哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2022 年秋季

课程名称：操作系统

实验名称：系统调用

实验性质：课内实验

实验时间：9 月 28 日 地点：T2507

学生班级：5 班

学生学号：200110513

学生姓名：宗晴

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2022 年 9 月

一、 回答问题

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？

通过脚本文件 user/usys.pl 中的`print " li a7, SYS ${name}\n";`可知，系统调用的编号存储于 a7 寄存器中。在 kernel/syscall.c 的 syscall()函数中，首先利用 myproc()函数得到指向当前进程的 PCB 的指针 p。然后通过 PCB 中的 trapframe 指针，可以利用 p->trapframe->a7，得到 a7 寄存器中存储的系统调用编号，如下图：

```
num = p->trapframe->a7;
```

该系统调用编号与 kernel/syscall.c 程序中的 syscalls 数组的下标所对应，即每个下标的对应位置存储了指向相应系统调用编号所对应的系统调用处理函数（例如 sys_fork）的指针。这些系统调用处理函数在上方由 extern 声明，同时在 kernel/sysproc.c 中被具体实现。

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num])
```

如上图，判断系统调用编号 num 是否在合法范围内，同时判断 syscalls[num]，即该编号所对应的系统调用处理函数是否存在。若存在，即可通过该函数指针，调用对应的系统调用处理函数，如下图：

```
return_value = syscalls[num]();
```

syscall()将具体的系统调用的返回值存储在 a0 寄存器中，这样在用户态就可以根据约定，从 a0 寄存器中获取返回值。与上述获取 a7 寄存器中存储的内容类似的方式，可将返回值 return_value 存储到 a0 寄存器中：

```
p->trapframe->a0 = return_value;
```

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。

```
int
argint(int n, int *ip)
```

```
int
argaddr(int n, uint64 *ip)
```

```
int
argstr(int n, char *buf, int max)
```

如上图，在 kernel/syscall.c 中 m，argint()、argaddr()、argstr()均可用于传递系统调用参数。其中 argint()用于获取第 n 个 32bit 的系统调用参数，且参数类型为 int，返回值为 0。argaddr()用于获取第 n 个系统调用参数，且参数类型为 uint64，为地址，返回值为 0。argstr()用于获取第 n 个系统调用参数，且参数类型为字符串，字符串的最大长度为 max，若成功则返回值为字符串长度，否则返回值为-1。

argstr()是调用 argaddr()实现的，而 argint()与 argaddr()均是通过调用 argraw()实现的。

```
static uint64
argraw(int n)
```

argraw()的输入参数为 n，返回值类型为 uint64。其中 n（0≤n≤5）表示寄存器 an，通过 p->trapframe->an，可以获取寄存器 an 中的内容并作为返回值返回。

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？

进程控制块保存在 kernel/proc.c 的 proc 数组中，如下图：

```
struct proc proc[NPROC];
```

其中 NPROC 为 64，表示最大的进程个数。类型 proc 为存储每个进程控制块的数据结构，定义在 kernel/proc.h 中。

进程控制块中的成员 state 指示了进程的状态：

```
enum procstate state;
```

一共有五种状态：新建态 UNUSED、阻塞态 SLEEPING、就绪态 RUNNABLE、运行态 RUNNING、终止态 ZOMBIE。如下图：

```
enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

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？Xv6 中的一个页有多少字节？

在 kernel/kalloc.c 中，结构体 kmem 的成员 freelist 可以指示空闲的内存页：

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

xv6 中的一个页有 4096 字节，程序中用 PGSIZE 表示，定义在 kernel/riscv.h 中：

```
#define PGSIZE 4096 // bytes per page
```

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

由于内核页表对于指针的映射和用户页表对于指针的映射不同，因此，当内核需要使用某些系统调用传递的指针参数来读写用户空间时便会发生错误。所以就需要进行内核空间和用户空间的地址转换。在 xv6 中，我们可以使用 copyout/copyin 这两个函数分别完成内核空间到用户空间、用户空间到内核空间的数据拷贝。

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
```

copyout()函数的参数如上图所示，它用于将数据从内核态拷贝到用户态。其中，pagetable 表示进程的页表，dstva 为目标空间的虚拟地址，src 为指向待拷贝的源数据的指针，len 表示待拷贝数据的长度。所以该函数表示，结合进

程所给出的页表 pagetable，将长度为 len 字节的内容从 src 拷贝到目标虚拟地址 dstva。

二、 实验详细设计

1、任务一要求在 xv6 中加入具有跟踪功能的 trace 系统调用，用于打印所有使用到的系统调用的信息。打印格式如下：

