Lab2 System calls

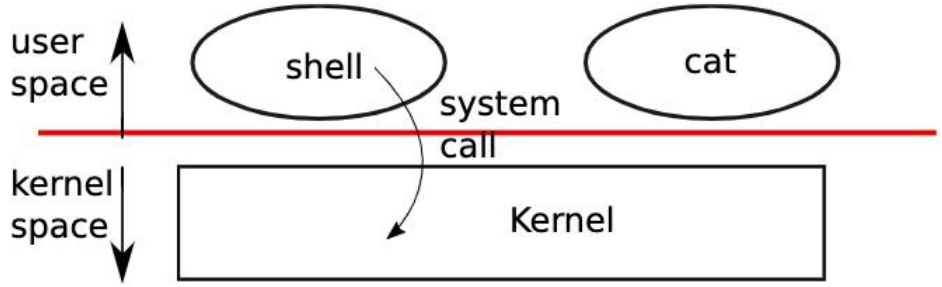
2023.10.10

目录

- 操作系统的强隔离性
- 内核组织方式
- 系统调用
- 实验提交

操作系统的强隔离性

- **应用程序与操作系统的硬边界**
  - 强隔离需要**应用程序**和**操作系统**之间有硬边界。当某个**应用程序**出错时，不会导致操作系统或其他应用程序出现错误，操作系统应当有能力清理出错的应用程序并继续其他应用的运行，
  - 为了实现强隔离，操作系统必须安排应用程序不能修改（甚至读取）操作系统的数据结构和指令，并且应用程序不能访问其他进程的内存。
- **CPU支持**
  - CPU为强隔离提供硬件支持。例如，RISC-V具有CPU执行指令的三种模式：**机器模式**、**管理模式（内核态）**和**用户模式（用户态）**。

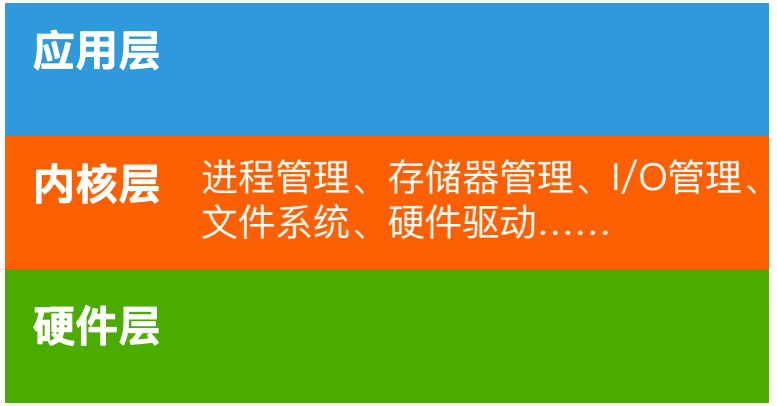


内核组织方式

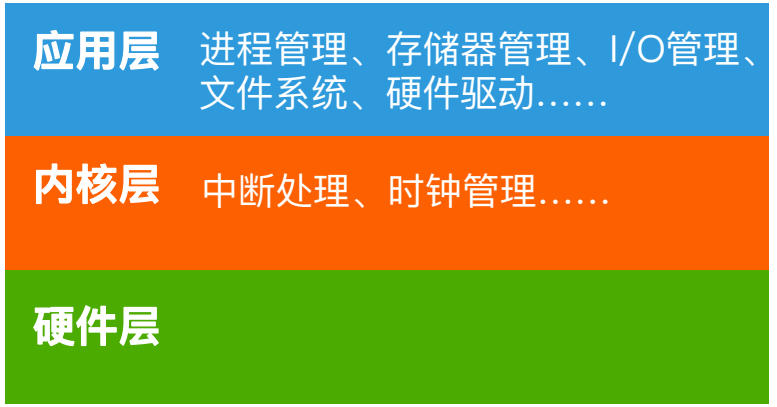
- **宏内核（xv6）**
  - 宏内核是将进程管理、存储器管理、I/O管理等管理计算机资源的功能整合在一起；
  - 若将进程管理等各项功能看作一个个模块。在宏内核中，这些模块都是集成在一起的，运行在内核进程中，只有处于**内核态**下才能运行。功能模块间可通过**方法调用**进行交互。

