Lab 1: Xv6 and Unix Utilities

本实验的主要目的是通过实现一些基本的Unix实用程序来深入理解xv6操作系统和C编程。

Boot xv6

一、实验目的

切换到xv6-labs-2021代码的util分支，利用QEMU模拟器启动xv6系统。

二、实验步骤

1. 下载WSL

```
wsl --install
```

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，其对应代码如下：

```
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，我学习了如何使用git来管理和切换代码版本，以及如何使用QEMU模拟器来启动xv6操作系统。这些技能对于我后续的学习和研究都非常有用。

sleep

一、实验目的

为xv6系统实现UNIX的sleep程序。sleep程序应该使当前进程暂停相应的时钟周期数，时钟周期数由用户指定。

二、实验步骤

1. 在user目录下，创建一个名为sleep.c的文件；
2. 在sleep.c中，编写一个程序，该程序接受一个命令行参数（ticks数量），并使当前进程暂停相应的ticks数量。在该程序中，需要考虑到如果用户没有提供参数或者提供了多个参数，程序应该打印出错误信息。
3. 将程序以`$U/_sleep\`的形式，添加到Makefile的UPROGS中；
4. 使用`make qemu`在xv6 shell中测试运行该程序；
5. 使用`./grade-lab-util sleep`进行单元测试。

三、实验中遇到的问题和解决办法

暂无

四、实验心得

通过这个实验，我学习了如何为xv6操作系统添加新的程序。我了解了如何处理命令行参数，以及如何使用系统调用来控制进程的行为。这些知识对于我理解操作系统的工作原理非常有帮助。

pingpong

一、实验目的

使用UNIX系统调用编写一个程序pingpong，在一对管道上实现两个进程之间的通信。

二、实验步骤

思路：开两个pipe，一个pipe负责子进程写父进程读，另一个pipe负责父进程写子进程读。

1. 在user目录下，创建一个名为pingpong.c的文件；
2. 在pingpong.c中，编写程序以实现功能。
    - 先创立两个管道，分别表示从父到子（ping）从子到父（pong）；
    - 通过`fork()`的返回值来分辨父子进程：若为父，则先向父到子管道`write()`发出”ping”，接着开始等待`wait()`子进程结束，读取`read()`子到父管道的信号”pong”，并做出回应；若为子，则先等待父到子管道的信号”ping”，接收到后做出回应，再向子到父的管道内发出信号”pong”，子进程结束。
    - 最后使用`exit(0)`来正常退出程序。
3. 将程序以`$U/_pingpong\`的形式，添加到Makefile的UPROGS中；
4. 在xv6 shell中测试运行该程序；
5. 使用`./grade-lab-util pingpong`进行单元测试。

三、实验中遇到的问题和解决办法

暂无

四、实验心得

通过实现pingpong程序，我深入理解了进程间通信的基本原理和使用管道进行进程通信的方法。通过使用fork、pipe和相关的系统调用函数，我成功地实现了父子进程之间的通信，并能够正确地发送和接收信号。

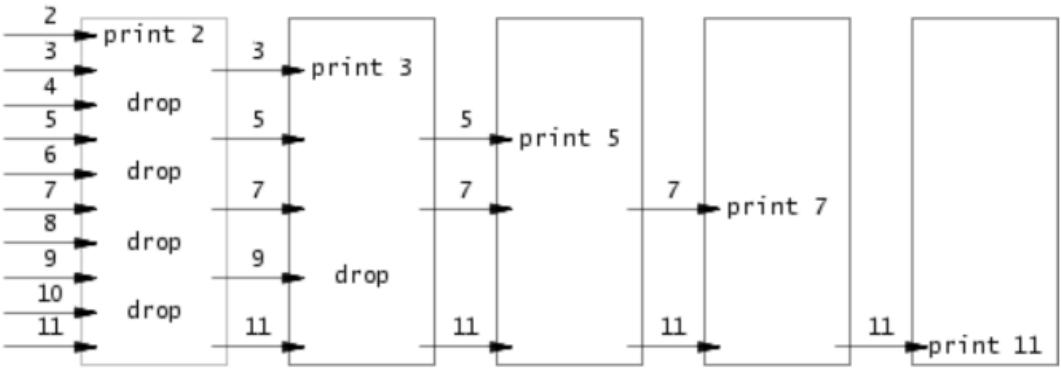
primes

一、实验目的

使用管道将2至35中的素数筛选出来。第一个进程将数字2到35送入管道中。对于每个质数，要安排创建一个进程，从其左邻通过管道读取，并在另一条管道上写给右邻。

二、实验步骤

1. 思路：将一组数输入到一个进程里，先打印出最小的一个数，这是一个素数，然后用其他剩下的数依次尝试整除这个素数，如果可以整除，则将其drop， 不能整除则将其feed到下一个进程中，直到最后打印出所有的素数。
    1. 在user目录下，创建一个名为primes.c的文件；
    2. 在primes.c中，编写程序以实现功能，其实现原理如下图所示：



    - 定义常量`READEND`、`WRITEEND`、`ERROREND`，分别表示进程自己独立的文件描述符fd，标准输入（0）、标准输出（1）、标准错误（2）；
    - `pipe(p)`创建一个管道p，将用于存放2~35之间的所有数字；
    - 在父进程当中：先关闭管道的读取端`close(p[READEND])`【只需要写入，关闭读取端可以防止意外的读操作】，然后依次将34个数字`write(p[WRITEEND], &i, sizeof(int))`写到管道p的写入端，关闭管道的写入端`close(p[WRITEEND])`，等待子进程结束`wait(NULL)`。
    - 在子进程当中：调用函数`child(p)`：先关闭父进程的管道p的写通道，`read(p[READEND], &n, sizeof(int))`读取p中的第一个整数，若读取失败（p中为空），则read返回0，此时子进程退出。再创建一个管道pr。
      再一次分为父子进程：
      在父进程中，关闭pr的读取端，打印pr的第一个元素prime（肯定为素数）；之后，依次处理p中所有元素，若能够被prime整除，则不将其写入pr中；最后，关闭pr通道的写端，等待子进程结束。
      在子进程中：调用函数`child(pr)`，继续筛选剩下的元素，直至管道内没有元素。
    3. 将程序以`$U/_primes\`的形式，添加到Makefile的UPROGS中；
    4. 在xv6 shell中测试运行该程序；
    5. 使用`./grade-lab-util primes`进行单元测试。

三、实验中遇到的问题和解决办法

1. **问题一：素数筛选逻辑错误**
   在我编写primes.c最初的实现中，我错误地处理了素数筛选的逻辑。我没有正确地使用素数去筛选其他的数，导致输出的结果不是素数。

   解决方案： 我重新审查了代码，并意识到我在处理素数筛选逻辑时犯了错误。采用埃拉托斯特尼筛法进行素数的筛选，确保在每轮筛选数时，只有当一个数不能被当前的素数整除时，我才将它写入到下一个进程的管道中。这样，我就可以正确地筛选出素数了。

四、实验心得

实现primes程序的过程中，我学习了如何使用管道进行进程间通信，并运用了进程的创建、关闭、读写管道等操作。通过编写筛选素数的逻辑，我深入理解了进程之间的协作和管道的作用。

find

一、实验目的

实现UNIX的find程序。你的find程序应该遍历指定的目录及其子目录，打印出与指定模式匹配的文件名。

二、实验步骤

思路：用递归方式找到指定的文件夹下符合某个名字的文件，参考user/ls.c的实现方法

1. 在user目录下，创建一个名为find.c的文件；
2. 在find.c中，编写程序以实现功能：
    1. 在main函数中，首先检查用户是否提供了正确数量的命令行参数。如果参数数量不等于3（包括程序名），则打印错误消息并退出程序。如果参数数量正确，将第一个参数（路径）和第二个参数（要查找的文件名）传递给find函数。
    2. 在find函数中，首先尝试打开给定的路径。如果无法打开，打印错误消息并返回。如果路径可以打开，使用`fstat`函数获取路径的状态信息。如果无法获取状态信息，关闭文件描述符，打印错误消息并返回。
    3. 使用`read`函数读取目录中的每个条目，直到没有更多条目为止。
    4. 对于每个目录条目，检查其索引节点号。如果索引节点号为0（表示条目为空），则跳过该条目。
    5. 对于非空条目，将条目的名称添加到路径的末尾，形成完整的文件或子目录路径。
    6. 使用`stat`函数获取新路径的状态信息。如果无法获取状态信息，打印错误消息并继续下一个条目。
    7. 根据新路径的类型进行不同的处理：
        - 如果新路径是文件，并且文件名与要查找的文件名相同，打印文件路径。
        - 如果新路径是目录，且目录名不是"."或".."(这两个目录名分别代表当前目录和父目录)，则递归地在这个目录中查找目标文件。
    8. 关闭文件描述符并返回。
3. 将程序以`$U/_find\`的形式，添加到Makefile的UPROGS中；
4. 在xv6 shell中测试运行该程序；

5. 使用`./grade-lab-util find`进行单元测试。

三、实验中遇到的问题和解决办法

1. **问题一：错误地处理了`.`和`..`目录**
   在我编写find.c最初的实现中，我没有正确地处理`.`和`..`这两个特殊的目录名。我没有检查目录名，导致我在`.`和`..`目录中进行了递归查找。这个错误导致陷入了无限的递归，程序无法正确地运行。

   解决方案： 我重新审查了代码，并意识到我在处理目录时犯了错误。修改了代码，确保在处理目录时，会检查目录名。如果目录名是`.`或`..`，那么我会跳过该目录，不进行递归查找。只有当目录名既不是`.`也不是`..`时，我才进行递归查找。

2. **问题二：路径处理错误**
   在我编写find.c最初的实现中，我没有正确地处理文件和目录的路径。我忘记在目录的名称前添加斜杠，导致路径拼接错误。这个错误导致我无法正确地打开文件或子目录，我花了很多时间才找到这个问题。

   解决方案： 我重新审查了代码，并意识到我在处理路径时犯了错误。我修改了代码，确保在添加目录项的名称到路径的末尾时，先添加一个斜杠，然后再添加目录项的名称。此外，我还在路径的末尾添加了一个字符串结束符，以确保路径是一个正确的字符串。修改的代码段如下：

   ```
   strcpy(buf,path);
   p = buf + strlen(buf);
   *p++ = '/';
   ```

四、实验心得

通过实现find程序，我学习了如何使用递归的方式遍历目录及其子目录，并使用系统调用函数获取文件和目录的信息。这个实验让我更加熟悉了文件系统的结构和相关的系统调用。

xargs

一、实验目的

实现UNIX的xargs程序，实现将标准输入作为参数一起输入到xargs后面跟的命令中

二、实验步骤

1. 在user目录下，创建一个名为xargs.c的文件；
2. 在xargs.c中，编写程序以实现功能:
   1. 从标准输入读取命令和参数：程序首先从标准输入读取用户输入的命令和参数。这些输入被读取到一个字符串数组中，每个字符串代表一个参数。
   2. 解析命令和参数：程序会解析用户输入的命令和参数。它会将输入的字符串分割成多个参数，每个参数都是一个独立的字符串。这个过程是通过检查空格字符来实现的，因为在命令行中，参数通常是由空格分隔的。
   3. 执行命令：程序会创建一个新的进程来执行用户输入的命令。这是通过调用`fork`和`exec`函数来实现的。`fork`函数会创建一个新的进程，而`exec`函数则会在这个新进程中执行指定的命令。

4. 等待命令执行完成：在命令开始执行后，程序会等待命令执行完成。这是通过调用`wait`函数来实现的。`wait`函数会阻塞当前进程，直到子进程（也就是执行命令的进程）结束。
5. 循环处理：程序会一直重复上述步骤，直到从标准输入读取到EOF（表示输入结束）。这样，用户就可以连续输入多个命令，程序会依次执行这些命令。

3. 将程序以`$U/_xargs\`的形式，添加到Makefile的UPROGS中；
4. 在xv6 shell中测试运行该程序；
5. 使用`./grade-lab-util xargs`进行单元测试。

三、实验中遇到的问题和解决办法

1. **问题一：参数长度限制**

   在我编写xargs.c最初的实现中，我设置了一个固定的参数长度**MAX_LEN**。然而，如果不进行设置，当输入的参数长度超过**MAX_LEN**时，这会导致参数被截断，或者在极端情况下，可能会导致缓冲区溢出。

   解决方案： 我重新检查了代码，并意识到我在处理参数长度时犯了错误。我修改了代码，确保在处理参数时能够正确处理参数长度超过**MAX_LEN**的情况。例如，我可以通过截断参数、显示错误消息或者拒绝执行命令等方式来处理这种情况。此外，我还需要确保**MAX_LEN**足够大，可以处理预期的最大参数长度。

四、实验心得

通过实现xargs程序，我学习了如何处理标准输入并将其作为参数传递给后续的命令。这个实验让我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

实验成果截图如下：

```
root@david:~/xv6-labs-2021# ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.7s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (1.2s)
== Test time ==
time: OK
Score: 100/100
```

Lab 2: System Calls

本实验主要是让读者更加了解操作系统的进程的组织形式。

System call tracking

一、实验目的

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

二、实验步骤

1. 使用如下命令切换实验分支，获取实验资源。

```
git fetch
git checkout syscall
make clean
```

2. 根据hints中的提示修改相应的文件：
    1. 在Makefile中添加`$U/_trace\`到UPROGS，但此时系统调用的用户空间存根尚不存在；
    2. 在user/user.h中添加`int trace(int);`声明trace的系统调用的原型；
    3. 在user/usys.pl中添加存根`entry("trace");`；
    4. 在kernel/syscall.h中添加系统调用号`#define SYS_trace 22`；
3. 在kernel/sysproc.c中添加一个sys_trace()函数，通过在proc结构中记住其参数来实现新的系统调用。
    1. 在kernel/sysproc.c中，定义sys_trace()函数，将输入的掩码传递到proc的trace_mask中；
    2. 在kernel/syscall.c中，仿照其他系统调用，增添`extern uint64 sys_trace(void);`与`[SYS_trace]  sys_trace,`；在syscall()中，取得掩码，获得其对应的系统调用名称，将其打印出来。
4. 在kernel/proc.c中，修改fork()以从父进程复制跟踪掩码到子进程：`np->trac_mask=p->trac_mask;`
5. 测试代码输出结果是否正确。

三、实验中遇到的问题和解决办法

1. **问题一：参数传递问题**

    在sysproc.c中可以仿照`sys_exit`的方式（`argint(0, &n) < 0`）拿到输入进来的mask，但mask对应的系统调用是在syscall.c中才被调用，应该如何将mask传递，以实现trace的功能。

    解决方案：在开发中有两种方式：全局变量共享内存、信号量传递。后者通过形参的方式传递，在此处不太可行。查看syscall的代码，发现参数是从`proc`里面来获得的，`proc`是进程控制块，在sysproc中也有如sys_sbrk等函数调用到`myproc`。故最终通过进程进行通信，设置了`proc->trace_mask`来标记输入进来的掩码，实现参数传递。此外，也需要到proc.h中去修改`struct proc`的变量，引入`int trace_mask;`。相关代码段如下：

```
//sysproc.c
//Add a sys_trace() function in kernel/sysproc.c
```

```
uint64
sys_trace(void)
{
  int mask;
  if(argint(0, &mask) < 0)
    return -1;
  struct proc *p = myproc();
  p->trace_mask = mask;
  return 0;
}

//syscall.c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();//process
  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();//return value stored in register a0
    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;
  }
}
```

2. **问题二：不用trace仍然打印系统调用**

   测试时，输入`grep hello README`仍然展示系统调用。

   解决方案：查阅资料，发现与进程的`freeproc()`有关，释放进程时需要重置变量的值，故增加`p->trace_mask=0;`到其中，以实现重置，问题解决。修正前后截图如下：

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
5: syscall sbrk -> 16384
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 968
5: syscall read -> 235
5: syscall read -> 0
5: syscall close -> 0
```

