护同步屏障数据结构中的共享变量。在多个线程访问和修改同步屏障状态时，需要使用互斥锁来避免竞态条件。
- `pthread_cond_t barrier_cond`：这是一个条件变量，用于在同步屏障上等待的线程之间进行通信。线程可以在条件变量上等待，直到满足某个条件时被唤醒。
- `int nthread`：表示已经到达当前屏障点的线程数量。每当一个线程到达同步屏障，它会增加这个计数值。当计数值达到预定的线程数时，意味着所有线程都已经到达，可以解除屏障并继续执行。
- `int round`：用于标识当前屏障的轮次或阶段。每次所有线程到达屏障并被释放后，轮次会递增，下一轮重新开始。

4. 测试结果

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (3.2s)
== Test answers-thread.txt == answers-thread.txt: FAIL
    Cannot read answers-thread.txt
== Test ph_safe == make[1]: Entering directory '/home/zmc/xv6/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/zmc/xv6/xv6-labs-2021'
ph_safe: OK (12.8s)
== Test ph_fast == make[1]: Entering directory '/home/zmc/xv6/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/zmc/xv6/xv6-labs-2021'
ph_fast: OK (24.8s)
== Test barrier == make[1]: Entering directory '/home/zmc/xv6/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/zmc/xv6/xv6-labs-2021'
barrier: OK (3.3s)
```

Lab7: Networking

1. Networking

1) 实验目的

背景：使用名为E1000的网络设备来处理网络通信，与xv6交互的E1000由qemu模拟。***kernel/e1000.c***文件包含E1000的初始化代码以及用于发送和接收数据包的空函数，您将填写这些函数。***kernel/e1000_dev.h***包含E1000定义的寄存器和标志位的定义，并在《英特尔E1000软件开发人员手册》中进行了描述。***kernel/net.c***和***kernel/net.h***包含一个实现IP、UDP和ARP协议的简单网络栈。这些文件还包含用于保存数据包的灵活数据结构（称为`mbuf`）的代码。最后，***kernel/pci.c***包含在xv6引导时在PCI总线上搜索E1000卡的代码。

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

`e1000_transmit()`：

1. 根据提示，读取`E1000_TDT`控制寄存器，向E1000询问等待下一个数据包的TX环索引（即`tail`）：

```
acquire(&e1000_lock); // 加锁应对多线程
uint32 index = regs[E1000_TDT]; // 下一个数据包的TX环索引
struct tx_desc *desc = &tx_ring[index];
```

2. 然后检查环是否溢出。如果`E1000_TXD_STAT_DD`未在`E1000_TDT`索引的描述符中设置，则E1000尚未完成先前相应的传输请求，因此返回错误：

```
// 检查描述符状态，如果没有空闲位置则直接返回
if(!(desc->status & E1000_TXD_STAT_DD)){
    release(&e1000_lock);
    return -1;
}
```

3. 检查这个描述符对应的`mbuf`的状态，如果`mbuf`已经使用传输完毕，则用`mbuffree()`释放从该描述符传输的最后一个`mbuf`：

```
// 之前地址还未释放，则释放
if(tx_mbufs[index])
  mbuffree(tx_mbufs[index]);
```

4. 填写描述符。`m->head`指向内存中数据包的内容，`m->len`是数据包的长度，设置必要的`cmd`标志，保存指向`mbuf`的指针，以便稍后释放：

```
tx_mbufs[index] = m; // 保存指向mbuf的指针，以便稍后释放
desc->addr = (uint64)m->head;
desc->length = m->len;
desc->cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
```

5. `E1000_TDT`加一取模来更新环位置：

```
regs[E1000_TDT] = (index + 1) % TX_RING_SIZE;
```

6. 最后解锁并返回，完整函数如下：

```
// kernel/e1000.c
int
e1000_transmit(struct mbuf *m)
{
  acquire(&e1000_lock); // 加锁应对多线程
  uint32 index = regs[E1000_TDT]; // 下一个数据包的TX环索引
  struct tx_desc *desc = &tx_ring[index];

  // 检查描述符状态，如果没有空闲位置则直接返回
  if(!(desc->status & E1000_TXD_STAT_DD)){
    release(&e1000_lock);
    return -1;
  }
  // 之前地址还未释放，则释放
  if(tx_mbufs[index])
    mbuffree(tx_mbufs[index]);

  tx_mbufs[index] = m; // 保存指向mbuf的指针，以便稍后释放
  desc->addr = (uint64)m->head;
  desc->length = m->len;
  desc->cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;

  regs[E1000_TDT] = (index + 1) % TX_RING_SIZE;

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

`e1000_recv()`：

1. 首先通过提取`E1000_RDT`控制寄存器并加一对`RX_RING_SIZE`取模，向E1000询问下一个等待接收数据包（如果有）所在的环索引：

```
uint32 index = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
struct rx_desc *desc = &rx_ring[index];
```