- **微内核**
  - 微内核结构则和宏内核结构相反，它提倡内核中的功能模块尽可能的少。
  - 内核只提供最核心的功能，比如任务调度，中断处理等等。其他实际的模块功能如进程管理等则被移出内核，变成服务进程，和**用户进程**同等级，只是它们是一种特殊的用户进程。

宏内核

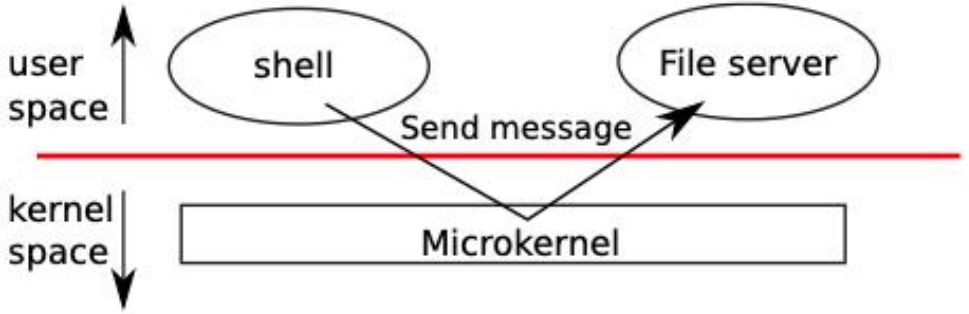


微内核



系统调用

- 应用程序与系统过程的接口
  - OS内核设置一组用于实现各种系统功能的子程序/过程，并将它们提供给应用程序调用。
  - 由于这些程序/过程是 OS 系统本身程序模块中的一部分，为了保护操作系统程序不被用户程序破坏，一般都不允许用户程序访问操作系统的程序和数据，所以也不允许应用程序采用一般的过程调用方式来直接调用这些过程，而是向应用程序提供了一系列的系统调用命令，让应用程序通过**系统调用**去调用所需的系统过程。
  - 宏内核通过函数调用完成模块间合作
  - 微内核通过进程间通信调用系统过程



实验准备：添加新系统调用

1. 在user/user.h中添加procnum系统调用的prototype
2. 在user/usys.pl中添加系统调用的存根
3. 在kernel/syscall.h为procnum添加系统调用编号
4. 在kernel/sysproc.c中添加主体函数
5. 修改kernel/syscall.c

Open ~/Desktop/xv6-labs-2022/user — user.h — Save

```
1 struct stat;
2
3 // system calls
4 int fork(void);
5 int exit(int) __attribute__((noreturn));
6 int wait(int*);
7 int pipe(int*);
8 int write(int, const void*, int);
9 int read(int, void*, int);
10 int close(int);
11 int kill(int);
12 int exec(const char*, char**);
13 int open(const char*, int);
14 int mknod(const char*, short, short);
15 int unlink(const char*);
16 int fstat(int fd, struct stat*);
17 int link(const char*, const char*);
18 int mkdir(const char*);
19 int chdir(const char*);
20 int dup(int);
21 int getpid(void);
22 char* sbrk(int);
23 int sleep(int);
24 int uptime(void);
25 int new_syscall(void);
```

1

Open ~/Desktop/xv6-labs-2022/user — usys.pl

```
17
18 entry("fork");
19 entry("exit");
20 entry("wait");
21 entry("pipe");
22 entry("read");
23 entry("write");
24 entry("close");
25 entry("kill");
26 entry("exec");
27 entry("open");
28 entry("mknod");
29 entry("unlink");
30 entry("fstat");
31 entry("link");
32 entry("mkdir");
33 entry("chdir");
34 entry("dup");
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("new_syscall")
```

2

Open ~/Desktop/xv6-labs-2022/ke... — syscall.h

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open   15
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_new_syscall  22
```

3

Open ~/Desktop/xv6-labs-2022/kernel — *sysproc.c

```
80 // return how many clock tick interr
81 // since start.
82 uint64
83 sys_uptime(void)
84 {
85   uint xticks;
86
87   acquire(&tickslock);
88   xticks = ticks;
89   release(&tickslock);
90   return xticks;
91 }
92
93 uint64
94 sys_new_syscall(void)
95 {
96   // Your implementation here.
97 }
98
```

4

Open ~/Desktop/xv6-labs-2022/kernel — syscall.c

```
98 extern uint64 sys_write(void);
99 extern uint64 sys_mknod(void);
100 extern uint64 sys_unlink(void);
101 extern uint64 sys_link(void);
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_new_syscall(void);
```

Open ~/Desktop/xv6-labs-2022/kernel — syscall.c