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$
```

四、实验心得

通过这次实验，我深入理解了系统调用的工作原理，并学习了如何添加新的系统调用。我发现，系统调用是操作系统提供给用户程序的接口，它是用户程序与操作系统内核进行交互的关键机制。在实验过程中，我遇到了一些问题，例如参数传递和进程释放，但通过查阅资料和不断尝试，我成功解决了这些问题。这次实验让我更加熟悉了操作系统的工作原理，并提高了我的问题解决能力。我期待进行更多这样的实验，以进一步提高我的技能和知识。

Sysinfo

一、实验目的

添加一个新的系统调用sysinfo，它可以收集和返回关于正在运行的系统的信息（需要获取当前空闲的内存大小填入struct sysinfo.freemem中，获取当前所有不是UNUSED的进程数量填入struct sysinfo.nproc中。）

二、实验步骤

1. 根据hints中的提示修改相应的文件：
    1. 在Makefile中添加`$U/_sysinfotest\`到UPROGS，但此时系统调用的用户空间存根尚不存在；
    2. 在user/user.h中添加`int sysinfo(struct sysinfo*);`，并在此之前声明结构体`struct sysinfo;`（其在kernel/sysinfo.h中有定义）；
    3. 在user/usys.pl中添加存根`entry("sysinfo");`；
    4. 在kernel/syscall.h中添加系统调用号`#define SYS_sysinfo 23`；
    5. 在kernel/syscall.c中，增添`extern uint64 sys_trace(void);`与`[SYS_sysinfo] sys_sysinfo`
2. 参考kernel/sysfile.c中的`sys_fstat()`与kernel/file.c中的`filestat()`使用`copyout()`来编写kernel/sysproc.c中的`sys_sysinfo()`函数。参考kernel/kalloc.c与 kernel/proc.c来获得对应的info参数。
3. 在qemu中输入`sysinfotest`测试代码输出结果是否正确。
4. 使用命令`./grade-lab-syscall`来获取最终得分

三、实验中遇到的问题和解决办法

1. **问题一：模仿以使用copyout()函数**

    在hints中有提到使用copyout()函数以实现功能，最初对其实现较为困惑。

    解决方案：

    1. 了解kernel/sysfile.c中的`sys_fstat()`与kernel/file.c中的`filestat()`的作用和实现原理：
        - `sys_fstat()`允许用户程序获取文件的状态信息，其中调用`argfd(0, 0, &f)`和`argaddr(1, &st)`获取系统调用的参数。
            - `argfd(int n, int *pfd, struct file **pf)`：用于获取系统调用的第n个参数（文件描述符）。若成功，函数会将文件描述符存储在`pfd`指向的位置，将对应的文件结构的指针存储在`pf`指向的位置，并返回0。如果失败，函数返回-1。

    - `argaddr(int n, uint64 *pp)`：用于获取系统调用的第n个参数（用户空间的地址）。如果成功，函数会将地址存储在`pp`指向的位置，并返回0。如果失败，函数返回-1。
- `filestat()`使用`copyout(p->pagetable, addr, (char *)&st, sizeof(st))`将状态信息从内核空间复制到用户空间
    - `int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len);`

      其中`pagetable`：当前进程的页表，用于进行虚拟地址到物理地址的转换；`dstva`：目标地址，即用户空间的地址，这是要复制到的地方；`src`：源地址，即内核空间的地址，这是要复制的数据的位置；`len`：要复制的字节数。如果复制成功，返回0；如果在复制过程中出现错误（例如，目标地址不在用户空间，或者源地址不在内核空间），返回-1。

2. 编写测试函数以测试是否真正理解

   测试代码如下，发现输出*FAIL: there is no free mem, but sysinfo.freemem=-2*表明参数传递基本正确

```
//Add a sys_sysinfo() function in kernel/sysproc.c
uint64
sys_sysinfo()
{
  struct sysinfo info;
  uint64 addr;

  info.nproc=-1;
  info.freemem=-2;
  struct proc *p = myproc();

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

3. 编写代码，以获得`info.nproc`与`info.freemem`

    - 参考 kernel/kalloc.c其中的空闲空间是以链表`struct run *freelist`的方式存储的，故采用遍历的方式获得链表的长度，在乘上一个节点的大小PGSIZE=4096即可获得总空闲空间。
    - 参考kernel/proc.c中的

   代码如下：

```
// kalloc.c
uint64 acquire_freemem(){
  struct run *r;
  uint64 cnt = 0;

  acquire(&kmem.lock);
  r=kmem.freelist;
  while(r){
    r=r->next;
```

```
    cnt++;
  }
  release(&kmem.lock);
  return cnt*PGSIZE;
}
// proc.c
uint64 acquire_nproc(){
  struct proc *p;
  int cnt=0;
  for(p=proc;p<&proc[NPROC];p++){
    acquire(&p->lock);
    if(p->state==UNUSED){
      cnt++;
    }
    release(&p->lock);
  }
  return cnt;
```

4. 在`kernel/sysproc.c`中导入并应用对应的函数即可。

2. **问题二：hints中给出的操作的用途**

   在Lab2的两个实验中，hints都需要修改一系列文件，但其用途暂未清晰。

   解决方案：回顾课程内容，了解各个文件的作用。kernel中有以下文件及作用

文件	描述
file.c	File descriptor support.
kalloc.c	Physical page allocator.
proc.c	Processes and scheduling.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.

   经过整理后，理解如下：

   1. 在Makefile中添加`$U/_sysinfotest\`到UPROGS，这是为了让编译系统知道需要编译和链接`sysinfotest`这个用户程序。
   2. 在user/user.h中添加`int sysinfo(struct sysinfo*);`，这是为了在用户空间提供`sysinfo`系统调用的接口。在这之前声明结构体`struct sysinfo;`是因为`sysinfo`系统调用需要这个结构体类型的参数。
   3. 在user/usys.pl中添加存根`entry("sysinfo");`，这是为了生成用户空间的系统调用存根，这个存根会在用户程序调用`sysinfo`时转到内核空间的实现。
   4. 在kernel/syscall.h中添加系统调用号`#define SYS_sysinfo 23`，这是为了给`sysinfo`系统调用分配一个唯一的编号。
   5. 在kernel/syscall.c中，增添`extern uint64 sys_trace(void);`与`[SYS_sysinfo] sys_sysinfo`，这是为了在系统调用分派函数中添加`sysinfo`的入口。

6. 参考kernel/sysfile.c中的 `sys_fstat()` 与kernel/file.c中的 `filestat()` 使用 `copyout()` 来编写kernel/sysproc.c中的 `sys_sysinfo()` 函数。这是因为 `sysinfo` 系统调用需要将内核空间的信息复制到用户空间， `copyout` 函数正是用于这个目的。

7. 参考kernel/kalloc.c与 kernel/proc.c来获得对应的info参数，这是因为 `kalloc.c` 中实现了物理页面的分配和释放，可以提供系统当前的内存使用情况；而 `proc.c` 则负责进程的创建、调度和销毁，可以提供系统当前的进程数量和CPU使用情况。这些信息被收集并填充到 `sysinfo` 结构体中，然后通过 `sys_sysinfo` 系统调用返回给用户空间。

3. **问题三：nproc计算出错**

   在最后提交成果后，使用命令 `./grade-lab-syscall` 来获取最终得分，发现报错如下图，表明nproc的计算错误。

```
== Test sysinfotest == sysinfotest: FAIL (1.5s)
    ...
        $ sysinfotest
        sysinfotest: start
    BAD  sysinfotest: FAIL nproc is 60 instead of 62
    GOOD sysinfotest: OK
        $ qemu-system-riscv64: terminating on signal 15 from pid 89890 (make)
    unexpected lines in output
    QEMU output saved to xv6.out.sysinfotest
```

   解决方案：找到kernel/proc.c中实现计算空闲进程的函数 `acquire_nproc()` ，发现写作 `p->state == UNUSED` 判断式子写错了，将其修正为只有不等时才计数（因为要求的是进行中的进程数量），修正后正确。此外，查阅相关资料、仿照上面sleep()等的实现，可以在本处，使用 `acquire(&p->lock);` 与 `release(&p->lock);` 来解决并发问题：如果没有这个锁，可能会在检查进程状态的过程中，进程状态被其他线程或进程改变，导致统计的结果不准确。（在本实验中不使用也没什么问题，因为期间不涉及进程状态的变化。）最终相关代码如下：

```
// Calculate used process
uint64 acquire_nproc(){
  struct proc *p;
  int cnt=0;
  for(p=proc;p<&proc[NPROC];p++){
    acquire(&p->lock);  // lock
    if(p->state != UNUSED){
      cnt++;
    }
    release(&p->lock); // unlock
  }
  return cnt;
}
```

四、实验心得

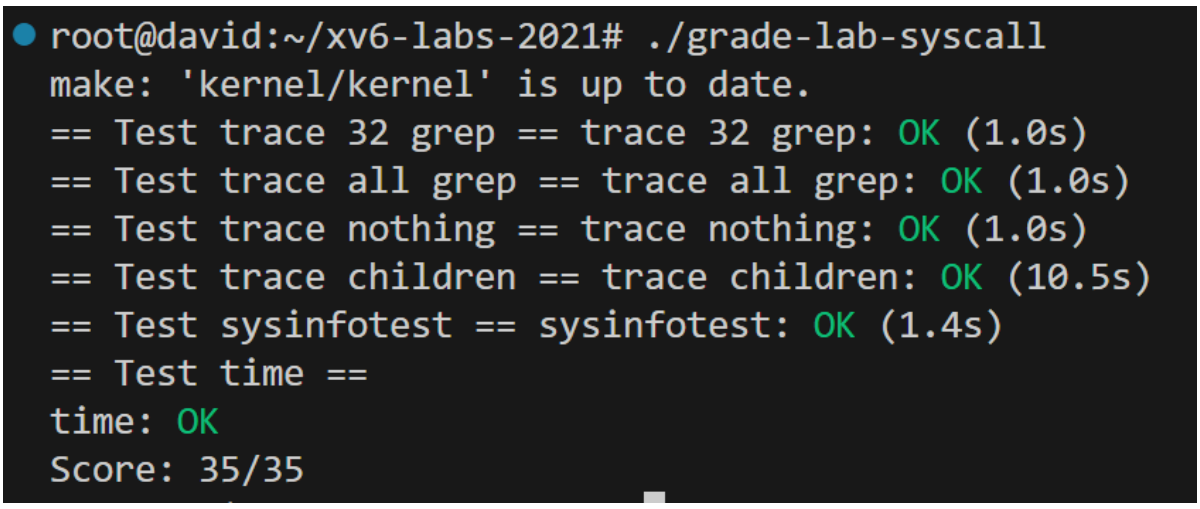
在这次实验中，我学习了如何在xv6操作系统中添加一个新的系统调用。这个过程包括了在用户空间和内核空间之间建立连接，以及如何在内核中实现这个系统调用的功能。这个过程让我更深入地理解了操作系统的工作原理，特别是系统调用的工作机制。

在实验过程中，我遇到了一些问题，比如如何使用 `copyout()` 函数，以及如何理解hints中给出的操作的用途。通过查阅资料和进行实验，我逐渐理解了这些问题的答案。这个过程虽然有些困难，但是我从中学到了很多。

总的来说，这次实验让我对操作系统有了更深的理解，也提高了我的编程能力和解决问题的能力。我期待在未来的学习中能够学到更多的知识。

实验结果

实验结果截图如下：



Lab 3: Page Tables

Speed up system calls

一、实验目的

一些操作系统（例如 Linux）通过在用户空间和内核之间共享只读区域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. 采用以下操作，切换到Lab3实验环境中：

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

2. 查看kernel/memlayout.h了解内存的排列方式，具体代码如下：

```
// User memory layout.
// Address zero first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
//   ...
//   USYSCALL (shared with kernel)
//   TRAPFRAME (p->trapframe, used by the trampoline)
//   TRAMPOLINE (the same page as in the kernel)
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
#ifdef LAB_PGTBL
#define USYSCALL (TRAPFRAME - PGSIZE)
```

3. 根据hints以及实验要求编写程序：
    1. 在kernel/proc.c中的`proc_pagetable()`中添加虚拟页的映射关系：（仿照将trapframe映射到TRAMPOLINE的方式来写）：
        - 在proc.h中添加USYSCALL的结构体：`struct usyscall *usyscall;`
        - 使用`mappages()`函数，将usyscall结构（其物理地址由(uint64)(p->usyscall)给出）映射到用户空间的USYSCALL地址处，映射的页面大小为PGSIZE，允许读和**访问**（不是写）；
        - 执行失败时，将USYSCALL、TRAMPOLINE从页表中删去并释放。
    2. 在`allocproc()`函数中为页面进行初始化：（参考trapframe的代码）
        - 使用`kalloc()`分配一个usyscall页面到进程p中；
        - 将usyscall页面的pid定义为进程的pid；
        - 处理请求页面失败的情况：使用`freeproc(p)`来释放进程p占用的资源，然后释放进程的锁，并返回0表示失败。

3. 确保在`freeproc()`中释放页面：（参考trapframe的代码）
    - 在`freeproc()`中若指针不为空（即为有usyscall），则使用`kfree()`来释放资源，并将该指针指向0（NULL）；
    - 在`freeproc()`所调用的用于释放页面的子函数`proc_freepagetable()`中使用`uvmunmap()`来移除USYSCALL的映射关系。

三、实验中遇到的问题和解决方案

1. **问题一：访问一个没有权限的地址**

   在测试时，发现错误如下图所示，错误信息显示，进程遇到了一个意外的陷阱，具体的陷阱类型是由`scause`字段给出的。在这种情况下，scause的值是0x000000000000000d，这通常表示进程试图访问一个它没有权限访问的内存地址。`sepc`字段给出的是发生陷阱时的程序计数器值，也就是发生错误的指令的地址。`stval`字段给出的是触发陷阱的异常地址或异常指令。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
usertrap(): unexpected scause 0x000000000000000d pid=4
            sepc=0x0000000000000478 stval=0x0000003fffffd000
```

   解决方案：寻找与权限相关的自己写的代码，发现是在`mappages()`建立USYSCALL与TRAMPOLINE的联系时，使用了PTE权限，一开始照搬TRAPFRAME与TRAMPOLINE写成了`PTE_R | PTE_W`，后查阅书籍，发现有相应的文字如下：

   > P30
   >
   > PTE_U controls whether instructions in user mode are allowed to access the page; if PTE_U is not set, the PTE can be used only in supervisor mode.

   且hints中有提示说是*"Choose permission bits that allow userspace to only read the page."*故在此处的权限应该为`PTE_R | PTE_U`，修正后运行正确。

   此外，在查阅资料时有提及在此处若执行`mappages()`失败应该是`uvmunmap()`TRAPFRAME和TRAMPOLINE，而非USYSCALL和TRAMPOLINE，原因在于需要处理的是映射USYSCALL部分内存失败的情况，在USYSCALL没有映射成功的情况下`uvmunmap()`USYSCALL会崩溃（因为在将USYSCALL页map的时候失败了，proc_pagetable这个函数已经不能再继续进行了。所以要把之前加好的TRAMPOLINE和TRAPFRAME给unmap，然后再把该pagetable全部free，最后返回0表示proc_pagetable函数出错了）故最终的相应代码应该如下：

```
// kernel/proc.c
// in function proc_pagetable
// map the the usyscall just below TRAPFRAME
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    //执行失败则将TRAPFRAME、TRAMPOLINE从页表中删去并释放
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
```

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。我学习了如何在xv6的getpid()系统调用中实现这种优化，以加速系统调用。我也了解了内存的布局方式，如何在内核中添加新的内存映射，以及如何处理内存映射失败的情况。

在实验过程中，我遇到了一些问题，例如试图访问没有权限的地址。通过查阅资料和参考书籍，我了解到这是因为我在建立内存映射时使用了错误的权限。我应该使用`PTE_R | PTE_U`来允许用户空间的代码读取这个页面，而不是`PTE_R | PTE_W`。这个经验让我更深入地理解了内存权限的重要性。

总的来说，这次实验让我对操作系统的内存管理有了更深入的理解，也让我更熟悉了xv6操作系统的内部工作机制。我期待在未来的实验中能够继续深化我对这些知识的理解。

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中定义函数`vmprint()`接受一个`pagetable_t`参数，并按照格式打印页表：
    1. 在defs.h中声明`void vmprint(pagetable_t pagetable);`，这样可以在exec.c中调用此函数
    2. 在kernel/vm.c中实现`vmprint()`函数（参考`freewalk()`函数，递归实现`printwalk()`）
2. 将`if(p->pid==1) vmprint(p->pagetable)`复制到exec.c的`exec()`中的`return argc;`之前，以实现打印第一个进程（shell程序）的页表。
3. 在qemu中输入`make grade`以测试实验是否正确。

三、实验中遇到的问题和解决方案

1. **问题一：打印相关的问题**

   解决方案：查阅相关函数的实现与hints总结的打印方案如下：
    1. 用`%p`在printf中输出64位的pte；
    2. 使用`PTE2PA(pte)`获得pte对应的pa；

3. 在打印过程中，使用深度优先搜索（DFS）算法，递归地打印每个页表条目的详细信息，包括其地址和权限等。在每个递归层级，打印额外的`..`来表示当前的深度。

   DFS的具体体现：首先遍历页表中的每一个页表条目（PTE）。对于每一个有效的PTE（即PTE_V位被设置），你首先打印出当前深度的缩进（由`..`表示），然后打印出PTE的信息。如果这个PTE指向的是一个更低级别的页表（即它不包含任何读/写/执行权限，由`PTE_R|PTE_W|PTE_X`检查），就递归地对这个更低级别的页表调用`printwalk`函数，深度加一。

最终的代码如下所示：

```
// simulate freewalk to print vm
void printwalk(pagetable_t pagetable, int depth)
{
  // 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){
      for (int j = 0;j < depth;j++){
        printf("..");
        if (j != depth - 1)
          printf(" ");
      }
      // this PTE points to a lower-level page table.
      // type cast
      uint64 child = PTE2PA(pte);
      printf("%d: pte %p pa %p\n", i, pte, child);