2. 然后通过检查描述符`status`部分中的`E1000_RXD_STAT_DD`位来检查新数据包是否可用：

```
if(!(desc->status & E1000_RXD_STAT_DD)) return;
```

3. 将`mbuf`的`m->len`更新为描述符中报告的长度。使用`net_rx()`将`mbuf`传送到网络栈：

```
rx_mbufs[index]->len = desc->length;
net_rx(rx_mbufs[index]);
```

4. 然后使用`mbufalloc()`分配一个新的`mbuf`，以替换刚刚给`net_rx()`的`mbuf`。将其数据指针（`m->head`）编程到描述符中。将描述符的状态位清除为零：

```
if((rx_mbufs[index] = mbufalloc(0)) == 0)
    panic("mbuf alloc failed!");
desc->addr = (uint64)rx_mbufs[index]->head;
desc->status = 0;
```

5. 将`E1000_RDT`寄存器更新为最后处理的环描述符的索引：

```
regs[E1000_RDT]= index;
```

6. 在某刻，曾经到达的数据包总数将超过环大小（16），将其放入循环即可，完整函数如下：

```
static void
e1000_recv(void)
{
  while(1){
    uint32 index = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    struct rx_desc *desc = &rx_ring[index];

    if(!(desc->status & E1000_RXD_STAT_DD)) return;

    rx_mbufs[index]->len = desc->length;
    net_rx(rx_mbufs[index]);

    if((rx_mbufs[index] = mbufalloc(0)) == 0)
      panic("mbuf alloc failed!");
    desc->addr = (uint64)rx_mbufs[index]->head;
    desc->status = 0;

    regs[E1000_RDT]= index;
  }
}
```

3) 问题与解决方法

对E1000的学习：

1. E1000 使用了`DMA（direct memory access）`技术，可以直接把接收到的数据包写入计算机的内存；在发送时也可以把描述符写入内存的特定位置，E1000 就会自己去找到待发送的数据并发送。接收和发送的数据包都是以描述符数组描述的，接收描述符在xv6中的定义如下：

```
// kernel/e1000_dev.h
// [E1000 3.2.3]
struct rx_desc
{
  uint64 addr;       /* Address of the descriptor's data buffer */
  uint16 length;     /* Length of data DMAed into data buffer */
  uint16 csum;       /* Packet checksum */
  uint8 status;      /* Descriptor status */
  uint8 errors;      /* Descriptor Errors */
  uint16 special;
};
```

- `addr`：存储描述符所关联数据缓冲区的地址。当接收到一个数据包时，数据将被传输到这个地址指定的缓冲区。
- `length`：已经 DMA到数据缓冲区中的数据的长度。
- `csum`：数据包的校验和，用于验证数据包是否被正确传输和接收。
- `status`：描述符状态，主要用到`DD (Descriptor Done)`这个标志位。

描述符数组会被解读为一个环形队列，`head`到`tail`之间区域的数据包已经被软件处理好，网卡接收新数据时，会向`head`位置描述符的`addr`地址的缓冲区写入数据。读取时，从`tail+1`位置开始处理（此位置是未处理数据中等待时间最长的），处理完后`tail++`。

2. 发送描述符定义如下：

```
// kernel/e1000_dev.h
// [E1000 3.3.3]
struct tx_desc
{
  uint64 addr;
  uint16 length;
  uint8 cso;
  uint8 cmd;
  uint8 status;
  uint8 css;
  uint16 special;
};
```

- `cso`：数据校验和偏移，用于验证数据包是否被正确传输。
- `cmd`：命令字段，用于控制发送过程中的一些操作，常用的包括：
  - RPS (Report Packet Sent)：设置之后，网卡会报告数据包发送的状态。
  - EOP (End of Packet)：表明这个描述符是数据包的结尾。
- `css`：发送状态报告的状态，用于告知发送结果。

描述符数组同样解读为环形队列，但`head`到`tail`之间的区域表示还未发送的数据，`head`等待时间最长，因此从这里开始发送，完成后`tail++`。新加入的描述符从`tail`处加入。

