Lab2：应用程序与系统调用（Utilities and System Calls）

该任务中应用程序部分需要同学们认识xv6操作系统，并熟悉其运行环境；在复习并巩固系统调用、进程等理论知识的基础上，掌握在xv6上编写用户程序的方法。

该任务中系统调用部分需要同学们了解xv6系统调用的工作原理的基础上，对操作系统的系统调用模块进行修改，尽可能在真正修改操作系统之前，先对操作系统有一定的了解；并且熟悉xv6通过系统调用给用户程序提供服务的机制。

用户程序的实现

用户程序相当于用户使用操作系统的桥梁，用户通过命令行执行用户程序，用户程序则使用系统提供的一系列服务完成我们想要的功能。这里"系统提供的服务"即所谓的 **系统调用**（syscall）。本次实验就需要利用xv6提供的系统调用实现一些实用的用户程序。

系统调用接口

系统调用被封装为函数以向用户提供接口，用户程序可以通过函数调用的方式请求操作系统的服务，常见的系统调用接口定义如下：（系统调用接口头文件user/user.h）

函数名	功能描述
`fork(void)`	创建一个新的进程，该进程是调用进程的副本。返回新进程的 PID 在父进程中，返回 0 在子进程中。
`exit(int)`	终止调用进程，并将状态码返回给父进程。
`wait(int*)`	阻塞调用进程，直到其一个子进程终止。返回终止子进程的 PID。
`pipe(int*)`	创建一个管道，用于进程间通信。返回两个文件描述符，分别用于读和写。
`write(int, const void*, int)`	将缓冲区的数据写入到指定的文件描述符。返回写入的字节数。
`read(int, void*, int)`	从指定的文件描述符读取数据到缓冲区。返回读取的字节数。
`close(int)`	关闭指定的文件描述符。
`kill(int)`	向指定的进程发送一个信号，通常用于终止进程。
`exec(char*, char**)`	用新的程序替换当前进程的内存映像。成功时不返回，失败时返回 -1。
`open(const char*, int)`	打开指定的文件并返回文件描述符。
`mknod(const char*, short, short)`	创建一个文件系统节点（文件、设备、管道）。

函数名	功能描述
`unlink(const char*)`	删除指定的文件名（取消链接）。
`fstat(int fd, struct stat*)`	获取与文件描述符相关联的文件状态信息。
`link(const char*, const char*)`	创建一个新的硬链接，将文件与新的路径名关联。
`mkdir(const char*)`	创建一个新目录。
`chdir(const char*)`	改变当前工作目录。
`dup(int)`	复制一个文件描述符。返回新的文件描述符。
`getpid(void)`	返回当前进程的进程标识符（PID）。
`sbrk(int)`	调整进程的数据段的大小（动态分配内存）。返回新内存的起始地址。
`sleep(int)`	使进程休眠指定的秒数。
`uptime(void)`	返回系统自启动以来的时间（时钟滴答数）。
`stat(const char*, struct stat*)`	获取与文件名相关联的文件状态信息。
`strcpy(char*, const char*)`	将字符串从源地址复制到目标地址。
`memmove(void*, const void*, int)`	在内存中移动数据块，支持重叠区域的安全复制。
`strchr(const char*, char c)`	在字符串中查找字符的首次出现位置。
`strcmp(const char*, const char*)`	比较两个字符串。返回 0 表示相等，非 0 表示不等。
`fprintf(int, const char*, ...)`	将格式化的输出写入到指定的文件描述符。
`printf(const char*, ...)`	将格式化的输出写入到标准输出（屏幕）。
`gets(char*, int max)`	从标准输入读取一行字符到缓冲区（不安全，不建议使用）。
`strlen(const char*)`	返回字符串的长度（不包括终止符 `\0`）。
`memset(void*, int, uint)`	将内存区域设置为指定的字节值。
`malloc(uint)`	分配指定大小的内存块。返回指向该内存的指针。
`free(void*)`	释放先前分配的内存块。

函数名	功能描述
`atoi(const char*)`	将字符串转换为整数。
`memcmp(const void*, const void*, uint)`	比较两个内存区域。返回值与 `strcmp` 类似。
`memcpy(void*, const void*, uint)`	复制内存区域的数据。

user.h 相关函数讲解

system calls

```
// 将地址：addr处的数据copy到ip指向的地址，返回值代表操作的成功与否
int
fetchaddr(uint64 addr, uint64 *ip)// addr为内存地址，ip为目标指针
{
  struct proc *p = myproc();
  if(addr >= p->sz || addr+sizeof(uint64) > p->sz) // 判断地址是否越界
    return -1;
  if(copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0) // 将用户进程的数据copy到内核空间
    return -1;
  return 0;
}

// 将addr处的string copy到buffer中，返回string的长度（不包括结尾的/0）
int
fetchstr(uint64 addr, char *buf, int max)// addr为内存地址，buf为目标缓冲区指针，max为单次copy的最大长度
{
  struct proc *p = myproc();
  int err = copyinstr(p->pagetable, buf, addr, max);
  if(err < 0)
    return err;
  return strlen(buf);
}

// 从当前进程的陷阱帧（trap frame）中获取特定编号（a0-a5）的系统调用参数。
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
```

```
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}

// Fetch the nth 32-bit system call argument.
// 获取第n个系统调用的参数，放在ip指针处
int
argint(int n, int *ip)
{
  *ip = argraw(n);
  return 0;
}

// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
// 将第n个系统调用的参数放在ip指针处
int
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
  return 0;
}

// Fetch the nth word-sized system call argument as a null-terminated
string.
// Copies into buf, at most max.
// Returns string length if OK (including nul), -1 if error.
// 将第n个系统调用的参数作为字符串放置在buf缓冲区中
int
argstr(int n, char *buf, int max)
{
  uint64 addr;
  if(argaddr(n, &addr) < 0)
    return -1;
  return fetchstr(addr, buf, max);// 上文提到的copy string的函数，将addr处的
string存放到buf处
}

extern uint64 sys_chdir(void);
extern uint64 sys_close(void);
extern uint64 sys_dup(void);
extern uint64 sys_exec(void);
extern uint64 sys_exit(void);
extern uint64 sys_fork(void);
extern uint64 sys_fstat(void);
extern uint64 sys_getpid(void);
extern uint64 sys_kill(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_mknod(void);
```

```
92  extern uint64 sys_open(void);
93  extern uint64 sys_pipe(void);
94  extern uint64 sys_read(void);
95  extern uint64 sys_sbrk(void);
96  extern uint64 sys_sleep(void);
97  extern uint64 sys_unlink(void);
98  extern uint64 sys_wait(void);
99  extern uint64 sys_write(void);
100 extern uint64 sys_uptime(void);
101
102 static uint64 (*syscalls[])(void) = { // 存放各系统调用的服务函数指针
103 [SYS_fork]    sys_fork,
104 [SYS_exit]    sys_exit,
105 [SYS_wait]    sys_wait,
106 [SYS_pipe]    sys_pipe,
107 [SYS_read]    sys_read,
108 [SYS_kill]    sys_kill,
109 [SYS_exec]    sys_exec,
110 [SYS_fstat]   sys_fstat,
111 [SYS_chdir]   sys_chdir,
112 [SYS_dup]     sys_dup,
113 [SYS_getpid]  sys_getpid,
114 [SYS_sbrk]    sys_sbrk,
115 [SYS_sleep]   sys_sleep,
116 [SYS_uptime]  sys_uptime,
117 [SYS_open]    sys_open,
118 [SYS_write]   sys_write,
119 [SYS_mknod]   sys_mknod,
120 [SYS_unlink]  sys_unlink,
121 [SYS_link]    sys_link,
122 [SYS_mkdir]   sys_mkdir,
123 [SYS_close]   sys_close,
124 };
125
126 // 获取系统调用信息
127 void
128 syscall(void)
129 {
130   int num;
131   struct proc *p = myproc();
132
133   num = p->trapframe->a7; // a7处存了系统调用编号
134   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { // 判断系统调用编号
    是否合法
135     p->trapframe->a0 = syscalls[num](); // 取出对应的系统调用函数指针
136   } else { // 返回错误信息
137     printf("%d %s: unknown sys call %d\n",
138             p->pid, p->name, num);
139     p->trapframe->a0 = -1;
140   }
141 }
```