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

void vmprint(pagetable_t pagetable){
  // Use %p in your printf calls to print out full 64-bit hex PTEs and
addresses as shown in the example.
  printf("page table %p\n", pagetable);
  printwalk(pagetable, 1);
}
```

四、实验心得

在这个实验中，我学习了如何在xv6操作系统中实现页表的可视化。这个过程涉及到了对页表的深入理解，包括页表条目（PTE）的结构，页表的层级结构，以及如何通过页表进行地址转换。我也学习了如何在C语言中使用递归来遍历页表的所有条目，并打印出它们的信息。

在实验过程中，我遇到了一些问题，主要是关于如何正确地打印页表的信息。通过查阅相关的文档和函数的实现，我了解到了如何使用`%p`在printf中输出64位的pte，如何使用`PTE2PA(pte)`获得pte对应的pa，以及如何在打印过程中，使用深度优先搜索（DFS）算法，递归地打印每个页表条目的详细信息，包括其地址和权限等。

这个实验让我对页表有了更深入的理解，也让我更熟悉了C语言的递归和打印操作。这将对我未来在操作系统或其他低级编程中的学习和工作有所帮助。

Detect 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. 阅读user/pgtlbtest.c中的`pgaccess_test()`函数，了解`pgaccess`系统调用的使用方式（`pgaccess(起始地址,查找页数,位图地址)`）。
2. 在kernel/sysproc.c中实现`sys_pgaccess()`函数（这个函数将会被用户程序通过系统调用来调用）：
    1. 接收参数：使用`argaddr()`和`argint()`函数接收到的系统调用的参数；
    2. 防御性编程：为查找页数设置上限32；
    3. 计算bitmask：循环下面过程，知道bitmask被填充完成。
        - 在kernel/riscv.h中加上PTE_A的定义：`#define PTE_A (1L << 6)`；
        - 参考`kernel/vm.c`中的`walk()`来正确查找到PTE；
        - 检查PTE的访问位（PTE_A）。如果访问位被设置，说明该页面已经被访问过；
        - 清除访问位；
    4. 输出bitmask：对于输出的位掩码，可以先在内核中创建一个临时缓冲区，在填充完相关的位信息后，再通过`copyout()`函数将其复制到用户空间。
    5. 最后，如果所有操作都成功，函数返回0。如果有任何错误（例如参数解析失败，或者复制到用户空间失败），函数返回-1。
3. 最后，在根目录下添加time.txt和answers-pgtbl.txt，并向其中添加内容，在xv6中输入`./grade-lab-pgtbl`以获得自己的评分。

三、实验中遇到的问题和解决方案

1. **问题一：获得系统调用的参数**

    不同于Lab1中的获得用户输入的参数，在此处是接收系统调用的参数；前者可以使用`argv`来以字符串的形式获得输入，`argv[0]`表示需要执行的操作，其后面的参数表示其他输入参数，如果为整数，需要用`atoi()`函数将其转化。且后面懂得用`argaddr()`和`argint()`后仍然会有错误如下：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
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
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: pgaccess failed, pid=3
```

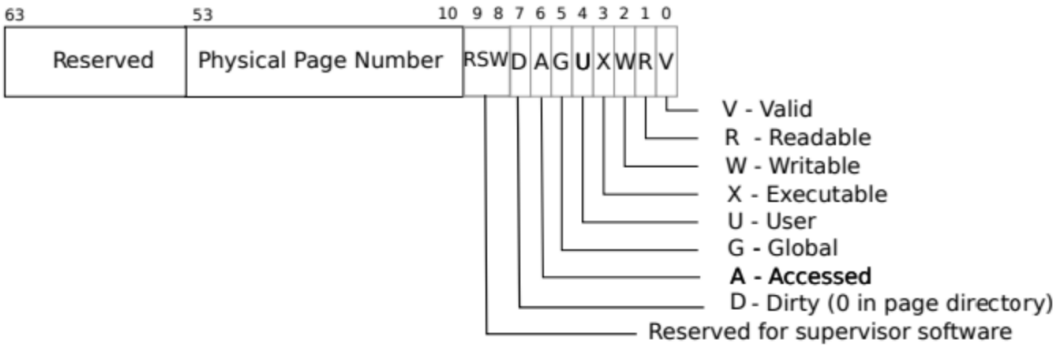
解决方案：根据hints知道应该与`argaddr()`和`argint()`函数有关，模仿sysproc.c中的其他函数接收系统参数的方式来写，第一个参数表明参数的位置**（从0开始）**，第二个表示存储的地址；同时，仿照其他函数，在获取失败时返回-1。错误原因为忽略了参数从0开始。以下为对应的修正后的代码段：

```
// kernel/sysproc.c
// in function sys_pgaccess
  uint64 addr;
  int len;
  int bitmask;

  if(argaddr(0,&addr)<0){
    return -1;
  }
  if(argint(1,&len)<0){
    return -1;
  }
  if(argint(2,&bitmask)<0){
    return -1;
  }
```

2. **问题二：标志位的定义问题**

   解决方案：参考RISC-V架构手册和课本中的内容，我们可以确定`PTE_A`（访问位）的位置应在页表条目（PTE）的第6位。在RISC-V架构中，页表条目（PTE）的每一位都有特定的含义，这些位被定义为`PTE_*`。例如，`PTE_W`表示该页是否允许写操作，`PTE_V`表示该页是否有效。同样，`PTE_A`表示该页是否被访问过。这些位的设置和查询对于页表的管理和虚拟内存的操作至关重要。

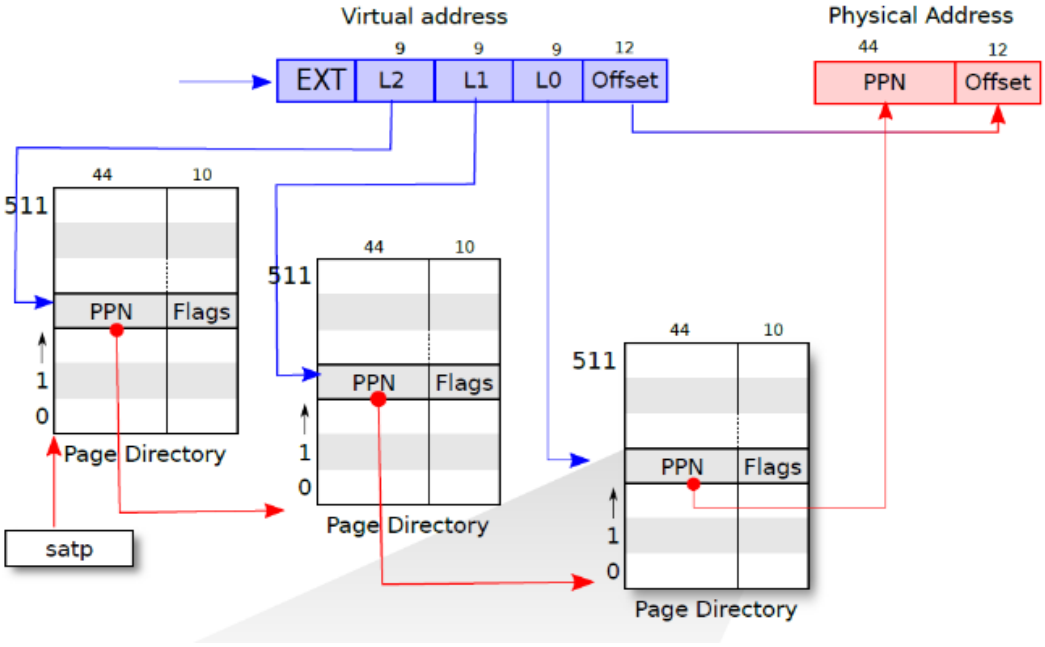


3. **问题三：bitmask的获取**

   在最开始看`walk()`函数时感觉其用法不是很清晰，在编写bitmask的获取的代码时十分模糊。

   解决方案：查阅相关论坛+打印输出+代码调试

   1. `walk()`函数实际上实现虚拟地址到物理地址的映射（三级页表的映射），也会对合法但未创建的页表页进行创建，如下图所示，其输入为页表、虚拟地址、是否有分配空间，返回的是Level0的PTE地址。



   2. 在`walkaddr()`中有实现对于用户页面的查找，可以将其应用于本实验中，仿照其编写`vm_pgaccess()`。其中使用了`*pte = *pte & (~PTE_A);`通过位运算以获得清除PTE_A后的PTE，具体实现代码如下：

```
// kernel/vm.c
int vm_pgaccess(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;
}
```

此外，PTE_A的初值是由硬件设置的，在本实验中不需要实现。

4. **问题四：如何获得满分**

   在使用./grade-lab-pgtbl时一直无法得到满分，在`Test answers-pgtbl.txt`中一直FAIL，报错*Cannot read answers-pgtbl.txt*。

   解决方案：查阅论坛以及阅读操作文档，发现是少在根目录下添加answers-pgtbl.txt文本，添加完成之后，需要向其中加入较多文字，之后提交便可以通过测试，获取满分。

四、实验心得

这次的实验让我对操作系统中的内存管理有了更深入的理解，特别是对于页表管理和虚拟内存的操作。通过实现`pgaccess()`系统调用，我了解到了如何通过检查RISC-V页表中的访问位来报告哪些页面已被访问。这个过程中，我学习到了如何使用`walk()`函数来查找正确的页表条目（PTE），以及如何通过位运算来清除和设置PTE的访问位。

在解决实验中遇到的问题的过程中，我也学习到了很多。例如，我了解到了如何使用`argaddr()`和`argint()`函数来获取系统调用的参数，以及如何通过查阅RISC-V架构手册和课本来确定PTE的访问位的位置。此外，我也学习到了如何使用位掩码来存储和查询页面的访问信息。

总的来说，这次的实验不仅提高了我的编程技能，也加深了我对操作系统中内存管理的理解。我期待着在未来的学习中，能够更深入地探索这些知识。

实验结果

```
root@david:~/xv6-labs-2021# ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (1.4s)
== 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 == (145.9s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

Lab 4: Traps

当xv6系统中出现中断、异常、系统调用三种情况时都会执行陷入机制，陷入到内核态执行一系列操作。本次实验学习如何在trap中实现系统调用。

RISC-V assembly

一、实验目的

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

二、实验步骤

1. 采用以下操作，切换到Lab4实验环境中：

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

2. 在xv6的命令行中输入运行`make fs.img`，编译user/call.c程序，得到可读性比较强的user/call.asm文件（其代码如下）：

```
// user/call.c
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
int g(int x) {
  return x+3;
}
int f(int x) {
  return g(x);
}
void main(void) {
  printf("%d %d\n", f(8)+1, 13);
  exit(0);
}
```

3. 阅读生成的user/call.asm中函数`g`,`f`,`main`实现的汇编代码，并回答问题：（使用`vim user/call.asm`可以用vim打开汇编代码）

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

       函数参数存储在寄存器a0~a7中，例如main函数的`printf`中的13寄存在a2寄存器中。

    2. ***Where is the call to function `f` in the assembly code for main? Where is the call to `g`? (Hint: the compiler may inline functions.)***

       没有这样的代码。`g`被内联inline到 f(x) 中，然后`f`又被进一步内联到`main`中。

```
000000000000001c <main>:

void main(void) {
  1c:   1141                    addi    sp,sp,-16
  1e:   e406                    sd      ra,8(sp)
  20:   e022                    sd      s0,0(sp)
  22:   0800                    addi    s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24:   4635                    li      a2,13
  26:   45b1                    li      a1,12
  28:   00000517                auipc   a0,0x0
  2c:   7a850513                addi    a0,a0,1960 # 7d0 <malloc+0xea>
  30:   00000097                auipc   ra,0x0
  34:   5f8080e7                jalr    1528(ra) # 628 <printf>
  exit(0);
  38:   4501                    li      a0,0
  3a:   00000097                auipc   ra,0x0
  3e:   276080e7                jalr    630(ra) # 2b0 <exit>
```

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

   0x0000000000000628, main 中使用 pc 相对寻址来计算得到这个地址。

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

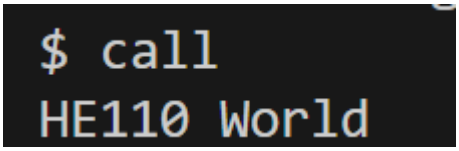
   0x0000000000000038, jalr 指令的下一条汇编指令的地址。

5. ***Run the following code.***

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

   What is the output? [Here's an ASCII table](#) that maps bytes to characters.

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

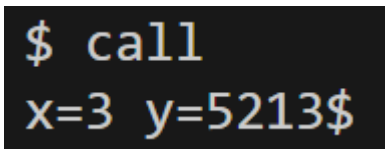
   `HE110 World`;

   需要将`unsigned int i = 0x00646c72;`替换成`unsigned int i = 0x00726c64;`

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

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

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

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

三、实验中遇到的问题和解决方案

1. **问题一：通过调试查看对应参数**

   解决方案：查阅教程发现可以通过gdb来调试，获得对应的值，相应操作如下：

   ```
   make CPU=1 qemu-gdb
   # 新开一个终端
   gdb-multiarch kernel/kernel
   # 进入gbd后执行
   (gdb) set confirm off
   (gdb) set architecture riscv:rv64
   (gdb) set riscv use-compressed-breakpoints yes
   (gdb) target remote localhost:25000
   ```

   接下来，进入调试过程。在调试过程中，通常会使用以下操作：

   - `layout split`：view src-code & asm-code
   - `file xxx`： 调试特定（用户空间）的文件（如`file user/_call`）
   - `b xxx`：在`xxx`处添加断点（如`b main`、`b *0x34`）
   - `c`：继续执行程序，直到遇到下一个断点
   - `s`：单步执行程序源码，如果当前行有函数调用，会进入该函数（`si`为执行汇编）
   - `n`：单步执行程序源码，但在遇到函数调用时不会进入该函数（`ni`为执行汇编）
   - `d`：删除所有的断点
   - `p xxx`：打印对应硬件中的值
     - `p $a0` # 打印a0寄存器的值
     - `p/x 1536` # 以16进制的格式打印1536
   - `i xxx`：查看`xxx`的信息
     - `i r a0` # info registers a0
     - `x/i 0x630` # 查看0x630地址处的指令
     - `x/g 0x80000000` # 查看0x80000000地址处的值（g表示值的长度有64位）

2. **问题二：`jalr`指令的用途**

   在回答第四个问题时，`jalr`这个指令有些令人费解

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

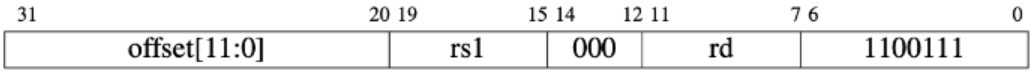
   解决方案：查阅riscv指令集（https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html）以及计算机组成原理相关知识，并gdb调试运行，总结出其功能是：① 把pc + 4 的值记为t ② 把pc的值设置成`$ra + 1528`（0x30+0x5f8=0x628，刚好为`printf`的入口）③ 把ra寄存器的值设置成t。

jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t`

Jump and Link Register. I-type, RV32I and RV64I.

Sets the *pc* to x[*rs1*] + *sign-extend*(*offset*), masking off the least-significant bit of the computed address, then writes the previous *pc*+4 to x[*rd*]. If *rd* is omitted, x1 is assumed.

Compressed forms: **c.jr** rs1; **c.jalr** rs1



3. 问题三：`HE110 World`的输出原因

   解决方案：仔细思考，结论如下：

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

   57616=0xe110，十六进制输出即为`e110`

   `0x64`/`0x6c`/`0x72`分别对应d/l/r，小端序，从低地址（i的最右边）往高地址读，对应`rld`

四、实验心得

通过这次实验，我对RISC-V汇编有了更深入的理解，特别是函数调用和参数传递的过程。我了解到函数参数是通过寄存器传递的，这是因为寄存器的访问速度比内存快得多，所以在函数调用中使用寄存器来传递参数可以提高程序的运行速度。此外，我还了解到了函数调用的过程中，如何保存和恢复调用者的环境，这是通过使用ra寄存器和栈来实现的。

此外，我还了解到了一些关于编译器优化的知识。例如，编译器可能会将一些简单的函数内联到调用它的函数中，以减少函数调用的开销。这在我查看汇编代码时给我带来了一些困扰，因为我没有找到我期望看到的函数调用。但是通过查阅资料和进行实验，我理解了这是编译器为了优化程序性能做的改变。

在解决问题的过程中，我也遇到了一些困难，例如理解某些汇编指令的含义，以及理解printf函数的工作原理。但是通过查阅资料和进行实验，我最终解决了这些问题。这个过程虽然有些困难，但是我从中学到了很多知识，也提高了我的问题解决能力。

总的来说，这次实验让我对汇编语言和函数调用有了更深入的理解，也让我对编译器的工作原理有了更深的理解。虽然过程中遇到了一些困难，但是通过解决这些困难，我收获了很多知识和经验。

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。这对于调试非常有用。

二、实验步骤

1. 将`void backtrace(void);`添加到def.h中声明函数；
2. 向sysproc.c的sys_sleep函数返回之前加入`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;
  // The GCC compiler stores the frame pointer of the currently executing
function in
  // the register s0. This function uses "in-line assembly" to read s0.
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

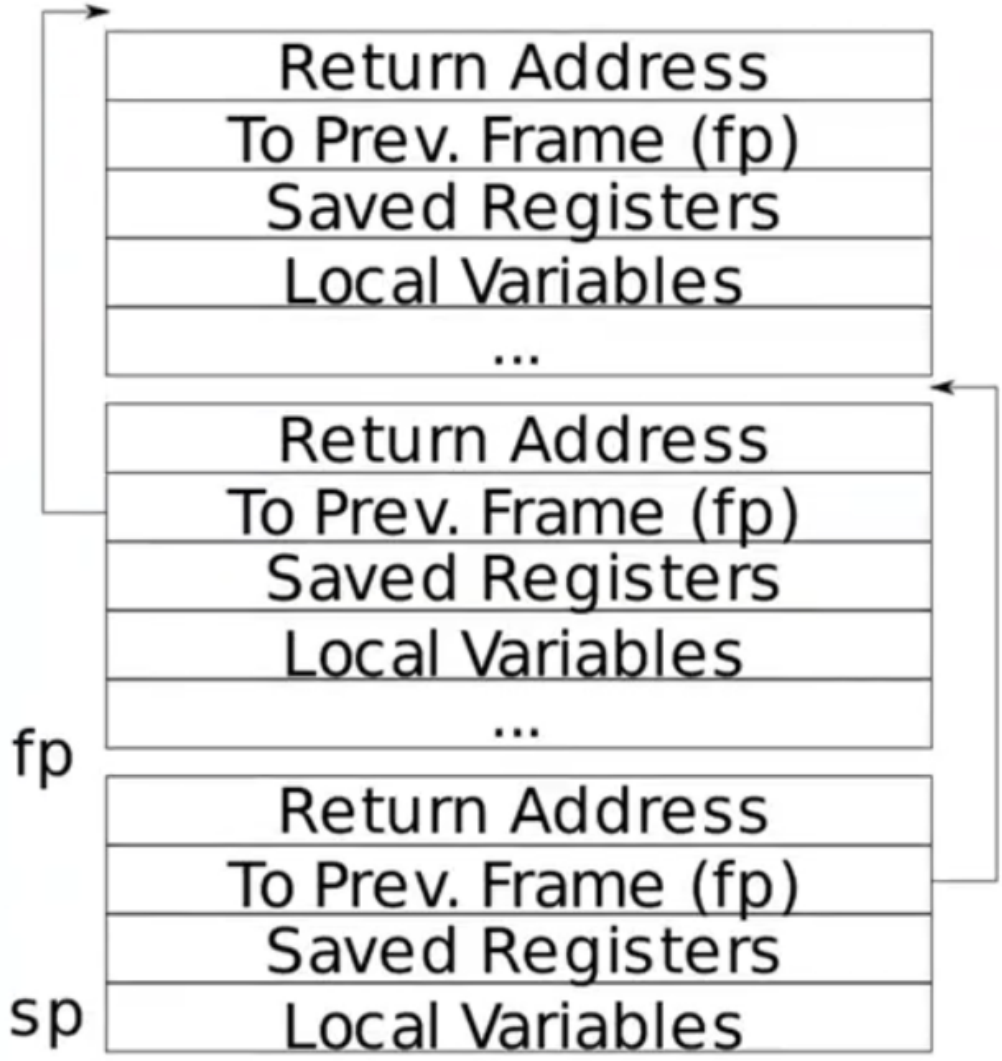
4. 在kernel/printf.c中实现`backtrace()`函数：
5. 使用`make qemu`，在qemu环境中使用`bttest`得到返回地址；
6. 退出qemu，在终端运行`addr2line -e kernel/kernel`，从上面的地址中粘贴地址，得到的结果所对应的代码位置应该就是该函数的返回值位置；
7. 将`backtrace()`加到kernel/printf.c中的`panic()`函数中，应用程序崩溃了就可以把他的调用关系打印出来。

三、实验中遇到的问题和解决方案

1. **问题一：函数调用栈相关知识**

   在看这一节的时候，对这个概念不是很清晰，对“*Note that the return address lives at a fixed offset (-8) from the frame pointer of a stackframe, and that the saved frame pointer lives at fixed offset (-16) from the frame pointer.*”感到十分困惑。

   解决方案：查阅相关资料后总结如下：



1. **函数调用栈（Stack)**
   - 由高地址往低地址增长；
   - 在xv6里，有一页大小（4KB）；
   - 栈指针（stack pointer）保存在sp寄存器里
2. **栈帧（Stack Frame)**
   - 当前栈帧的地址保存在 s0/fp寄存器里
   - 当前栈帧的地址也叫栈帧的指针（frame pointer， fp），指向该栈帧的最高处
   - 栈帧指针往下偏移8个字节是函数返回地址 return address，往下偏移16个字节是上一个栈帧的栈帧指针（previous frame pointer）
   - sp是stack pointer，用于指向栈顶（低地址），保存在寄存器中;fp是frame pointer，用于指向当前帧底部（高地址），保存在寄存器中，同时每个函数栈帧中保存了调用当前函数的函数（父函数）的fp（保存在to prev frame那一栏中）；这些栈帧都是由编译器编译生成的汇编文件生成的。

相应的有如下指令：

```
#查看当前的栈帧(info frame)
i f 0
i f 1
i f 2
#查看当前的函数调用关系(backtrace)
bt
```

最后backtrace()的实现如下：

```
// kernel/printf.c
void backtrace(void){
  printf("backtrace:\n");
  uint64 fp = r_fp();
  uint64 *frame = (uint64 *) fp;
  uint64 up = PGROUNDUP(fp);
  uint64 down = PGROUNDDOWN(fp);
  while (fp < up && fp > down){
    printf("%p\n", frame[-1]);
    fp = frame[-2];
    frame = (uint64 *)fp;
  }
}
```

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。我了解到函数调用栈是程序运行时用来保存函数调用关系的一种数据结构，每次函数调用时都会在栈上创建一个新的栈帧，用来保存函数的局部变量和返回地址等信息。当函数返回时，对应的栈帧会被销毁，函数的返回地址会被取出来，用来恢复调用者的执行环境。

此外，我还了解到了如何使用栈帧指针和返回地址来实现函数的回溯。这是通过在每个栈帧中保存上一个栈帧的栈帧指针和当前函数的返回地址来实现的。通过这种方式，我们可以从任何一个栈帧开始，沿着栈帧指针一直回溯到最初的函数调用。

在实现backtrace函数的过程中，我遇到了一些困难，例如如何正确地读取和解析栈帧中的信息。但是通过查阅资料和进行实验，我最终解决了这些问题。这个过程虽然有些困难，但是我从中学到了很多知识，也提高了我的问题解决能力。

总的来说，这次实验让我对函数调用栈和栈帧有了更深入的理解，也让我对函数回溯有了更深的理解。虽然过程中遇到了一些困难，但是通过解决这些困难，我收获了很多知识和经验。

Alarm

一、实验目的

xv6添加一个功能，该功能可以周期性地提醒一个进程它正在使用CPU时间。这对于想要限制自己消耗的CPU时间的计算绑定进程，或者想要进行计算但也想要定期执行某些操作的进程非常有用。

二、实验步骤

1. 修改Makefile，在UPROGS中加入`$U/_alarmtest\`；
2. 在user/user.h里添加如下声明：

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

3. 同Lab2，分别在添加其他：
    1. 在user/usys.pl中添加存根`entry("sigalarm");`与`entry("sigreturn");`；
    2. 在kernel/syscall.h中添加系统调用号：

```
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

    3. 在kernel/syscall.c中，增添`extern uint64 sys_sigalarm(void);`、`extern uint64 sys_sys_sigreturn(void);`与`[SYS_sigalarm] sys_sigalarm`、`[SYS_sigreturn] sys_sigreturn;
    4. 需要让`sys_sigalarm`记录时钟间隔，为此，先在kernel/proc.h的`proc()`中增添属性：

```
  int alarm_interval;
  int alarm_ticks;
  uint64 alarm_handler;
  struct trapframe alarm_trapframe;
```

    并在kernel/proc.c的`allocproc()`中初始化这些属性:

```
  p->alarm_interval = 0;
  p->alarm_ticks = 0;
  p->alarm_handler = 0;
```

    5. 在kernel/trap.c的`usertrap()`里的`if(which_dev == 2)`分支中处理中断，寻找中断时RISC-V上决定用户空间代码恢复执行的指令地址的地方（`p->trapframe->epc`），将其改为handler;
    6. 在kernel/sysproc.c中实现两个系统调用函数`sys_sigalarm()`与`sys_sigreturn()`；
    7. 完成之后，一旦通过`test0`、`test1`和`test2`，就运行`usertests`以确保没有破坏内核的任何其他部分；
    8. 最后，在终端输入`./grade-lab-traps`以获得最终得分。

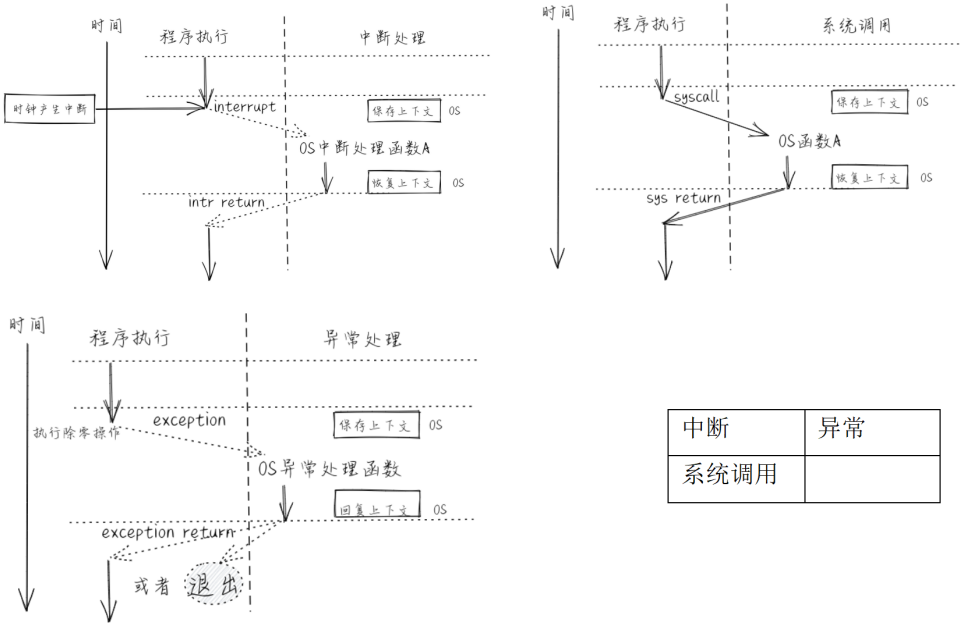
三、实验中遇到的问题和解决方案

1. **问题一：XV6中的陷阱（Trap）机制的实现原理**

   解决方案：查阅书籍、听课、查看笔记与源码，整理笔记如下：

    1. 3种可能的情况使得CPU暂停对正常指令的执行：①系统调用syscall，移交给kernel；②exception，指令执行了非法操作；③设备中断。以上情况合并称为*trap*。xv6对trap的处理分为四个阶段：①RISC-V CPU硬件执行的动作；②汇编文件为内核C文件做的准备；③用C实现的陷阱处理程序；④系统调用或设备驱动服务例程。（下图参考自：https://tarplkpqsm.fei

shu.cn/docs/doccnoBgv1TQlj4ZtVnP0hNRETd#)



中断	异常
系统调用	

2. **RISC-V陷阱机制**：RISC-V CPU有一系列控制寄存器，可以通知内核发生了陷阱，也可以由内核写入来指导CPU如何处理陷阱。当发生陷阱时，CPU会执行一系列步骤，包括清除`SIE`以禁用所有中断，将PC复制到`sepc`，保存当前模式（用户或监督者）到SPP，设置`scause`寄存器以指示陷阱的原因，将当前模式设置为监督者supervisor，将`stvec`的值复制到`pc`，然后开始执行陷阱处理程序。
3. **用户空间的陷阱**：当用户空间发生陷阱时，会将`stvec`的值复制到`pc`，然后跳转到`uservec`，保存一些现场寄存器，恢复内核栈指针和内核页表到`satp`寄存器，再跳转到`usertrap`陷阱处理程序，然后返回`usertrapret`，跳回到内核的`trampoline.S`，最后通过`sret`跳回到用户空间。（保存现场寄存器 -> 恢复内核栈指针和内核页表 -> 跳转到陷阱处理程序 -> 返回用户空间。）
4. **系统调用**：用户执行系统调用时，会将参数放入a0和a1寄存器中，将系统调用的代码放入a7寄存器中，然后执行`ecall`指令陷入内核。内核陷阱代码将保存寄存器的值在当前进程的陷阱帧中。系统调用将提取陷阱帧中的a7寄存器的值，查找对应的系统调用种类并执行，然后将返回值放置在`p->trapframe->a0`中。(用户执行系统调用时，会将参数和系统调用代码放入寄存器中，然后执行陷入内核的指令。系统调用将提取陷阱帧中的值，查找并执行对应的系统调用，然后将返回值放置在陷阱帧中。）
5. **内核空间的陷阱**：当执行内核代码发生CPU陷阱时，`stvec`指向内核向量的汇编代码。内核向量将寄存器的值保存在被中断的内核线程的栈中，而不是陷阱帧中。然后跳转到`kerneltrap`陷阱处理程序。如果是设备中断，调用`devintr`进行处理；如果是异常则`panic`；如果是计时器中断，就调用`yield`让其他内核线程运行。最后返回到内核向量中，内核向量将保存的寄存器值从堆栈中弹出，执行`sret`，将`sepc`复制到`pc`来执行之前被打断的内核代码。
6. 对于上面涉及到的函数，解释如下：
    - **ecall指令**：触发系统调用，涉及寄存器stvec、sepc、scause、sstatus，执行过程包括检查陷阱类型、禁用中断、模式切换、保存和更新程序计数器、设置陷阱原因、保存当前模式。
    - **uservec函数**：用户空间陷阱处理，执行过程包括保存现场、准备内核环境、跳转到usertrap。
    - **usertrap函数**：内核空间陷阱处理，执行过程包括分情况处理陷阱、更新陷阱处理程序地址。

- **usertrapret函数**：从内核空间返回用户空间，执行过程包括准备下一次陷阱、恢复陷阱处理程序地址。
- **userret函数**：从内核空间返回用户空间，执行过程包括恢复现场、准备用户环境、返回用户空间。
- **sret指令**：从管理模式返回用户模式，执行过程包括模式切换、更新程序计数器、重新打开中断。

2. **问题二：实现信号处理机制**

在实验中需要实现一个基本的信号处理机制，即当定时器到达设定的时间间隔时，调用一个预先设定的处理函数。然而，如何正确地设置定时器和处理函数，以及在信号处理函数执行完毕后如何恢复进程的执行状态，是一大挑战。

解决方案：理解并使用系统调用以及内存拷贝函数：

1. 首先，理解并实现`sys_sigalarm`系统调用。这个系统调用用于设置一个定时器，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。这个函数有两个参数，第一个参数是定时器的时间间隔，第二个参数是定时器到时需要调用的处理函数的地址。代码如下：

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

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

  myproc()->alarm_interval = interval;
  myproc()->alarm_handler = handler;
  return 0;
}
```

2. 其次，理解并实现`sys_sigreturn`系统调用。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。它会将进程的`trapframe`（保存进程执行状态的结构体）恢复为信号发生前的状态。在这个函数中，使用了`memmove`函数进行内存拷贝（`void *memmove(void *dest, const void *src, size_t n);`），将`alarm_trapframe`（保存信号处理函数执行前的进程状态）的内容拷贝到`trapframe`（保存当前进程状态），从而恢复进程的执行状态。代码如下：

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

3. 最后，通过实验和代码调试，确保系统调用的正确性，并确保在信号处理函数执行完毕后，进程的执行状态能够正确恢复。

四、实验心得

在这次实验中，我深入理解了操作系统中的信号处理机制，以及如何在xv6系统中实现这一机制。我学习了如何使用系统调用来设置定时器和处理函数，以及如何在信号处理函数执行完毕后恢复进程的执行状态。这个过程中，我对系统调用的工作原理有了更深入的理解，也学习了如何使用内存拷贝函数来保存和恢复进程的状态。

此外，我也遇到了一些挑战，比如理解xv6中的陷阱（Trap）机制的实现原理。通过查阅书籍、听课、查看笔记与源码，我对这个问题有了更深入的理解。我了解了RISC-V CPU如何处理陷阱，以及xv6如何在用户空间和内核空间处理陷阱。我也学习了如何使用系统调用来实现特定的功能，以及如何从内核空间返回用户空间。

总的来说，这次实验让我对操作系统的工作原理有了更深入的理解，也提高了我的编程和调试技能。我期待在未来的实验中继续学习和探索。

实验结果

