XV6 实验报告

2253984 施帅乾

目录

Lab1: Xv6 and Unix utilities

Boot xv6

一、实验目的

切换到 xv6-labs-2021 代码的 util 分支。

二、实验步骤

1. 下载 WSL 与 ubuntu 20.04，配置好环境。
2. 将 xv6-labs-2021 代码克隆到本地，进入到相应的目录，建立自己的分支。

```
git clone git://g.csail.mit.edu/xv6-labs-2021
cd xv6-labs-2021
git checkout util
```

3. 使用 make qemu 命令来实现 xv6 操作系统的启动。

三、实验中遇到的问题和解决办法

无

四、实验心得

通过这个实验，我配置好了实验的环境，并学习了如何启动 xv6 操作系统，为接下来的实验项目做好准备。

sleep

一、实验目的

实现 xv6 的 UNIX 程序 sleep：暂停到用户指定的计时数

二、实验步骤

1. 在 user 目录下，创建一个名为 sleep.c 的文件。
2. 程序接受一个参数，使当前进程暂停相应的 ticks 数量。在该程序中，如果用户提供错误参数量，程序应该打印出错误信息。使用系统提供的函数 sleep 函数完成操作即可。
3. 将程序添加到 Makefile 的 UPROGS 中。
4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

Sleep 函数只接受 int 类型参数，使用 atoi 参数将 argv[1]转换为 int 类型。

四、实验心得

通过这个实验，我知道了 main 函数的两个参数的含义，如何添加新程序并测试。

argv 数组是主程序接收到的所有变量，其中 argv[0]无实际作用，实际参数从 argv[1]开始。argc 指参数个数（包括 argv[0]）。

pingpong

一、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通信一个字节，使用两个管道，每个方向一个。

二、实验步骤

1. 在 user 目录下，创建一个名为 pingpong.c 的文件。
2. 创建一对管道，用 fork()函数创建子进程。在父进程中向管道 1 中写入“ping”，等待子进程结束后读取管道 2 中的数值并将其打印，最后退出；在子进程中读取管道 1 内容并打印，再向管道 2 中写入“pong”，最后退出。
3. 将程序添加到 Makefile 的 UPROGS 中。
4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

无

四、实验心得

通过这个实验，我配置好了实验的环境，并学习了如何启动 xv6 操作系统，为接下来的实验项目做好准备。

管道相关操作：创建容量为 2 的 int 数组 p，用 pipe(p)创建管道。p[0]为标准输出端口，p[1]为标准输入端口使用 close 函数关闭当前进程中管道的端口。

primes

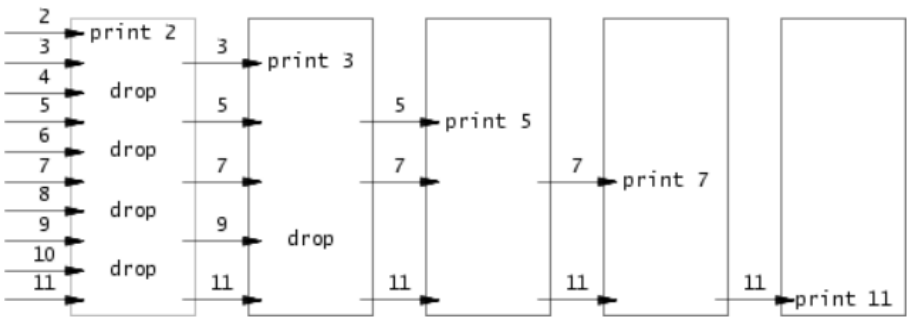
一、实验目的

使用管道编写 prime sieve(筛选素数)的并发版本。

二、实验步骤

思路：

主进程将 2-35 数推入 1 号进程(主进程的子进程)，1 号进程将第一个数打印，认为其是素数，并排除传进来的数中能被它整除的数，将剩下的数传到 2 号进程(1 号进程的子进程)，重复之前过程，直到没有数剩余。



1. 在 user 目录下，创建一个名为 primes.c 的文件。
2. 主要操作为一个递归函数 prime_treat：

   父进程中，将数据写入一条管道 p，关闭该管道的输入与输出接口并创建一个子进程，这个子进程关闭 p 的输入接口，通过输出接口逐个读取管道内数据，打印第一个数据 n 并筛去能整除此数的数据，将剩余数据写入一条新开设的管道 new_p。重复以上步骤直至 read 操作返回值为 0，即没有剩余数据。
3. 将程序添加到 Makefile 的 UPROGS 中。
4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

输出乱序：需要合理地关闭不需要的端口以及设置 wait 时机。

四、实验心得

通过这个实验，我加深了对管道原理与其相关操作的理解，学习了进程间的通信方式与读写操作。

find

一、实验目的

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

二、实验步骤

思路：

参照 ls 操作(打印当前目录下所有文件与文件夹名)的方法，在其基础上对文件与文件夹采取不同的操作：文件：对比文件名与要找的名字，匹配则打印；文件夹：进入文件夹路径，再次执行 find 操作，目录树全部查找完后停止。(递归)

匹配函数可以调用 grep.c 中的 match 方法。

1. 在 user 目录下，创建一个名为 find.c 的文件。
2. 先打开给的路径，如果有问题就报错，接着查看当前操作的 st 路径类型：若是文件就调用 match 函数判断是否匹配，是则打印路径；若是文件夹且路径名不为"."或".."，就进入此文件夹，并递归地在该文件夹中查找文件。
3. 将程序添加到 Makefile 的 UPROGS 中。
4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

无

四、实验心得

通过这个实验，我理解了 xv6 操作系统的目录格式与其相关参数、操作，加强了对递归思路的使用，熟悉了文件结构。

xargs

一、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。

二、实验步骤

思路：

|xargs 前的命令结果会由标准输出端口输出至命令行中，使用标准输入端口可以获取这些输出，输出结果由换行分隔。

将这些输出结果切片，获得的切片就是|xargs 后的命令的额外参数。此时传入参数执行即可。

1. 在 user 目录下，创建一个名为 xargs.c 的文件。
2. 通过标准输入端口 0 一个一个读入字符，当字符为'\n'时，说明一个额外参数已经出现，此时新建一个参数列表 new_argv，new_argv[0]=argv[1]，new_argv[1]=argv[2]，new_argv[2]=额外参数，使用 exec(argv[1],new_argv)对这个额外的参数调用命令。重复以上操作直至无法读入字符。
3. 将程序添加到 Makefile 的 UPROGS 中。
4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

读取参数：一开始我试图直接读入一整行参数，但是这样会造成数组空间的浪费，于是我一个一个字符读取，避免这些问题。

四、实验心得

通过这个实验，我学习了 exec 函数的使用，更加熟悉了标准输入输出的处理和进程的创建与执行。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (2.9s)
== Test sleep, returns == sleep, returns: OK (1.7s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.7s)
== Test pingpong == pingpong: OK (1.9s)
== Test primes == primes: OK (2.0s)
== Test find, in current directory == find, in current directory: OK (2.1s)
== Test find, recursive == find, recursive: OK (2.2s)
== Test xargs == xargs: OK (2.3s)
== Test time ==
time: OK
Score: 100/100
```

Lab 2: System Calls

System call tracking

一、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。

二、实验步骤

1. 修复编译问题。

   在 user/user.h 中添加 int trace(int);，声明 trace 的系统调用的原型。

   在 user/usys.pl 中添加存根 entry("trace")。

   在 kernel/syscall.h 中添加系统调用号#define SYS_trace 22。

   在 kernel/syscall.c 中声明 sys_trace 系统调用并添加至系统调用列表中。

2. 在 kernel/sysproc.c 中添加一个 sys_trace()函数。

   在结构体 proc 中加入参数 trace_mask 并实例化为 p，将其设置为参数 mask。

```
uint64
sys_trace(void)
{
  int mask;
  if(argint(0, &mask) < 0)
    return -1;

  struct proc *p= myproc();
  p->trace_mask=mask;
  return 0;
}
```

3. 在 kernel/proc.c 中，修改 fork()。

   实例化结构体 proc，添加一条语句将 p 的 trace_mask 赋值给 np，即从父进程复制给子进程。

4. 修改 kernel/syscall.c 中的 syscall()函数以打印跟踪输出。

   找到系统调用号后，根据 trace_mask=1 << 系统调用号 确认 trace_mask 是否正确。