```
PID: sys_$name(arg0) -> return_value
```

其中 PID 为当前进程的 PID，$name 为被追踪的系统调用的名称，arg0 为被追踪的系统调用的第一个参数，return_value 为被追踪的系统调用的返回值。

为此，首先在 Makefile 中给 UPROGS 加入`$U/_trace\`，在 user/user.h 中加入`int trace(int);`即系统调用 trace 的函数接口，使用户能够调用该 trace 系统调用。

这只是封装的接口，然后我们根据系统调用的真正实现过程添加相应的代码。

脚本文件 user/usys.pl 会在编译期间被执行，生成汇编文件 usys.S。在该脚本文件中，对每一个系统调用的抽象接口都生成了一个叫做 entry 的具体实现。因此，我们需要在该文件中增加`entry("trace");`。

entry 函数如下：

```
9   sub entry {
10      my $name = shift;
11      print ".global $name\n";
12      print "${name}:\n";
13      print " li a7, SYS_${name}\n";
14      print " ecall\n";
15      print " ret\n";
16  }
```

(*)

print 的内容就是 RISCV 指令集的汇编语言。entry 的参数（如“trace”）会替换 11、12、13 行中的$name。第 11 行会将系统调用（如 SYS_trace）的编号存入 a7 寄存器中，然后内核就可以通过该寄存器得知当前要处理的是什么系统调用。因此，需要在 kernel/syscall.h 中添加 trace 系统调用的编号，如下`#define SYS_trace  22`。同时，在 kernel/syscall.c 中，数组*syscalls[]保存了每个编号对应到的系统调用函数指针，因此需要在该数组中增加 trace 系统调用`[SYS_trace]   sys_trace,`，[ ]中表示的是系统调用的编号，该语句表示将 sys_trace 系统调用的函数指针存储在数组中下标为其相应编号的位置上。

事实上，在调用 user/user.h 中的函数接口后，所需要的参数就已经被存储在相应的寄存器中了（如 trace 系统调用的参数即待追踪的系统调用编号被存入了 a0 寄存器中）。

接下来第 12 行的 ecall 指令使得 CPU 从用户态转入了内核态，然后内核就可以根据存储在 a7 寄存器中的系统调用编号以及存储在其它寄存器中的参数来执行不同的系统调用函数。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。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。

此时，kernel/syscall.c 中的 syscall()函数会根据 a7 寄存器中的编号来进行系统调用的分发，因此，我们可以在分发的同时来实现 trace 追踪并打印系

统调用的功能。

在此之前，为补充 trace 系统调用，我们首先需要在该文件中加上以 extern 进行标识的函数接口`extern uint64 sys_trace(void);`。

extern 声明了 trace 函数，但该函数的具体实现是在 kernel/sysproc.c 中进行的：

```
uint64
sys_trace(void)
{
  int n;  // 表示待追踪的系统调用的编号
  if(argint(0, &n) < 0)  // 从寄存器a0中读取该编号
    return -1;
  myproc() -> mask = n;  // 将PCB中的mask值设置为该编号
  return 0;
}
```

首先，调用 argint 从寄存器 a0 中读取待追踪的系统调用的编号，若失败则返回-1。然后将该编号存储到当前进程 PCB 中的 mask 成员变量中。注意我们需要在 kernel/proc.h 中定义 PCB 数据结构的`struct proc`中添加 mask 成员变量`int mask;`，用以存储待追踪的系统调用的编号。mask 的每一位对应一个系统调用，位的比特值指示是否需要追踪其对应的系统调用。我们在 kernel/proc.c 初始化 PCB`procinit(void)`时，将 mask 初始化为 0`p->mask = 0;`（在不调用 trace 系统调用时保证 mask 为 0，这样不会由于 mask 被初始化为非 0 值而产生错误的打印输出）。同时，由于 fork 后子进程会继承父进程的 mask，因此我们需要在 kernel/proc.c 的 fork()函数中添加该语句`np->mask = p->mask;`。至此我们完成了 trace 系统调用函数的具体实现。

然后我们要在 kernel/syscall.c 的 syscall()函数分发各系统调用的同时，打印待追踪的系统调用的信息，也即实现 trace 系统调用的真正功能。在 syscall()函数中，首先通过 myproc()获取指向当前进程的 PCB 的指针 p：`struct proc *p = myproc();`。然后核心逻辑如下图所示：

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();
  int arg0;
  int return_value;

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    arg0 = p->trapframe->a0;  // 保存系统调用的arg0
    return_value = syscalls[num]();  // 保存系统调用的return_value
    p->trapframe->a0 = return_value;
    if((p -> mask) & (1 << num)){  // 判断当前系统调用是否为待追踪的系统调用
      printf("%d: %s(%d) -> %d\n", p->pid, syscalls_name[num], arg0, return_value);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

在 PCB 中，trapframe 保存了用户进程的寄存器状态，即上下文。因此可通过 trapframe 获取各个寄存器中的值。如上所述，当前系统调用的编号存储在寄存器 a7 中，通过 p->trapframe->a7，获取当前的系统调用编号 num。然后判断系统调用编号 num 是否在合法范围内，同时判断 syscalls[num]，即该编号所对应的系统调用处理函数是否存在。若存在，即可通过该函数指针，调用对应的系统调用处理函数。在调用之前，首先需要保存寄存器 a0 中的值，该值为当前系统调用的第一个参数 arg0。然后将该系统调用的返回值 return_value 存储到 a0 寄存器中，因为函数的返回值默认存储在 a0 寄存器中。接着判断当前系统调用是否为待追踪的系统调用，即判断 p -> mask 从低至高的第 num 位是否为 1（因为 mask 的每一位对应一个系统调用，位的比特值指示是否需要追踪其对应的系统调用），即判断(p -> mask) & (1 << num)是否为 1。若当前系统调用是待追踪的系统调用，则打印该系统调用的相关信息：