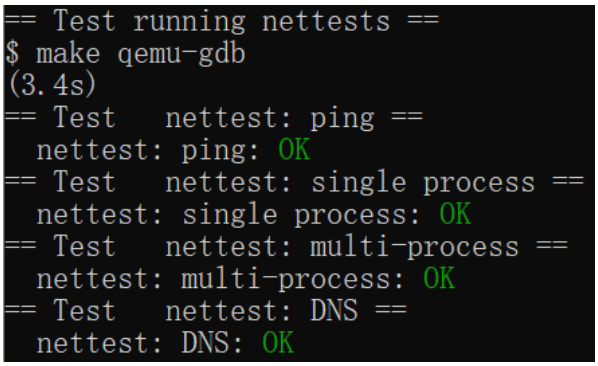
3. xv6还定义了一个保存数据包的灵活数据结构：

```
// kernel/net.h
struct mbuf {
  struct mbuf  *next; // the next mbuf in the chain
  char         *head; // the current start position of the buffer
  unsigned int len;   // the length of the buffer
  char         buf[MBUF_SIZE]; // the backing store
};
// The above functions manipulate the size and position of the buffer:
//            <- push            <- trim
//             -> pull            -> put
// [-headroom-][------buffer------][-tailroom-]
// |----------------MBUF_SIZE-----------------|
```

`buffer`部分存储数据正文，`headroom`是缓冲区，可以用`push`和`pull`与`buffer`交互，当需要更大的空间时，可以缩小`headroom`的大小以增大`buffer`大小：

```
// an Ethernet packet header (start of the packet).
struct eth {
  uint8  dhost[ETHADDR_LEN];
  uint8  shost[ETHADDR_LEN];
  uint16 type;
} __attribute__((packed));
```

2. 测试结果



Lab8: Locks

1. Memory allocator

1) 实验目的

原本的`kalloc()`有一个`freelist`，由一个锁保护。任何程序申请内存，都需要竞争该锁（见`kalloc()`和`kfree()`实现），这大大降低了内存分配的效率。为了解决这个问题，需要为每个处理器核心都分配一个`freelist`，并各自上锁。

主要的挑战将是处理一个CPU的`freelist`为空，而另一个CPU的`freelist`有空闲内存的情况。在这种情况下，一个CPU必须"偷取"另一个CPU`freelist`的一部分。偷取可能会发生锁争用，但这种情况希望不会经常发生。

所有锁的命名必须以"`kmem`"开头。也就是说，应该为每个锁调用`initlock`，并传递一个以"`kmem`"开头的名称。运行`kalloctest`以查看是否减少了锁争用，运行`usertests sbrkmuch`检查它是否仍然可以分配所有内存。

2) 实验步骤

1. 首先，将`kmem`改为数组，大小为CPU核心数：

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

2. 修改`kinit()`：

```
void
kinit()
{
  char lockname[7];
  for(int i = 0; i < NCPU; i++){
    snprintf(lockname, sizeof(lockname), "kmem_%d", i);
    initlock(&kmem[i].lock, lockname);
  }
  freerange(end, (void*)PHYSTOP);
}
```

3. 修改`kfree()`，需要先用`push_off()`禁用中断：

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

4. 修改`kalloc()`，同样需要开关中断：

```
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;
  else{
    int cpu2;
    for(cpu2 = 0; cpu2 < NCPU; cpu2++){
      if(cpu2 == cpu) continue;
      acquire(&kmem[cpu2].lock);
      r = kmem[cpu2].freelist;
```

```
      if(r){
        kmem[cpu2].freelist = r->next;
        release(&kmem[cpu2].lock);
        break;
      }
      release(&kmem[cpu2].lock);
    }
  }
  release(&kmem[cpu].lock);
  pop_off();

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

3) 问题与解决方法

虽然不禁用中断在测试中不会导致错误，但还是需要先关闭中断，否则可能会因为在偷页过程中跳出去处理其他进程，若这一进程调用`kalloc()`，则会导致重复偷页。

2. Buffer cache

1) 实验目的

在xv6中，我们是不能直接访问硬盘设备的，如果想要读取硬盘中的数据，需要先把数据拷贝到一个缓存中，然后读取缓存中的内容。磁盘数据的最小单位是一个块，大小为 1024 kb。在读写磁盘时，需要通过`bread()`函数得到相应的缓存，其中的`bget()`函数会判断缓存中是否已经缓存过该磁盘块，所有缓存是以双向链表的方式存储的，因此`bget()`会遍历一遍链表。

多个进程同时使用文件系统的时候，`bcache.lock`上会发生严重的锁竞争。`bcache.lock`锁用于保护磁盘区块缓存（位于`kernel/bio.c`中），在原本的设计中，由于该锁的存在，多个进程不能同时操作（申请、释放）磁盘缓存。

修改块缓存，以便在运行`bcachetest`时，`bcache`（`buffer cache`的缩写）中所有锁的`acquire`循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但只要总和小于500就可以。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（例如，不必全部等待`bcache.lock`）。

提示的方法是实现一个散列表，将块号映射到块缓存的桶，这样在查找块缓存时不需要遍历所有的缓存。同理，当桶中缓存不足时，可以从别的桶中偷。

2) 实验步骤

1. 首先修改`bcache`，将链表方式改为散列表方式存储，通的数量取质数13：