ulibs.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/fcntl.h"
#include "user/user.h"
// 将字符串t copy 到s处，并返回s的首地址
char*
strcpy(char *s, const char *t)
{
  char *os;

  os = s;
  while((*s++ = *t++) != 0) //逐个字符复制
    ;
  return os;
}

// 将字符串p与字符串q按字母序比较，若p>q返回正值,p=q返回0,p<q返回负值
int
strcmp(const char *p, const char *q)
{
  while(*p && *p == *q) //逐个字符进行比较
    p++, q++;
  return (uchar)*p - (uchar)*q;
}

// 返回字符串长度
uint
strlen(const char *s)
{
  int n;

  for(n = 0; s[n]; n++)
    ;
  return n;
}

// 将从dst开始，n个字节，全部赋值为c
void*
memset(void *dst, int c, uint n)
{
  char *cdst = (char *) dst;
  int i;
  for(i = 0; i < n; i++){
    cdst[i] = c;
  }
  return dst;
}

// 将字符串s从第一次出现字符c的地方截取下来作为新字符串，并返回新字符串的初始地址，返回0代表失败
char*
strchr(const char *s, char c)
{
```

```
  for(; *s; s++)
    if(*s == c)
      return (char*)s;
  return 0;
}

// 从buf中获取max个字符，遇到换行即停，常用于获取标准输入中的一整行字符
char*
gets(char *buf, int max)
{
  int i, cc;
  char c;

  for(i=0; i+1 < max; ){
    cc = read(0, &c, 1);
    if(cc < 1)
      break;
    buf[i++] = c;
    if(c == '\n' || c == '\r')
      break;
  }
  buf[i] = '\0';
  return buf;
}

// 将文件n的信息放到st指针处，返回值为操作结果
// struct stat 是 Unix/Linux 系统中定义的一个结构体，包含了文件的多种属性，如大小、权限、创建时间等。
int
stat(const char *n, struct stat *st)
{
  int fd; // 文件描述符
  int r;

  fd = open(n, O_RDONLY); // open函数打开文件n后会返回文件描述符
  if(fd < 0)
    return -1;
  r = fstat(fd, st); // 将文件n的文件信息通过fstat函数将文件信息放到st指针处
  close(fd);
  return r;
}

// 将字符串数字转化为int类型数字
int
atoi(const char *s)
{
  int n;

  n = 0;
  while('0' <= *s && *s <= '9')
    n = n*10 + *s++ - '0';
  return n;
}

//将vsrc处的数据copy到vdst处，大小为n字节
```

```
107 void*
108 memmove(void *vdst, const void *vsrc, int n)
109 {
110   char *dst;
111   const char *src;
112 
113   dst = vdst;
114   src = vsrc;
115   if (src > dst) { //为了防止copy过程中vdst占用了vsrc的空间，将未copy的数据覆盖，需要根据地址大小不同，从不同方向copy
116     while(n-- > 0)
117       *dst++ = *src++;
118   } else {
119     dst += n;
120     src += n;
121     while(n-- > 0)
122       *--dst = *--src;
123   }
124   return vdst;
125 }
126 
127 // 对从s1和s2开始的数据进行逐字节比较，比较方法与strcmp类似
128 int
129 memcmp(const void *s1, const void *s2, uint n)
130 {
131   const char *p1 = s1, *p2 = s2;
132   while (n-- > 0) {
133     if (*p1 != *p2) {
134       return *p1 - *p2;
135     }
136     p1++;
137     p2++;
138   }
139   return 0;
140 }
141 
142 // 将src处的数据copy到dst处，大小为n字节
143 void *
144 memcpy(void *dst, const void *src, uint n)
145 {
146   return memmove(dst, src, n);
147 }
```

实验步骤

1. **部署实验环境**

   实验环境主要分为三部分：xv6运行环境、xv6源码、xv6的编译与运行。

   切换分支

   每个实验项目都在不同的分支上完成，请注意切换分支，例如，实验一需切换到util分支后进行开发。

```
1 $ git branch -a
2 $ git checkout util
```

编译并且运行xv6

Step1 在代码总目录xv6-labs-2020下输入“make qemu”， 编译并运行xv6;

Step2 当可以看到“init: starting sh”的字样表示xv6已经正常启动，此时在“$”提示符后可输入xv6支持的shell命令。

```
问题    输出    调试控制台    终端    端口
yhz@yhz-None:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kerne
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

> **qemu退出方式**
> 先按 `ctrl+a` 组合键，接着完全松开, 再按 `x`

2. **准备工作**
   本次实验需要编写实验内容中介绍的5个 `Unix` 实用程序。初次接触操作系统实验的你可能会感到不知所措，因此不妨先体验一下这些程序的运行效果。实际上， `Linux` 中具备本次实验要实现的一些程序，例如 `sleep` 、 `find` 、 `xargs` 。你可以先尝试在 `Linux` 中使用这些命令，充分体会功能后再开始编程。当然， `Linux` 中命令的功能较为复杂，我们仅要求实现简化版。
3. **编写用户程序**
   在编写程序之前，你要充分理解程序功能，我们将提供一份示例及引导，还希望各位同学可以给出自己的思考及创新！

sleep

sleep命令是Linux系统中的一个非常常用的命令，它用于在终端或脚本执行的过程中让系统进行休眠等待。使用sleep命令可以让程序暂停执行一定的时间，从而达到控制程序执行节奏的效果