```
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]();

    int trace_mask=p->trace_mask;
    if((trace_mask>>num)&1){
      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;
  }
}
```

5. 将程序添加到 Makefile 的 UPROGS 中。
6. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

无法切换分支：从 lab1 切换至 lab2 时，使用 git checkout 指令切换分支后，lab1 的内容仍然存在且 commit 也有问题。我保存 lab1 代码后重新部署了一次环境，并且创建 github 仓库，使用 git remote add 指令将其与 wsl 上的 xv6-labs-2021 连接，处理完一个 lab 后先 commit 一次，再用 git push 命令传到 github 仓库中，此时切换分支就没有问题。

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = git://g.csail.mit.edu/xv6-labs-2021
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "util"]
        remote = origin
        merge = refs/heads/util
[remote "github"]
        url = https://github.com/CH4MBER/XV6_Labs.git
        fetch = +refs/heads/*:refs/remotes/github/*
[branch "syscall"]
        remote = origin
        merge = refs/heads/syscall
```

如图，添加了远程连接[remote “github”]。

编译出错：修复编译问题多次还是有问题，后来发现 syscall.c 中不能只在 syscall 数组中添加一条数据，在上方要添加 extern uint64 sys_trace(void)的声明。

参数无法传递：函数没有形参，难以直接传递参数。参照 hints 中的提示在 proc(即进程控制块)中添加新参数，让参数能够通过进程进行传递。核心代码大致如下：

传递参数：

```
struct proc *p= myproc();
p->trace_mask=mask;
```

接收参数：

```
np->trace_mask=p->trace_mask;
```

四、实验心得

通过这次实验，我学习了系统调用的工作原理，以及如何添加新的系统调用。在创建系统调用时，首先应该将其声明好，防止编译出错，其次由于底层代码与平常我写的代码不一样，参数传递等基础功能需要另辟蹊径，细心参考给的提示并查阅网络上的相关资料就可以解决问题。此外我还理解了 git 的相关技巧，解决了托管问题。

System call tracking

一、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。

二、实验步骤

1. 修复编译问题。

   在 user/user.h 中添加 int sysinfo (struct sysinfo *);，声明 sysinfo 的系统调用的原型。

   在 user/usys.pl 中添加存根 entry("sysinfo")。

   在 kernel/syscall.h 中添加系统调用号#define SYS_sysinfo 23。

   在 kernel/syscall.c 中声明 sys_sysinfo 系统调用并添加至系统调用列表中。

2. 在 kernel/sysproc.c 中添加一个 sys_sysinfo()函数。

   参照 kernel/sysfile.c 中的 sys_fstat()和 kernel/file.c 中的 filestat()使用 copyout()将一个 struct sysinfo 复制回用户空间。

```
uint64
sys_sysinfo(void)
{
  struct sysinfo info;
  uint64 addr;
  struct proc *p= myproc();

  info.freemem=get_freemem();
  info.nproc=get_processnum();

  if(argaddr(0, &addr) < 0)
    return -1;

  if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
      return -1;

  return 0;
}
```

3. 在 kernel/kalloc.c 中，添加一个函数用于获取空闲字节数

```
uint64 get_freemem(void)
{
  struct run *r;
  uint64 num=0;
  acquire(&kmem.lock);
  r = kmem.freelist;

  while(r){
    r = r->next;
    num++;
  }
  release(&kmem.lock);
  return num*PGSIZE;

}
```

4. 在 kernel/proc.c 中，添加一个函数用于获取进程数。

```
uint64 get_processnum(void)
{
  uint64 num = 0;
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state != UNUSED)
      num++;
    release(&p->lock);
  }
  return num;
}
```

5. 将程序添加到 Makefile 的 UPROGS 中。
6. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

Copyout 的使用：copyout 函数参数众多，难以理解其意义，在深入研究后可以发现在 sys_fstat 函数中,先用 argfd(0, 0, &f)于 argddr(1,&st)获取 f 与 st 这两个参数，这两个函数的第一个参数指系统调用的第 n 个参数, 在这里获取了文件指针与 stat 地址, 调用 filestat(f,st)。在 filestat 函数中，语句 copyout(p->pagetable, addr, (char *)&st, sizeof(st))中并没有 f，所以可以将其忽略, 其意义是在 p 进程的页表中将输入的内核地址 st 复制给用户控件地址 addr。如果想要复制结构体 info 就用它替换 st 即可。

获取空闲字节数：观察 kalloc.c 的结构体 run，可以发现这是一个链表，而 kmem 装着这个链表的头指针与锁。循环获取当前节点的下一个节点并计数，可以得到节点数，结果乘以节点大小 PGSIZE 就可以得到答案。

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

获取进程数：观察 proc.c 可以发现进程的结构体都存放在在 proc 列表中，遍历其每一个元素，统计 state 参数不为 UNUSED 的进程数，返回最终结果即可。

锁：在解决上两个问题时，我发现文件中的函数几乎都用到了 aquire 与 release 两个函数，我没有加这两个函数的时候 info 的两个参数有时会有错误。加上这两个函数后就是创建一个临界区，防止其他进程打断操作，保证了数据的准确性。

四、实验心得

在这次实验中，我学习了如何添加新的系统调用。从逻辑声明解决编译问题到连接用户空间与内核空间，再到如何在内核中实现这个系统调用功能。这个过程让我理解了系统调用的工作机制。

相比于 lab1，lab2 更加考察了我对底层代码的阅读理解能力，在整个实验过程中，我看了许多已有文件中的代码，将这些相关代码与问题联系起来，完成编程。这个过程比较困难，但是我从中学到了很多。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.4s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (0.9s)
== Test trace children == trace children: OK (11.4s)
== Test sysinfotest == sysinfotest: OK (3.1s)
== Test time ==
time: OK
Score: 35/35
```

Lab3: page tables

Speed up system calls

一、实验目的

一些操作系统通过在用户空间和内核之间共享只读区域的数据来加速某些系统调用。这消除了执行这些系统调用时进行内核交叉的需要。本实验是为 xv6 的 getpid() 系统调用实现这种优化，进而加速系统调用。

二、实验步骤

1. 在 kernel/proc.c 中的 proc_pagetable() 中添加虚拟页的映射关系。

先在 proc 结构体中添加新的参数：usyscall 结构体。

使用 mappages 函数进行映射，注意在未成功时释放先前的映射（如 TRAMPOLINE）。

在 proc_freepagetable 函数中相应的添加释放指令。

初始化 pid，将 usyscall 页面的 pid 定义为当前进程的 pid。

```
if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, 0);
  return 0;
}
p->usyscall->pid=p->pid;
```

```
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. 在 allocproc() 函数中为页面进行初始化。

使用 kalloc 函数进行分配，如果失败(返回 0)要释放内存与相关的锁。

```
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
```

3.在 freeproc() 中释放页面。

如果指针不为空就用 kfree 函数释放资源，重置指针。

```
if(p->usyscall)
  kfree((void*)p->usyscall);
p->usyscall = 0;
```

4. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

权限控制：映射 USYSCALL 时，我复制了上面的映射 TRAPFRAME 的代码并将 trampframe 改为 syscall，但是在实际测试过程中发现有错误，经过多重对比研究发现不同的映射的最后一个参数(包含 PTE _字段)都不一样，查阅资料后发现这种字段代表了权限，仔细读题后发现 USYSCALL 给予的是只读权限，修改为 PTE_R | PTE_U 后测试正确。