```
root@david:~/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 (0.6s)
== Test running alarmtest == (3.5s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (127.8s)
== Test time ==
time: OK
Score: 85/85
```

Lab 5: Copy-on-Write Fork for xv6

一、实验目的

实现写时拷贝（Copy-on-Write，简称COW）的`fork()`。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. 采用以下操作，切换到Lab5实验环境中：

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

2. 在kernel/riscv.h中定义`PTE_COW`，表示RISC-V中的RSW（reserved for software）位。这是为了在页表项中标记哪些页是COW页：`#define PTE_COW (1L << 8)`
3. 在kernel/kalloc.c中执行以下操作：
   - 定义`INDEX`宏，用于将物理地址转换为引用计数数组的索引。
   - ```
     #define INDEX(pa) (((char*)pa - (char*)PGROUNDUP((uint64)end)) >> 12)
     ```
   - 修改`kmem`结构，增加一个用于记录每个物理页引用计数的数组`ref_count`，增加一把锁`ref_lock`。
   - 修改`kinit()`函数，使其在初始化物理内存的同时，也初始化引用计数数组和锁。

     ```
     void kinit()
     {
       initlock(&kmem.lock, "kmem");
       initlock(&kmem.ref_lock, "ref");

       uint64 phypages = ((PHYSTOP - (uint64)end) >> 12) + 1;
       phypages = ((phypages * sizeof(uint)) >> 12) + 1;
       kmem.ref_count = (uint*) end;
       uint64 offset = phypages << 12;
       freerange(end + offset, (void*)PHYSTOP);
     }
     ```
   - 修改`freerange()`函数，使其在将物理页添加到空闲列表时，也初始化相应页的引用计数：`kmem.ref_count[INDEX((void*)p)] = 1;`
   - 修改`kfree()`函数，使其在释放物理页时，先减少相应页的引用计数，只有当引用计数为0时，才真正将页放回到空闲列表。
   - 修改`kalloc()`函数，使其在分配物理页时，将相应页的引用计数设置为1。
4. 在kernel/defs.h中声明并在kernel/kalloc.c中实现一些操作引用计数的函数`get_kmem_ref()`、`add_kmem_ref()`。这些函数的目的是抽象引用计数的相关操作，使代码更清晰。
5. 修改kernel/vm.c中的`uvmcopy()`函数，使其在复制页表时不再复制物理页，而是让子进程共享父进程的物理页（使用`mappages()`进行映射），并增加相应页的引用计数。同时，将这些页的写权限去掉，标记为COW页。
     ```

6. 修改kernel/trap.c中的`usertrap()`函数，使其能处理写页错误。
    - 当一个COW页发生写页错误（`r_scause() == 15`）时，为其`kalloc()`分配一个新的物理页，并将旧页的内容复制到新页中`memmove()`，然后将新页`mappages()`安装到页表中，并设置写权限`PTE_W`，最后`kfree()`释放旧页面。
    - 为了实现其功能，还需要修改如下代码：
        - 修改kernel/vm.c中的`mappages()`函数，使其在映射已经映射过的页时不再panic。修改`copyout()`函数，使其在遇到COW页面时，使用与页面错误处理相同的方案。
7. 修改kernel/vm.c中的`copyout()`函数，使其在遇到COW页时，也能处理写页错误。（具体方法与`usertrap()`类似）
8. 最后，在终端输入`./grade-lab-cow`以获得最终得分。

三、实验中遇到的问题和解决方案

1. **问题一：出现异常代码**

   在进行测试时，异常提示如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000009da stval=0x0000000000003f98
usertrap(): unexpected scause 0x0000000000000002 pid=2
            sepc=0x0000000000001004 stval=0x0000000000000000
usertrap(): unexpected scause 0x000000000000000c pid=1
            sepc=0x0000000000001000 stval=0x0000000000001000
panic: init exiting
```

   解决方案：查看RISC-V异常和中断的原因，明白出错的原因。

Exception Code	Description	Reason
2	**非法指令** illegal instruction	程序包含了无效的指令代码，或者尝试执行一些特定的、但是在当前的执行环境下不允许的操作
c（12）	**指令页错误** instruction page fault	程序尝试访问的内存区域并没有被映射到物理内存，或者虽然被映射了，但是并没有被加载到内存中
f（15）	**存储页错误** store page fault	因为程序尝试写入的内存区域并没有被映射到物理内存，或者虽然被映射了，但是并没有被加载到内存中，或者该内存区域被标记为只读

   发现应该是uvmunmap中不恰当的释放导致了问题的出现（某页面已经释放了，但其依赖并未取消，导致异常）。采用*reasonable plan of attack*中的建议用引用计数的方法来记录，具体实现过程中，一开始打印出涉及到的页面数量，发现总物理页表个数为32729，最初是想用全局数组来存放各个页表对应的引用次数，发现不现实（不能根据kinit中计算得到的页面数量动态申请数组大小），后改用链表进行存储，涉及的代码段如下：

```
//kernel/kalloc.c
struct {
  struct spinlock lock;
  struct run *freelist;
  // add ref
  struct spinlock ref_lock;
  uint *ref_count;
} kmem;
int get_kmem_ref(void *pa){
  return kmem.ref_count[INDEX(pa)];
}
void add_kmem_ref(void *pa){
  kmem.ref_count[INDEX(pa)]++;
}
//vm.c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;

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

    // set unwrite and set rsw
    *pte = ((*pte) & (~PTE_W)) | PTE_COW;

    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);

    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
    //add ref_count
     add_kmem_ref((void*)pa);
  }
  return 0;
 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

2. **问题二：将物理地址转换为引用计数数组的索引**

   解决方案：`(char*)pa - (char*)PGROUNDUP((uint64)end)`计算的是物理地址`pa`与内核空间结束地址`end`之间的偏移量，然后右移12位（即除以4096，物理页的大小为4KB）得到的结果就是该物理地址对应的页号。最终通过宏来计算，更加方便：

```
#define INDEX(pa) (((char*)pa - (char*)PGROUNDUP((uint64)end)) >> 12)
```

3. **问题三：处理写页错误I**

解决方案：查阅相关资料、借鉴他人做法，将实现方式总结如下：

1. 获取引发错误的虚拟地址`va`并将其向下舍入到最近的页边界。
2. 使用`walk`函数获取该虚拟地址对应的页表项（PTE）。
3. 检查该PTE是否有效，是否设置了`PTE_COW`位（标记页面是否为COW页面），以及是否允许用户访问。如果任一条件不满足，终止进程。
4. 获取该PTE对应的物理地址`pa`，并获取该物理页面的引用计数`ref`。
5. 如果，将该页面设置为可写，并清除`PTE_COW`位。
6. 如果引用计数大于1，分配新的物理页面，将旧页面内容复制到新页面，将新页面映射到引发错误的虚拟地址，设置`PTE_W`位并清除`PTE_COW`位，然后释放旧页面。（这样，当前进程就可以写入新页面，而不会影响到其他进程）

4. **问题四：处理写页错误II**

解决方案：需要将各个函数的用法弄清楚，然后将其整合于如下代码（对于usertrap而言）。注意，当一个COW页发生写页错误时，我们需要为其分配一个新的物理页，并将旧页的内容复制到新页中，然后将新页安装到页表中，并设置写权限。这就需要我们能够修改已经被映射的虚拟地址的页表项，而不是导致panic，故需要修改`mappages()`的`panic`情况。

```
//kernel/trap.c
extern pte_t* walk(pagetable_t, uint64, int);
//in function usertrap
if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);
    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 (r_scause() == 15) { // write page fault
    // 获取引发错误的虚拟地址va并将其向下舍入到最近的页边界。
    uint64 va = PGROUNDDOWN(r_stval());
    pte_t *pte;
    // 使用walk函数获取该虚拟地址对应的页表项（PTE）
    if (va > p->sz || (pte = walk(p->pagetable, va, 0)) == 0){
      p->killed = 1;
      goto end;
    }
    // 检查该PTE是否有效
    // 检查是否设置了PTE_COW位（标记页面是否为COW页面）
    // 以及是否允许用户访问
    // 如果任一条件不满足，终止进程。
    if (((*pte) & PTE_COW) == 0 || ((*pte) & PTE_V) == 0 || ((*pte) & PTE_U)
== 0){
      p->killed = 1;
      goto end;
    }
    // 获取该PTE对应的物理地址pa，并获取该物理页面的引用计数ref。
    uint64 pa = PTE2PA(*pte);
    acquire(&kmem.ref_lock); //lock
    uint ref = get_kmem_ref((void*)pa);
```

```
    // 如果引用计数为1，将该页面设置为可写，并清除PTE_COW位。
    if (ref == 1){
      *pte = ((*pte) & (~PTE_COW)) | PTE_W;
    }
    else {
      // 分配新的物理页面
      char* mem = kalloc();
      if (mem == 0){
        p->killed = 1;
        release(&kmem.ref_lock);// unlock
        goto end;
      }
      // 将旧页面内容复制到新页面
      memmove(mem, (char*)pa, PGSIZE);
      // 设置PTE_W位并清除PTE_COW位
      uint flag = (PTE_FLAGS(*pte) | PTE_W) & (~PTE_COW);
      if (mappages(p->pagetable, va, PGSIZE, (uint64)mem,flag) != 0){
        kfree(mem);
        p->killed = 1;
        release(&kmem.ref_lock);// unlock
        goto end;
      }
      // 释放旧页面
      kfree((void*)pa);
    }
    release(&kmem.ref_lock);// unlock
  } else if((which_dev = devintr()) != 0){
    // ok
  } 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;
  }
```

对于copyout()而言，其实现类似，分为以下几步：

1. 获取目标虚拟地址对应的页表项。
2. 检查页表项是否有效和用户可访问。
3. 如果页表项是只读的并且被标记为COW页面，执行以下操作：
   - 如果物理页的引用计数为1，直接将物理页设置为可写。
   - 如果引用计数大于1，分配新的物理页，复制旧页内容到新页，然后将新页安装到页表中，并设置写权限。
4. 重新获取目标虚拟地址对应的物理地址，因为可能已经将目标虚拟地址映射到了新的物理页。
5. **问题五：各个文件的作用**

在本节中涉及到了 许多内核文件，其作用和功能一直困惑我。

解决方案：查阅课本，发现了 **图2.2：XV6内核源文件**，以下为本题中涉及到的文件及其作用：

文件	描述
bio.c	文件系统的磁盘块缓存

文件	描述
console.c	连接到用户的键盘和屏幕
entry.S	首次启动指令
exec.c	`exec()`系统调用
file.c	文件描述符支持
fs.c	文件系统
kalloc.c	物理页面分配器
main.c	在启动过程中控制其他模块初始化
pipe.c	管道
printf.c	格式化输出到控制台
proc.c	进程和调度
sleeplock.c	Locks that yield the CPU
spinlock.c	Locks that don’t yield the CPU.
swtch.c	线程切换
syscall.c	Dispatch system calls to handling function.
sysfile.c	文件相关的系统调用
sysproc.c	进程相关的系统调用
trampoline.S	用于在用户和内核之间切换的汇编代码
trap.c	对陷入指令和中断进行处理并返回的C代码
vm.c	管理页表和地址空间

四、实验心得

通过本次实验，我深入理解了写时复制（Copy-on-Write，简称COW）的fork()的工作原理和实现方法。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。

在实验过程中，我遇到了一些问题，包括异常代码的处理、物理地址转换为引用计数数组的索引、处理写页错误等。通过查阅相关资料和借鉴他人的做法，我成功解决了这些问题。我学习了如何修改内核代码，包括定义宏、修改函数、添加新的函数等。这些操作使我对内核编程有了更深的理解。

通过本次实验，我对虚拟内存管理有了更深的理解，包括页表、物理页、虚拟地址和物理地址的关系等。

实验结果

```
root@david:~/xv6-labs-2021# ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (4.5s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests == (138.6s)
== 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
```

Lab 6: Multithreading

本实验与多线程有关，具体包括：在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现一个屏障。

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. 使用如下命令切换实验分支，获取实验资源。

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

2. 将如下汇编代码加入到user/uthread_switch.S中。

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

3. 在user/uthread.c中修改以下内容：
    - 定义一个结构体`tcontext`，用于保存寄存器内容，并将其加入到`thread`结构体中（命名为`context`），这样每个线程都有自己的上下文，可以在切换线程时保存和恢复。

```
// 用于线程切换时保存/恢复寄存器信息。
struct tcontext{
  uint64 ra; // 返回地址
  uint64 sp; // 堆栈指针
  // 被调用者保存的寄存器
  // callee-save registers
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

- 补全函数`thread_create()`，实现找到空闲位置、设置接下来要跑的线程栈以及运行起始位置、然后切换线程。
- 补全函数`thread_schedule()`，根据提示调用`thread_switch()`

```
thread_switch((uint64)&t->context,(uint64)&next_thread->context);
```

4. 在`qemu`中输入`uthread`查看执行情况。

三、实验中遇到的问题和解决方案

1. **问题一：堆栈溢出**

   错误地认为栈是从低地址向高地址增长的，那么在向栈中推入新的元素时，可能会覆盖掉栈之外的其他内存区域，引发程序崩溃。

   解决方案：需要在设置栈指针时，正确地考虑栈的增长方向。将栈指针设置为栈的顶部（高地址），然后每次推入新的元素时，栈指针向下移动（即地址减小）。这样，我们就可以确保不会覆盖掉栈之外的其他内存区域，避免栈溢出的问题，相应代码如下：

   ```
   // 设置新线程的堆栈指针sp为线程堆栈的顶部
   t->context.sp=(uint64)&t->stack+(STACK_SIZE);
   // 设置新线程的返回地址ra为传入的函数func的地址
   t->context.ra=(uint64)func;
   ```

2. **问题二：进程上下文切换的实现**

   解决方案：跟随教程，查阅源码：

   1. kernel/trap.c中的`usertrap`函数是用于将用户空间切换到内核空间，会保存用户空间的32个寄存器然后切换至内核空间中来，在其中有如下代码段：用于查看是否是由于时钟中断导致的，如果是则调用`yield()`函数处理，将CPU资源给让渡出来。（在`kerneltrap`中也有使用）

      ```
      79  // give up the CPU if this is a timer interrupt.
      80  if(which_dev == 2)
      81    yield();
      ```

2. kernel/proc.c中有`yield`函数的实现过程（如下），将进程状态设置为可执行（`RUNNABLE`），下次在调度时可以将其执行，后调用`sched`函数。

```
494  // Give up the CPU for one scheduling round.
495  void
496  yield(void)
497  {
498    struct proc *p = myproc();
499    acquire(&p->lock);
500    p->state = RUNNABLE;
501    sched();
502    release(&p->lock);
503  }
```

3. kernel/proc.c中有`sched`函数的实现过程（如下），在判断是否符合上下文切换的条件后处理中断，后调用`swtch`函数进行上下文切换。

```
467  // Switch to scheduler.  Must hold only p->lock
468  // and have changed proc->state. Saves and restores
469  // intena because intena is a property of this
470  // kernel thread, not this CPU. It should
471  // be proc->intena and proc->noff, but that would
472  // break in the few places where a lock is held but
473  // there's no process.
474  void
475  sched(void)
476  {
477    int intena;
478    struct proc *p = myproc();
479
480    if(!holding(&p->lock))
481      panic("sched p->lock");
482    if(mycpu()->noff != 1)
483      panic("sched locks");
484    if(p->state == RUNNING)
485      panic("sched running");
486    if(intr_get())
487      panic("sched interruptible");
488
489    intena = mycpu()->intena;
490    swtch(&p->context, &mycpu()->context);
491    mycpu()->intena = intena;
492  }
```

4. 在kernel/swtch.S中有`switch`函数的实现过程（如下），将当前的14个callee register全部放到a0以及其偏移量上，再将a1上的内容全部放到14个callee register上。（不同于trap中保存和回复现场需要保存/恢复32个寄存器）

   对于问题“*为什么RISC-V中有32个寄存器，但是swtch函数中只保存并恢复了14个寄存器？*回答如下：

   > 这个问题涉及到RISC-V架构下的寄存器使用规则和函数调用规约（calling conventions）。在RISC-V中，有32个寄存器，但并非所有的寄存器都需要在函数调用时保存。RISC-V的函数调用规约将这些寄存器分为两种类型：Callee Saved Register和Caller Saved Register。

1. Callee Saved Register（被调用者保存的寄存器）：当一个函数（被调用者）准备使用这些寄存器时，它需要首先保存这些寄存器的原始值，以便在函数返回时可以恢复这些值。这样做是为了保护调用者函数的执行环境。
2. Caller Saved Register（调用者保存的寄存器）：**调用者在调用其他函数（被调用者）之前，如果希望这些寄存器的值在函数调用后仍然保持，那么它需要先保存这些寄存器的值。**这样做是因为被调用的函数可能会改变这些寄存器的值。

swtch函数只保存并恢复了14个寄存器，这些寄存器应该就是被定义为Callee Saved Register的寄存器。其他寄存器（Caller Saved Register）的保存和恢复则是由swtch函数的调用者来完成的。在函数调用中，swtch函数只需要关心Callee Saved Register，因为这部分是swtch函数需要负责保存和恢复的。而Caller Saved Register的保存和恢复则是由swtch函数的调用者来负责的，因为它们可能在swtch函数执行过程中被修改。

```
# Context switch
#
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.


.globl swtch
swtch:
        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)

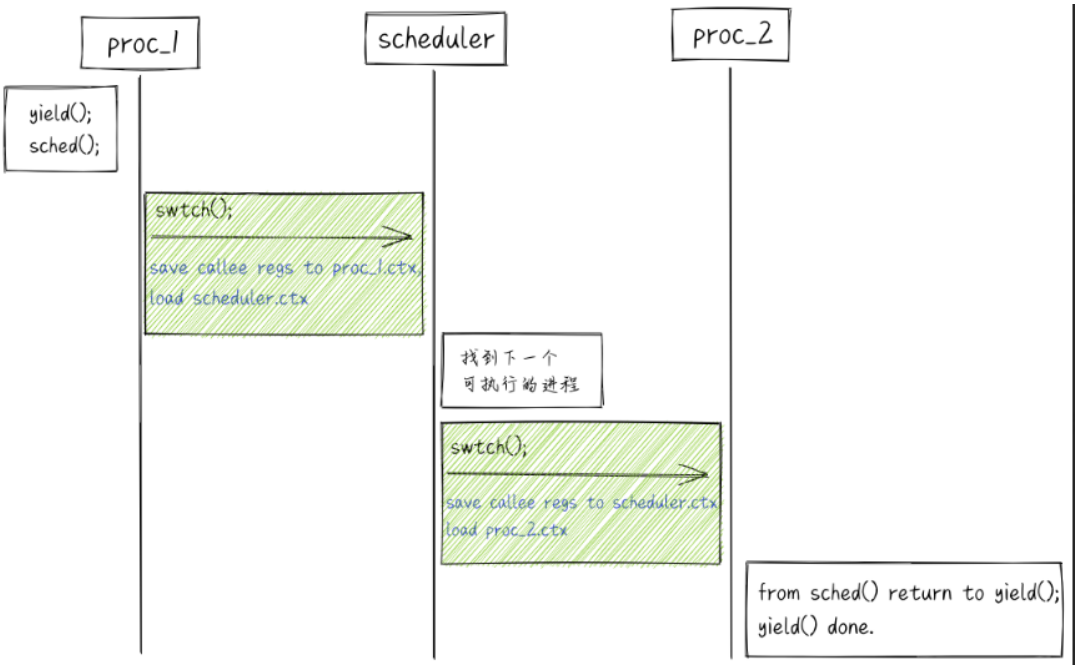
        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

5. 运行完成之后，调试时发现其到`scheduler`调度器中运行，调度策略：**时间片轮转调度**，使用`swtch`来实现上下文切换。

```
430  // Per-CPU process scheduler.
431  // Each CPU calls scheduler() after setting itself up.
432  // Scheduler never returns.  It loops, doing:
433  //  - choose a process to run.
434  //  - swtch to start running that process.
435  //  - eventually that process transfers control
436  //    via swtch back to the scheduler.
437  void
438  scheduler(void)
439  {
440    struct proc *p;
441    struct cpu *c = mycpu();
442
443    c->proc = 0;
444    for(;;){
445      // Avoid deadlock by ensuring that devices can interrupt.
446      intr_on();
447
448      for(p = proc; p < &proc[NPROC]; p++) {
449        acquire(&p->lock);
450        if(p->state == RUNNABLE) {
451          // Switch to chosen process.  It is the process's job
452          // to release its lock and then reacquire it
453          // before jumping back to us.
454          p->state = RUNNING;
455          c->proc = p;
456          swtch(&c->context, &p->context);
457
458          // Process is done running for now.
459          // It should have changed its p->state before coming back.
460          c->proc = 0;
461        }
462        release(&p->lock);
463      }
464    }
465  }
```

综上，可以用如下的流程图描述进程切换过程



四、实验心得

在进行这个实验的过程中，我对多线程编程以及用户级线程系统的设计有了深入的理解，特别是如何设计线程的上下文切换机制。通过这个实验，我也对堆栈的增长方向有了更直观的认识，理解了为什么在大多数现代操作系统中，堆栈是从高位向低位增长的，以及这种设计的好处。

同时，这个实验也让我认识到了在进行多线程编程时，需要格外注意线程间的同步问题，以及堆栈溢出的问题。特别是堆栈溢出问题，如果没有处理好，可能会导致程序崩溃，甚至覆盖掉其他内存区域的数据，引发不可预知的错误。

总的来说，这是一个非常有价值的实验，它帮助我理解了线程的上下文切换是如何工作的，也让我对多线程编程有了更深的理解和实践经验。我相信这个实验对我的编程技能和计算机系统知识都有很大的提升。

Using threads

一、实验目的

探索使用线程和锁的并行编程。将使用一个哈希表。需要在一个真实的Linux或MacOS计算机上完成这个任务，这台计算机需要有多个核心。这个任务使用UNIX pthread线程库。需要修改哈希表，使其在多线程使用时仍然正确。

二、实验步骤

1. 在notxv6/ph.c中进行如下修改：
    - 声明锁：`pthread_mutex_t lock[NBUCKET];`
    - 在main里初始化锁：

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

    - 在`put()`与`get()`中上锁（`pthread_mutex_lock(&lock[i]);`）与开锁（`pthread_mutex_unlock(&lock[i]);`）
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方案

1. **问题一：C中pthread库实现并行**

   解决方案：自助编写代码以运用pthread（用多线程并行实现斐波那契数列计算）

   ```
   #include <stdio.h>
   #include <stdlib>
   #include <pthread.h>

   //按照pthread_create定义规范进行函数定义
   void* p_fib(void* arg){
       //p是在栈中
       int *p=arg;
       //res是在堆中，函数消亡堆还不变
       int *res=malloc(sizeof(int));
   ```

```
    if(*p<2){
        *res=1;
        return res;
    }

    pthread_t pthread1;
    pthread_t pthread2;

    int arg1=*p-1,arg2=*p-2;
    pthread_create(&pthread1,NULL,p_fib,(void*)&arg1);//spawn thread1
    pthread_create(&pthread2,NULL,p_fib,(void*)&arg2);//spawn thread2

    //sync
    int *res1;
    int *res2;
    pthread_join(pthread1,(void*)&res1);
    pthread_join(pthread2,(void*)&res2);
    *res=*res1+*res2;
    free(res1);
    free(res2);
    return res;
}
int main(){
    pthread_t pthread;
    int arg = 4;
    //用arg发送参数
    pthread_create(&pthread,NULL,p_fib,(void*)&arg);
    //等待pthread执行完后才能继续执行return,用res接收结果
    int *res;
    pthread_join(pthread,(void*)&res);
    printf("res: %d",*res);
    return 0;
}
```

2. **问题二：C中pthread库中的锁机制**

   解决方案：自助编写代码以运用pthread+回顾操作系统的课件

   为了避免并发导致的错误可以有两种解决方式：

   1. 使用原子指令：`__sync_...`（只有加减、与或非）
   2. 使用互斥锁：`pthread_mutex_lock`与`pthread_mutex_unlock`，解决哲学家进餐问题、生产者消费者问题（顺序答应ABC，代码如下），最佳实践：
      - 调用pthread_cond_*相关函数时，先获取相关的锁;
      - 调用pthead_cond_wait的时候，最好和while循环搭配使用。

```
#include <stdio.h>
#include "pthread.h"

int bIsDone=0;
int bCouldPrint=0;

pthread_cond_t cond_b;      //条件变量
pthread_cond_t cond_c;      //条件变量
pthread_mutex_t lock;       //锁
```

```
void *f1(void *arg){
    pthread_mutex_lock(&lock);
    //防止f2先创建导致f1阻塞
    while(bCouldPrint==0){
        pthread_cond_wait(&cond_b,&lock);
    }
    printf("B\n");

    bIsDone=1;
    pthread_cond_signal(&cond_c);
    pthread_cond_unlock(&lock);

    return NULL
}
void *f2(void* arg){

    pthread_mutex_lock(&lock);
    printf("A\n");
    bCouldPrint=1;
    pthread_cond_signal(&cond_b);
    pthread_cond_unlock(&lock);

    pthread_mutex_lock(&lock);
    while(bIsDone==0){
        pthread_cond_wait(&cond_c,&lock);
    }
    printf("C\n");
    pthread_cond_unlock(&lock);

    return NULL;
}
void init(){
    pthread_cond_init(&cond_b,NULL);
    pthread_mutex_init(&lock,NULL);
}
int main(){
    init();
    pthread_t pthread1;
    pthread_t pthread2;
    pthread_create(&pthread1,NULL,f1,NULL);
    pthread_create(&pthread2,NULL,f2,NULL);
    pthread_join(pthread1,NULL);
    pthread_join(pthread2,NULL);
    return 0;
}
```

四、实验心得

在这个实验中，我深入学习了如何使用线程和锁进行并行编程。通过实现斐波那契数列和生产者消费者问题，我理解了线程的创建、同步、数据传递和互斥锁的使用。这个实验提高了我在实际项目中应用并行编程的能力，同时激发了我对并行编程和多核系统的进一步学习的兴趣。

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。将使用pthread条件变量，这是一种类似于xv6的sleep和wakeup的序列协调技术。需要在一个真实的计算机上完成这个任务。目标是实现所需的屏障行为。需要在put和get中插入锁和解锁语句，以使两个线程的键丢失数量始终为0。

二、实验步骤

1. 在notxv6/barrier.c中按照要求补充完成`barrier()`函数，其步骤可以分解如下：
    1. **获取互斥锁**：通过`pthread_mutex_lock(&bstate.barrier_mutex);`获取互斥锁，以保护共享资源`bstate`的访问。
    2. **增加线程计数**：通过`bstate.nthread++;`增加线程计数，表示有一个新的线程已经到达了屏障。
    3. **检查是否所有线程都已到达屏障**：通过`if (bstate.nthread < nthread)`检查是否所有线程都已到达屏障。
        - 如果还有线程没有到达屏障（即`bstate.nthread < nthread`），则当前线程会被阻塞，并释放互斥锁，等待其他线程到达屏障。这是通过`pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);`实现的。
        - 如果所有线程都到达屏障（即`bstate.nthread`等于线程总数`nthread`），则进行下一步。
    4. **开始新一轮的屏障**：当所有线程都到达屏障时，通过`bstate.round++;`增加`bstate.round`的值，表示新一轮的屏障开始。
    5. **重置线程计数**：通过`bstate.nthread = 0;`将线程计数`bstate.nthread`重置为0。
    6. **唤醒所有在屏障处等待的线程**：通过`pthread_cond_broadcast(&bstate.barrier_cond);`唤醒所有在屏障处等待的线程，让它们继续执行。
    7. **释放互斥锁**：通过`pthread_mutex_unlock(&bstate.barrier_mutex);`释放互斥锁，允许其他线程访问共享资源`bstate`。

    这些步骤共同实现了一个线程屏障，确保所有线程都到达屏障后才能继续执行。

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

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

三、实验中遇到的问题和解决方案

暂无

四、实验心得

在这个实验中，通过实现一个线程屏障，深入理解了多线程编程中的同步机制。线程屏障是一种常见的同步手段，它可以确保所有线程在继续执行之前都达到某个点。这在需要所有线程同时开始执行某个任务，或者在所有线程都完成某个任务后才能进行下一步操作的场景中非常有用。

实验结果

```
root@david:~/xv6-labs-2021# ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.0s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (11.5s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (28.8s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (4.2s)
== Test time ==
time: OK
Score: 60/60
```

Lab 7: Networking

一、实验目的

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

二、实验步骤

1. 使用如下命令切换实验分支，获取实验资源。

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

2. 依照**hints**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（一步一步来）
   - 声明两个锁：

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

   - 用于传输数据包的函数为`int e1000_transmit(struct mbuf *m)`，其由驱动程序主动调用来传输数据包，其实现如下：
     - 首先，通过读取`E1000_TDT`控制寄存器，向E1000询问等待下一个数据包的TX环索引。

```
acquire(&e1000_txlock);
uint32 tail = regs[E1000_TDT];
```

     - 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。

```
if(!(tx_ring[tail].status & E1000_TXD_STAT_DD)){
    release(&e1000_txlock);
    return -1;
}
```

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

```
if(tx_mbufs[tail])
        mbuffree(tx_mbufs[tail]);
```

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

```
    tx_ring[tail].addr = (uint64)m->head;
    tx_ring[tail].length = (uint16)m->len;
    tx_ring[tail].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
    tx_mbufs[tail] = m;
```

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

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

- 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`。（在第二步中有实现）

- 用于接受数据的函数为`static void e1000_recv(void)`。当操作系统接受到数据包时，其将产生设备中断并在`devintr`中调用`e1000_intr`，该函数调用`e1000_recv`完成对数据包的具体接受工作，其实现如下：
  - 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。

```
    struct mbuf *newmbuf;
    acquire(&e1000_rxlock);
    uint32 tail = regs[E1000_RDT];
    uint32 curr = (tail + 1) % RX_RING_SIZE;
```

  - 然后通过检查描述符`status`部分中的`E1000_RXD_STAT_DD`位来检查新数据包是否可用。如果不可用，请停止。

```
  if(!(rx_ring[curr].status & E1000_RXD_STAT_DD))
      break;
```

  - 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。

```
        rx_mbufs[curr]->len = rx_ring[curr].length;
        net_rx(rx_mbufs[curr]);
        tail = curr;
```

  - 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。

```
        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;
```

  - 最后，将`E1000_RDT`寄存器更新为最后处理的环描述符的索引。

```
    regs[E1000_RDT] = tail;
    release(&e1000_rxlock);
```

三、实验中遇到的问题和解决方案

1. **问题一：E1000硬件的数据包接受与传输机制**

   为了完成本实验，我们必须要了解E1000硬件的数据包接受与传输机制。

   解决方案：查阅提供的"E1000 Software Developer's Manual" + 查看kernel/e1000_dev.h中相应的实现。

   在软件层面上，E1000的接受和传输主要都是基于一种名为**描述符**（descriptor）的数据结构完成的。描述符就像是一个信息中心，它包含了关于网络数据包的所有重要信息，包括数据的存储位置（地址）、数据的大小（长度）、数据的完整性信息（校验和）、描述符的当前状态以及可能的错误信息。通过读取和修改描述符，硬件和软件可以共享数据，进行有效的通信。

   > 1. 地址（addr）：这是描述符数据缓冲区的地址。这个地址是在内存中的位置，它指向的地方是硬件将数据（在这个情况下是网络数据包）直接存储的地方。
   > 2. 长度（length）：这是已经通过DMA（直接内存访问）传输到数据缓冲区的数据的长度。这告诉我们数据包有多大。
   > 3. 校验和（csum）：这是数据包的校验和。校验和是一种检查数据是否在传输过程中被更改的方法。如果数据在传输过程中被更改，那么校验和也会改变，这样我们就知道数据可能已经被破坏。
   > 4. 状态（status）：这是描述符的状态。它告诉我们描述符（以及相关的数据包）的当前状态。例如，它可能会告诉我们数据包是否已经被硬件接收。
   > 5. 错误（errors）：这是描述符的错误信息。如果在处理数据包时发生错误，这个字段就会包含相关的错误信息。
   > 6. 特殊（special）：这是一些特殊的信息，可能会被用于特定的情况。

```
// 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;
};
// [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;
};
```

为了提升E1000设备的大规模数据承载能力，E1000驱动程序维护了**描述符环**数据结构（存放描述符的循环队列）作为数据包传输和接受的缓冲机制。具体实现代码如下：

```
// kernel/e1000_dev.h
#define TX_RING_SIZE 16
static struct tx_desc tx_ring[TX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *tx_mbufs[TX_RING_SIZE];

#define RX_RING_SIZE 16
static struct rx_desc rx_ring[RX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *rx_mbufs[RX_RING_SIZE];
```

四、实验心得

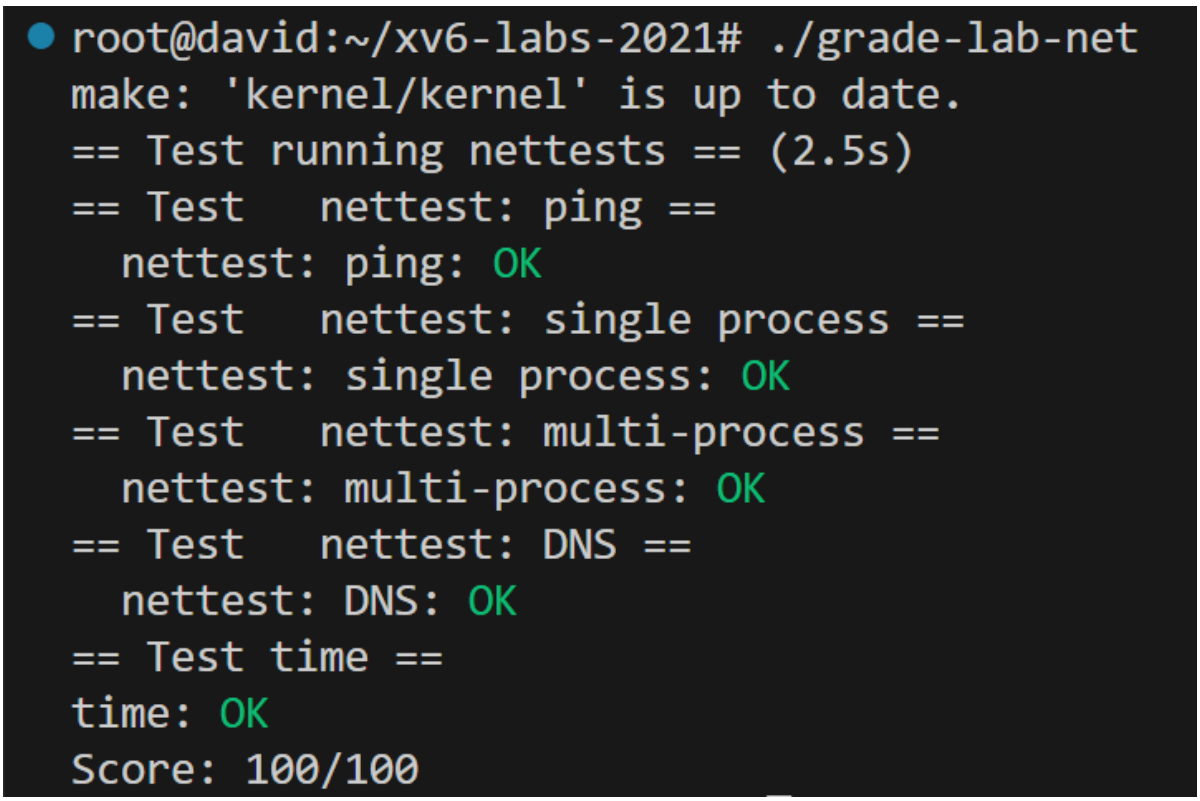
这次实验让我对网络编程有了更深入的理解。通过实现一个基本的网络协议栈，我理解了网络通信的核心机制，包括创建以太网驱动程序，处理ARP请求和响应，实现IP数据包的发送和接收，处理ICMP Echo请求和响应，实现UDP数据报的发送和接收，以及实现一个简单的DHCP客户端以从DHCP服务器获取IP地址。