```
#define NBUFMAP 13
#define NBUFMAP_HASH(dev, blockno) ((((dev)<<27)|(blockno))%NBUFMAP)

struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  struct buf bufmap[NBUFMAP];
  struct spinlock bufmap_lock[NBUFMAP];
} bcache;
```

2. 修改`binit()`，最开始将所有缓存分配在桶0：

```
void
binit(void)
{
  char lockname[17];
  for(int i = 0; i < NBUFMAP; i++){
    snprintf(lockname, sizeof(lockname), "bcache_bufmap_%d", i);
    initlock(&bcache.bufmap_lock[i], lockname);
    bcache.bufmap[i].next = 0;
  }

  for(int i = 0; i < NBUF; i++){
    struct buf *b = &bcache.buf[i];
    snprintf(lockname, sizeof(lockname), "buffer_%d", i);
    initsleeplock(&b->lock, lockname);
    b->lru = 0;
    b->refcnt = 0;
    b->next = bcache.bufmap[0].next;
    bcache.bufmap[0].next = b;
  }

  initlock(&bcache.lock, "bcache");
}
```

分配块内存时采用LRU算法，在之前用链表存储时直接取表尾即可，但对于新的`bcache`设计，需要在`struct buf`中加入`lru`字段记录：

```
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
  uint lru;
};
```

3. 修改`brelse()`，修改`lru`为最后使用的时间，即`ticks`：

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

  releasesleep(&b->lock);

  uint key = NBUFMAP_HASH(b->dev, b->blockno);

  acquire(&bcache.bufmap_lock[key]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->lru = ticks;
  }

  release(&bcache.bufmap_lock[key]);
}
```

4. 修改`bpin()`和`bunpin()`：

```
void
bpin(struct buf *b) {
  uint key = NBUFMAP_HASH(b->dev, b->blockno);
  acquire(&bcache.bufmap_lock[key]);
  b->refcnt++;
  release(&bcache.bufmap_lock[key]);
}

void
bunpin(struct buf *b) {
  uint key = NBUFMAP_HASH(b->dev, b->blockno);
  acquire(&bcache.bufmap_lock[key]);
  b->refcnt--;
  release(&bcache.bufmap_lock[key]);
}
```

5. 修改`bget()`，若没有被缓存，则用LRU算法找到最近最久未使用的桶，并进行更新。在这个过程中要注意避免死锁情况：
    - 先查找是否已在缓存区内：

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  uint key = NBUFMAP_HASH(dev, blockno);
  acquire(&bcache.bufmap_lock[key]);

  // Is the block already cached?
  for(b = bcache.bufmap[key].next; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
```

```
      release(&bcache.bufmap_lock[key]);
      acquiresleep(&b->lock);
      return b;
    }
  }
  ...
```

- 不在缓冲区内，为了避免死锁，释放自己的小锁并上大锁，随后再检查一遍是否被修改：

```
...
  // Not cached.
  // 为了避免死锁，先释放自己的小锁，然后上大锁
  release(&bcache.bufmap_lock[key]);
  acquire(&bcache.lock);
  // 再次检查是否被修改了
  for(b = bcache.bufmap[key].next; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      acquire(&bcache.bufmap_lock[key]);
      b->refcnt++;
      release(&bcache.bufmap_lock[key]);
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
...
```

- 用LRU算法寻找最合适的桶，并进行更新：

```
...
  struct buf *last = 0;
  uint lru_key = -1;

  for(int i = 0; i < NBUFMAP; i++){
    acquire(&bcache.bufmap_lock[i]);
    int found = 0;
    for(b = &bcache.bufmap[i]; b->next; b = b->next){
      if(b->next->refcnt == 0 &&
        (!last || b->next->lru < last->next->lru)){
          last = b;
          found = 1;
      }
    }
    if(!found) // 没有找到更合适的选择（LRU），保持锁
      release(&bcache.bufmap_lock[i]);
    else{ // 找到更合适的，释放之前的锁并更新记录
      if(lru_key != -1) release(&bcache.bufmap_lock[lru_key]);
      lru_key = i;
    }
  }

  if(!last) panic("bget: no buffers");

  b = last->next;
```

```
  if(lru_key != key){
    last->next = b->next;
    release(&bcache.bufmap_lock[lru_key]);
    acquire(&bcache.bufmap_lock[key]);
    b->next = bcache.bufmap[key].next;
    bcache.bufmap[key].next = b;
  }
  b->dev = dev;
  b->blockno = blockno;
  b->refcnt = 1; // 第二次检查时有refcnt++
  b->valid = 0;
  release(&bcache.bufmap_lock[key]);
  release(&bcache.lock);
  acquiresleep(&b->lock);