要求：利用sleep系统调用，编写一个程序sleep，让程序休眠指定的时长，程序需要在qemu中运行，调用格式为:sleep <second*10>

1. 创建 `user/sleep.c` 文件
2. 由于 `sleep.c` 为新增的用户程序文件，请在 `Makefile` 文件中找到 `UPROGS` ，在 `UPROGS` 上增加一行 `$U/_sleep` :
3. 编译 `xv6` 并运行 `sleep`
4. 执行 `./grade-lab-util sleep` 测试程序

```
135 UPROGS=\
136     $U/_cat\
137     $U/_echo\
138     $U/_forktest\
139     $U/_grep\
140     $U/_init\
141     $U/_kill\
142     $U/_ln\
143     $U/_ls\
144     $U/_mkdir\
145     $U/_rm\
146     $U/_sh\
147     $U/_stressfs\
148     $U/_usertests\
149     $U/_grind\
150     $U/_wc\
151     $U/_zombie\
152     $U/_sleep\
153
```

```
yhz@yhz-None:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kerne
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sleep 10
$
```

问题 输出 调试控制台 终端 端口

```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-util sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.0s)
== Test sleep, returns == sleep, returns: OK (0.8s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
yhz@yhz-None:~/xv6-labs-2020$
```

如果在执行打分程序时遇到`/usr/bin/env: "python": 没有那个文件或目录`报错
可以尝试将`grade-lab-util`文件中`#!/usr/bin/env python`更改为`#!/usr/bin/env python3`

pingpong

利用pipe和fork系统调用，编写一个程序 pingpong ， 实现两个进程之间的通信。

要求：父进程给子进程发送一个信息 “ping”，子进程接收信息后打印 "<子进程pid>: received ping" 。然后子进程发送一个信息 “pong” 给父进程，父进程接收信息后打印 "<父进程pid>: received pong" ，然后退出。程序需要在qemu中运行，调用格式为:pingpong

主要的点有两个:

```c
// 第一点 pipe的使用
int fd[2];
pipe(fd);
// pipe()会创建一个单向通信的通道文件，fd[1]为写端的file descriptor，fd[0]为读端的file descriptor
// 使用read,write函数可以对通道文件进行读写

// 第二点 父子进程的管道
// fork创建完子进程后，子进程依然可以使用父进程创建的fd[2]进行通道文件的读写
// 由此我们可以利用先开好pipe再fork，完成父子进程的communication
// 假设父进程负责写，子进程负责读，则在父进程中需要close(fd[0])，子进程中需要clode(fd[1])以避免错误操作和file descriptor的数量超过上限
```

1. 创建`user/pingpong.c`文件
2. 由于`pingpong.c`为新增的用户程序文件，请在`Makefile`文件中找到`UPROGS`，在`UPROGS`上增加一行`$U/_pingpong`：
3. 编译`xv6`并运行`pingpong`
4. 执行`./grade-lab-util pingpong`测试程序

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (2.0s)
```

primes

利用fork、pipe、wait系统调用，编写一个程序primes，实现一个素数筛（2~35）

要求：第一个进程创建一个子进程，向管道里放入数字2~35，子进程会依次读取这些数字，每当子进程读到一个素数时，该子进程会输出该素数（prime <n>），并创建一个新的子进程来完成后续素数的筛选，请想办法将父进程创建的管道传递下去。所有进程需要等待子进程执行完毕才可以结束。

1. 创建`user/primes.c`文件
2. 由于`primes.c`为新增的用户程序文件，请在`Makefile`文件中找到`UPROGS`，在`UPROGS`上增加一行`$U/_primes`：
3. 编译`xv6`并运行`primes`
4. 执行`./grade-lab-util primes`测试程序

```
hart 2 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
```

```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (1.4s)
```

find

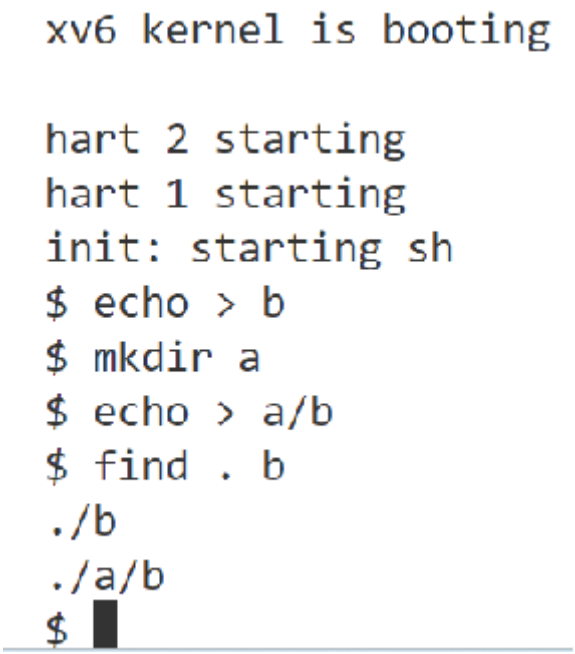
参考ls指令，编写一个程序find

要求：给定一个初始路径和目标文件名，要不断递归的扫描找到所有子目录中叫该名字的文件，并输出路径。程序需要在qemu中运行，调用格式为:find <des_path> <des_file_name>

从`ls`用户函数的实现里，主要可以领悟到4个点:

```
// 1. 如何知道一个路径是文件还是目录（文件夹）?
int fd = open(curr_path, 0);
fstat(fd, &st);
// st.type:T_DIR--目录, T_FILE--文件

// 2. 如何递归地扫描一个目录?
// 如果已知一个fd对应的是一个目录，我们可以不断读取DIRSIZ大小bytes来便利每个子文件/子文件目录
struct dirent de;
while(read(fd, &de, sizeof(de)) == sizeof(de)) {
  // de.name 得到文件名
  ...
}

// 3. 我们需要一路上自己不断构造完整路径，需要保存一个buffer，并在合适的时候加上'/[subdirectory]'进入递归

// 4. 记得及时关闭不再使用的file descriptor,
```

1. 创建`user/find.c`文件
2. 由于`find.c`为新增的用户程序文件，请在`Makefile`文件中找到`UPROGS`，在`UPROGS`上增加一行`$U/_find`：
3. 编译`xv6`并运行`find`
4. 执行`./grade-lab-util find`测试程序




```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-util find
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK (1.5s)
== Test find, recursive == find, recursive: OK (1.4s)
```

xargs

利用fork和exec系统调用，编写一个程序xargs

要求：实现一个指令参数拼接器程序需要在qemu中运行，调用格式为:xargs <command> <args_0> \n（换行） <args_1> \n <args_2> \n...该程序会将<args_i>依次与<args_0>拼接，并交由<command>执行即：<command> <args_0+args_1> ;<command> <args_0+args_2>; <command> <args_0+args_3> ...将标准输入里每一个以`'\n'`分割的行作为单独`1`个额外的参数，传递并执行下一个命令. 这题主要是考察`fork` + `exec`的使用.

有以下这几个点需要思考的:

```
# 1. 什么时候知道不会有更多的行输入了?
当键入Ctrl + D时结束

# 2. 怎么能抓出每一个以'\n'分割的行?
我们不能方便地从file descriptor里读到空行符为止, xv6没有这样的库函数支持.
我们需要自己管理一个滑动窗口buffer, 如下:

假设我们有1个长度为10的buffer, 以 . 代表为空
buf = [. . . . . .]
我们read(buf, 10)读进来10个bytes
buf = [a b \n c d \n . . . .]