在实现这些功能的过程中，我遇到了一些挑战，比如理解E1000硬件的数据包接受与传输机制，以及理解描述符和描述符环的概念。但是，通过查阅相关资料和反复调试，我最终成功地完成了这些任务。

这次实验也让我对操作系统如何处理网络通信有了更深入的理解。我了解到，操作系统不仅仅是管理硬件和运行应用程序，它还需要处理复杂的网络通信。这让我对操作系统的复杂性和强大性有了更深入的理解。

总的来说，这次实验虽然有一定的难度，但是它提供了一个很好的机会，让我能够深入理解网络通信的核心机制。我相信这次实验的经验将对我未来的学习和研究大有帮助。

实验结果



Lab 8: Locks

在并发编程中，锁常被用于解决同步与互斥问题，但在多核机器上，若使用锁的方式不当，会引发许多称为“锁竞争”（lock contention）的问题。此实验的目标便是修改涉及锁的数据结构，以减少锁的竞争情况。

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. 使用如下命令切换实验分支，获取实验资源。

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

2. 在kernel/kalloc.c中，进行以下操作：
    1. 为每个CPU分配kmem：

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

    2. 修改 `kinit` 以便对这些锁进行初始化，适应新的数据结构：

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

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

```
// 只有在中断关闭时调用函数cpuid返回当前的核心编号，并使用其结果才是安全的。
// 使用push_off()和pop_off()来关闭和打开中断。
push_off();
int id = cpuid();
acquire(&kmem[id].lock);
r->next = kmem[id].freelist;
kmem[id].freelist = r;
release(&kmem[id].lock);
pop_off();
```

4. 修改`kalloc`以便对申请空间过程中的锁进行处理，其逻辑如下：
    - 关闭中断并获取当前CPU的ID。
    - 获取当前CPU的内存锁，尝试从当前CPU的freelist中取出一个空闲块并返回。如果当前CPU的freelist为空，进入下一步。
    - 开始遍历所有的CPU，尝试从其他CPU的freelist中"偷取"空闲块。具体过程包括：
        - 跳过当前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，偷取其最多1024个空闲块。然后更新两个CPU的freelist，断开偷取的空闲块与其后的空闲块的连接，最后释放其他CPU的内存锁。
    - 在成功偷取到内存后，再次尝试从当前CPU的freelist中取出一个空闲块并返回。如果freelist非空，更新freelist。
    - 释放当前CPU的内存锁，返回空闲块。如果成功分配到内存，使用memset初始化分配的内存块为特定的值。

三、实验中遇到的问题和解决方案

1. **问题一：解决锁竞争**

    xv6将空闲的物理内存kmem组织成一个空闲链表kmem.freelist，同时用一个锁kmem.lock保护freelist，所有对kmem.freelist的访问都需要先取得锁，所以会产生很多竞争。

    解决方案：给每个CPU单独开一个freelist以及其对应的锁lock，这样只有同一CPU上的进程同时获取对应锁时才会产生竞争，其数据结构如下：

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

    在编写过程中，查阅相关资料时，发现有一个相对复杂的问题是：当一个CPU的freelist为空时，需要向其他 CPU的`freelist`“借”空闲块，具体实现如下：

    - 当前CPU`freelist`不空：取出该空闲块，更新freelist，然后将其分配给请求内存的任务。
    - 当前CPU`freelist`为空：从其他CPU的freelist中"借用"空闲块。
        - 遍历所有CPU
        - 获取CPU的锁和freelist
        - 检查freelist是否为空
        - 从非空freelist里面借用空闲块

```
// kernel/kalloc.c
// in function kalloc
  if(r)
    kmem[cpu].freelist = r->next;
  else // steal page from other CPU
  {
    struct run* tmp;
    for (int i = 0; i < NCPU; ++i)
    {
      if (i == cpu) continue;
      acquire(&kmem[i].lock);
      tmp = kmem[i].freelist;
      if (tmp == 0) {
        release(&kmem[i].lock);
        continue;
      } else {
        for (int j = 0; j < 1024; j++) {
          // steal 1024 pages
          if (tmp->next)
            tmp = tmp->next;
          else
            break;
        }
        kmem[cpu].freelist = kmem[i].freelist;
        kmem[i].freelist = tmp->next;
        tmp->next = 0;
        release(&kmem[i].lock);
        break;
      }
    }
```

2. **问题二：字符串格式化**

   提示中有提到可以用kernel/sprintf.c中的`snprintf`函数进行格式化字符串，但之前没有使用过类似的方法。

   解决方案：阅读kernel/sprintf.c中的相关代码，实现字符串的格式化打印。`snprintf`函数的主要用途是以特定的格式将数据写入字符缓冲区，它是C语言中常用的字符串格式化工具。对其实现过程总结如下：

   > 1. 函数接收三个参数：一个字符缓冲区`buf`，缓冲区的大小`sz`和一个格式化字符串`fmt`。此外，`...`表示可变数量的额外参数。
   > 2. 首先，函数检查格式化字符串`fmt`是否为null，如果为null则导致程序崩溃。
   > 3. 使用`va_start`宏来初始化可变参数列表`ap`。
   > 4. 函数通过一个循环来处理`fmt`中的每个字符。对于不是`%`的字符，直接将其放入缓冲区`buf`中。如果字符是`%`，则读取下一个字符来确定要格式化的类型，并从可变参数列表中取出相应的参数进行处理。
   > 5. 函数支持四种格式化类型：
   >   - `%d`：以十进制格式打印整数。
   >   - `%x`：以十六进制格式打印整数。
   >   - `%s`：打印字符串。
   >   - `%%`：打印一个`%`字符。

> 6. 对于未知的格式化类型，函数会打印出`%`和未知类型字符以提醒用户。
> 7. 最后，函数返回实际写入缓冲区的字符数。

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。同时，我也明白了在多处理器环境下，通过关闭和开启中断来保证数据安全的重要性。

通过实验，我掌握了`snprintf`等C语言字符串格式化函数的使用，这对我进一步理解C语言的字符串处理功能有很大帮助。最后，我感到阅读和理解现有代码对解决问题和实现新功能有着重要的作用。

总之，这次实验提升了我的编程技能，同时也让我对操作系统的内存管理有了更深入的理解。

Buffer cache

一、实验目的

Buffer cache是xv6文件系统的一部分，它的作用是保存磁盘的一部分数据，减少磁盘操作的时间消耗。但这也导致所有进程（包括在不同CPU上的进程）都会共享这个数据结构。如果我们仅使用一个锁bcache.lock来保证对它修改的原子性，将会产生大量的竞争，这可能导致性能下降。本实验便是自己设计来解决这个问题。

二、实验步骤

1. 在kernel/bio.c中进行如下操作：
    - 定义buckets数量（可以使用固定数量的散列桶，而不动态调整哈希表的大小。使用素数个存储桶（例如13）来降低散列冲突的可能性。）：

      ```
      #define NBUCKET 13
      ```

    - 修改定义`bcache`
    - 定义哈希函数，以便更快找到对应的桶：

      ```
      int hash(uint blockNo){
        return blockNo % NBUCKET;
      }
      ```

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

      ```
      void
      binit(void)
      {
        struct buf *b;

        initlock(&bcache.lock, "bcache");
        for (int i = 0; i < NBUCKET; i++) {
          initlock(&bcache.hashlock[i], "bcache");
      ```

```
    // Create linked list of buffers
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
    for(b = bcache.hash[i]; b < bcache.hash[i]+NBUF; b++){
      b->next = bcache.head[i].next;
      b->prev = &bcache.head[i];
      initsleeplock(&b->lock, "buffer");
      bcache.head[i].next->prev = b;
      bcache.head[i].next = b;
    }
  }
}
```

- 修改`bget()`函数：首先计算块号的哈希值，然后在对应的桶中查找是否已经有这个块的缓冲区。如果有，就直接返回这个缓冲区。如果没有，就从其他桶中找一个未使用的缓冲区，将其移动到当前桶中，并设置为需要的块。用`&bcache.hashlock[hashcode]`来替换`&bcache.lock`
- 修改`breles`函数：减少缓冲区的引用计数。如果引用计数变为 0，说明没有进程正在使用这个缓冲区，将其移动到链表的头部，表示这个缓冲区是最近最少使用的。用`bcache.head[hashcode]`来替换`bcache.head`，用`&bcache.hashlock[hashcode]`来替换`&bcache.lock`即可

三、实验中遇到的问题和解决方案

1. **问题一：Buffer Cache的原子性修改**

   如果只用一个锁 bcache.lock保证对其修改的原子性的话，势必会造成很多的竞争。

   解决方案：根据数据块的blocknumber将其保存进一个**哈希表**，而哈希表的每个bucket都有一个相应的锁来保护，这样竞争只会发生在两个进程同时访问同一个bucket内的block。bcache的数据结构如下，之后需要对使用到其的地方进行修改。

```
struct {
  struct spinlock lock;
  struct buf head[NBUCKET];
  struct buf hash[NBUCKET][NBUF];
  struct spinlock hashlock[NBUCKET]; // lock per bucket
} bcache;
```

四、实验心得

这个实验的目标是优化 xv6 操作系统的缓冲区缓存。主要方法是引入哈希表，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。同时，使用双向链表管理缓冲区，实现 LRU（最近最少使用）策略。这个实验有助于理解操作系统的缓冲区管理和数据结构性能优化。

实验结果

```
== Test running kalloctest == (93.1s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (8.7s)
== Test running bcachetest == (6.3s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests == usertests: OK (104.0s)
== Test time ==
time: OK
Score: 70/70
```

Lab 9: File System

向xv6文件系统添加大型文件和符号链接。

Large files

一、实验目的

增加xv6文件的最大大小，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. 使用如下命令切换实验分支，获取实验资源。

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

2. 修改kernel/fs.h中宏定义，如下：

   `NDIRECT`：这是直接指向数据块的地址数量；`NINDIRECT`：这是一个间接块可以包含的地址数量；`NDINDIRECT`：这是一个二级间接块可以包含的地址数量；`MAXFILE`：这是一个文件可以包含的最大数据块数量。

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

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

4. 修改kernel/fs.c中的`bmap()`与`itrunc()`函数，使其能够处理二级间接块，实现逻辑如下：

   - 补充二级页表的寻找（仿照第一级的写法来写）：
     检查块号`bn`是否在二级间接块的范围内。如果是，它会进行以下操作：

     1. 检查二级间接块的地址是否已经存在。如果不存在，它会分配一个新的块，并将地址存储在`ip->addrs[NDIRECT + 1]`中。
     2. 使用`bread()`函数读取二级间接块的内容，并将其转换为一个整数数组`a`。
     3. 检查`bn / NINDIRECT`索引处的地址是否存在。如果不存在，它会分配一个新的块，并将地址存储在`a[bn / NINDIRECT]`中。然后，它会使用`log_write()`函数将修改写入日志。
     4. 释放对二级间接块的引用。
     5. 使用`bread()`函数读取`bn / NINDIRECT`索引处的块的内容，并将其转换为一个整数数组`a`。
     6. 检查`bn % NINDIRECT`索引处的地址是否存在。如果不存在，它会分配一个新的块，并将地址存储在`a[bn % NINDIRECT]`中。然后，它会使用`log_write()`函数将修改写入日志。

7. 释放对`bn / NINDIRECT`索引处的块的引用。
8. 返回`bn % NINDIRECT`索引处的块的地址。
9. 如果`bn`不在二级间接块的范围内，它会触发一个panic，表示块号超出了范围。

- 补充二级页表的块的释放（仿照第一级的写法来写）：
  1. 检查`ip->addrs[NDIRECT+1]`是否存在，如果不存在，说明没有二级索引，直接跳过。
  2. 使用`bread()`函数读取二级索引块到缓冲区`bp`，然后将`bp->data`转换为`uint`指针`a`。
  3. 遍历二级索引块中的所有一级索引块地址`a[j]`，如果`a[j]`存在，执行以下步骤：
     1. 使用`bread()`函数读取一级索引块到缓冲区`d_bp`，然后将`d_bp->data`转换为`uint`指针`d_a`。
     2. 遍历一级索引块中的所有数据块地址`d_a[i]`，如果`d_a[i]`存在，使用`bfree()`函数释放这个数据块。
     3. 使用`brelse()`函数释放一级索引块的缓冲区`d_bp`，然后使用`bfree()`函数释放一级索引块。
     4. 将一级索引块地址`a[j]`设置为0，表示这个一级索引块已经被释放。
  4. 使用`brelse()`函数释放二级索引块的缓冲区`bp`，然后使用`bfree()`函数释放二级索引块。
  5. 将二级索引块地址`ip->addrs[NDIRECT+1]`设置为0，表示这个二级索引块已经被释放。

三、实验中遇到的问题和解决方案

1. **bread()和brelse()的使用**

   在实验指南里有提到“别忘了把你`bread()`的每一个块都`brelse()`”为什么？

   解决方案：查阅资料，发现`bread()`和`brelse()`是xv6文件系统中用于操作缓冲区的两个函数：

   - `bread()`：这个函数的作用是读取一个块到缓冲区。它接受两个参数：设备号和块号。首先，它会在缓冲区中查找这个块。如果找到了，就直接返回这个块。如果没有找到，就分配一个新的缓冲区，从磁盘中读取这个块的内容到缓冲区，然后返回这个缓冲区。这个函数的主要作用是减少磁盘I/O操作，因为如果一个块已经在缓冲区中，就不需要再从磁盘中读取。
   - `brelse()`：这个函数的作用是释放一个缓冲区。它接受一个参数：一个缓冲区。首先，它会检查这个缓冲区是否被其他进程使用。如果没有，就将这个缓冲区放回到空闲列表。如果有，就等待其他进程释放这个缓冲区。这个函数的主要作用是管理缓冲区的使用，确保每个缓冲区在同一时间只被一个进程使用。

   在使用`bread()`读取一个块到缓冲区后，一定要记得使用`brelse()`释放这个缓冲区。因为如果不释放，这个缓冲区就会一直被占用，其他需要这个块的进程就无法获取到这个块，可能会导致死锁。

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，特别是如何通过一级和二级间接块来扩展文件的最大大小。我学习了如何修改文件系统的数据结构和相关函数，以支持二级间接块的使用。在实验过程中，我遇到了一些问题，比如如何正确使用`bread()`和`brelse()`函数，通过查阅资料和实践，我解决了这些问题。

我认识到，操作系统中的许多设计，如间接块的使用，都是为了解决实际问题，如文件大小的限制。同时，我也体验到了操作系统设计的复杂性，如何在保证系统性能的同时，处理各种可能出现的情况，如缓冲区的管理，需要仔细的设计和实现。

总的来说，这次实验提升了我对文件系统和操作系统设计的理解，也锻炼了我的编程和问题解决能力。

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. 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（同Lab2），向kernel/stat.h添加新的文件类型（`T_SYMLINK`）以表示符号链接。在kernel/fcntl.h中添加一个新标志（`O_NOFOLLOW`），该标志可用于`open`系统调用。