```
125 [SYS_unlink]   sys_unlink,
126 [SYS_link]     sys_link,
127 [SYS_mkdir]    sys_mkdir,
128 [SYS_close]    sys_close,
129 [SYS_new_syscall]   sys_new_syscall,
130 };
```

5

实验1： Process counting

- 系统调用功能 procnum：**统计系统总进程数**
- 实验步骤

1.添加用户文件（与lab0, lab1相同）

1. 在user文件夹下添加 procnum.c
2. 修改Makefile文件

2.修改系统调用文件（参考实验准备1-3）

- make qemu（failed，SYS_procnum未实现）

3. 添加procnum调用（参考实验准备4-5，在sysproc.c完成功能）

- make qemu（success）

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ procnum
3 procnum: unknown sys call 24
procnum failed!
$
```

2

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ procnum
Number of process: 3
$
```

3

```c
#include "kernel/types.h"
#include "kernel/riscv.h"
#include "kernel/sysinfo.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if (argc >= 2) {
    fprintf(2, "procnum: Too many arguments\n");
    exit(1);
  }

  int num = -1;

  if (procnum(&num) < 0) {
    fprintf(2, "procnum failed!\n");
    exit(1);
  }

  printf("Number of process: %d\n", num);
  exit(0);
}

```

- **提示：统计当前运行进程时，只需统计当前状态为unused的进程。（请阅读kernel/proc.c中代码，进程状态维护在 *struct proc proc[NPROC]* 数组中）**

实验2：Free Memory Counting

- 系统调用功能 **freemem：统计当前空闲内存块的总字节数**
- 实验步骤
  1. 添加用户文件（与lab0, lab1相同）
     1. 在user文件夹下添加 freemem.c
     2. 修改Makefile文件
  2. 修改系统调用文件（同实验1）
  3. 添加freemem调用（同实验1）

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ procnum
Number of process: 3
$ freemem
Number of bytes of free memory: 133386240
$
```

- **提示：统计当前空闲内存，请阅读kernel/kalloc.c**

```c
#include "kernel/types.h"
#include "kernel/riscv.h"
#include "kernel/sysinfo.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if (argc >= 2) {
    fprintf(2, "freemem: Too many arguments\n");
    exit(1);
  }

  int num = -1;

  if (freemem(&num) < 0) {
    fprintf(2, "freemem failed!\n");
    exit(1);
  }

  printf("Number of bytes of free memory: %d\n", num);
  exit(0);
}
```

实验3：System call tracing

- 系统调用功能 trace：跟踪特定的系统调用
  - 参数： mask（整数），其二进制位指定要跟踪哪些系统调用
    - 跟踪fork调用：trace(1 << SYS_fork)，其中 SYS_fork 是来自 kernel/syscall.h 的系统调用号
  - 要求：
    1. 在每个跟踪的系统调用即将返回时打印一行信息，需包含进程ID、系统调用名称、返回值，无需打印系统调用参数；
    2. 对于所跟踪的系统调用，需要同时启用 调用它的进程 及 其后派生的任何子进程 的跟踪，且不影响之外的其它进程

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

pid　　系统调用名称　返回值

- **具体要求参考：**
  https://pdos.csail.mit.edu/6.S081/2022/labs/syscall.html

实验4：流程概述

1.请概述用户从发出系统调用指令到得到返回结果的执行的流程。

2.搜索资料，概述malloc的底层实现原理。

实验提交

- 提交到邮箱**fduos2023lab25@163.com**：
  - 命名为：**学号-姓名-授课教师-lab2.pdf**，报告内容包含：
    1. 概述题标清练习题编号；
    2. 实验题标清练习题编号，说明实验思路，实验过程（必须包括但不限于实验代码、系统调用流程等）实验效果截图；
    3. 实验过程中遇到的问题及解决方案；
- 截止日期：**2023年10月20日23时**
- 注意：请各位同学独立完成实验，参考代码需注明

参考资料

- 实验所需代码（procnum.c、freemem.c）
  - https://docs.qq.com/doc/DR2doWVFjV0NpTkxX
- xv6 Lab: System calls
  - https://pdos.csail.mit.edu/6.S081/2022/labs/syscall.html
- xv6 book: Chapter2, Sections 4.3 and 4.4 of Chapter 4
  - https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf