

同济大学操作系统课程设计——Lab2: System calls

2151422武芷朵 Tongji University, 2024 Summer

同济大学操作系统课程设计——Lab2: System calls

2151422武芷朵 Tongji University, 2024 Summer

综述

1. System call tracing (moderate)

- 1.1 实验目的
- 1.2 实验步骤
- 1.3 实验中遇到的问题和解决办法
- 1.4 实验心得

2.Sysinfo (moderate)

- 2.1 实验目的
- 2.2 实验步骤
- 2.3 实验中遇到的问题和解决办法
- 2.4 实验心得

3 实验检验得分

Lab2: System calls: 系统调用实验

项目地址: [wzd232604/TJOS-xv6-2024-labs](https://github.com/wzd232604/TJOS-xv6-2024-labs): 同济大学操作系统课程设计-xv6实验(github.com)

综述

Lab2中向 XV6 添加一些新的系统调用, 并展示一些 xv6 内核的内部结构。

- 系统调用的用户空间代码在 `user/user.h` 和 `user/usys.pl` 中。
- 内核空间代码在 `kernel/syscall.h` 和 `kernel/syscall.c` 中。
- 与进程相关的代码在 `kernel/proc.h` 和 `kernel/proc.c` 中。

切换到 `syscall` 分支:

```
git fetch
```

```
git checkout syscall
```

```
make clean
```

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout syscall
Already on 'syscall'
Your branch is up to date with 'origin/syscall'.
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
    user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln u
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie \
ph barrier
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

1. System call tracing (moderate)

1.1 实验目的

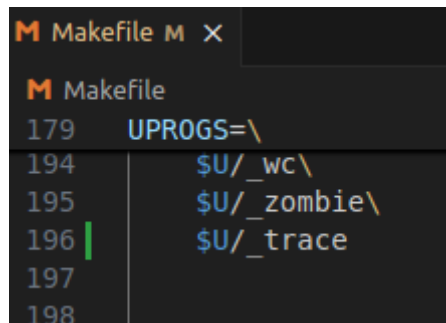
追踪系统调用：了解系统调用跟踪功能的实现。创建一个新的跟踪系统调用（`trace system call`），用于控制跟踪操作，追踪用户程序使用系统调用的情况。

该系统调用应接受一个整数参数 "mask"，其中的位数表示要跟踪的系统调用。例如，要跟踪 fork 系统调用，程序应调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是来自 `kernel/syscall.h` 的系统调用号。该进程调用过 `trace(1 << SYS_fork)` 后，如果该进程后续调用了 fork 系统调用，调用 fork 时内核则会打印形如 `<pid>: syscall fork -> <ret_value>` 的信息。

需要修改 xv6 内核以便在每个系统调用即将返回时打印一行输出，输出行应包含进程ID、系统调用名称和返回值，不需要打印系统调用的参数。`trace` 系统调用应启用调用它的进程以及其后续 `fork` 出的所有子进程的跟踪，但不应影响其他进程。

1.2 实验步骤

1. 在 Makefile 的 `UPROGS` 环境变量中添加 `$U/_trace`

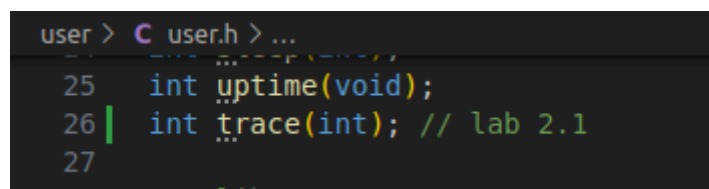


```
M Makefile M X
M Makefile
179     UPROGS=\
194         $U/_wc\
195         $U/_zombie\
196         $U/_trace
197
198
```

2. 添加系统调用的声明和存根：

在 `user/user.h` 中添加 `trace` 系统调用原型：(作为 `trace` 系统调用在用户态的入口)

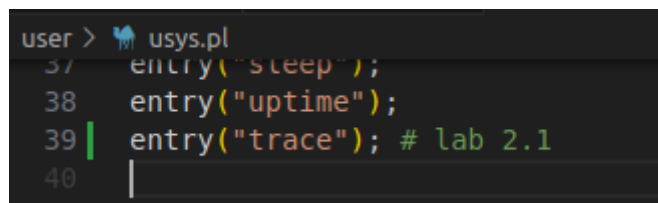
```
int trace(int); // lab 2.1
```



```
user > C user.h > ...
25     int uptime(void);
26     int trace(int); // lab 2.1
27
28
```