  return b;
}
```

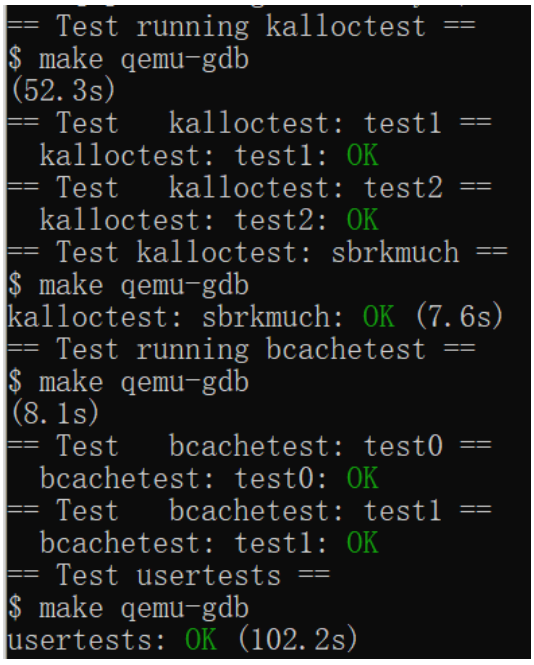
3) 问题与解决方法

1. 为什么不能像上一个实验一样通过将缓存分配到不同核心的方法：分配页帧和回收页帧的时候，只需要有一个核心参与，而且分配后某个页帧只会被一个进程访问。但分配出去的快缓存很有可能会被多个不同进程访问，若进程需要的缓存不属于本核心，就需要遍历查找，导致性能下降。
2. 如果仅仅是每个桶有自己的一个锁，会出现死锁情况：桶1中的块试图获得桶2的锁，桶2中的试图获得桶1的锁，循环等待造成死锁。

   解决以上死锁的一个方法是：偷取之前释放掉自己的锁。
3. 如果偷取之前释放掉自己的锁，则会使第一遍查找是否已被缓存的过程失去锁的保护，另一个CPU可能会在锁释放后访问同一个`blockno`，进而导致一个块中有两份缓存。

   解决方法可以是牺牲一些并行性，保留`bcache`原有的大锁，在释放小锁准备偷取前获取大锁（先获取大锁后释放小锁也会导致死锁），随后再次遍历查找`blockno`对应的缓存是否已存在，若存在则直接返回，不存在则用LRU算法进行后续查找和更新。这样一来保证了不会出现重复缓存的问题，但会导致原本并行的过程变为串行的，且在每次`cache`不命中时，会多一次额外的遍历开销。

3. 测试结果



Lab9: File System

1. Large files

1) 实验目的

目前，xv6文件限制为268个块，即`268*BSIZE`字节（在xv6中`BSIZE`为1024）。此限制是因为：一个xv6 `inode`包含12个"直接"块号和一个"一级间接"块号（`singly-indirect block`，指一个最多可容纳256个块号的块），总共12+256=268个块。

更改xv6文件系统代码，以支持每个`inode`中可包含256个"一级间接"块地址的"二级间接"块（`doubly-indirect block`），每个一级间接块最多可以包含256个数据块地址。结果将是一个文件将能够包含多达65803（256*256+256+11）个块。

2) 实验步骤

1. 为了加入二级间接索引，需要修改原本的宏定义：

```
// fs.h
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
```

2. 由于将一个直接块改为了二级间接块，需要对应修改`inode`和`dinode`：

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

3. 仿照对一级索引的操作，为`bmap()`新增支持二级索引：

```
static uint
bmap(struct inode *ip, uint bn)
{
  ...
  if(bn < NINDIRECT){
    ...
  }
  bn -= NINDIRECT;

  if(bn < NDINDIRECT){
    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 index_1 = bn / NINDIRECT; // bn对应的一级间接块下标
    uint index_2 = bn % NINDIRECT; // bn对应的二级间接块下标
    if((addr = a[index_1]) == 0){  // 检查一级块是否存在
      a[index_1] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    if((addr = a[index_2]) == 0){
      a[index_2] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

