同济大学操作系统课程设计 - Xv6 - Lab2: System calls

【实验目的】

1. **理解和实现系统调用追踪机制：** 通过创建一个新的 `trace` 系统调用，学习如何在操作系统内核中添加和管理系统调用，理解进程级别的系统调用跟踪机制，掌握如何控制和记录特定系统调用的执行情况。
2. **增强对 `fork` 系统调用的理解：** 通过追踪 `fork` 系统调用，进一步理解父进程和子进程的关系，以及系统调用在进程间的行为表现。
3. **学习系统信息的获取与管理：** 通过实现 `sysinfo` 系统调用，学习如何在操作系统中获取和管理系统资源信息，理解内存管理和进程管理的基本概念。
4. **掌握内核代码修改和扩展技巧：** 通过修改 `xv6` 内核，熟悉内核源码结构，掌握在操作系统中添加功能、修改内核行为的基本方法。
5. **强化调试和测试技能：** 通过编写和运行测试程序 `sysinfotest`，验证系统调用的正确性，提升调试、验证内核修改的能力。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 syscall 分支：

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

1 System call tracing

在本作业中添加一个系统调用跟踪功能，该功能 可能会在调试后续实验时有所帮助。首先创建一个新的跟踪系统调用，它将控制跟踪。它应该取一个参数，一个整数 “mask”，其位指定哪个 对 trace 的系统调用。例如，要跟踪 fork 系统调用， 程序调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是一个 来自 `kernel/syscall.h` 的 syscall 编号。必须修改 xv6 内核在每次系统调用即将 return，如果在掩码中设置了系统调用的号码。 该行应包含 进程 ID、系统调用的名称和 返回值；无需打印 System Call 参数。 `trace` 系统调用应启用跟踪 对于调用它的进程以及它随后分叉的任何子进程， 但不应影响其他进程。

1. 在 Makefile 中向 UPROGS 添加 `$U/_trace`
2. 修改代码，添加trace系统调用
    - 根据提示，修改`user/user.h`， `user/usys.pl`与`kernel/syscall.h`，分别添加语句，使其功能完整

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

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

//kernel/syscall.h
#define SYS_close  21
#define SYS_trace  22
```

- 在 `kernel/sysproc.c` 中添加一个 `sys_trace()` 函数，这个函数通过将参数保存到 `proc` 结构体里的一个新变量中来实现新的系统调用。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:

```
struct proc {
  char name[16];               // Process name (debugging)
  int tracemask;              // trace system call argument
}
```

- 然后是 `sys_trace` 函数的实现：

```
uint64 sys_trace(void)
{
  int maskValue;
  if (argint(0, &maskValue) < 0) {
    return -1;
  }
  myproc()->trace_mask = maskValue;
  return 0;
}
```

- 修改 `fork()` 函数将跟踪掩码从父进程复制到子进程，即把父进程的 `trace_mask` 赋值给子进程的 `trace_mask`：

```
// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
  freeproc(np);
  release(&np->lock);
  return -1;
}
np->sz = p->sz;

  // copy mask
np->tracemask=p->tracemask;
```

- 最后在 `syscall.c` 中加上函数引用：

```
extern uint64 sys_uptime(void);
extern uint64 sys_trace(void);
```

- 再添加一个数组，使用系统调用名称来索引。

```
static char *syscall_names[] = {
  "", "fork", "exit", "wait", "pipe",
  "read", "kill", "exec", "fstat", "chdir",
  "dup", "getpid", "sbrk", "sleep", "uptime",
  "open", "write", "mknod", "unlink", "link",
  "mkdir", "close", "trace"};
```

3. 结果测试：

- 输入`trace 32 grep hello README`，输出为

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
```

- 输入`trace 2147483647 grep hello README`，输出为

```
$ trace 2147483647 grep hello README
5: syscall trace -> 0
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 966
5: syscall read -> 70
5: syscall read -> 0
5: syscall close -> 0
```