在 `user/usys.pl` 脚本中添加 `trace` 对应的 entry:(生成调用 `trace(int)` 入口时在用户态执行的汇编代码)

```
entry("trace"); # lab 2.1
```



```
user > usys.pl
37     entry("sleep");
38     entry("uptime");
39     entry("trace"); # lab 2.1
40
```

在 `kernel/syscall.h` 中添加 `trace` 的系统调用号(为 `trace` 分配一个系统调用的编号)

```
#define SYS_trace 22 // lab 2.1
```

```
kernel > C syscall.h > SYS_trace
22 #define SYS_close 21
23 | #define SYS_trace 22 // lab 2.1
24
```

3. 编写 trace 的系统调用函数

结构体 `struct proc` 的定义在 `kernel/proc.h` 中，该结构体记录着进程的转态。需要为 trace 系统调用添加一个变量 `tracemask` 来记录其参数。因为 trace 只会在本进程发挥作用，所以 `tracemask` 应该作为进程的私有变量。

```
kernel > C proc.h > ...
86 struct proc {
105 struct file *ofile[NOFILE]; // Open files
106 struct inode *cwd; // Current directory
107 char name[16]; // Process name (debugging)
108 | int tracemask; //trace
109 };
```

```
kernel > C sysproc.c > sys_trace(void)
98
99 //将trace(int) 的参数保存到struct proc
100 uint64 sys_trace(void) {
101     int mask;
102     // 获取整数类型的系统调用参数
103     if (argint(0, &mask) < 0) {
104         return -1;
105     }
106     // 存入proc 结构体的 mask 变量中
107     myproc()->tracemask = mask;
108     return 0;
109 }
```

4. 修改 fork() 函数，将父进程的跟踪掩码复制到子进程。

```
kernel > C proc.c > fork(void)
273 fork(void)
289 }
290 np->sz = p->sz;
291
292 // 用于拷贝mask
293 np->tracemask=p->tracemask;
294
295 // copy saved user registers.
```

5. 在 `syscall.c` 中添加函数引用

```
kernel > C syscall.c > syscalls
105 extern uint64 sys_write(void);
106 extern uint64 sys_uptime(void);
107 | extern uint64 sys_trace(void);
108
```

在 `syscall.c` 中添加一个 `syscall_names` 数组，使用系统调用名称来索引。

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

```

kernel > C syscall.c > [0] syscalls
135 };
136
137 static char *syscall_names[] = {
138     "", "fork", "exit", "wait", "pipe",
139     "read", "kill", "exec", "fstat", "chdir",
140     "dup", "getpid", "sbrk", "sleep", "uptime",
141     "open", "write", "mknod", "unlink", "link",
142     "mkdir", "close", "trace"};
143
144
145 void
146 syscall(void)
147 {
148     int num;
149     struct proc *p = myproc();
150
151     num = p->trapframe->a7; //从寄存器 a7 中读取系统调用号
152     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
153         p->trapframe->a0 = syscalls[num](); //通过调用 syscalls[num](); 函数, 把返回值保存在了 a0 寄存器中
154         int mask = p->tracemask;
155         //trace syscall masked
156         if ((1 << num) & mask) {
157             printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
158         }
159     } else {
160         printf("%d %s: unknown sys call %d\n",
161             p->pid, p->name, num);
162         p->trapframe->a0 = -1;
163     }
164 }

```

- 保存后在终端里执行make qemu编译运行xv6;
- 在命令行中输入 `trace 32 grep hello README`, 该语句就是一个进程, 包含了多个系统函数, 但我们只跟踪 `read()`: (其中 32 是 `1 << SYS_read` 即 `1 << 5`)

```

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$

```

- 在命令行中输入 `trace 2147483647 grep hello README`, 该语句就是一个进程, 该进程包含了多个系统函数, 同时, 全部调用到的系统函数我们都跟踪:

```

$ 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
$

```

- 在命令行中输入 `grep hello README`, 程序没有被跟踪, 因此没有打印跟踪输出“:

```

$ grep hello README
$

```

- 在命令行中输入 `trace 2 usertests forkforkfork`, 跟踪了 usertests 中 `forkforkfork` 测试的所有后代的 fork 系统调用:

```
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
10: syscall fork -> 12
9: syscall fork -> 13
10: syscall fork -> 14
12: syscall fork -> 15
13: syscall fork -> 16
9: syscall fork -> 17
11: syscall fork -> 18
10: syscall fork -> 19
9: syscall fork -> 20
11: syscall fork -> 21
10: syscall fork -> 22
12: syscall fork -> 23
11: syscall fork -> 24
9: syscall fork -> 25
17: syscall fork -> 26
```

```
9: syscall fork -> 63
12: syscall fork -> 64
24: syscall fork -> 65
10: syscall fork -> 66
9: syscall fork -> 67
26: syscall fork -> 68
10: syscall fork -> -1
27: syscall fork -> -1
9: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
$
```