这时我们需要做的是,
1. 找到第一个'\n'的下标, 用xv6提供的strchr函数, 得到下标为2
2. 把下标0~1的byte转移到另一个buffer去作为额外参数
3. 执行fork+exec+wait组合拳去执行真正执行的程序, 使用我们parse出来的额外的参数
4. 修建我们的buffer, 把0~2的byte移除, 把3~9的byte移到队头
此时buffer变成:
buf = [c d \n . . . . . .]

# 3. 上述操作要一直循环直到
```

```
file descriptor 0 已经关闭了 && buffer里的所有读入的bytes都处理完了
```

```
$ xargs echo good   # 指定要执行的命令：echo，同时输入参数'good'
  bye               # 换行后继续输入echo的参数'bye'
  good bye          # 执行"echo good bye"，输出"good bye"
  hello too         # 换行后输入参数'hello too'
  good hello too    # 执行"echo good hello too"，输出"good hello too"
  通过ctrl+D结束输入
```

```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-util xargs
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (2.7s)
```

系统调用原理说明

> 这一章讲述本次实验的实验原理，以下知识会大大帮助你理解xv6是如何工作的。但是你需要加以思考，并且将其一个一个串接起来，以形成一个整体的思维。

1. 系统调用

系统调用就是调用操作系统提供的一系列内核函数。由于用户程序运行在`CPU`的用户态（又称非特权模式，用户模式），无法直接访问系统硬件和操作系统中的系统数据，用户程序只能发出请求，然后由内核调用相应的内核函数来帮着处理，最后将结果返回给应用程序。

1.1 系统调用的使用：用户的权力

在第一个实验中，我们将系统调用和C语言的函数放在一起做了一定的说明。

“系统调用被`封装`为函数以向用户提供接口，用户程序可以通过函数调用的方式请求操作系统的服务”。

从这里我们需要注意，虽然两者看起来一样，但是这是因为封装的结果。接下来我们简要介绍xv6是怎样封装的。

首先看`user/user.h`文件，我们可以看到对应有许多的函数接口，其中标识了系统调用和用户库（`ulib`）。但是我们只看到了封装的接口，那么实际上系统调用的实现是什么样的呢?

1.2 系统调用的接口：操作系统内核和用户程序的中间体

现在我们看到`user/usys.pl`文件，该脚本文件会在编译期间被执行，生成一个汇编文件`usys.S`。其对每一个系统调用抽象接口都生成了一个具体的实现，叫做`entry`。

```
#!/usr/bin/perl -w

# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
entry("exit");
```

```
20  ......
```

这段脚本中的`print`的内容实际就是`RISC-V`指令集的汇编语言。`$name`就是系统调用的名字，如`fork`，`exit`等，当脚本执行时，`18`行之后`entry`的内字符串会被填入`$name`。
系统调用步骤包括以下几步：

- `li a7, SYS_${name}\n`：将系统调用号`SYS_${name}`（来自`kernel/syscall.h`）传给`RISC-V CPU`上的`a7`寄存器，这样内核就可以通过`a7`寄存器知道现在要处理的是什么系统调用。
- `ecall`：特殊指令，用来转入操作系统内核（关键指令）。
- `ret`：操作系统内核执行完后会返回到这里，执行`ret`就结束了用户看到的系统调用，返回至用户程序。
  你可以在编译后阅读`user/usys.S`，将其与上述脚本对应以便更好地理解。

1.3 系统调用的参数：寄存器传参

这里我们还忽略了一点，那就是系统调用还需要传递参数。实际上，从汇编的角度来看，当我们调用一个函数的时候，传入的参数一般会按照位置依次放在`a0，a1，a2`等以此类推的寄存器中。

寄存器	接口名称	描述	在调用中是否保留？
Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6–7	t1–2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10–11	a0–1	Function arguments/return values 函数参数	No /返回值
x12–17	a2–7	Function arguments 函数参数	No
x18–27	s2–11	Saved registers 保存寄存器	Yes
x28–31	t3–6	Temporaries 临时寄存器	No
f0–7	ft0–7	FP temporaries 浮点临时寄存器	No
f8–9	fs0–1	FP saved registers 浮点保存寄存器	Yes
f10–11	fa0–1	FP arguments/return values 浮点参数/返回值	No
f12–17	fa2–7	FP arguments 浮点参数	No
f18–27	fs2–11	FP saved registers 浮点保存寄存器	Yes
f28–31	ft8–11	FP temporaries 浮点临时寄存器	No

图 3.2 RISC-V 整数和浮点寄存器的汇编助记符。RISC-V 有足够的寄存器，如果过程或方法不产生其它调用，就可以自由使用由 ABI 分配的寄存器，不需要保存和恢复。调用前后不变的寄存器也称为“由调用者保存的寄存器”，反之则称为“由被调用者保存的寄存器”。浮点寄存器将第 5 章进行解释。（这张图源于[Waterman and Asanovi´c 2017]的表 20.1。）

也就是说，调用`user/user.h`中的函数接口后，参数已经存储于寄存器了，这个时候我们就原封不动地继续调用`ecall`，操作系统就可以通过`a0，a1，a2`等寄存器来获取参数了。此外，函数的返回值一般存储于`a0`寄存器。

1.4 系统调用的分发和实现：解耦合

按照之前所述，定义一个系统调用实际上很简单：将系统调用号的宏定义`SYS_${name}`添加在`kernel/syscall.h`，每次系统调用时根据`a7`寄存器传入的值判定具体的系统调用类型即可。 那么接下来我们面临的第一个问题是：内核怎么通过系统调用号（存在`a7`)来执行不同的系统调用函数。

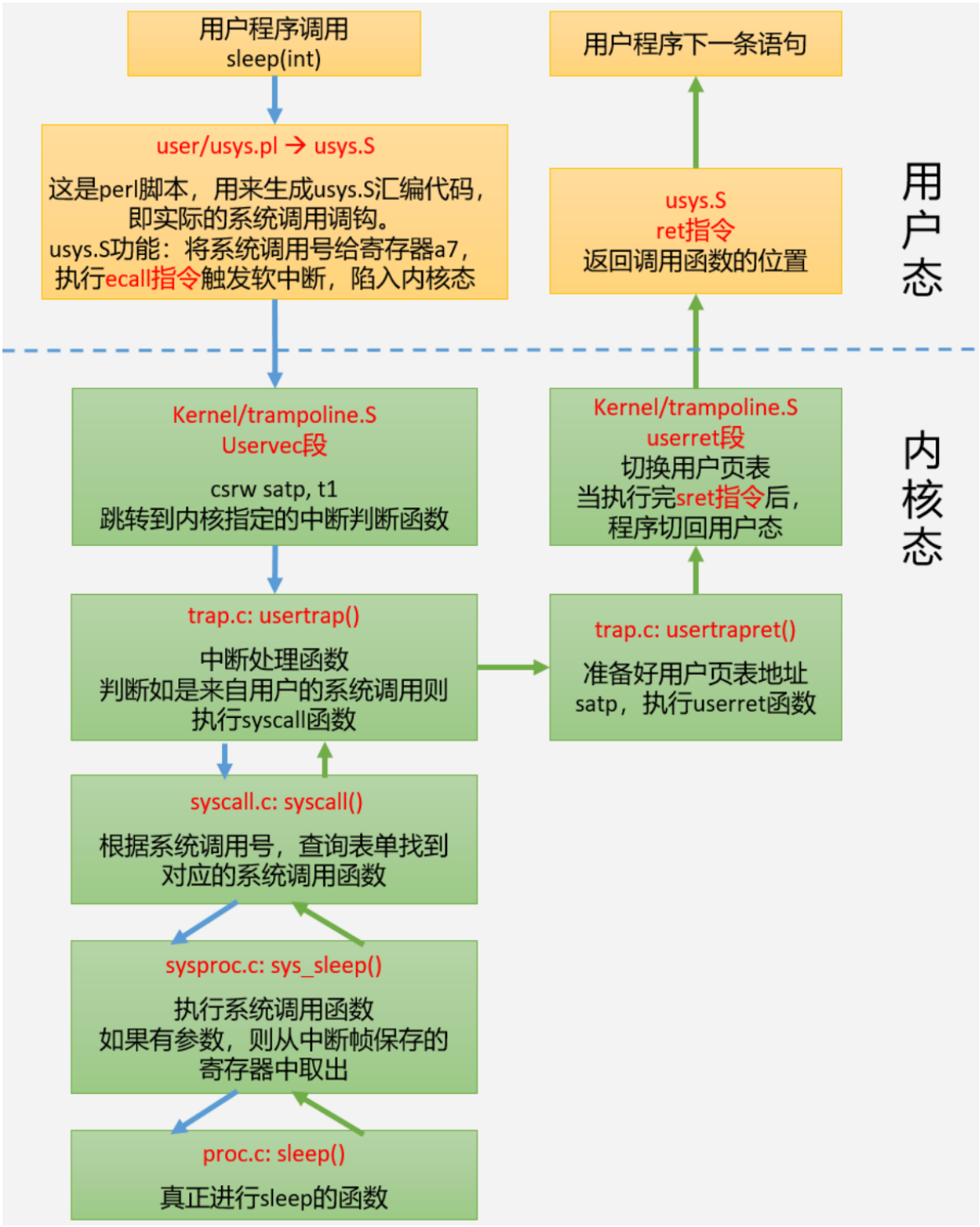
> 我们需要关注`kernel/syscall.c`中的代码，`line 86-130`定义了一大段看起来就和系统调用有关的部分。这里可以分为两个部分，一个是`extern`进行标识的函数接口，另一个是以`syscall`为名的数组。
> 前者实际上声明了这些函数，这些函数的实现不必在这个文件中，而是分布在各个相关的代码文件

> 中（一般放在`sys`开头的文件中，包括`sysproc.c`与`sysfile.c`），我们在这些代码文件中实现好对应的函数，最后就可以编译出对应名字的汇编代码函数，`extern`就会找到对应的函数实现了。
> 后者则是将这些函数的指针都放在统一的数组里，并且数组下标就是系统调用号，这样我们在分辨不同系统调用的时候就可以很方便地用数组来进行操作了。`kernel/syscall.c`中的`syscall()`函数就根据这一方法实现了系统调用的分发（通过不同系统调用号调用不同系统调用函数），请仔细阅读并尝试理解。
> 将两者合起来使用，可以使得系统调用的**实现**和系统调用的**分发**彼此分离，这对函数编写者非常友好，但是会让初学者有些迷惑，这是需要注意的。

2. 举例：系统调用的实现

现在我们以一个具体的例子，来看看`xv6`是怎么实现系统调用的。我们准备挑选看起来朴实无华的系统调用`sleep`，因为它实际上暗藏凶险。

先来看张总图把握一下整体流程：



以下只对系统调用过程某几个关键点进行分析，其他相关代码/流程需要大家自行查阅代码：

2.1 用户程序调用

xv6 关于 sleep 系统调用接口的声明在 user/user.h。

user.h

user > C user.h