- 输入`grep hello README`，无输出，这是因为没有跟踪程序，因此无打印内容

```
$ grep hello README
```

- 输入`trace 2 usertests forkforkfork`，输出为

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

```
13: syscall fork -> 24
14: syscall fork -> 25
12: syscall fork -> 26
13: syscall fork -> 27
12: syscall fork -> 28
14: syscall fork -> 29
12: syscall fork -> 30
13: syscall fork -> 31
12: syscall fork -> 32
14: syscall fork -> 33
12: syscall fork -> 34
13: syscall fork -> 35
12: syscall fork -> 36
14: syscall fork -> 37
12: syscall fork -> 38
13: syscall fork -> 39
12: syscall fork -> 40
13: syscall fork -> 41
12: syscall fork -> 42
14: syscall fork -> 43
12: syscall fork -> 44
13: syscall fork -> 45
14: syscall fork -> 46
12: syscall fork -> 47
13: syscall fork -> 48
14: syscall fork -> 49
12: syscall fork -> 50
13: syscall fork -> 51
14: syscall fork -> 52
12: syscall fork -> 53
13: syscall fork -> 54
12: syscall fork -> 55
13: syscall fork -> 56
12: syscall fork -> 57
13: syscall fork -> 58
14: syscall fork -> 59
12: syscall fork -> 60
13: syscall fork -> 61
12: syscall fork -> 62
13: syscall fork -> 63
12: syscall fork -> 64
13: syscall fork -> 65
14: syscall fork -> 66
12: syscall fork -> 67
13: syscall fork -> 68
12: syscall fork -> 69
13: syscall fork -> 70
12: syscall fork -> 71
14: syscall fork -> -1
13: syscall fork -> -1
12: syscall fork -> -1
OK
9: syscall fork -> 72
ALL TESTS PASSED
```

2 Sysinfo

在此作业中添加一个系统调用 `sysinfo` 以收集有关正在运行的系统的信息。系统调用接受一个参数：指向 `struct sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核应填写字段：应设置 `FreeMem` 字段置为可用内存的字节数，`并且 nproc` 字段应设置为状态为`UNUSED` 的进程数。 我们提供了一个测试程序 `sysinfotest`;通过这个如果打印 “sysinfotest： OK” 则已分配。

1. 在 Makefile 中向 UPROGS 添加 `$U/_sysinfotest`
2. 在 `user/user.h` 中需要预先声明

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

3. 在 `user/usys.pl` 中添加系统调用入口：

```
entry("sysinfo");
```

4. 在 `kernel/syscall.h` 文件中，为sysinfo 分配一个系统调用编号：

```
#define SYS_sysinfo  23
```

5. 在 `syscall.c` 中添加函数引用：

   在 `syscall.c` 文件中，确保正确引用 `sys_sysinfo` 函数，确保系统调用能够被识别和调用。

6. 实现获取空闲内存大小的函数：

   在 `kernel/kalloc.c` 中实现一个函数，通过遍历空闲链表计算空闲内存的大小。通过累加链表中的空闲页面数，并乘以每个页面的大小来获得总的空闲内存。

7. 实现统计空闲进程控制块数量的函数：

   在 `kernel/proc.c` 中实现一个函数，通过遍历进程控制块数组来统计空闲的进程控制块数量。数组中的每个元素表示一个进程，未使用的元素即为空闲的进程控制块。

8. 实现 `sys_sysinfo` 系统调用：

   在 `kernel/sysproc.c` 中实现 `sys_sysinfo(void)` 函数，该函数负责填充 `sysinfo` 结构体中的字段，如空闲内存大小和活跃进程数量。

9. 在 `kernel/defs.h` 中声明函数原型：

   在 `kernel/defs.h` 文件中添加相应函数的原型声明，以确保在其他地方可以正确引用这些函数。

10. 编译并运行 xv6：

    保存所有更改后，在终端执行 `make qemu`，编译并运行 xv6 系统，输出为：

```
hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

```
hart 2 starting
hart 1 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

【分析讨论】

- 通过本次实验，我深入了解了如何在 xv6 内核中添加新的系统调用，以及如何修改进程控制块以支持跟踪掩码的功能。
- 我掌握了系统调用的实现过程，并成功在用户级程序中调用和验证了新添加的系统调用。
- 实验加深了我对操作系统系统调用机制的理解，特别是在用户空间与内核空间交互方面，以及对 xv6 内核的数据结构和进程管理的认识。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.8s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.9s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.6s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (21.4s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.5s)
== Test time ==
time: OK
Score: 35/35
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab2$
```