11. 在终端里运行 `./grade-lab-syscall trace` 可进行评分：

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (3.5s)
== Test trace all grep == trace all grep: OK (3.6s)
== Test trace nothing == trace nothing: OK (3.0s)
== Test trace children == trace children: OK (57.4s)
(Old xv6.out.trace_children failure log removed)
```

1.3 实验中遇到的问题和解决办法

1. 问题：系统调用跟踪的输出与预期不符。

- 解决办法：检查代码中的跟踪输出部分，确保正确获取并打印所需的信息。确保在每个系统调用返回之前进行跟踪输出，检查系统调用的返回值。

2. 问题：用户态和内核态之间的数据传递失败。

- 解决办法：用户态和内核态之间的数据传递和状态切换的方面：

1. 系统调用参数传递：在实验中，用户程序通过调用 `sys_trace` 系统调用来传递一个整数参数 `mask` 给内核，用于指定要追踪的系统调用。用户程序通过 `argint` 函数将参数从用户空间传递到内核空间。

2. 内核态与用户态切换：当用户程序调用 `sys_trace` 系统调用时，会触发用户态切换到内核态。这个切换由系统调用机制完成。
3. 数据传递和复制：用户程序传递给内核的 `tracemask` 参数需要在内核中进行数据复制。当用户程序传递参数给内核时，内核需要正确地从用户空间复制数据到内核空间，以确保内核能够访问到正确的数据。
4. 数据验证和权限检查：在实验中，内核需要验证用户传递的 `tracemask` 参数是否合法，并确保用户程序有权限调用 `sys_trace` 系统调用。内核可以通过对 `tracemask` 进行合法性检查来防止恶意传递错误的参数。

3. 问题：Timeout! trace children 超时。

- 解决办法：通过查阅资料发现可能是电脑性能不够强导致出现：

```
== Test trace children ==
$ make qemu-gdb
Timeout! trace children: FAIL (30.2s)
...
9: syscall fork -> 56
6: syscall fork -> -1
7: syscall fork -> -1
8: syscall fork -> -1
qemu-system-riscv64: terminating on signal 15 from pid 5581 (make)
MISSING '^ALL TESTS PASSED'
QEMU output saved to xv6.out.trace_children
```

最后选择在 `py` 文件 `gradelib.py` 中改变超时判断时间 70s。

```
gradelib.py
415 class Runner():
419     def run_qemu(self, *monitors, **kw):
425         """
426         argument gives the make target to run. The make_args argument
427         should be a list of additional arguments to pass to make. The
428         timeout argument bounds how long to run before returning."""
429         def run_qemu_kw(target_base="qemu", make_args=[], timeout=70):
430             return target_base, make_args, timeout
430         target_base, make_args, timeout = run_qemu_kw(**kw)
```

1.4 实验心得

- 通过本实验，我了解了如何在 xv6 内核中添加新的系统调用，如何修改进程控制块以支持跟踪掩码，并且理解了如何在内核中实现系统调用的功能。此外，我还了解了如何在用户级程序中调用新增的系统调用，并在实验中验证系统调用的正确性。
- 通过完成实验，我更深入地理解了操作系统的系统调用机制。我了解了用户空间和内核空间之间的交互方式，并学会了如何在用户程序中调用新添加的系统调用。此外，我还加深了对 xv6 内核的理解，包括系统调用的执行流程、内核数据结构和进程管理。

2.Sysinfo (moderate)

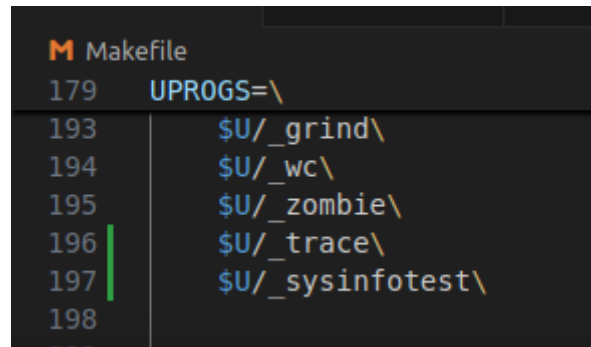
2.1 实验目的

本实验添加一个系统调用 `sysinfo`，用于收集有关正在运行的系统的信息。该系统调用接受一个参数：指向 `struct sysinfo` 结构体的指针（见 `kernel/sysinfo.h`）。内核应填充该结构体的字段：
`freemem` 字段应设置为空闲内存的字节数，`nproc` 字段应设置为状态不是 `UNUSED` 的进程数量。

提供了一个名为 `sysinfotest` 的测试程序；如果该程序打印出 `"sysinfotest: OK"`，则表示通过此任务。

2.2 实验步骤

1. 在 Makefile 的 `UPROGS` 中添加 `$U/_sysinfotest`

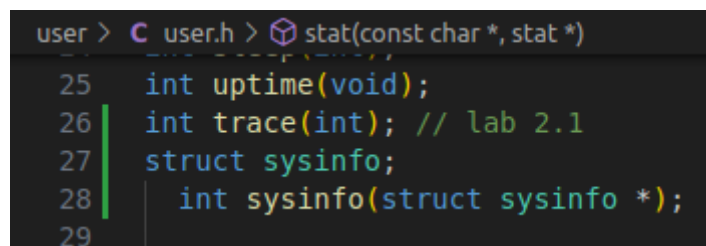