四、实验心得

通过本次实验，我了解了操作系统如何通过在用户空间和内核之间共享只读区域的数据来加速某些系统调用的过程。了解了内存的布局方式以及如何在内核中添加新的内存映射。

在实验的过程中，我遇到权限相关问题。通过查阅资料与个人研究后发现建立内存映射时使用了错误的权限。我也在解决这个问题的过程周学习到了更多的相关知识。

Print a page table

一、实验目的

为了帮助我们更好地理解 RISC-V 的页表，我们将编写一个函数来打印页表的内容，实现页表可视化，这将有助于我们对页表的理解和未来的调试工作。

二、实验步骤

1. 在 kernel/vm.c 中创建函数 vmprint() 接受一个 pagetable_t 参数，并按照格式打印页表。
   在 defs.h 中声明 void vmprint(pagetable_t pagetable)，解决编译问题。
   参考 freewalk 函数，使用递归的方法打印页表。

```
void printpte(pagetable_t pagetable,int depth)
{
 for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){
      for(int j=0;j<depth-1;j++)
      {
        printf(".. ");
      }
      printf("..");

      uint64 child = PTE2PA(pte);

      printf("%d: pte %p pa %p\n",i,pte,child);



      if((pte & (PTE_R|PTE_W|PTE_X)) == 0)
      {
        printpte((pagetable_t)child,depth+1);
      }
    }
  }

}

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

2. 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。

3. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

递归条件：pte & PTE_V 为真，说明是有效的页表，根据 depth 打印其深度与页表信息（使用%p），PTE_R|PTE_W|PTE_X 为真，说明页表在更第一级，令 depth+1 开始递归调用。

四、实验心得

在这个实验中，我学习了页表的相关知识，比如页表条目（PTE）的结构、页表的层级结构、以及页表的地址转换等等。通过学习理解其他函数，使用递归来遍历页表的所有条目，并打印出它们的信息。

Detecting which pages have been accessed

一、实验目的

某些自动内存管理机制，如垃圾收集器，可以从页面访问信息中受益。在这部分实验中，我们将通过检查 RISC-V 页表中的访问位，向用户空间报告哪些页面已被访问。

二、实验步骤

1. 在 kernel/sysproc.c 中实现 sys_pgaccess() 函数。

观察 user/pgtlbtest.c 中的 pgaccess_test() 函数，了解 pgaccess 系统调用的必要参数（3 个：起始地址,查找页数,位图地址）。

```
if (pgaccess(buf, 32, &abits) < 0)
```

使用 argaddr() 和 argint() 函数接收到的系统调用的参数。

```
if(argaddr(0, &addr) < 0)
  return -1;

if(argint(1, &n) < 0)
  return -1;
if(argint(2, &bitmask) < 0)
  return -1;
```

计算 bitmask。首先声明访问位 PTE_A，参考 walk 函数新建一个 vmpgaccess 函数完成对 PTE 的查找工作，访问位被设置过时说明改页被访问过，输出这些页面。

```
#define PTE_A (1L << 6)
```

```c
int
vmpgaccess(pagetable_t pagetable, uint64 va){
  pte_t *pte;

  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  if(pte == 0){
    return 0;
  }
  if((*pte & PTE_A) != 0){
    *pte = *pte & (~PTE_A);// clear 6th flag (PTE_A)
    return 1;
  }


  return 0;
}
```

将所有的输出掩码通过 copyout 函数复制到用户空间。

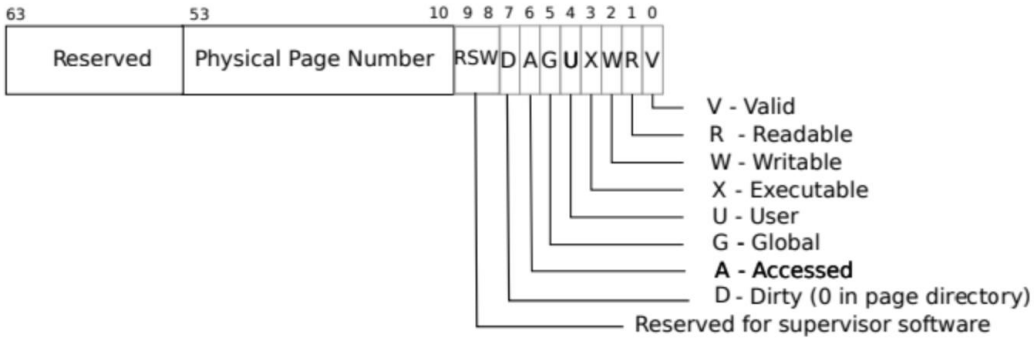
```c
int res=0;

struct proc *p=myproc();

for(int i=0;i<n;i++){
  int va=addr+i*PGSIZE;
  int abit=vmpgaccess(p->pagetable,va);
  res=res|abit<<i;
}


if(copyout(p->pagetable, bitmask, (char*)&res, sizeof(res))<0){
  return -1;
}
```

访问位置零。

3. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

标志位问题：在声明 PTE_A 时，我认为值不重要，直接参照上方的 PTE_U (1L << 4)为 PTE_A 设置为(1L << 5)，但是测试结果出现错误。经过查阅资料，发现 PTE 具有特定意义的位置会被固定在一个位置，而 A 对应的位置时第第 6 位，所以应当改为(1L << 6)。



vmpgaccess 的设计：根据提示，使用 walk 函数进行对 pte 的查找，通过分析可以得知 walk 函数内置了对页表的三级映射的功能，将虚拟地址映射到物理地址上，所以传入参数 pagetale、虚拟地址 va 与 0(判定作用)后，会自动输出存在的 pte，让后再根据 PTE_A 的值筛选需要的 pte 并将其返回。

四、实验心得

这次的实验是一个困难且相对综合的实验，让我对于页表管理和虚拟内存的操作有了更加深入的了解。通了解到了如何用 walk 函数查找页面并通过检查 RISC-V 页表中的访问位来报告哪些页面已被访问，并加深了对 pte 地址结构的印象。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-pgtbl
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_PGTBL -DLAB_PGTBL -MD -mcmodel=medany -ffreestanding -fno-
common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie  -c -o kernel/vm.o kernel/vm.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/kalloc.o kernel/string.o kernel
/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o ker
nel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o
kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
== Test pgtbltest == (1.8s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests == (148.6s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

Lab4: traps

RISC-V assembly

一、实验目的

理解 RISC-V 汇编的基本概念。

二、实验步骤

1. 运行 make fs.img 指令，编译 user/call.c 程序，得到可读性比较强的 user/call.asm 文件。
2. 回答 6 个问题，并答案保存在 answers-traps.txt 中。

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

寄存器 a0~a7，13 存在 a2 中。

2.2 Where is the call to function `f` in the assembly code for main? Where is the call to `g`?

没有调用。 `g`被内联 inline 到 f(x) 中，然后 `f`又被进一步内联到 `main`中。

2.3 At what address is the function `printf` located?

0x0000000000000630

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

0x0000000000000038

2.5 Run the following code.

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

What is the output?

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

`HE110 World`;

反过来将 i 替换成 0x00726c64;

不需要，57616 的十六进制是 110，无论端序

2.6 In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

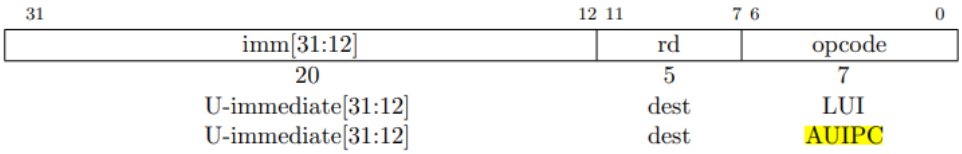
```
printf("x=%d y=%d", 3);
```

第二个参数没有指定，但是"y="后的参数存储在对应的寄存器 a2 中，因此会直接输出之前存在 a2 中的值。

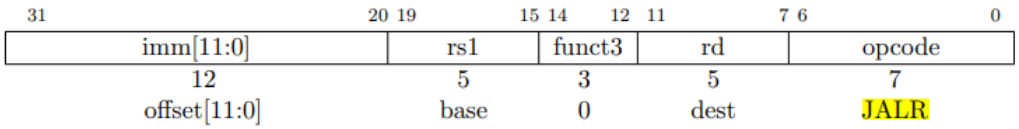
三、实验中遇到的问题和解决办法

auipc 指令与 jalr 指令的意义与用法：

auipc 指令格式：auipc rd imm，将高位立即数加到 PC 上，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，图中为 rd 寄存器



jalr 指令格式：jalr rd, offset(rs1)跳转并链接寄存器。jalr 指令会将当前 PC+4 保存在 rd 中，然后跳转到指定的偏移地址 offset(rs1)。



第四题中两条指令：

30: 00000097　　auipc ra,0x0

34: 600080e7　　jalr　1536(ra) # 630 <printf>

第一条指令从中地址 00000097 中提取 imm=0，左移 12 位仍然为 0，此时 PC=0x30，ra 最终存储的值为 0x30。

第一条指令从中地址 610080e7 中提取 imm=0x610，加上 ra 的值后=0x640，ra 最终存储的值为 pc+4=0x38。

小端存储与大端存储：一个十六进制数 i=0x00646c72，小端存储从右开始，按原有的数字顺序，每次选取两个数字，即为 72-6c-64-00，大端存储则为从左开始，按原有的数字顺序，每次选取两个数字，即为 00-64-6c-72。

四、实验心得

通过这次实验，我对 RISC-V 汇编的原理有了更深入的理解，包括了函数调用和参数传递等等过程。在我尝试理解某些汇编指令的含义时，我遇到了困难，通过查阅资料我最终解决了这些问题，我从中学到了许多。

Backtrace

一、实验目的

在 kernel/printf.c 中实现一个 backtrace()函数，该函数能够打印出当前调用栈上的函数调用列表。

二、实验步骤

1. 在 def.h 中声明 backtrace 函数。
2. 向 sysproc.c 的 sys_sleep 函数返回之前调用 backtrace() 函数;
3. 在 kernel/riscv.h 中加入如下函数，以便 backtrace() 能返回当前页指针;

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

4. 在 kernel/printf.c 中实现 backtrace() 函数。

   计算栈顶与栈底指针，便于终止循环。

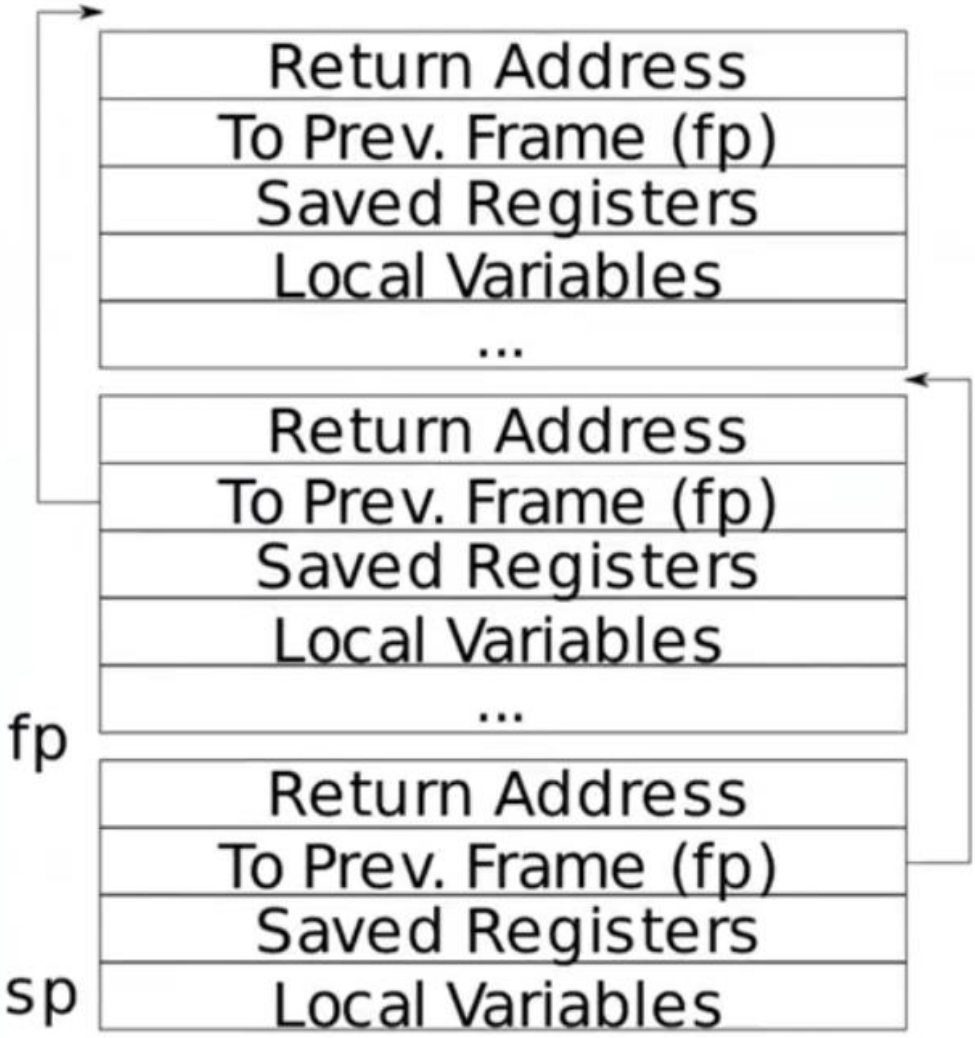
   打印每个栈页面的返回地址。