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

4. 修改`itrunc()`：

```
void
itrunc(struct inode *ip)
{
  ...

  if(ip->addrs[NDIRECT + 1]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){ // 遍历一级间接块
      if(a[i]){ // 如果此一级块有数据，遍历其二级块
        struct buf *bp2 = bread(ip->dev, a[i]);
        uint *a2 = (uint*)bp2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(a2[j])
            bfree(ip->dev, a2[j]);
        }
        brelse(bp2);
        bfree(ip->dev, a[i]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
  }

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

3) 问题与解决方法

1. 原本磁盘索引节点的格式由`fs.h`中的`struct dinode`定义：

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

   `addrs`维护文件的实际存储位置，前12（`NDIRECT`）个直接指向文件存储的块，最后一个间接块储存指向其他位置（实际存储数据的块）的指针。

2. `bmap()`：接收一个`inode`指针和`bn`，`bn`表示`inode`中的第几个块，返回对应的块号。
   - 如果`bn`小于`NDIRECT`，说明数据块在`inode`的直接块内。检查`inode`的`addrs[bn]`，如果该地址为0，表示对应的数据块尚未分配，调用`balloc()`函数进行分配，并将得到的地址赋给`addrs[bn]`。
   - 如果`bn`不小于`NDIRECT`，但小于`NDIRECT + NINDIRECT`，说明数据块在`inode`的间接块内。检查`inode`的`addrs[NDIRECT]`，如果该地址为0，表示间接块尚未分配，调用`balloc()`函数进行分配，并将得到的地址赋给`addrs[NDIRECT]`。然后，从磁盘读取该间接块（`bp = bread(ip->dev, addr)`），检查其中的第`bn - NDIRECT`个块地址，如果为0，表示对应的数据块尚未分配，调用`balloc()`函数进行分配，并将得到的地址赋给对应位置。最后，调用`log_write(bp)`标志这个块被修改了，随后会更新到磁盘的日志区盘，并用`brelse()`释放已读取的间接块缓存。
   - 如果`bn`超出了有效范围，即大于等于`NDIRECT + NINDIRECT`，则发生了错误，调用`panic()`函数终止程序。
3. `itrunc()`：通过反复调用`bfree()`和`brelse()`来清理`inode`，前者释放磁盘块，后者释放块缓存。
4. `bigfile`结果为523：523=11+256+256，所以是把二级间接块也当作一级间接块了，经检查发现是宏定义`MAXFILE`定义错误，修改为正确大小即可。

2. Symbolic links

1) 实验目的

在本练习中，向xv6添加符号链接。符号链接（或软链接）是指按路径名链接的文件；当一个符号链接打开时，内核跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管xv6不支持多个设备，但实现此系统调用是了解路径名查找工作原理的一个很好的练习。

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) 实验步骤

1. 首先，与Lab2类似，为`symlink`创建一个新的系统调用号，在***user/usys.pl***、***user/user.h***中添加一个条目，在***kernel/syscall.c***、***kernel/syscall.h***中添加相关内容：

```
// user/usys.pl
...
entry("symlink");
```

```
// user/user.h
// system calls
...
int symlink(char *, char *);
```

```
// kernel/syscall.h
...
#define SYS_symlink 22
```

```
// kernel/syscall.c
...
extern uint64 sys_symlink(void);
static uint64 (*syscalls[])(void) = {
...,
[SYS_symlink] sys_symlink,
};
```

2. 按照提示，添加宏定义：新的文件类型（`T_SYMLINK`）以表示符号链接，和新标志位（`O_NOFOLLOW`）以标志不递归打开软链接路径：

```
// kernel/stat.h
#define T_SYMLINK 4   // 软链接
// kernel/fcntl.h
#define O_NOFOLLOW 0x800
```

3. 完成`sys_symlink()`，创建一个`inode`并向其中写入`target`路径：

```
// kernel/sysfile.c
uint64
sys_symlink(void)
{
  char target[MAXPATH], path[MAXPATH];
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;

  begin_op();
  // 创建一个文件
  struct inode *ip = create(path, T_SYMLINK, 0, 0);
  if(ip == 0){
    end_op();
    return -1;
  }
  // 向文件中写入target路径
```

```
  if(writei(ip, 0, (uint64)target, 0, strlen(target)) < strlen(target)){
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlockput(ip);
  end_op();
  return 0;
}
```

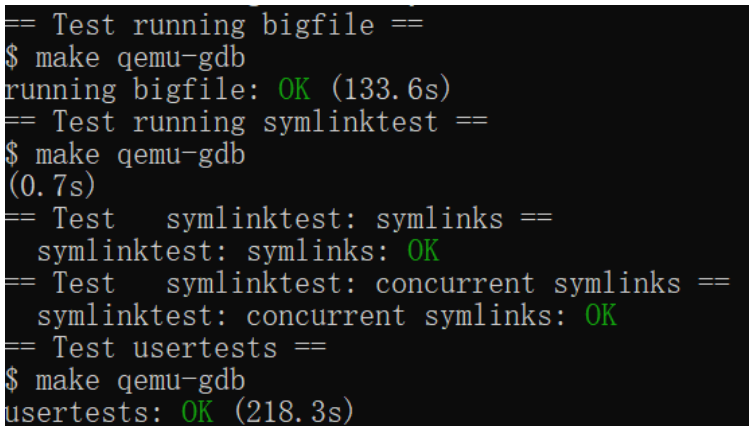
4. 修改`sys_open()`以支持打开软链接：