```
M Makefile
179 UPROGS=\
193 $U/_grind\
194 $U/_wc\
195 $U/_zombie\
196 $U/_trace\
197 $U/_sysinfotest\
198
```

2. 添加系统调用的声明和存根：

在 `user/user.h` 中添加 `sysinfo()` 调用原型：

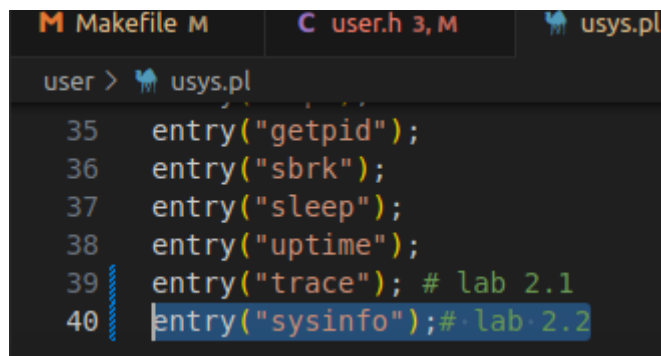
```
struct sysinfo;
int sysinfo(struct sysinfo *);
```



```
user > C user.h > stat(const char *, stat *)
25 int uptime(void);
26 int trace(int); // lab 2.1
27 struct sysinfo;
28 int sysinfo(struct sysinfo *);
29
```

在 `user/usys.pl` 脚本中添加 对应的 entry:

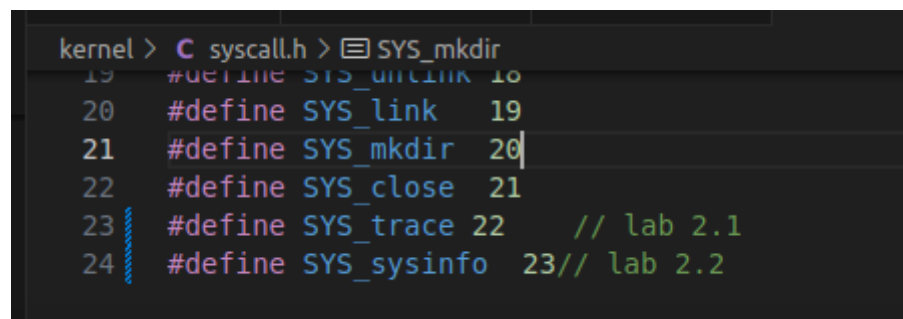
```
entry("sysinfo");# lab 2.2
```



```
M Makefile M C user.h 3, M usys.pl
user > usys.pl
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("trace"); # lab 2.1
40 entry("sysinfo");# lab 2.2
```

在 `kernel/syscall.h` 中添加 `trace` 的系统调用号(为 `trace` 分配一个系统调用的编号)

```
#define SYS_sysinfo 23// lab 2.2
```



```
kernel > C syscall.h > SYS_mkdir
19 #define SYS_unlink 10
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_trace 22 // lab 2.1
24 #define SYS_sysinfo 23// lab 2.2
```

在 `syscall.c` 中添加函数引用


```

kernel > C syscall.c > [?] syscalls
...
106 extern uint64 sys_uptime(void);
107 extern uint64 sys_trace(void);
108 extern uint64 sys_sysinfo(void);
109
110

```

3. 编写函数, 获得空闲内存大小:

在 `kernel/kalloc.c` 实现一个函数。由于 xv6 管理内存空闲空间使用的是空闲链表, 参照链表的结构后, 只需遍历链表并计算数量, 然后乘以页面大小即可。