```
PID: sys_$name(arg0) -> return_value
```

其中，pid 可从 PCB 中获取，name 为该系统调用的名称，arg0 即刚刚保存的该系统调用的第一个参数，return_value 即刚刚保存的该系统调用的返回值。对于 name，我们需要在该文件中定义一个 syscalls_name 的数组，用于存储每个系统调用的编号所对应的系统调用的名称：

```
// 存储每个编号对应的系统调用的名称
static char *syscalls_name[] = {
[SYS_fork]    "sys_fork",
[SYS_exit]    "sys_exit",
[SYS_wait]    "sys_wait",
[SYS_pipe]    "sys_pipe",
[SYS_read]    "sys_read",
[SYS_kill]    "sys_kill",
[SYS_exec]    "sys_exec",
[SYS_fstat]   "sys_fstat",
[SYS_chdir]   "sys_chdir",
[SYS_dup]     "sys_dup",
[SYS_getpid]  "sys_getpid",
[SYS_sbrk]    "sys_sbrk",
[SYS_sleep]   "sys_sleep",
[SYS_uptime]  "sys_uptime",
[SYS_open]    "sys_open",
[SYS_write]   "sys_write",
[SYS_mknod]   "sys_mknod",
[SYS_unlink]  "sys_unlink",
[SYS_link]    "sys_link",
[SYS_mkdir]   "sys_mkdir",
[SYS_close]   "sys_close",
[SYS_trace]   "sys_trace",
[SYS_sysinfo]   "sys_sysinfo",
};
```

若当前的系统调用编号 num 不合法，则输出错误信息，并将寄存器 a0（用于保存返回值）置为-1。

syscall()函数执行完成后，程序返回到上图（*）user/usys.pl 中的第 15 行，执行 ret 指令后就结束了系统调用，从内核态返回至用户态。

至此，完成了 trace 系统调用的实现。

2、任务二要求在 xv6 中加入 sysinfo 系统调用，参数为指向结构体 sysinfo 的指针。结构体 sysinfo 的定义如下：

```
struct sysinfo {
  uint64 freemem;   // amount of free memory (bytes)
  uint64 nproc;     // number of process
  uint64 freefd;    // number of free file descriptor
};
```

该系统调用用于计算出当前剩余的内存字节数、状态为 UNUSED 的进程个数、当前进程可用文件描述符的数量（即尚未使用的文件描符数量），并将它们填入结构体 sysinfo 的相应变量中。

与完成 trace 系统调用的过程类似。首先在 Makefile 中给 UPROGS 加入 `$U/_sysinfotest\`（该程序是用户级应用程序，依次测试剩余的内存字节数、UNUSED 的进程个数、未被使用的文件描述符数量），在 user/user.h 中加入

```
struct sysinfo;  // 需要预先声明结构体，参考fstat的参数stat
int sysinfo(struct sysinfo *);
```

即系统调用 sysinfo 的函数接口，使用户能够调用该 sysinfo 系统调用。

这只是封装的接口，然后同样我们根据系统调用的真正实现过程添加相应的代码。

脚本文件 user/usys.pl 会在编译期间被执行，生成汇编文件 usys.S。在该脚本文件中，对每一个系统调用的抽象接口都生成了一个叫做 entry 的具体实现。因此，我们需要在该文件中增加`entry("sysinfo");`。