```
void
backtrace()
{
  printf("backtrace:\n");
  uint64 fp=r_fp();
  uint64 *frame = (uint64 *) fp;
  uint64 maxfp = PGROUNDUP(fp);
  uint64 minfp = PGROUNDDOWN(fp);

  while(minfp<fp&&fp<maxfp){
    printf("%p\n", frame[-1]);
    fp = frame[-2];
    frame = (uint64 *)fp;

  }

}
```

5. 在 kernel/printf.c 的 panic 中调用 backtrace() 函数。
6. 使用 make qemu 在 xv6 shell 中使用 bttest 测试运行该程序。

三、实验中遇到的问题和解决办法

函数调用栈：下图时函数调用栈的基本格式。



在图中，上方是高地址，下方是低地址，函数调用栈由高地址向低地址增长，每次增长一个帧（即图中的一个大矩形），sp 为栈顶指针，fp 为当前帧的底指针，在每一个帧中，都有返回地址（return address）、指向前一个帧的指针（to prev.frame）以及其他相关信息，帧指针(fp)往下偏移 8 个字节是函数返回地址 return address，往下偏移 16 个字节是上一个帧的帧指针。函数调用栈的整体结构类似链表，可以得知函数调用的顺序关系。

Backtrace 函数的实现方式就是通过 r_fp 指令获取当前帧指针，并在栈顶与栈底间通过链表方式循环获取下一帧，打印帧的返回地址。

四、实验心得

通过这次实验，我对函数调用栈和栈帧有了更深入的理解。理解了函数调用栈与栈帧的数据结构形式与调用方式，可以用来恢复调用者的执行环境。

Alarm

一、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，实现用户级中断/故障处理程序的一种初级形式。

二、实验步骤

1. 解决编译问题。

   在 user/user.h 里添加 int sigalarm(int ticks, void (*handler)())与 int sigreturn(void)声明。

   在 user/usys.pl 中添加存根 entry("sigalarm"); 与 entry("sigreturn")。

   在 kernel/syscall.h 中添加系统调用号#define SYS_sigalarm 22 与 SYS_sigreturn 23。

   在 kernel/syscall.c 中，声明 sys_sigalarm 与 sys_sigreturn 调用并添加至系统调用列表中。

   修改 Makefile，在 UPROGS 中加入$U/_alarmtest\ 。

2. 在 kernel/proc.h 的结构体 proc() 中增添必要属性。

   int alarm_interval; 记录间隔

   int alarm_ticks; 记录 tick 数量

   uint64 alarm_handler; 记录调用函数地址

   struct trapframe alarm_trapframe; 记录函数调用栈帧

3. 在 kernel/sysproc.c 中实现两个系统调用函数 sys_sigalarm() 与 sys_sigreturn() 。
4. 修改 kernel/trap.c 的 usertrap()函数。
5. 使用 make qemu 在 xv6 shell 中测试运行该程序。

三、实验中遇到的问题和解决办法

程序计数器流程：

1. ecall 指令中将 PC 保存到 SEPC
2. 在 usertrap 中将 SEPC 保存到 p->trapframe->epc
3. p->trapframe->epc 加 4 指向下一条指令
4. 执行系统调用
5. 在 usertrapret 中将 SEPC 改写为 p->trapframe->epc 中的值
6. 在 sret 中将 PC 设置为 SEPC 的值

需要通过 p->trapframe->epc 来决定执行系统调用后返回到用户空间继续执行的指令地址。这个过程可以在 usertrap 中进行修改，当间隔=tick 数，即计时结束后，修改 epc 值，回到原来的位置。

usertrap 的修改：

```
if(which_dev == 2)
{
  if (p->alarm_interval>0){
    p->alarm_ticks++;
    if (p->alarm_ticks == p->alarm_interval){
      memmove(&(p->alarm_trapframe), p->trapframe, sizeof(*(p->trapframe)
      p->trapframe->epc = p->alarm_handler;
    }
  }
```

两个系统调用的实现：

sys_sigalarm 的实现。这个系统调用可以设置一个定时器，当定时器到达设定的时间间隔时，会调用一个预先设定的处理函数。

