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

实验设计报告

开课学期：2023 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：2023.10.10 地点：T2507

学生班级：8

学生学号：210110812

学生姓名：李春阳

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 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？

在 user/usys.pl 文件中，可以观察到 entry 中存在一行代码“li a7, SYS_${name}\n”，意思是将系统调用号 SYS_${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。

syscall()首先通过 myproc()函数获取指向当前进程 PCB 的指针 p，然后利用 p->trapframe->a7 获取寄存器 a7 中储存的当前系统调用编号（在 kernel/syscall.h 中定义）。如果获取到的系统调用编号在合理范围内，那么通过 p->trapframe->a0 = syscalls[num]()调用编号对应的处理函数（例如 sys_fork），并将具体的系统调用的返回值储存在 a0 寄存器中，否则返回-1。

```
static uint64 (*syscalls[])(void) = {
    [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_rename] sys_rename, [SYS_yield] sys_yield,
};
```

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。

有 3 个函数用于传递系统调用参数，分别是：

int argint(int n, int *ip)：获取第 n 个寄存器的信息，并用 int 型指针指向它

int argaddr(int n, uint64 *ip)：获取第 n 个寄存器的信息，并用无符号 int64 型指针指向它(内容是地址)

int argstr(int n, char *buf, int max)：获取第 n 个寄存器的信息，并用 char 型指针指向它(内容是字符串，最大长度为 max)

argraw()函数的含义：

argraw()输入参数为 n，返回类型为 uint64。argraw 函数中首先通过 myproc()函数获得当前进程的指针，然后依据参数 n 访问该进程的 trapframe 结构体，获取寄存器 an 中的内容并将其作为函数返回值。

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？

1）进程控制块储存在 kernel/proc.c 中的 struct proc proc[NPROC]数组中

2）观察 kernel/proc.h 可知，进程控制块中的成员 enum procstate state 指示了进程的状态，共有五个状态，分别是：UNUSED（新建态），SLEEPING（阻塞态），RUNNABLE（就绪态），RUNNING（运行态），ZOMBIE（终止态）

4. 在任务一当中，为什么子进程（4、5、6 号进程）的输出之前会 **稳定的** 出现一个$符号？（提示：shell 程序(sh.c)中什么时候打印出$符号？）

   $符号是命令行提示符，表示此时正在等待用户输入命令。用于测试的 exittest 程序通过 fork 系统调用创建 3 个子进程，然后使父进程先退出，3 个子进程再退出。当父进程退出时，命令执行结束，会出现$符号等待用户输入下一个命令。随后，3 个子进程退出，此时子进程已被 initproc 管理，会在用户还未输入下一个命令前打印出相关信息（这 3 个子进程再无子进程，因此只能打印出当前进程的父进程的信息）。

5. 在任务三当中，我们提到测试时需要指定 CPU 的数量为 1，因为如果 CPU 数量大于 1 的话，输出结果会出现乱码，这是为什么呢？（提示：多核心调度和单核心调度有什么区别？）

   因为当 CPU 数量大于 1 时，多个 CPU 同时访问和修改共享资源（如标准输出）可能导致竞争条件，导致可能出现打印一个/几个字符后就被别的 CPU 抢去的情况，从而导致输出混乱。

   不指定 CPU 数量为 1 的结果如下：

```
$ yieldtest
yield test
paressnwwiittt cchhy ite otl odc
hstart to yield, user pc 0x0 0000000000003e2
islwpdia tr0c
ehn t[INFO] proc 4 exit, parent pid 3, name yieldtest, state running
tch iyoile lddc  hfii1n
il[INFO] proc 5 exit, parent pid 3, name yieldstest, state running
dh ed2

[INFO] proc 6 exit, parent pid 3, name yieldtest, state running
[INFO] proc 3 exit, parent pid 2, name sh, state sleeping
```

二、 实验详细设计

任务 1：进程信息收集

在回收完当前进程打开的所有文件且唤醒初始进程后，由于进程控制块 PCB 中存在成员变量 parent，所以很容易得到当前进程的父进程控制块并通过指针 original_parent 保存。

```
struct proc *original_parent = p->parent;
```

在当前进程的子进程的父进程指针更改成 initproc（执行 reparent 函数）之前，我们需要通过封装好的 exit_info 函数输出当前进程的父进程及子进程控制块信息，否则当父进程退出、子进程变为孤儿进程后，其父进程 pid 将统一变为 1。

较难获取的是当前进程的子进程，对此，我们需要对所有进程控制块进行遍历，通过 pp->parent == p 条件判断此进程是否为 p 的子进程，并设置 int 型变量 i 对当前进程的子进程进行计数及排序。

```
struct proc *pp;
int i=0;
for (pp = proc; pp < &proc[NPROC]; pp++){
  if (pp->parent == p){
    acquire(&pp->lock);
    exit_info("proc %d exit, child %d, pid %d, name %s, state %s\n", p->pid, i, pp->pid, pp->name, pstate[pp->state]);
    i++;
    release(&pp->lock);
  }
}
```