```
9   sub entry {
10      my $name = shift;
11      print ".global $name\n";
12      print "${name}:\n";
13      print " li a7, SYS_${name}\n";
14      print " ecall\n";
15      print " ret\n";
16  }
```

entry 函数中的第 11 行会将系统调用（如 SYS_sysinfo）的编号存入 a7 寄存器中，然后内核就可以通过该寄存器得知当前要处理的是什么系统调用。因此，需要在 kernel/syscall.h 中添加 sysinfo 系统调用的编号，如下`#define SYS_sysinfo  23`。同时，在 kernel/syscall.c 中，数组*syscalls[]保存了每个编号对应到的系统调用函数指针，因此需要在该数组中增加 sysinfo 系统调用`[SYS_sysinfo]   sys_sysinfo,`，[]中表示的是系统调用的编号，该语句表示将 sys_sysinfo 系统调用的函数指针存储在数组中下标为其相应编号的位置上。

事实上，在调用 user/user.h 中的函数接口后，所需要的参数就已经被存储在相应的寄存器中了（如 sysinfo 系统调用的参数即指向结构体 sysinfo 的指针被存入了 a0 寄存器中）。

与 1 中的 trace 类似，第 12 行的 ecall 指令使得 CPU 从用户态转入了内核态，然后 kernel/syscall.c 中的 syscall()函数就可以根据存储在 a7 寄存器中的系统调用编号以及存储在其它寄存器中的参数来执行不同的系统调用函数。

为补充 sysinfo 系统调用，我们首先需要在该文件中加上以 extern 进行标识的函数接口`extern uint64 sys_sysinfo(void);`。

extern 声明了 sysinfo 函数，但该函数的具体实现是在 kernel/sysproc.c 中进行的：

```
uint64
sys_sysinfo(void)
{
  uint64 p_info;
  if(argaddr(0, &p_info) < 0)  // 获取指向sysinfo结构体的指针
    return -1;

  struct sysinfo info;
  info.freemem = sizeof_freemem();  // 计算剩余的内存空间
  info.nproc = numof_nproc();  // 计算空闲进程数量
  info.freefd = numof_freefd();  // 计算可用文件描述符数量

  struct proc *p = myproc();

  // 将内核态的info结构体，结合进程的页表，写到进程内存空间内的p_info指针指向的地址处。
  if(copyout(p->pagetable, p_info, (char *)&info, sizeof(info)) < 0)
    return -1;

  return 0;
}
```

首先,调用 argaddr 从寄存器 a0 中读取指向结构体 sysinfo 的指针到 p_info 中，若失败则返回-1。由于内核页表对于指针的映射和用户页表对于指针的映射不同，因此，当内核直接使用系统调用传递的指针参数来读写用户空间时便会发生错误。所以就需要使用 copyout 完成内核空间到用户空间的数据拷贝。

我们新定义一个指向结构体 sysinfo 的指针 info(该结构体的地址为内核态的地址），通过 sizeof_freemem()函数计算剩余的内存空间存入 info.freemem 中，通过 numof_nproc()函数计算空闲进程数量存入 info.nproc 中，通过 numof_freefd()函数计算可用文件描述符数量存入 info.freefd 中。这三个函数我们稍后实现。

然后通过 myproc()获取指向当前进行 PCB 的指针 p，利用 copyout 函数，将内核态的 info 结构体，结合进程的页表 p->pagetable，写到进程内存空间内的 p_info 指针指向的结构体 sysinfo 的地址处。

接下来，我们实现上述三个函数。首先，我们在 kernel/kalloc.c 中实现计算空闲内存空间大小的函数 sizeof_freemem:

```
// 计算空闲内存大小
uint64
sizeof_freemem(void)
{
  struct run *r;
  uint64 freememsize = 0;

  acquire(&kmem.lock);  // 获取锁
  r = kmem.freelist;  // 指向空闲的内存页

  // 通过计算空闲页数来计算空闲内存大小
  while (r)
  {
    freememsize += PGSIZE;  // 加上每一页的字节数
    r = r -> next;  // 指向下一个空闲的内存页
  }

  release(&kmem.lock);  // 释放锁

  return freememsize;
}
```

我们通过计算空闲的内存页数来计算空闲内存的大小，用 freememsize 表示。首先获取 kmem 的锁，然后通过 kmem.freelist 获取指向空闲内存页的指针 r。遍历所有的空闲内存页，每轮给 freememsize 加上一页所占的字节数。遍历完成后释放锁，同时返回 freememsize，即空闲内存大小。

然后我们在 kernel/proc.c 中实现计算空闲进程数量的函数 numof_nproc：