```
22    int getpid(void);
23    char* sbrk(int);
24    int sleep(int);
25    int uptime(void);
26
27    // ulib.c
28    int stat(const char*, struct stat*);
29    char* strcpy(char*, const char*);
30    void *memmove(void*, const void*, int);
```

2.2 usys.S 汇编

然而，user/user.h 只是对函数原型进行了声明。具体做了什么事呢？这个定义是在 usys.S 中，详见 1.2 小节。

需要注意的是，usys.S 汇编是由 usys.pl （perl 脚本）自动生成的。也就是，当你要增加新的系统调用时，不要修改 usys.S，而是参考其他系统调用接口来修改 usys.pl。

2.3 uservec 汇编

当执行 ecall 指令后，内核会切换到 supervisor mode。接着，内核执行的第一个指令是来自 trampoline.S 文件的 uservec 汇编函数。

2.4 usertrap

之后，代码跳转到了由 C 语言实现的 usertrap 函数中（trap.c），判断如果是来自用户的系统调用则执行 syscall 函数。

> 一般进入 usertrap 函数有三种情况：
>
> 1. syscall 系统调用，是由用户进程主动调用 ecall 指令来实现的。
> 2. exception 异常，是用户进程或内核程序的 illegal instruction 导致的，比如：除 0 错误，或引用无效的虚拟地址。
> 3. device interrupt 设备中断，比如：磁盘设备完成了读写请求操作。
>
> **2.5 syscall**
>
> syscall 函数里有一个表单，根据传入的代表系统调用的数字进行查找，并执行具体实现系统调用功能的函数。对于这个例子来说，这个函数就是 sys_sleep。
>
> **2.6 sys_sleep**
>
> 其代码可见 kernel/sysproc.c line55-74：

```
55  uint64
56  sys_sleep(void)
57  {
58    int n;
59    uint ticks0;
60
61    if(argint(0, &n) < 0)
```

```
62      return -1;
63    acquire(&tickslock);
64    ticks0 = ticks;
65    while(ticks - ticks0 < n){
66      if(myproc()->killed){
67        release(&tickslock);
68        return -1;
69      }
70      sleep(&ticks, &tickslock);
71    }
72    release(&tickslock);
73    return 0;
74  }
```

- 参数：系统调用`sleep`有一个参数，用来告知操作系统`sleep`的持续时间。但是这里怎么是`void`？

  - 这是因为`xv6`又进行了一次解耦合，通过一个额外的函数获取参数，他们以`arg`开头，具体见`kernel/syscall.c`。

- 获得进程的状态：执行系统调用的时候，参数不会告诉`xv6`自己到底是什么程序。因此，`xv6`需要额外的调用来弄清楚，现在到底是什么程序正在运行？

  - 关键的函数是`myproc()`，这个函数将返回指向当前进程的`PCB`（也就是进程控制块）的指针（`struct proc *`），里面有程序的各种信息。
    `sleep`的实现逻辑：这里，主要就是根据各个不同的系统调用，实现自身应的逻辑即可。现在简要介绍`sleep`逻辑。

> line 61：获取参数
> line 63：给时钟加锁，获取当前的时间
> line 65：比较是否到了sleep结束的时间
> line 66-69：进程结束了就退出，什么也不做
> line 70：否则继续睡眠
> line 72-73：到时间了则释放时间的锁，返回程序，sleep结束。

2.7 执行完后返回用户空间

当`sleep`完成之后，返回至`syscall`函数。在`syscall`函数中调用`usertrapret`，用于完成部分方便在`C`代码实现的返回用户空间的工作。还有一些工作只能在汇编语言完成，即`trampoline.s`中的`userret`函数。最后，这个函数执行`sret`指令，切回用户空间，执行用户空间`ecall`的下一条指令`ret`。当`ret`执行完后，返回调用函数的位置，执行用户程序的下一条语句。
以上，即是系统调用过程中大致的代码执行流程。

实验步骤

切换分支到`syscall`并同步上游仓库。