需要注意的是，进程块中的状态以枚举变量表示，而枚举变量无法直接打印（只能打印出对应 int 值），因此我们需要额外设置一个数组来存放状态字符串，并通过 exit_info(“%s”, pstate[original_parent->state])的方式输出。

```
char * pstate[] = {"unused", "sleeping", "runnable", "running", "zombie"};
```

任务 2：wait 系统调用的非阻塞选项实现

为实现 wait 系统调用的非阻塞选项实现，我们在 wait 系统调用增加 int flags 来表示是否进行非阻塞 wait。当 flags=1 时，不需要阻塞等待，而是解锁后直接返回-1；否则通过 sleep 函数实现父进程等待。

```
int wait(uint64 addr, int flag)
```

```
  // Wait for a child to exit.
  if (flag != 1){
    sleep(p, &p->lock);  // DOC: wait-sleep
  } else {
    release(&p->lock);
    return -1;
  }
```

除了需要更改 kernel/proc.c 中的 wait 函数之外，还需要更改系统调用函数 sys_wait()。利用寄存器传参，我们通过 argint(1,&n)获取用户态传入的第二个系统调用参数（即储存在 a1 寄存器中的 flag 值），并传递给 wait(p, n)返回。

```
uint64 sys_wait(void) {
  uint64 p;
  int n;
  if (argaddr(0, &p) < 0) return -1;
  if (argint(1, &n) < 0) return -1;
  return wait(p, n);
}
```

最后，更改 kernel/defs.h 头文件中的 wait 的定义。

```
int             wait(uint64, int flag);
```

任务 3：实现 yield 系统调用

在用户部分，我们需要：在 user.h 添加相关的系统调用声明；在 usys.pl 文件中新增一个 entry；在 Makefile 的 UPROGS 变量中新增一个用户程序_yieldtest。

```
entry("yield");
```

```
void yield(void);
```

```
$U/_yieldtest\
```

在 kernel/proc.c 中，我们已经实现了 yield 具体的系统调用函数功能，但此函数并未成为一个系统调用。对此，我们在 kernel/sysproc.c 中新增系统调用函数 sys_yield()。sys_yield()首先通过 myproc()函数获取指向当前进程 PCB 的指针 p，并在 PCB 中的 trapframe 结构体取得用户态陷入内核时对应的 PC 值（epc），然后输出。最后，系统调用 sys_yield 调用函数 yield，以实现系统调用的实际功能，即将当前进程让出 CPU，从而调度到别的进程。

```
uint64 sys_yield(void) {
  struct proc *p = myproc();
  printf("start to yield, user pc %p\n", p->trapframe->epc);
  yield();
  return 0;
}
```

接着，在 sys_call.h 中加入系统调用号的定义。

```
#define SYS_yield  23
```

最后，在 syscall.c 中注册该函数，将系统调用标识 SYS_yield 和系统调用函数 sys_yield 关联起来。

```
extern uint64 sys_yield(void);
```

```
[SYS_yield] sys_yield,
```

至此，我们成功新增了 yield 的系统调用。

三、 实验结果截图

```
$ exittest
exit test
[INFO] proc 3 exit, parent pid 2, name sh, state sleeping
[INFO] proc 3 exit, child 0, pid 4, name child0, state sleeping
[INFO] proc 3 exit, child 1, pid 5, name child1, state sleeping
[INFO] proc 3 exit, child 2, pid 6, name child2, state sleeping
$ [INFO] proc 4 exit, parent pid 1, name init, state runnable
[INFO] proc 6 exit, parent pid 1, name init, state runnable
[INFO] proc 5 exit, parent pid 1, name init, state running
```

```
$ waittest
wait test
no child exited yet, round 0
no child exited yet, round 1
no child exited yet, round 2
[INFO] proc 4 exit, parent pid 3, name waittest, state sleeping
child exited, pid 4
wait test OK
[INFO] proc 3 exit, parent pid 2, name sh, state sleeping
```

```
$ yieldtest
yield test
parent yield
start to yield, user pc 0x00000000000003e2
switch to child 0
[INFO] proc 4 exit, parent pid 3, name yieldtest, state runnable
switch to child 1
[INFO] proc 5 exit, parent pid 3, name yieldtest, state runnable
switch to child 2
[INFO] proc 6 exit, parent pid 3, name yieldtest, state runnable
parent yield finished
[INFO] proc 3 exit, parent pid 2, name sh, state sleeping
```

```
$ make qemu-gdb LAB_SYSCALL_TEST=on
exit test: OK (7.3s)
== Test wait test ==
$ make qemu-gdb LAB_SYSCALL_TEST=on
wait test: OK (1.6s)
== Test yield test ==
$ make qemu-gdb CPUS=1 LAB_SYSCALL_TEST=on
yield test: OK (1.2s)
== Test time ==
time: OK
Score: 100/100
```