```
// 计算状态为UNUSED的进程个数
uint64
numof_nproc(void)
{
  struct proc *p;
  uint64 nprocnum = 0;

  for(p = proc; p < &proc[NPROC]; p++){  // 遍历所有进程的PCB
    if(p -> state == UNUSED) nprocnum++;  // 若状态为UNUSED，则个数加一
  }

  return nprocnum;
}
```

在该函数中，我们用 nprocnum 存储状态为 UNUSED 的进程个数。我们遍历所有进程的 PCB。具体来说，我们遍历存储所有进程 PCB 的数组 proc`struct proc proc[NPROC];`。若当前指针所指向的 PCB 中存储的 state 为 UNUSED，即表示当前进程为空闲进程，那么空闲进程数量 nprocnum++。最终返回 nprocnum，即状态为 UNUSED 的进程个数。

同样，我们在 kernel/proc.c 中实现计算可用文件描述符数量的函数

numof_freefd:

```
// 计算当前进程可用文件描述符的数量
uint64
numof_freefd(void)
{
  uint64 freefdnum = 0;
  struct proc *p = myproc();

  for(int fd = 0; fd < NOFILE; fd++){  // 遍历所有文件描述符
  // ofile中存储了当前进程的所有文件描述符，若该文件描述符尚未使用，则数量加一
    if(!(p->ofile[fd])) freefdnum++;
  }

  return freefdnum;
}
```

在该函数中，我们用 freefdnum 存储当前进程可用的文件描述符的数量。我们遍历进程的所有文件描述符。具体来说，我们遍历存储了当前进程的所有文件描述符的数组 ofile，NOFILE 为每个进程的文件描述符数量。若当前遍历到的文件描述符为空，即该文件描述符尚未使用，则 freefdnum++。最终返回 freefdnum，即当前进程可用的文件描述符的数量。

在具体实现这三个函数的同时，我们还需要在 kernel/defs.h 中添加这三个函数的声明：

```
uint64          sizeof_freemem(void);
uint64          numof_nproc(void);
uint64          numof_freefd(void);
```

最后，还需要在 kernel/syscall.c 的 syscalls_name 数组中加入该系统调用：`[SYS_sysinfo]   "sys_sysinfo",`，以便上一任务中的 trace 系统调用也能追踪到这一系统调用。

至此，完成了 sysinfo 系统调用的实现。

三、 实验结果截图

任务 1：trace 系统调用

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: sys_read(3) -> 1023
3: sys_read(3) -> 966
3: sys_read(3) -> 70
3: sys_read(3) -> 0
$ trace 2147483647 grep hello README
4: sys_trace(2147483647) -> 0
4: sys_exec(12240) -> 3
4: sys_open(12240) -> 3
4: sys_read(3) -> 1023
4: sys_read(3) -> 966
4: sys_read(3) -> 70
4: sys_read(3) -> 0
4: sys_close(3) -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: sys_fork(0) -> 7
test forkforkfork: 6: sys_fork(1) -> 8
8: sys_fork(0) -> 9
9: sys_fork(-1) -> 10
10: sys_fork(-1) -> 11
9: sys_fork(-1) -> 12
10: sys_fork(-1) -> 13
9: sys_fork(-1) -> 14
10: sys_fork(-1) -> 15
11: sys_fork(-1) -> 16
9: sys_fork(-1) -> 17
10: sys_fork(-1) -> 18
9: sys_fork(-1) -> 19
10: sys_fork(-1) -> 20
```

```
10: sys_fork(-1) -> 57
9: sys_fork(-1) -> 58
11: sys_fork(-1) -> 59
10: sys_fork(-1) -> 60
9: sys_fork(-1) -> 61
10: sys_fork(-1) -> 62
9: sys_fork(-1) -> 63
10: sys_fork(-1) -> 64
11: sys_fork(-1) -> 65
9: sys_fork(-1) -> 66
10: sys_fork(-1) -> 67
9: sys_fork(-1) -> 68
10: sys_fork(-1) -> -1
9: sys_fork(-1) -> -1
12: sys_fork(-1) -> -1
OK
6: sys_fork(0) -> 69
ALL TESTS PASSED
```

任务 2：sysinfo 系统调用

```
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

整体实验结果截图

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.4s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.7s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.6s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (13.9s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.9s)
== Test time ==
time: OK
Score: 35/35
200110513@comp6:~/xv6-labs-2020$
```