```
yhz@yhz-None:~/xv6-labs-2020$ git fetch --all
yhz@yhz-None:~/xv6-labs-2020$ git checkout syscall
分支 'syscall' 设置为跟踪 'origin/syscall'。
切换到一个新分支 'syscall'
```

1. 任务一：系统调用信息的打印

要求：添加一个名为`trace`的系统调用，追踪当前进程的某一个/几个系统调用，当该进程使用了被追踪的系统调用时，输出如下信息（通过fork产生的子进程也要保持该功能）：

$$< pid > \ syscall \ < syscall\_name > \ \rightarrow \ < first\_arg >$$

1. 作为一个系统调用，我们先要定义一个系统调用的序号。系统调用序号的宏定义在`kernel/syscall.h`文件中。我们在`kernel/syscall.h`添加宏定义，模仿已经存在的系统调用序号的宏定义，

kernel > C syscall.h

```
17  #define SYS_write  16
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_trace  22
```

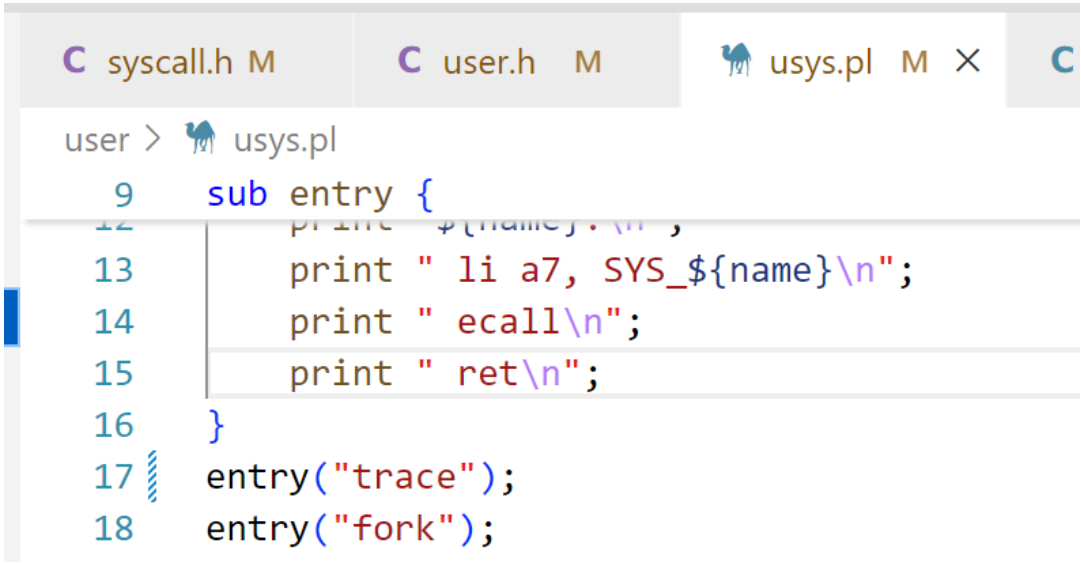
2. 查看了一下 user 目录下的文件，发现官方已经给出了用户态的`trace`函数`(user/trace.c)`，所以我们直接在`user/user.h`文件中声明用户态可以调用`trace`系统调用就好了，但有一个问题，该系统调用的参数和返回值分别是什么类型呢？接下来我们还是得看一看`trace.c`文件，可以看到`trace(atoi(argv[1])) < 0`，即`trace`函数传入的是一个数字，并和`0`进行比较，结合实验提示，我们知道传入的参数类型是`int`，并且由此可以猜测到返回值类型应该是`int`。这样就可以把`trace`这个系统调用加入到内核中声明了：

C syscall.h M | C user.h M × | usys.pl M | C

user > C user.h

```
1  struct stat;
2  struct rtcdate;
3
4  // system calls
5  int trace(int);
6  int fork(void);
7  int exit(int) __attribute__((noreturn));
```

3. 接下来我们查看`user/usys.pl`文件，这里`perl`语言会自动生成汇编语言`usys.S`，是用户态系统调用接口。所以在`user/usys.pl`文件加入下面的语句：



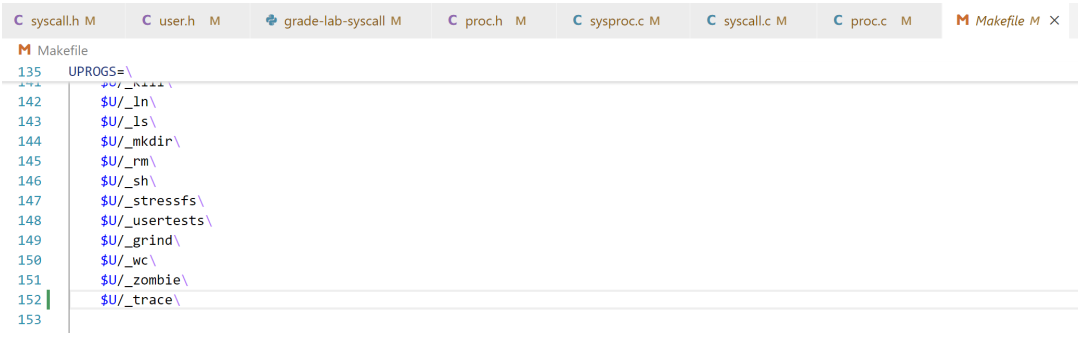
4. 我们把新增的`trace`系统调用添加到函数指针数组`*syscalls[]`上：