```

kernel > C kalloc.c > ...
...
83
84 uint64 getfreemem(void)
85 {
86     uint64 freemem = 0;
87     acquire(&kmem.lock);
88     struct run *r = kmem.freelist;
89     while (r) {
90         freemem += PGSIZE;
91         r = r->next;
92     }
93     release(&kmem.lock);
94     return freemem;
95 }
96

```

4. 编写函数, 统计空闲进程控制块的数量:

在 `kernel/proc.c` 实现一个函数。进程控制块是用静态数组管理的, 故而只需要用一个循环遍历该数组即可。

```

kernel > C proc.c > ...
660
661 // get the number of processes whose state is not UNUSED
662 uint64 getnproc(void)
663 {
664     struct proc *p;
665     uint64 nproc = 0;
666     for (p=proc; p<&proc[NPROC]; p++) {
667         if (p->state != UNUSED)
668             ++nproc;
669     }
670     return nproc;
671 }
672

```

5. 在 `kernel/sysproc.c` 中实现的 `sys_sysinfo(void)`

```

kernel > C sysproc.c > ...
6 #include "memlayout.h"
7 #include "spinlock.h"
8 #include "proc.h"
9 #include "sysinfo.h"

```



```

kernel > C sysproc.c > sys_sysinfo(void)
101  uint64 sys_trace(void) {
110  }
111
112  uint64
113  sys_sysinfo(void) {
114      uint64 addr;
115      if (argaddr(0, &addr) < 0)
116          return -1;
117      struct sysinfo info;
118      info.freemem = getfreemem(); // 获取系统空闲内存(在kernel/kalloc.c中实现)
119      info.nproc = getnproc();    // 获取系统当前的进程数量(在kernel/proc.c中实现)
120      struct proc *p = myproc();
121      if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
122          return -1;
123      return 0;
124  }

```

6. 在 `kernel/defs.h` 中添加函数原型

```

kernel > C defs.h > getnproc(void)
188
189  uint64      getfreemem(void);
190  uint64      getnproc(void);

```

7. 保存后在终端里执行 `make qemu` 编译运行 `xv6`;

8. 在命令行中输入 `sysinfotest`:

```

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
C: QEMU: Terminated

```

9. 在终端里运行 `./grade-lab-syscall sysinfo` 可进行评分:

```

wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest ==
sysinfotest: OK (15.0s)
wzd@ubuntu:~/Desktop/xv6-labs-2021$

```

2.3 实验中遇到的问题和解决办法

1. 问题: 如何根据现有的源码提取出可供我们利用的参数。

- 解决办法: 参考 `kalloc()` 和 `kfree()` 等函数, 发现内核通过 `kmem.freelist` 的一个链表维护未使用的内存, 同时链表的每个结点还对应了页表大小(PGSIZE), 所以需要乘上页表大小才获得了可用内存数。

2. 问题: `sysinfotest` 程序无法编译或输出不正确。

- 解决办法: 检查代码中的编译错误并修复。确保正确实现 `sysinfo` 系统调用, 并在该系统调用中正确填充 `struct sysinfo` 的字段。使用 `copyout()` 函数将数据正确地复制到用户空间。

2.4 实验心得

- 通过本次实验, 我学会为系统添加了一个新的系统调用 `sysinfo`, 实现了收集运行系统的信息, 如可用内存数、进程数等。

- 在完成实验的过程中, 我还了解了实现所需的几个基础数据结构, 比如kmem链表, 以及表示进程状态的字段 `UNUSED` 等。要实现这些能够反应系统运行信息功能, 首先需要了解这些信息分别由什么记录, 这样才能更有针对性地做出追踪和检测。
- xv6将空闲物理内存划分成一系列 4096 字节大小的 物理页. (每个物理页都是 4096 字节对齐的。将物理页组织成链表, 做法是在每个物理页的前 8 个字节记录下一页的起始地址。为实现这种管理方式, xv6 维护了一个数据结构 `kmem`。

`kmem.freelist` 是链表中第一个结点的起始地址 (链表为空时, `kmem.freelist == 0`);

`kmem.lock` 是一个互斥锁, 保护共享数据 `kmem.freelist`。

3 实验检验得分

1. 在实验目录下创建 `time.txt`, 填写完成实验时间数
2. 在终端中执行 `make grade`

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (15.7s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (2.7s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (2.7s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (54.3s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (13.4s)
== Test time ==
time: OK
Score: 35/35
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```