```
// kernel/sysfile.c
uint64
sys_open(void)
{
  ...
  // 处理符号链接
  if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
    int MAX_DEPTH = 10;
    for(int i = 0; i < MAX_DEPTH; i++){
      if(readi(ip, 0, (uint64)path, 0, strlen(path)) != strlen(path)){
        iunlockput(ip);
        end_op();
        return -1;
      }
      iunlockput(ip);
      ip = namei(path);
      if(ip == 0){
        end_op();
        return -1;
      }
      ilock(ip);
      if(ip->type != T_SYMLINK)
        break;
    }
    // 超出最大深度
    if(ip->type == T_SYMLINK){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }
  ...
}
```

3) 问题与解决方法

1. `create()`返回的`inode`是加锁的，需要用`iunlockput()`对其解锁，这一操作会使其引用计数减一，类似于之前的实验。
2. 对于`inode.type`的检查要在`ilock()`之后，因为`ilock()`中检查`inode.valid`，如果为0则读取磁盘将数据加载进`inode`，如果顺序调换有可能访问到空的`inode`。

3. 测试结果



Lab10: Mmap

1. Mmap

1) 实验目的

将把`mmap`和`munmap`添加到xv6中，它们可用于在进程之间共享内存，将文件映射到进程地址空间，并作为用户级页面错误方案的一部分。

`mmap`的声明如下：

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

映射描述符为`fd`的文件的前`length`个字节，到`addr`加上`offset`偏移量的位置，返回该地址，如果失败则返回`0xffffffffffffffff`。`addr`为0意味着内核应该决定映射文件的虚拟地址。`prot`指示内存是否应映射为可读、可写，可执行的。`flags`是`MAP_SHARED`（映射内存的修改应写回文件），或`MAP_PRIVATE`（映射内存的修改不应写回文件）。

`munmap(addr, length)`应删除指定地址范围内的`mmap`映射。如果进程修改了内存并将其映射为`MAP_SHARED`，则应首先将修改写入文件。但取消映射的位置要么在区域起始位置，要么在区域结束位置，要么就是整个区域(但不会在区域中间“打洞”)。

2) 实验步骤

1. 添加`mmap`、`munmap`系统调用以及`mmaptest`，与上个实验相同。
2. 根据提示，定义VMA（虚拟内存区域）对应的结构体，记录`mmap`创建的虚拟内存范围的地址、长度、权限、文件等。添加进进程结构体的VMA数组大小为16：

```
// kernel/proc.h
#define NVMA 16
struct mmap_vma {
  int used;           // 是否已被使用
  uint64 addr;        // 起始地址
  int length;         // 长度
  int prot;           // 权限
  int flags;          // 标志位
  int fd;             // 文件描述符
  struct file* file;  // 文件
```

```
  int offset;         // 偏移
};

// Per-process state
struct proc {
  ...
  struct mmap_vma vma[NVMA];   // 虚拟内存
};
```

全部初始化为0：

```
// kernel/proc.c
static struct proc*
allocproc(void)
{
  ...
  memset(&p->vma, 0, sizeof(p->vma));
  return p;
}
```

3. 实现`mmap()`：找到一个未使用的区域来映射文件，并将VMA添加到进程的映射区域表中，并用`filedup()`增加文件的引用计数，以便在文件关闭时结构体不会消失：

```
// kernel/sysfile.c
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  struct file *file;
  uint64 err = 0xffffffffffffffff;
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0
    || argint(2, &prot) < 0 || argint(3, &flags) < 0
    || argfd(4, &fd, &file) < 0 || argint(5, &offset) < 0)
    return err;
  // 本实验假定addr和offset为0
  if(addr != 0 || offset != 0 || length < 0)
    return err;
  // 若文件不可写，则不允许有写权限时映射为共享
  if((!file->writable) && (prot & PROT_WRITE) && (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;

  for(int i = 0; i < NVMA; i++){
    if(!p->vma[i].used){ // VMA未被使用
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].length = length;
      p->vma[i].prot = prot;
      p->vma[i].flags = flags;
```

```
      p->vma[i].file = file;
      p->vma[i].fd = fd;
      p->vma[i].offset = offset;
      filedup(file); // 增加引用计数
      p->sz += length;
      return p->vma[i].addr;
    }
  }

  return err;
}
```

4. 此时会产生页面错误，需要在`usertrap()`中处理，分配一页物理内存，将4096字节的相关文件读入该页面，并将其映射到用户地址空间：

```
// kernel/trap.c
void
usertrap(void)
{
  ...
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 13 || r_scause() == 15){
    if(mmap_handler(r_stval()) != 0)
      p->killed = 1;
  } else {
    ...
  }
 ...
}
```