```
109   static uint64 (*syscalls[])(void) = {
110   [SYS_fork]    sys_fork,
111   [SYS_exit]    sys_exit,
112   [SYS_wait]    sys_wait,
113   [SYS_pipe]    sys_pipe,
114   [SYS_read]    sys_read,
115   [SYS_kill]    sys_kill,
116   [SYS_exec]    sys_exec,
117   [SYS_fstat]   sys_fstat,
118   [SYS_chdir]   sys_chdir,
119   [SYS_dup]     sys_dup,
120   [SYS_getpid]  sys_getpid,
121   [SYS_sbrk]    sys_sbrk,
122   [SYS_sleep]   sys_sleep,
123   [SYS_uptime]  sys_uptime,
124   [SYS_open]    sys_open,
125   [SYS_write]   sys_write,
126   [SYS_mknod]   sys_mknod,
127   [SYS_unlink]  sys_unlink,
128   [SYS_link]    sys_link,
129   [SYS_mkdir]   sys_mkdir,
130   [SYS_close]   sys_close,
131   [SYS_trace]   sys_trace,
132   };
```

5. 然后我们就可以在`kernel/sysproc.c`给出`sys_trace`函数的具体实现了：我们可以将传入的参数记录到proc->mask中，后续的监测功能留在syscall()函数中完成（因为我们不可能一直执行sys_trace函数，更好的办法应该是在产生系统调用时再触发追踪功能）

6. `proc`结构体(见`kernel/proc.h`)里的`name`是整个线程的名字，不是函数调用的函数名称，所以我们不能用`p->name`，而要自己定义一个数组syscall_names[]，我这里直接在`kernel/syscall.c`中定义了，这里注意系统调用名字一定要按顺序，第一个为空，当然你也可以去掉第一个空字符串，但要记得取值的时候索引要减一，因为这里的系统调用号是从`1`开始的。

7. 接着就可以在syscall()函数中添加监测功能，并输出对应信息
8. 不要忘记在`kernel/proc.c`中`fork`函数调用时，添加子进程复制父进程的`mask`的代码
9. 最后在`Makefile`的`UPROGS`中添加:`$U/_trace\`



> 在实现过程中如果出现报错

```
user/sh.c: In function ‘runcmd’:
user/sh.c:58:1: error: infinite recursion detected [-Werror=infinite-recursion]
   58 | runcmd(struct cmd *cmd)
      | ^~~~~~
user/sh.c:89:5: note: recursive call
   89 |     runcmd(rcmd->cmd);
      |     ^~~~~~~~~~~~~~~~~
user/sh.c:109:7: note: recursive call
  109 |       runcmd(pcmd->left);
      |       ^~~~~~~~~~~~~~~~~~
user/sh.c:116:7: note: recursive call
  116 |       runcmd(pcmd->right);
      |       ^~~~~~~~~~~~~~~~~~~
user/sh.c:95:7: note: recursive call
   95 |       runcmd(lcmd->left);
      |       ^~~~~~~~~~~~~~~~~~
user/sh.c:97:5: note: recursive call
   97 |     runcmd(lcmd->right);
      |     ^~~~~~~~~~~~~~~~~~~
user/sh.c:127:7: note: recursive call
  127 |       runcmd(bcmd->cmd);
      |       ^~~~~~~~~~~~~~~~~
cc1: all warnings being treated as errors
make: *** [<内置>：user/sh.o] 错误 1
```

可以添加下面代码来解决：

syscall.h M | syscall.c M | grade-lab-syscall M | sh.c M | proc.c M

user > sh.c

```
54  struct cmd *parsecmd(char*);
55
56  // Execute cmd.  Never returns.
57  __attribute__((noreturn))
58  void
59  runcmd(struct cmd *cmd)
60  {
61    int p[2];
62    struct backcmd *bcmd;
63    struct execcmd *ecmd;
64    struct listcmd *lcmd;
```

2. 任务二：添加系统调用`sysinfo`

在本实验中，您将添加一个系统调用`sysinfo`

要求：收集有关正在运行的系统信息。系统调用接受一个参数：一个指向`struct sysinfo`的指针(参见`kernel/sysinfo.h`)，系统调用应将收集好的信息通过该指针传回。sysinfo系统调用程序应该填写这个结构体的两个字段：`freemem`字段应该设置为空闲内存的字节数，`nproc`字段应该设置为状态不是`UNUSED`的进程数。

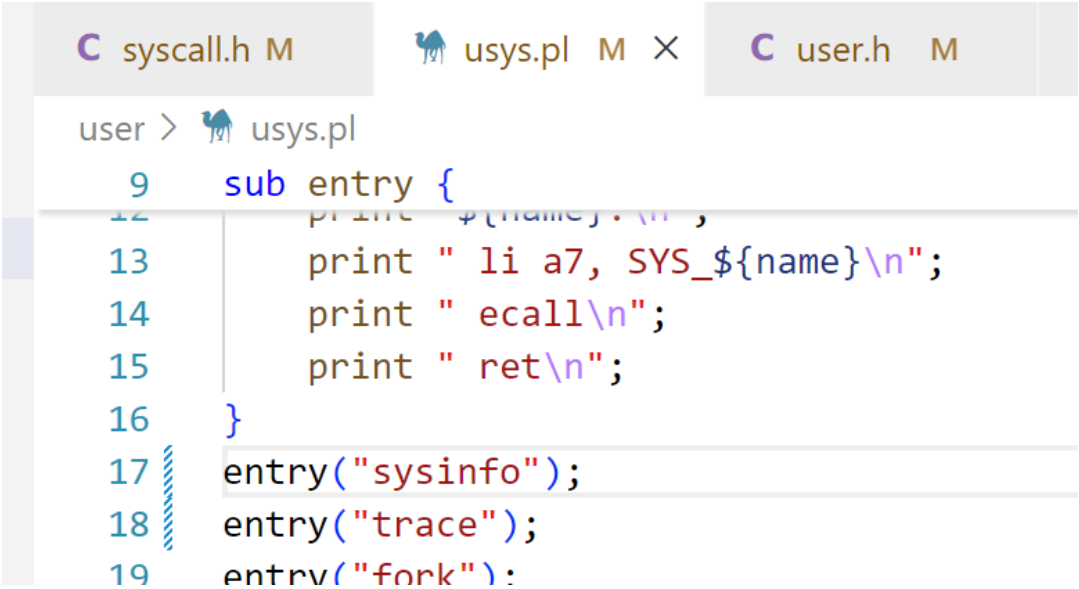
1. 首先定义一个系统调用的序号。系统调用序号的宏定义在`kernel/syscall.h`文件中。我们在`kernel/syscall.h`添加宏定义`SYS_sysinfo`如下：

syscall.h M | usys.pl M | user.h M | grade-la

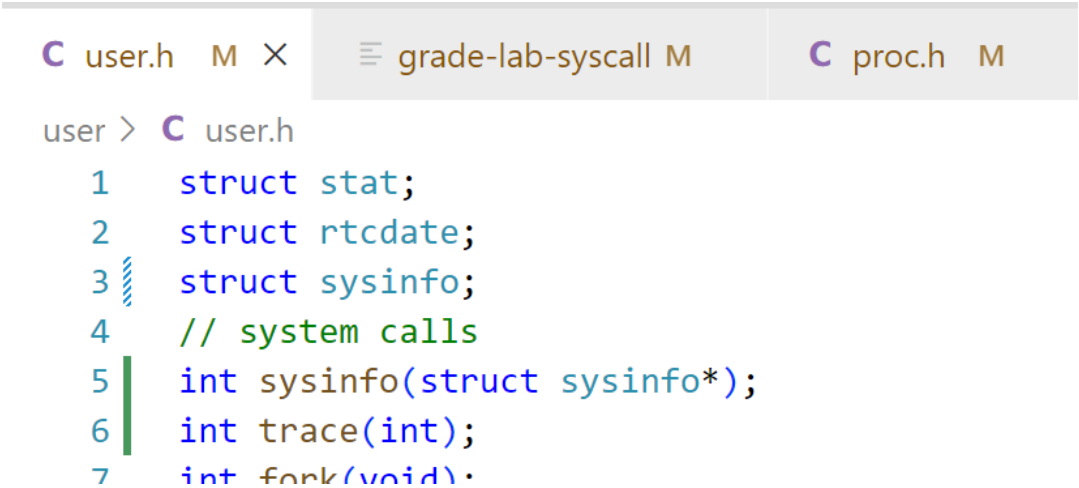
kernel > syscall.h