```
uint64 sys_sigalarm(void){
  int interval;
  uint64 handler;

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

  struct proc *p =myproc();
  p->alarm_interval = interval;
  p->alarm_handler = handler;
  return 0;
}
```

sys_sigreturn 的实现。这个系统调用在信号处理函数执行完毕后，恢复进程的执行状态。将进程的 trapframe 恢复为信号发生前的状态。

```
uint64 sys_sigreturn(void){
  struct proc *p =myproc();
  memmove(p->trapframe, &(p->alarm_trapframe), sizeof(struct trapframe))
  p->alarm_ticks = 0;
  return 0;
}
```

四、实验心得

本次实验让我详细了解了定时器与相关的函数、使用陷阱实现系统调用的方法与初步实现实现，这让我对操作系统的系统调用的工作原理有了更深刻的理解。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (2.0s)
== Test running alarmtest == (4.1s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (148.9s)
== Test time ==
time: OK
Score: 85/85
```

Lab5: Copy-on-Write Fork for xv6

Implement copy-on write

一、实验目的

实现写时拷贝的 fork() 。COW 的 fork()通过推迟分配和复制物理内存页面，直到实际需要复制的时候，从而提高了内存使用效率和系统性能。

二、实验步骤

1. 在 kernel/riscv.h 中定义 PTE_COW ， 表示 RISC-V 中的 RSW 位。

   #define PTE_COW (1L << 8)

2. 在 kernel/kalloc.c 中进行一系列增删改操作:

   2.1 定义引用计数的全局变量结构体 ref 并初始化

   定义:

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

   初始化,修改 kinit 函数

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

   2.2 修改 freerange、kalloc 和 kfree 函数

   freerange:

```
void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){

    ref.cnt[(uint64)p / PGSIZE] = 1;

    kfree(p);
  }
}
```

kfree:

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

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


  acquire(&ref.lock);
  if(--ref.cnt[(uint64)pa / PGSIZE] == 0) {
    release(&ref.lock);

    r = (struct run*)pa;

    memset(pa, 1, PGSIZE);

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

}
```

kalloc:

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

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    acquire(&ref.lock);
    ref.cnt[(uint64)r / PGSIZE] = 1;
    release(&ref.lock);
    kmem.freelist = r->next;
  }
  release(&kmem.lock);

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

2.3 增加 4 个函数，并且在 defs.h 中添加对应声明。

cowpage：

```
int cowpage(pagetable_t pagetable, uint64 va) {
  if(va >= MAXVA)
    return -1;
  pte_t* pte = walk(pagetable, va, 0);
  if(pte == 0)
    return -1;
  if((*pte & PTE_V) == 0)
    return -1;
  return (*pte & PTE_F ? 0 : -1);
}
```

cowalloc：

```
void* cowalloc(pagetable_t pagetable, uint64 va) {
  if(va % PGSIZE != 0)
    return 0;

  uint64 pa = walkaddr(pagetable, va);  // 获取对应的物理地址
  if(pa == 0)
    return 0;

  pte_t* pte = walk(pagetable, va, 0);  // 获取对应的PTE

  if(krefcnt((char*)pa) == 1) {
    // 只剩一个进程对此物理地址存在引用
    // 则直接修改对应的PTE即可
    *pte |= PTE_W;
    *pte &= ~PTE_F;
    return (void*)pa;
  } else {
    // 多个进程对物理内存存在引用
    // 需要分配新的页面，并拷贝旧页面的内容
    char* mem = kalloc();
    if(mem == 0)
      return 0;

    // 复制旧页面内容到新页
    memmove(mem, (char*)pa, PGSIZE);

    // 清除PTE_V，否则在mappagges中会判定为remap
    *pte &= ~PTE_V;

    // 为新页面添加映射
    if(mappages(pagetable, va, PGSIZE, (uint64)mem, (PTE_FLAGS(*pte) | PTE_W) & ~PTE_F) != 0) {
      kfree(mem);
      *pte |= PTE_V;
      return 0;
    }

    // 将原来的物理内存引用计数减1
    kfree((char*)PGROUNDDOWN(pa));
    return mem;
  }
}
```

krefcnt：

```
int krefcnt(void* pa) {
  return ref.cnt[(uint64)pa / PGSIZE];
}
```

kaddrefcnt：

```
int kaddrefcnt(void* pa) {
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    return -1;
  acquire(&ref.lock);
  ++ref.cnt[(uint64)pa / PGSIZE];
  release(&ref.lock);
  return 0;
}
```

在 defs.h 中添加对应声明（由于调用了 walk 函数，所以也要声明 walk）：

```
int cowpage(pagetable_t, uint64 );
void* cowalloc(pagetable_t , uint64 );
int krefcnt(void* );
int kaddrefcnt(void* );
pte_t* walk(pagetable_t, uint64,int);
```

3. 在 kernel/vm.c 中进行一系列修改操作。

3.1 修改 copyout，如果是 COW 页面，需要更换 pa0 指向的物理地址。

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

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);

    if(cowpage(pagetable, va0) == 0) {
      // 更换目标物理地址
      pa0 = (uint64)cowalloc(pagetable, va0);
    }

    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

3.2 修改 uvmcopy() 函数，使其在复制页表时不再创建一个新的物理页，而是让子进程共享父进程的物理页，并增加相应页的引用计数。

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

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

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

    if(mappages(new, i, PGSIZE, pa, flags) != 0) {
      uvmunmap(new, 0, i / PGSIZE, 1);
      return -1;
    }
    // 增加内存的引用计数
    kaddrefcnt((char*)pa);
  }
  return 0;

}
```

4. 修改 kernel/trap.c 中的 usertrap 函数，使其能够处理写页面错误。

```
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);
    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;
    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 13 || r_scause() == 15) {
  uint64 fault_va = r_stval();  // 获取出错的虚拟地址
  if(fault_va >= p->sz
    || cowpage(p->pagetable, fault_va) != 0
    || cowalloc(p->pagetable, PGROUNDDOWN(fault_va)) == 0)
    p->killed = 1;
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

三、实验中遇到的问题和解决办法

错误页面的处理：

对于一个传入的错误虚地址 va，需要对这个虚地址进行合理的处理，使操作系统运行正常。首先将 va 向下定位至最近的页边界，然后从页表中获取对于页表项，检查该页表项的有效性、PTE_COW 情况与用户访问权限，通过后获取对应的物理地址 pa 与物理页面的引用计数 ref。当 ref 小于 1 时，将页面设置为可写并清除 PTE_COW 位；否则分配一个新的物理页面，将旧页的信息赋值给新页、再将新页映射给 va，然后后设置 PTE_W 位、清除 PTE_COW 页，释放旧页。

可以将这个方法运用与 usertrap 等函数。

四、实验心得

通过本次实验，我深入理解了 Copy-on-Write 的 fork()的工作原理和实现方法。直到实际需要复制的时候才分配物理内存页面，从而减少了内存的浪费，提高了系统性能。学习到了页表相关的基础知识，理解进程初始化与执行过程中页表的处理过程

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (8.1s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests == (143.7s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```

Lab6: Multithreading

Uthread: switching between threads

一、实验目的

熟悉多线程编程，为用户级线程系统设计上下文切换机制。

二、实验步骤

1. 修改 user/uthread_switch.S，添加需要的汇编代码。

```
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 */
```

2. 在 user/uthread.c 中定义上下文结构体 tcontext 保存寄存器内容，并将其加入线程结构体 thread 中。

```
struct tcontext {
  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 {
  char        stack[STACK_SIZE];
  int         state;
  struct tcontext context;
};
```

3. 修改函数 thread_schedule 与 thread_create，使得前者可以切换线程，后者可以设置线程的返回地址与栈帧。

thread_schedule：

```
/* YOUR CODE HERE
 * Invoke thread_switch to switch from t to next_thread:
 * thread_switch(??, ??);
 */
 thread_switch((uint64)&t->context, (uint64)&current_thread->context);
```

thread_create：

```
// YOUR CODE HERE
t->context.ra = (uint64)func;                  // 设定函数返回地址
t->context.sp = (uint64)t->stack + STACK_SIZE;  // 设定栈指针
```

3. 在 qemu 中输入 uthread 进行测试。

三、实验中遇到的问题和解决办法

thread switch 实现：参见 LEC5 使用的文档《Calling Convention》

四、实验心得

通过这次实验，我实现了用户级线程系统的上下文切换机制，这让我对线程切换过程以及原理有了更深入的理解。

Using threads

一、实验目的

探索使用线程和锁的并行编程。

二、实验步骤

1. 在 notxv6/ph.c 中声明锁并初始化。

   声明：

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

   初始化：（main 函数中）

```
for (int i = 0; i < NBUCKET; i++){
  pthread_mutex_init(&lock[i], NULL);
}
```

2. 在 notxv6/ph.c 的 put 与 get 函数中设置锁。

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

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

}
```

3. 在计算机上执行程序验证实验结果。

三、实验中遇到的问题和解决办法

锁的设置：put 函数中的 insert 涉及对哈希表的更改，所以在这条语句之前要上锁，在这条语句后开锁。而 get 函数中并未涉及对哈希表的更改，所以不用设置锁。

为什么未修改时会丢失键?