```
// kernel/sysfile.c
// 处理mmap惰性分配导致的页面错误
int mmap_handler(uint64 addr)
{
  struct proc *p = myproc();
  // 寻找属于哪一个VMA
  int i;
  for(i = 0; i< NVMA; i++){
    if(p->vma[i].used && p->vma[i].addr <= addr
      && addr <= p->vma[i].addr + p->vma[i].length -1)
      break;
  }
  if(i == NVMA) return -1;

  struct mmap_vma *cur_vma = p->vma + i;

  if(r_scause() == 13 && (!cur_vma->file->readable) && (cur_vma->flags &
MAP_SHARED)){
    printf("mmap_handler:not readable\n");
    return -1;
  }
  if(r_scause() == 15 && (!cur_vma->file->writable) && (cur_vma->flags &
MAP_SHARED)){
```

```
    printf("mmap_handler:not writable\n");
    return -1;
  }

  void *pa = kalloc();
  if(pa == 0) return -1;
  memset(pa, 0, PGSIZE);

  // 读取文件
  ilock(cur_vma->file->ip);
  int offset = cur_vma->offset + PGROUNDDOWN(addr - cur_vma->addr);
  if(readi(cur_vma->file->ip, 0, (uint64)pa, offset, PGSIZE) == 0){
    iunlock(cur_vma->file->ip);
    kfree(pa);
    return -1;
  }
  iunlock(cur_vma->file->ip);

  int pte_flags = PTE_U;
  if(cur_vma->prot & PROT_READ) pte_flags |= PTE_R;
  if(cur_vma->prot & PROT_WRITE) pte_flags |= PTE_W;
  if(cur_vma->prot & PROT_EXEC) pte_flags |= PTE_X;
  // 添加页面映射
  if(mappages(p->pagetable, PGROUNDDOWN(addr),
    PGSIZE, (uint64)pa, pte_flags) != 0){
    kfree(pa);
    return -1;
  }

  return 0;
}
```

5. 实现`munmap()`：找到地址范围的VMA并用`uvmunmap()`取消映射指定页面。如果删除了先前`mmap()`的所有页面，应该减少相应`struct file`的引用计数。如果未映射的页面已被修改，并且文件已映射到`MAP_SHARED`，用`filewrite()`将页面写回该文件：

```
// kernel/sysfile.c
uint64
sys_munmap(void)
{
  uint64 addr;
  int length;
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0)
    return -1;

  struct proc *p = myproc();
  int i;
  for(i = 0; i < NVMA; i++){
    if(p->vma[i].used && p->vma[i].length >= length){
      if(p->vma[i].addr == addr){ // 起始位置
        p->vma[i].addr += length;
        p->vma[i].length -= length;
        break;
      }
      if(p->vma[i].addr + p->vma[i].length == addr + length){ // 末尾
```

```
        p->vma[i].length -= length;
        break;
      }
    }
  }
  if(i == NVMA) return -1;

  // 将MAP_SHARED页面写回
  if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0){
    filewrite(p->vma[i].file, addr, length);
  }
  uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
  // 当前VMA中页面映射全部被取消
  if(p->vma[i].length == 0){
    fileclose(p->vma[i].file);
    p->vma[i].used = 0;
  }

  return 0;
}
```

6. 理想情况下，将只写回程序实际修改的`MAP_SHARED`页面。RISC-V PTE中的脏位（`D`）表示是否已写入页面。但是，`mmaptest`不检查非脏页是否没有回写；因此，可以不用看`PTE_D`位就写回页面。但，如果对惰性分配的页面调用了`uvmunmap()`，或者子进程在`fork()`中调用`uvmcopy()`复制了父进程惰性分配的页面都会导致`panic`，因此需要修改检查`PTE_V`后不再`panic`：

```
// kernel/vm.c
if((*pte & PTE_V) == 0)
  //panic("uvmcopy: page not present");
  continue;
```

7. 修改`exit()`将进程的已映射区域取消映射：

```
// kernel/proc.c
void
exit(int status)
{
  ...
  // 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].file, p->vma[i].addr, p->vma[i].length);
      }
      fileclose(p->vma[i].file);
      uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length / PGSIZE, 1);
      p->vma[i].used = 0;
    }
  }
  ...
}
```

8. 修改`fork()`以确保子对象具有与父对象相同的映射区域。增加VMA的`struct file`的引用计数：

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

  // increment reference counts on open file descriptors.
  ...
  // 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].file);
    }
  }
  ...
}
```

3) 问题与解决方法

1. `struct file`的定义写在`file.h`中，但`defs.h`中只声明了`struct file`的存在，导致在`trap.c`中使用时出现"不允许指针指向不完整的类类型"错误。我的解决方法是将`mmap_handler()`移到`sysfile.c`，并在`defs.h`添加引用。
2. 为什么`munmap()`在`exit()`而非`freeproc()`中：`freeproc()`实际是在`wait()`中被调用以释放进程的，如果父进程不调用`wait()`那么这些文件就不会被写回文件。

2. 测试结果

```
== Test running mmaptest ==
$ make qemu-gdb
(3.7s)
== 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 (106.6s)
```