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

2. 在kernel/sysfile.c中实现的`sys_symlink`：
    1. 首先，函数获取两个参数：目标路径`target`和符号链接的路径`path`。这两个参数都是字符串，分别存储在`target`和`path`数组中。
    2. 然后，开始一个新的文件系统操作，保证文件系统的一致性。
    3. 接着，函数调用`create()`函数来创建一个新的符号链接。`create()`函数的参数包括路径、文件类型（这里是`T_SYMLINK`，表示符号链接）、主设备号和次设备号。`create()`函数返回一个指向新创建的inode的指针。
    4. 如果创建失败，函数会结束文件系统操作，并返回错误。
    5. 如果创建成功，函数会调用`writei()`函数将目标路径写入到新创建的符号链接的数据块中。`writei()`函数的参数包括inode指针、写入的数据、偏移量和写入的数据长度。
    6. 如果写入失败，函数会结束文件系统操作，并返回错误。
    7. 如果写入成功，函数会调用`iunlockput()`函数解锁并释放inode，然后结束文件系统操作。
    8. 最后，函数返回0，表示操作成功。

3. 在kernel/sysfile.c中，修改`sys_open`，设置最大搜索深度为10，如果到达10次，则说明打开文件失败。

```
// 判断是否为符号链接且并不打开符号链接文件本身--最多链接十层，防止循环链接
int layer=0;
while(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
  layer++;
  if(layer==10){
    // 很可能是循环链接
    iunlockput(ip);
    end_op();
    return -1;
  }
  else{
    // 读取 inode
    if(readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
      iunlockput(ip);
      end_op();
      return -1;
    }
    iunlockput(ip);
    // 文件名匹配inode
    ip = namei(path);
    if(ip==0){
      end_op();
      return -1;
    }
    ilock(ip);
  }
}
```

三、实验中遇到的问题和解决方案

1. **问题一：`begin_op()`与`end_op()`的使用方法**

   解决方案：查阅资料，总结如下：

   `begin_op()`和`end_op()`是xv6文件系统中用于同步文件系统操作的函数。在进行文件系统操作时，为了保证文件系统的一致性和防止数据竞争，需要使用这两个函数来包围文件系统操作。

   - `begin_op()`：这个函数的主要作用是开始一个新的文件系统操作。在开始操作之前，它会检查文件系统是否处于安全状态。如果不是，它会等待直到文件系统变为安全状态。然后，它会增加正在进行的操作的数量，并返回。
   - `end_op()`：这个函数的主要作用是结束一个文件系统操作。它会减少正在进行的操作的数量，然后检查是否有其他进程正在等待进行文件系统操作。如果有，它会唤醒这些进程。

   在使用时，需要注意以下几点：

   1. 每次调用`begin_op()`后，都必须调用`end_op()`来结束操作。否则，其他进程可能会无法进行文件系统操作。
   2. `begin_op()`和`end_op()`必须在同一层次的代码块中调用。也就是说，不能在一个函数中调用`begin_op()`，然后在另一个函数中调用`end_op()`。
   3. 在调用`begin_op()`和`end_op()`包围的代码块中，不应该有可能导致进程阻塞的操作。否则，可能会导致其他进程无法进行文件系统操作。

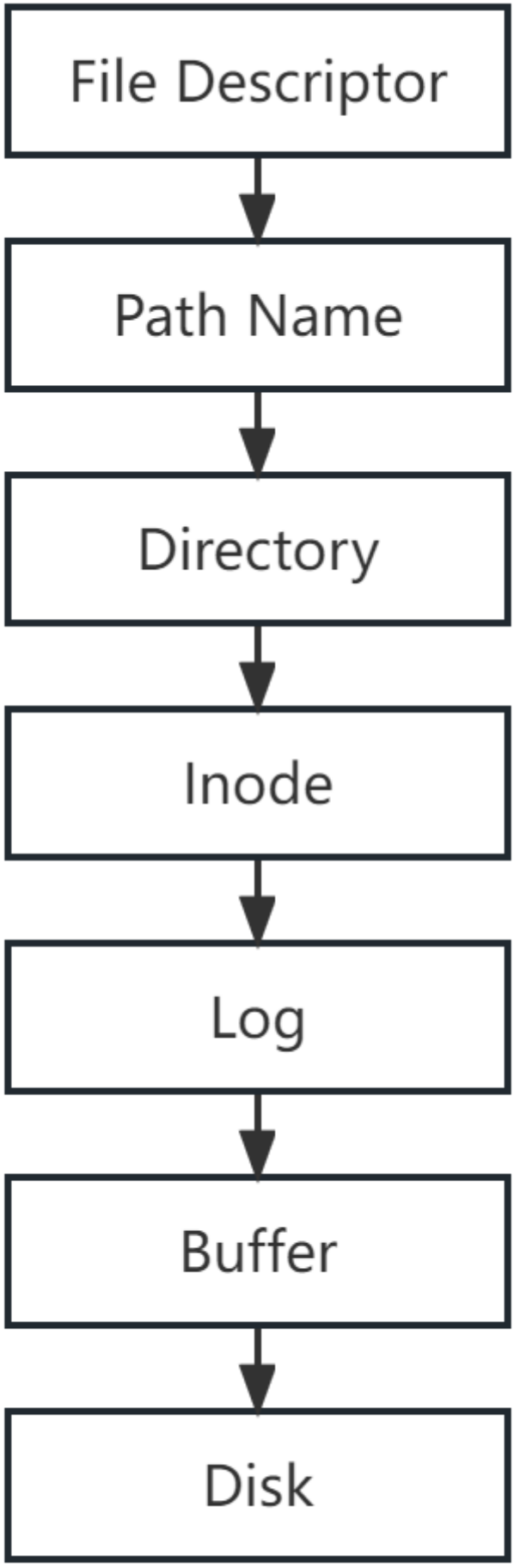
4. 在调用 `begin_op()` 和 `end_op()` 包围的代码块中，应该尽量减少操作的时间，以减少对其他进程的影响。

简而言之，与磁盘交互时调用系统调用都得先用这两句，只有执行成功和roll back，没有执行一半的说法。

2. **问题二：对于文件系统的理解**

在本题中发现出现了许多名词，需要对文件系统进行重新的学习。

解决方案：整个文件系统包括了7层：文件描述符、路径名、目录、inode、日志、缓冲区、磁盘，具体如下：


File Descriptor
Path Name
Directory
Inode
Log
Buffer
Disk

1. **文件描述符**：这是最高层，也是用户程序直接交互的层次。文件描述符是一个非负整数，用于标识一个已打开的文件。用户程序通过系统调用（如read、write、close等）和文件描述符进行交互。
2. **路径名**：这一层处理路径名到inode的转换。当用户程序打开一个文件时，它会提供一个路径名，文件系统需要将这个路径名解析为对应的inode。
3. **目录**：这一层处理目录的操作。目录是一种特殊的文件，它包含了一组文件名到inode的映射。当解析路径名时，文件系统需要查找目录来找到对应的inode。
4. **inode**：这一层处理inode的操作。每个文件都有一个对应的inode，其中包含了文件的元数据，如文件大小、文件类型、文件所有者和权限等，以及指向文件数据块的指针。
5. **日志**：这一层处理文件系统的日志操作。为了保证文件系统的一致性，xv6使用了日志文件系统。当进行修改文件系统的操作时，会先将操作记录到日志中，然后再实际执行操作。如果在执行操作的过程中发生了错误，可以使用日志来恢复文件系统的一致性。
6. **缓冲区**：这一层处理磁盘块的缓存。为了提高性能，文件系统会将经常使用的磁盘块缓存到内存中。当需要读写磁盘块时，首先会查找缓冲区，如果缓冲区中有对应的磁盘块，就可以直接使用，否则需要从磁盘中读取。
7. **磁盘**：这是最底层，处理实际的磁盘操作。当需要读写磁盘块时，如果缓冲区中没有对应的磁盘块，就需要进行磁盘操作。

四、实验心得

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，这是一个非常重要的文件系统功能。我理解了`begin_op()`和`end_op()`的使用方法，这两个函数是xv6文件系统中用于同步文件系统操作的关键函数。我也处理了符号链接的循环引用问题，这是一个常见的问题，需要在实现时进行特殊处理，否则可能会导致无限循环或栈溢出。总的来说，这个实验提供了一个很好的机会，让我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

```
root@david:~/xv6-labs-2021# ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (106.5s)
== Test running symlinktest == (0.8s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (279.8s)
== 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（memory-mapped files）。

二、实验步骤

1. 使用如下命令切换实验分支，获取实验资源。

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

2. 将程序以`$U/_mmaptest\`的形式，添加到Makefile的UPROGS中。
3. 同Lab2，如下修改相应文件：

```
// kernel/syscall.c
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
// kernel/syscall.h
#define SYS_mmap   22
#define SYS_munmap 23
// user/usys.pl
entry("mmap");
entry("munmap");
// user/user.h
void* mmap(void *, int, int, int, int, uint);
int munmap(void *, int);
```

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

```
// mmap
#define VMASIZE 16
struct vma {
  int valid;    // 有效位
  uint64 addr;  // 内存起始地址，可假设始终为0
  int length;   // 映射字节数

  struct file *f;
  int prot;     // 内存是否应映射为可读、可写
  int flags;    // MAP_SHARE 或者 MAP_PRIVATE
  int fd;       // 文件的描述符
  int offset;   // 偏移量，可假定为0
};
```

5. 接下来修改`usertrap`，惰性地填写页表，以响应page fault。即`mmap`不应该分配物理内存或读取文件，通常发生在延迟加载（lazy loading）或者内存映射文件（memory-mapped file）的上下文中，逻辑如下：
    - 首先，检查引发页错误的虚拟地址（`va`）是否有效。如果虚拟地址超出了进程的地址空间，或者与进程的堆栈重叠，那么进程将被标记为需要被杀死。
    - 然后，遍历进程的虚拟内存区域VMA，寻找包含这个虚拟地址的区域。如果找到了包含这个虚拟地址的VMA，那么将为这个地址分配一块新的物理内存，并将这块内存清零。
    - 接下来，从VMA关联的文件中`readi`读取数据到新分配的物理内存中。
    - 然后，根据VMA的权限设置页表项的权限。例如，如果VMA标记为可写，那么页表项也会被标记为可写。
    - 最后，更新页表，将虚拟地址`mappages`映射到新分配的物理内存。如果这个步骤失败（例如，因为内存不足），那么新分配的物理内存将被释放，进程将被标记为需要被杀死。
6. 在kernel/sysfile.c中实现函数`sys_mmap()`：在进程的地址空间中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。VMA应该包含指向映射文件对应`struct file`的指针；`mmap`应该增加文件的引用计数，以便在文件关闭时结构体不会消失（参考`filedup`）：
    - `argaddr` `argint` `argfd`获取系统调用的参数，包括映射的起始地址、长度、权限、标志、文件描述符和偏移量。
    - 检查文件的写权限是否与映射的权限匹配。如果文件不可写，但映射需要写权限，且映射类型为共享映射，那么返回错误。
    - 检查映射的长度是否会导致进程的虚拟内存空间超过最大限制(`p->sz > MAXVA - length`)。如果会，那么返回错误。
    - 遍历进程的虚拟内存区域（VMA）数组，找到一个未使用的区域。
    - 在找到的VMA中设置映射的信息，包括地址、长度、文件、权限、标志、文件描述符和偏移量。
    - 调用`filedup()`函数增加文件的引用计数。这是因为现在有一个新的引用（即VMA）指向这个文件，所以需要增加引用计数，防止文件在还有引用的情况下被关闭。
    - 更新进程的虚拟内存空间大小，并返回映射的起始地址。
7. 在kernel/sysfile.c中实现函数`sys_munmap()`：找到地址范围的VMA并取消映射指定页面（使用`uvmunmap`）。如果`munmap`删除了先前`mmap`的所有页面，它应该减少相应`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`，将页面写回该文件。查看`filewrite`以获得灵感。
    - 获取系统调用的参数，包括需要取消映射的起始地址和长度。
    - 将地址和长度分别向下和向上取整到页的边界，因为内存管理是以页为单位的。
    - 遍历进程的虚拟内存区域（VMA）数组，找到包含需要取消映射地址的区域。
    - 如果没有找到满足条件的VMA，那么直接返回。
    - 如果找到了满足条件的VMA，那么检查VMA的起始地址是否等于需要取消映射的地址。如果等于，那么进行以下操作：
        - 更新VMA的起始地址和长度，将取消映射的部分从VMA中移除。
        - 如果VMA的映射类型是共享映射，那么调用`filewrite()`函数将脏页的内容写回到文件。脏页是指被修改过的页，需要写回到文件以保证数据的一致性。
        - 调用`uvmunmap()`函数取消页表的映射。这个函数的参数包括页表、起始地址、页数和是否释放物理内存。这里设置为1，表示释放物理内存。

        - 如果VMA的长度变为0，那么关闭文件，将VMA标记为无效。
    - 最后，函数返回0，表示操作成功。

```
//munmap
uint64
sys_munmap(void){
  uint64 addr;
  int length;
  if(argaddr(0, &addr) || argint(1, &length)){
    return -1;
  }
  // 地址空间从低向高生长，优先使用了高位
  addr = PGROUNDDOWN(addr);
  length = PGROUNDUP(length);
  struct proc *p = myproc();
  struct vma *vma = 0;
  // 查找满足地址范围的vma
  for(int i = 0; i < VMASIZE; i++) {
    if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length) {
      vma = &p->vma[i];
      break;
    }
  }
    // 若未找到则直接返回
    if(vma == 0){
      return 0;
    }
    // 由实验要求，只需要取消地址与传入地址相同的文件的映射即可
    if(vma->addr == addr) {
      vma->addr += length;
      vma->length -= length;
      // 若需要写回则先将脏页内容写回文件
      if(vma->flags & MAP_SHARED){
        filewrite(vma->f, addr, length);
      }
      // 取消页表映射
      uvmunmap(p->pagetable, addr, length/PGSIZE, 1);
      if(vma->length == 0) {
        fileclose(vma->f);
        vma->valid = 0;
      }
    }
  return 0;
```

8. 修改kernel/vm.c中的uvmcopy和uvmunmap，避免因不合法而panic。
9. 在kernel/proc.c中修改`exit`将进程的已映射区域取消映射，就像调用了`munmap`一样。

```
// 删除文件映射
for(int i = 0; i < VMASIZE; i++) {
  if(p->vma[i].valid) {
    if(p->vma[i].flags & MAP_SHARED)
      filewrite(p->vma[i].f, p->vma[i].addr, p->vma[i].length);
    fileclose(p->vma[i].f);
    uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length/PGSIZE, 1);
    p->vma[i].valid = 0;
  }
}
```

10. 在kernel/proc.c中修改`fork`以确保子对象具有与父对象相同的映射区域。

```
np->state = RUNNABLE;
// 复制父进程的文件映射
for(int i = 0; i < VMASIZE; i++) {
  if(p->vma[i].valid){
    memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
    filedup(p->vma[i].f);
  }
}
```

三、实验中遇到的问题和解决方案

1. **问题一：VMA结构体的设计**

   解决方案：提示中有如下，结合资料对于其的设计如下。

   > 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。

   1. `valid`：有效位，用于标记这个VMA是否正在使用。如果为1，表示该VMA正在使用；如果为0，表示该VMA没有被使用。
   2. `addr`：VMA的起始地址。在xv6中，可以假设地址始终为0。
   3. `length`：VMA映射的字节数。它表示了VMA的大小。
   4. `f`：一个指向`file`结构体的指针，表示该VMA对应的被映射的文件。
   5. `prot`：VMA的保护位，用于标记这个内存区域是否应该被映射为可读或可写。
   6. `flags`：该VMA的标志位，表示该映射是共享的（`MAP_SHARED`）还是私有的（`MAP_PRIVATE`）。
   7. `fd`：这是这个VMA对应的文件的文件描述符。
   8. `offset`：这是这个VMA在文件中的偏移量。在xv6中，可以假设这个偏移量始终为0。

四、实验心得

在本实验中，将`mmap`和`munmap`系统调用添加到了xv6操作系统中，并重点关注了内存映射文件。通过实现相关函数和修改相应的文件，我成功地实现了在进程地址空间中映射文件，并能够根据权限和标志来控制映射的行为。

实验中遇到的问题主要是对VMA结构体的设计和使用，需要仔细理解VMA的作用和每个成员的含义，以确保正确地进行内存映射和取消映射的操作。

通过完成这个实验，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。这对于进程间共享内存、延迟加载和垃圾收集等方面都非常有用。同时，实验也增强了我对操作系统内存管理和文件系统的理解。

实验结果

```
root@david:~/xv6-labs-2021# ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (2.2s)
== 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 (106.4s)
== Test time ==
time: OK
Score: 140/140
```

参考资料

开源代码

https://github.com/PKUFlyingPig/MIT6.S081-2020fall

https://github.com/jlu-xiurui/MIT6.S081-2021-FALL/blob/master/lab8-net

https://github.com/Pikachudy/xv6-labs-2021

https://github.com/onevfall/MIT-6.S081

https://github.com/Baokker/my_xv6_lab

参考教程

官网：https://pdos.csail.mit.edu/6.828/2021

汉化：http://xv6.dgs.zone

笔记：https://fanxiao.tech/posts/2021-03-02-mit-6s081-notes/

课程翻译：https://mit-public-courses-cn-translatio.gitbook.io/mit6-s081/

视频教程：https://www.bilibili.com/video/BV1Qi4y1o7tN/?spm_id_from=333.788&vd_source=54848bbaacc95a6670b0f8ac0228b019

我的仓库地址：https://github.com/tjuDavidWang/xv6-labs-2022