当两个线程往同一个桶同时插入数据时，没有设置锁时，如果同时进入 insert 函数，将会往同一个头部插入数据。此时，先插入的数据会被覆盖，导致键的丢失。

四、实验心得

在这个实验中，我学习了如何使用线程和锁进行并行编程，防止线程间的冲突。

Barrier

一、实验目的

实现一个屏障：一个应用程序中的点，所有参与的线程必须等待所有其他参与的线程也到达那个点。

二、实验步骤

1. 在 notxv6/barrier.c 中实现 barrier() 函数。

当一个线程进入屏障，如果并非所有线程都进入了屏障，就开始等待

当一个线程进入屏障，线程计数+1.

当所有线程都进入了屏障，释放正所有在等待的线程，并使得轮数+1，重置线程计数。

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  pthread_mutex_lock(&bstate.barrier_mutex);

  bstate.nthread++;
  if(bstate.nthread == nthread) {
    // 所有线程已到达
    bstate.round++;
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  } else {
    // 等待
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }

  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

2. 在计算机上执行程序验证实验结果。

三、实验中遇到的问题和解决办法

锁的设置与线程等待：这两个操作的相关原语相似，开锁/关锁相对应，等待/唤醒相对应，但是要注意两者区别。

四、实验心得

在这个实验中，我学习了如何使用 barrier 函数在多个线程同步工作时防止线程间的冲突。进一步加深了我对多线程编程的同步的理解。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.4s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (7.7s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (18.6s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (3.2s)
== Test time ==
time: OK
Score: 60/60
```

Lab7: Networking

一、实验目的

为网络接口卡（NIC）编写一个 xv6 设备驱动程序。使用名为 E1000 的网络设备来处理网络通信。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。

二、实验步骤

1. 设置两个函数所需要的锁。

```
struct spinlock e1000_txlock;
struct spinlock e1000_rxlock;
```

2. 完成 e1000_transmit()函数。

首先，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。

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。

否则，使用 mbuffree()释放从该描述符传输的最后一个 mbuf（如果有）。

然后填写描述符。m->head 指向内存中数据包的内容，m->len 是数据包的长度。设置必要的 cmd 标志（参考 E1000 手册的第 3.3 节），并保存指向 mbuf 的指针，以便稍后释放。

最后，通过将一加到 E1000_TDT 再对 TX_RING_SIZE 取模来更新环位置。

如果 e1000_transmit()成功地将 mbuf 添加到环中，则返回 0。如果失败（例如，没有可用的描述符来传输 mbuf），则返回-1，以便调用方知道应该释放 mbuf。

```
int
e1000_transmit(struct mbuf *m)
{
  //
  // Your code here.
  //
  // the mbuf contains an ethernet frame; program it into
  // the TX descriptor ring so that the e1000 sends it. Stash
  // a pointer so that it can be freed after sending.
  //
  acquire(&e1000_txlock);
  uint32 tail = regs[E1000_TDT];
  if((tx_ring[tail].status & E1000_TXD_STAT_DD)==0){
    release(&e1000_txlock);
    return -1;
  }
  if(tx_mbufs[tail])
    mbuffree(tx_mbufs[tail]);

  tx_mbufs[tail] = m;
  tx_ring[tail].addr = (uint64)m->head;
  tx_ring[tail].length = m->len;
  tx_ring[tail].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;

  regs[E1000_TDT] = (tail+1) % TX_RING_SIZE;
  release(&e1000_txlock);
  return 0;

}
```

3. 完成 e1000_recv()函数。

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。

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。如果不可用，请停止。

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

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。

最后，将 E1000_RDT 寄存器更新为最后处理的环描述符的索引。

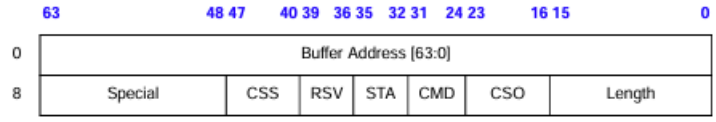
```
static void
e1000_recv(void)
{
  //
  // Your code here.
  //
  // Check for packets that have arrived from the e1000
  // Create and deliver an mbuf for each packet (using net
  //
  acquire(&e1000_rxlock);
  uint32 tail = regs[E1000_RDT];
  uint32 curr = (tail + 1) % RX_RING_SIZE;
  while(1){
      if((rx_ring[curr].status & E1000_RXD_STAT_DD)==0){
        break;
      }
      rx_mbufs[curr]->len = rx_ring[curr].length;
      net_rx(rx_mbufs[curr]);

      struct mbuf *newmbuf = mbufalloc(0);
      rx_mbufs[curr] = newmbuf;
      rx_ring[curr].addr = (uint64)newmbuf->head;
      rx_ring[curr].status = 0;
      regs[E1000_RDT] = curr;
      curr = (curr + 1) % RX_RING_SIZE;
  }

  release(&e1000_rxlock);
}
```

3. 使用 make grade 命令进行测试。

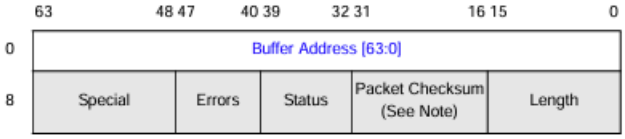
三、实验中遇到的问题和解决办法

描述符格式：描述符是一种数据结构，其中包含了网络数据包的各种重要信息。参考 E100 手册与代码文件中的 e1000_dev.h 可以看到这些信息

Transmit：



Receive：



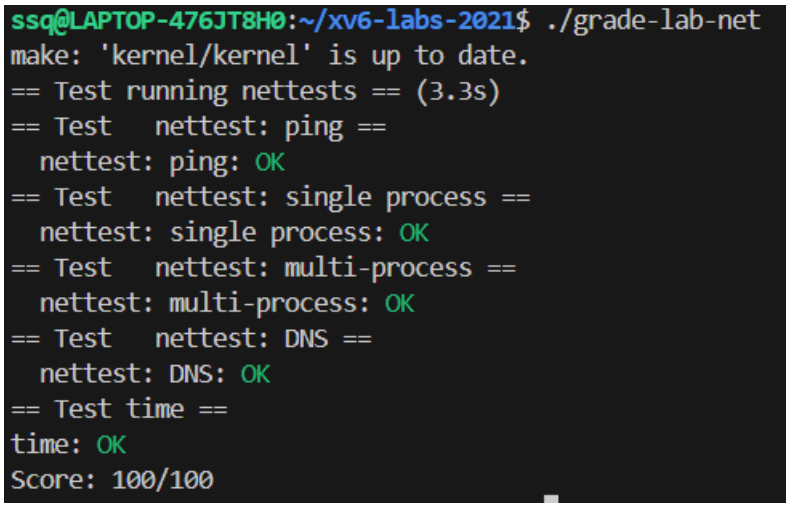
代码中的格式：

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

/* Receive Descriptor bit definitions [E1000 3
#define E1000_RXD_STAT_DD       0x01    /* Des
#define E1000_RXD_STAT_EOP      0x02    /* End

// [E1000 3.2.3]
struct rx_desc
{
  uint64 addr;       /* Address of the descrip
  uint16 length;     /* Length of data DMAed i
  uint16 csum;       /* Packet checksum */
  uint8 status;      /* Descriptor status */
  uint8 errors;      /* Descriptor Errors */
  uint16 special;
};
```