```
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_trace  22
24  #define SYS_sysinfo 23
```

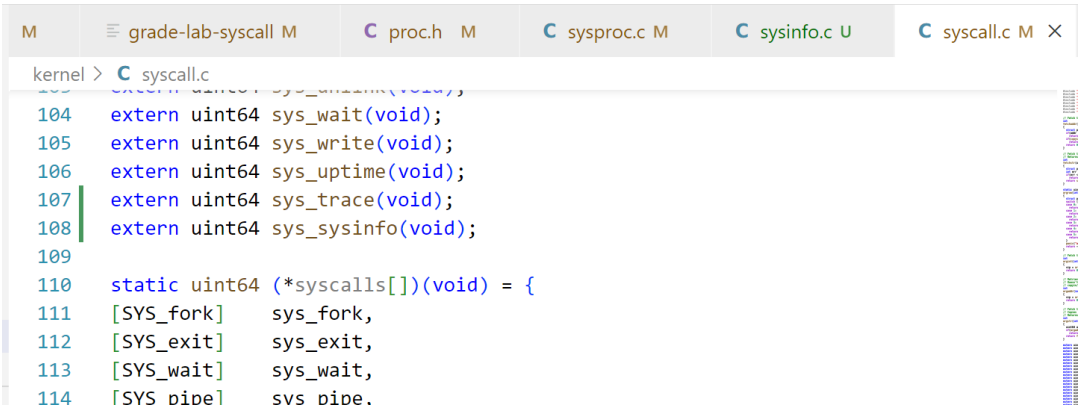
2. 在 `user/usys.pl` 文件加入下面的语句：



3. 要在 `user/user.h` 中声明 `sysinfo()` 的原型，您需要预先声明 `struct sysinfo`：



4. 在 `kernel/syscall.c` 中新增 `sys_sysinfo` 函数的定义：



5. 在 `kernel/syscall.c` 中函数指针数组新增 `sys_trace`，并且在 `kernel/syscall.c` 中的 `syscall_names` 数组新增上sysinfo的名称

6. 在 `kernel/proc.c` 中新增函数 `nproc`，通过该函数以获取可用进程数目

7. 在 `kernel/kalloc.c` 中新增函数 `free_mem`，以获取空闲内存数量

8. 在 `kernel/defs.h` 中添加上述两个新增函数的声明：

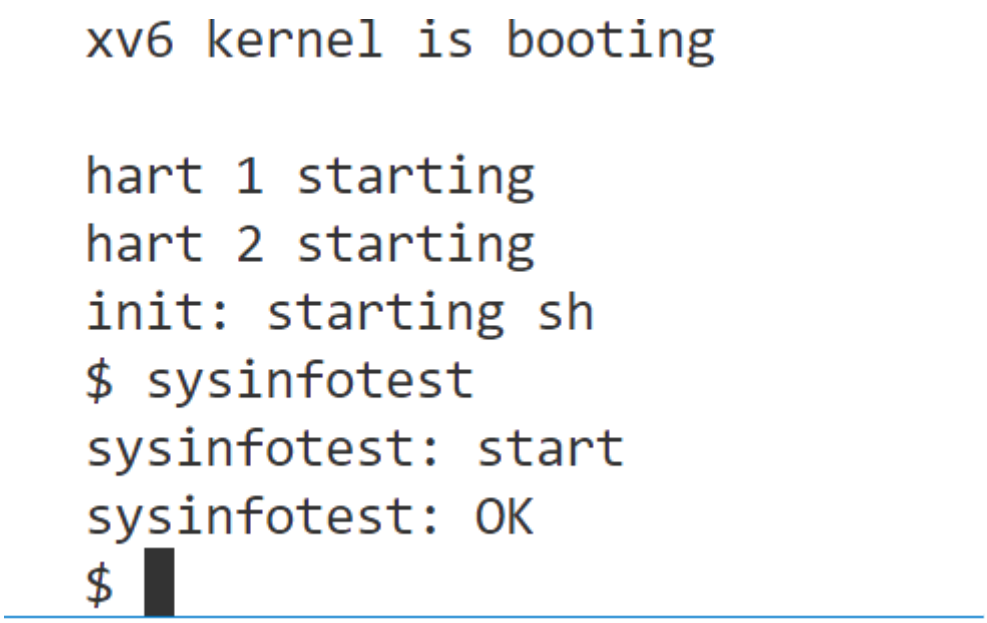
```
// kalloc.c
...
uint64          free_mem(void);

// proc.c
...
uint64          nproc(void);
```

9. 在 `kernel/sysproc.c` 文件中添加 `sysinfo.h` 头文件以及sys_sysinfo`函数的定义：要求将信息通过系统调用参数传入的指针传递回user，而不是直接在sys_sysinfo函数中直接输出

kernel > C sysproc.c

```
1   #include "types.h"
2   #include "riscv.h"
3   #include "defs.h"
4   #include "date.h"
5   #include "param.h"
6   #include "memlayout.h"
7   #include "spinlock.h"
8   #include "proc.h"
9   #include "sysinfo.h"
10
11  uint64
```

10. 最后在 `user` 目录下添加一个 `sysinfo.c` 用户程序输出对应的信息：（此为自测方式，不强制要求）
11. 将 `$U/_sysinfotest` 添加到 `Makefile` 的 `UPROGS` 中。（此为测试程序，必要）




```
yhz@yhz-None:~/xv6-labs-2020$ ./grade-lab-syscall sysinfo
make: “kernel/kernel”已是最新。
== Test sysinfotest == sysinfotest: OK (4.5s)
yhz@yhz-None:~/xv6-labs-2020$
```

问题探索

(1) 阅读`kernel/syscall.c`，试解释函数`syscall()`如何根据系统调用号调用对应的系统调用处理函数（例如`sys_fork`）？`syscall()`将具体系统调用的返回值存放在哪里？

(2) 阅读`kernel/syscall.c`，哪些函数用于传递系统调用参数？试解释`argraw()`函数的含义。

(3) 阅读`kernel/proc.c`和`proc.h`，进程控制块存储在哪个数组中？进程控制块中哪个成员指示了进程的状态？一共有哪些状态？

(4) 阅读`kernel/kalloc.c`，哪个结构体中的哪个成员可以指示空闲的内存页？

(5) 阅读`kernel/vm.c`，试解释`copyout()`函数各个参数的含义。