四、实验心得

这次实验让我对网络编程有了更深入的理解。通过实现一个基本的网络协议栈，以及经过对文档的阅读，我对 E1000 的工作原理有了初步的了解。

实验结果


ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (3.3s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100

Lab8: locks

Memory allocator

一、实验目的

实现每个 CPU 的空闲列表，并在 CPU 的空闲列表为空时进行窃取。

二、实验步骤

1. 将 kmem 定义为一个数组，每个元素对应一个 CPU

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

。

2. 修改 kinit 函数，为所有锁初始化以“kmem”开头的名称。

```
void
kinit()
{
  char name[10];
  for (int i = 0; i < NCPU; ++i) {
    snprintf(name, sizeof(name), "kmem_%d", i);
    initlock(&kmem[i].lock, name);
  }
  freerange(end, (void*)PHYSTOP);
}
```

3. 修改 kfree 函数，通过合理的关闭和打开中断，使当前 CPU 编号能够被正确获取。

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

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYS
    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 函数。

当前 CPU 没有空闲空间时：遍历其他 CPU，获取其他 CPU 的内存锁并检查其 freelist。如果 freelist 为空，释放该 CPU 的内存锁并尝试下一个 CPU。如果其他 CPU 的 freelist 非空，偷取其空闲块。然后更新两个 CPU 的 freelist，释放其他 CPU 的内存锁。

成功偷取到内存后，从当前 CPU 的 freelist 中取出一个空闲块并返回，释放当前 CPU 的内存锁。

```
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 {
    for (int i = 0; i < NCPU; ++i) {
      if (i == cpu) continue;
      acquire(&kmem[i].lock);
      r = kmem[i].freelist;
      if(r) {
        kmem[i].freelist = r->next;
        release(&kmem[i].lock);
        break;
      }
      release(&kmem[i].lock);
    }
  }
  release(&kmem[cpu].lock);
  pop_off();

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

三、实验中遇到的问题和解决办法

字符串格式化 snprintf 函数的使用：类似于常用的 printf 函数，但是在此基础上加入了两个参数：字符缓冲区 buf 与缓冲区的大小 sz 。

四、实验心得

这次实验让我更好地理解了操作系统中并发控制和内存管理的原理。我学习了如何通过为每个 CPU 独立分配空闲列表来降低锁的竞争，提高多处理器系统性能。

Buffer cache

一、实验目的

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。令 bcache 中不同块的并发查找和释放不太可能在锁上发生冲突。

二、实验步骤

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

```
#define MAXBUCKETS 13


struct bucket {
  struct spinlock lock;
  struct buf head;
};
struct {
  struct buf buf[NBUF];
  struct bucket buckets[MAXBUCKETS];
} bcache;
```

定义哈希函数，通过缓冲区号获取其在桶中的位置。

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

2. 修改 binit 函数 ， 初始化每个桶，并将所有缓冲区先全部挂载在 0 号桶上。

```
void
binit(void)
{
  struct buf *b;
  char name[20];
  for(int i=0; i<MAXBUCKETS; ++i) {
    snprintf(name, sizeof(name), "bcache_%d", i);
    initlock(&bcache.buckets[i].lock, name);
    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }
  // Create linked list of buffers
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.buckets[0].head.next;
    b->prev = &bcache.buckets[0].head;
    initsleeplock(&b->lock, "buffer");
    bcache.buckets[0].head.next->prev = b;
    bcache.buckets[0].head.next = b;
  }
}
```

4. 修改 bget 函数，当没有找到指定的缓冲区时进行分配。采用 LRU 算法。
没有找到缓冲区时：

```
b = 0;
struct buf* tmp;

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

  for(tmp = bcache.buckets[i].head.next; tmp != &bcache.buckets[i].head; tmp = tmp->next)
    // 使用时间戳进行LRU算法
    if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp))
      b = tmp;

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

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

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

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

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

3. 修改 brelse 函数，减少缓冲区引用次数，当次数为零时移至链表头。
先在 buf.h 中增加新字段 timestamp，用于做 LRU 算法的判定。

```
struct buf {
  int valid;   // has da
  int disk;    // does d
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // l
  struct buf *next;
  uchar data[BSIZE];

  uint timestamp;
};
```

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

  int bid = hash(b->blockno);

  releasesleep(&b->lock);

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

  // 更新时间戳
  acquire(&tickslock);
  b->timestamp = ticks;
  release(&tickslock);

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

5. 修改受影响的 bpin 和 bunpin 函数。

```
void
bpin(struct buf* b) {
  int bid = hash(b->blockno);
  acquire(&bcache.buckets[bid].lock);
  b->refcnt++;
  release(&bcache.buckets[bid].lock);
}

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

三、实验中遇到的问题和解决办法

Usertest 超时问题：代码写完后发现在这一步会因为超时导致 fail 的结果，阅读日志后发现是在 test:Writebig 中出现 panic: balloc: out of blocks 报错。此问题困扰了我很久，在网上查阅资料后发现将 param.h 中的 FSSIZE 值修改得更大一点即可，我将其修改至 10000，解决问题。

```
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (300.3s)
    ...
        OK
        test opentest: OK
        test writetest: OK
        test writebig: panic: balloc: out of blocks
        qemu-system-riscv64: terminating on signal 15 from pid 2131047 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
```

四、实验心得

通过这个实验，我学习了如何优化 xv6 操作系统的缓冲区缓存。将缓冲区分配到不同的哈希桶中，每个桶有自己的锁，减少锁的竞争，提高性能。并且复习了课上讲过的 LRU 算法，并进行实践。

实验结果

```
== Test running kalloctest == (119.1s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (10.0s)
== Test running bcachetest == (7.9s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests == usertests: OK (163.5s)
== Test time ==
time: OK
Score: 70/70
```

Lab9: file system

Large files

一、实验目的

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

二、实验步骤

1. 修改 kernel/fs.h 中宏定义。

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

2. 修改 dinode 和 inode 结构。

将 uint addrs[NDIRECT+1]改为 uint addrs[NDIRECT+2]

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

```
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];
};
```

3. 修改 kernel/fs.c 中的 bmap 函数与 itrunc 函数。

bmap：ip->addrs[NDIRECT + 1]用于存放二级间接块地址，a 为其内容。

在二级间接块中，要映射的块在第 bn/NINDIRECT 个一级间接块中，在一级间接块中的位置为 bn%NINDIRECT

```
if(bn < NDINDIRECT) {
    int level2_idx = bn / NINDIRECT;
    int level1_idx = bn % NINDIRECT;

    // 读出二级间接块
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    if((addr = a[level2_idx]) == 0) {
      a[level2_idx] = addr = balloc(ip->dev);
      // 更改了当前块的内容，标记以供后续写回磁盘
      log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[level1_idx]) == 0) {
      a[level1_idx] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
```

itrunc：遍历二级间接块中的所有一级间接块，仿照已有的一级间接块释放方式释放即可。

```
struct buf* bp1;
uint* a1;
if(ip->addrs[NDIRECT + 1]) {
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
  a = (uint*)bp->data;
  for(i = 0; i < NINDIRECT; i++) {
    // 每个一级间接块的操作都类似于上面的
    // if(ip->addrs[NDIRECT])中的内容
    if(a[i]) {
      bp1 = bread(ip->dev, a[i]);
      a1 = (uint*)bp1->data;
      for(j = 0; j < NINDIRECT; j++) {
        if(a1[j])
          bfree(ip->dev, a1[j]);
      }
      brelse(bp1);
      bfree(ip->dev, a[i]);
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 1]);
  ip->addrs[NDIRECT + 1] = 0;
}
```

三、实验中遇到的问题和解决办法

bread 块读入缓冲区与 brelse 缓冲区释放，这两个函数需要成对出现，否则会导致缓冲区被占用导致死锁。

四、实验心得

在这次的实验中，我深入理解了 xv6 文件系统中的数据块管理方式，学习了如何通过多级间接块来扩展文件的大小。

Symbolic links

一、实验目的

常见的硬链接（例如 A 链接到 B），会将 A 的 inode 号设置成和 B 文件一样，并且将对应 inode 的 ref 加一。而所谓符号链接（软链接），并不会影响 B 的 inode，而是将 A 标记成特殊的软链接文件，之后对 A 进行的所有文件操作，操作系统都会转换到对 B 的操作，类似于一个快捷方式。在本练习中，将向 xv6 添加符号链接。

二、实验步骤

1. 创建 symlink 系统调用并修复编译问题。
2. 实现 sys_symlink 函数。

使用 create 函数新建一个 inode 指针，若创建失败则返回错误；若成功就向该 inode 块中写入目标路径。最后使用 iunlockput 函数解锁 inode 并使其引用计数-1。

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

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

  begin_op();
  // 分配一个inode结点，create返回锁定的inode
  ip_path = create(path, T_SYMLINK, 0, 0);
  if(ip_path == 0) {
    end_op();
    return -1;
  }
  // 向inode数据块中写入target路径
  if(writei(ip_path, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip_path);
    end_op();
    return -1;
  }

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

3. 修改 sys_open 函数。
10 为最大搜索深度，如果搜索次数超过 10 次还未找到文件，则认为打开文件失败。
如果打开的文件仍然是符号链接文件，则递归继续查找。

```
if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  // 深度不能超过10
  for(int i = 0; i < 10; ++i) {
    // 读出符号链接指向的路径
    if(readi(ip, 0, (uint64)path, 0, MAXPATH) != MAXPATH) {
      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;
  }
  // 超过最大允许深度后仍然为符号链接，则返回错误
  if(ip->type == T_SYMLINK) {
    iunlockput(ip);
    end_op();
    return -1;
  }
}
```

三、实验中遇到的问题和解决办法

begin_op 与 end_op 函数：这对函数指明了一个文件系统操作的过程，会让操作在文件系统安全的时候开始进行，并且在结束后唤醒其他正在等待的操作。

四、实验心得

在这个实验中，我学习了如何在 xv6 操作系统中实现符号链接，深入理解文件系统的工作原理，以及如何在操作系统中实现复杂的文件系统功能。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (118.6s)
== Test running symlinktest == (1.1s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (222.5s)
== Test time ==
time: OK
Score: 100/100
```

Lab10: mmap

一、实验目的

mmap 和 munmap 系统调用允许 UNIX 程序对其地址空间进行详细控制。它们可用于在进程之间共享内存，将文件映射到进程地址空间，并作为用户级页面错误方案的一部分。在本实验中，重点关注内存映射文件

二、实验步骤

1. 创建 mmap 与 mummap 系统调用，并修复编译问题。
2. 定义 VMA 结构体并添加至进程结构体。

```
struct vm_area {
  int used;
  uint64 addr;
  int len;
  int prot;
  int flags;
  int vfd;
  struct file* vfile;
  int offset;
};
```

在 proc 结构体中添加:

```
struct vm_area vma[NVMA];
```

3. 在 allocproc 函数中初始化 VMA 数组为 0。

```
memset(&p->vma, 0, sizeof(p->vma));
return p;
```

4. 实现 mmap 系统调用。

在进程地址空间中查找一个未使用的区域用于映射文件，将 VMA 加入进程的映射区域表中。要注意添加文件的引用计数。

```
uint64
sys_mmap(void) {
  uint64 addr;
  int length;
  int prot;
  int flags;
  int vfd;
  struct file* vfile;
  int offset;
  uint64 err = 0xffffffffffffffff;

  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
    argint(3, &flags) < 0 || argfd(4, &vfd, &vfile) < 0 || argint(5, &offset) < 0)
    return err;

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

  // 文件不可写则不允许拥有PROT_WRITE权限时映射为MAP_SHARED
  if(vfile->writable == 0 && (prot & PROT_WRITE) != 0 && flags == MAP_SHARED)
    return err;

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

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

      filedup(vfile);

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

  return err;
}
```

5. 修改 usertrap 函数。

   此时访问页面会产生错误，在 usertrap 中处理这些错误。

   具体处理过程可以写在 mmap_handler 函数中。

```
} else if(r_scause() == 13 || r_scause() == 15) {
  if(PGROUNDUP(p->trapframe->sp) - 1 >= r_stval() || r_stval() >= p->sz)
    p->killed = 1;
  else if(mmap_handler(r_stval(), r_scause())) p->killed = 1;
```

mmap_handler 函数实现：

分为三步：查找 VMA，检查 pte 权限并修改对应的 pte 位，将读取的文件映射到进程的页表中。

```
int mmap_handler(int va, int cause) {
  int i;
  struct proc* p = myproc();

  for(i = 0; i < NVMA; ++i) {
    if(p->vma[i].used && p->vma[i].addr <= va && va <= p->vma[i].addr + p->vma[i].len - 1) {
      break;
    }
  }
  if(i == NVMA)
    return -1;

  int pte_flags = PTE_U;
  if(p->vma[i].prot & PROT_READ) pte_flags |= PTE_R;
  if(p->vma[i].prot & PROT_WRITE) pte_flags |= PTE_W;
  if(p->vma[i].prot & PROT_EXEC) pte_flags |= PTE_X;


  struct file* vf = p->vma[i].vfile;
  // 读错误
  if(cause == 13 && vf->readable == 0) return -1;
  // 写错误
  if(cause == 15 && vf->writable == 0) return -1;

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

  ilock(vf->ip);

  int offset = p->vma[i].offset + PGROUNDDOWN(va - p->vma[i].addr);
  int readbytes = readi(vf->ip, 0, (uint64)pa, offset, PGSIZE);

  if(readbytes == 0) {
    iunlock(vf->ip);
    kfree(pa);
    return -1;
  }
  iunlock(vf->ip);

  if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, pte_flags) != 0) {
    kfree(pa);
    return -1;
  }

  return 0;
}
```

6. 实现 munmap 系统调用。

遍历 VMA 数组，对于符合条件的 VMA，更新其初始地址与长度。使用 uvmumap 函数取消页表的映射。

若 VMA 映射类型为 MAP_SHARED，则将此页内容写回文件。

若 VMA 全部的映射都被取消时，关闭文件，将其设置为无效。

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

  int i;
  struct proc* p = myproc();
  for(i = 0; i < NVMA; ++i) {
    if(p->vma[i].used && p->vma[i].len >= len) {
      if(p->vma[i].addr == addr) {
        p->vma[i].addr += len;
        p->vma[i].len -= len;
        break;
      }
      if(addr + len == p->vma[i].addr + p->vma[i].len) {
        p->vma[i].len -= len;
        break;
      }
    }
  }
  if(i == NVMA)
    return -1;

  if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
    filewrite(p->vma[i].vfile, addr, len);
  }

  uvmunmap(p->pagetable, addr, len / PGSIZE, 1);

  if(p->vma[i].len == 0) {
    fileclose(p->vma[i].vfile);
    p->vma[i].used = 0;
  }

  return 0;
}
```

7. 修改 uvmunmap 函数与 uvmcopy 函数。

```
if((*pte & PTE_V) == 0)
  continue;
```

8. 修改 exit 函数，将进程已经映射过的区域取消映射。

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

9. 修改 fork 函数，复制父进程的 VMA，增加文件引用次数。

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

三、实验中遇到的问题和解决办法

VMA 结构设计：参考 lab 提示，使用其中参数，并进行必要的补充（used、file）。

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

四、实验心得

通过这个实验，我对 mmap 和 munmap 系统调用有了更深入的理解，并学会了在操作系统中实现内存映射文件的功能。

本次实验项目对于我来说是一个非常好的挑战，完成所有的实验后，我对操作系统的了解不仅仅只是书本中学习到的知识，对于操作系统的具体实现也有了一定的了解。

实验结果

```
ssq@LAPTOP-476JT8H0:~/xv6-labs-2021$ ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (3.1s)
== Test   mmaptest: mmap f ==
  mmaptest: mmap f: OK
== Test   mmaptest: mmap private ==
  mmaptest: mmap private: OK
== Test   mmaptest: mmap read-only ==
  mmaptest: mmap read-only: OK
== Test   mmaptest: mmap read/write ==
  mmaptest: mmap read/write: OK
== Test   mmaptest: mmap dirty ==
  mmaptest: mmap dirty: OK
== Test   mmaptest: not-mapped unmap ==
  mmaptest: not-mapped unmap: OK
== Test   mmaptest: two files ==
  mmaptest: two files: OK
== Test   mmaptest: fork_test ==
  mmaptest: fork_test: OK
== Test usertests == usertests: OK (153.0s)
== Test time ==
time: OK
Score: 140/140
```

代码托管库：https://github.com/CH4MBER/XV6_Labs