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：智能强基-计算机

学生学号：210010101

学生姓名：房煊梓

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？

(1) syscall()函数首先通过 myproc()函数来获得当前 CPU 上运行进程的指针，将其赋给指针变量 p。
(2) 再将 p->trapframe->a7 的值，即当前运行进程的系统调用号赋给 num。
(3) 接下来对 num 进行判断，如果 num>0，num<函数指针数组 syscalls 中的元素数量，且 syscalls[num]函数指针不为空，则根据 num，利用 syscalls 的成员来调用相应的系统调用处理函数。
例如调用 sys_fork：p->trapframe->a7 的值为 1 即系统调用号为 1，则 num 被赋值为 1，满足 num>0，num<syscalls 中的元素个数，且 syscalls[1]对应 sys_fork，不为空，所以执行 p->trapframe->a0 = syscalls[num]()语句，即执行 sys_fork()函数，并且将返回值存放到 p->trapframe->a0 处。
(4) 否则，为未知系统调用，打印相关信息，并将-1 赋给 p->trapframe->a0。

syscall()将具体系统调用的返回值存在 p->trapframe->a0 里。

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。

argint(),argaddr(),argstr(),argraw()函数用于传递系统调用参数。

argraw()函数的含义：根据输入的数字 n,返回当前用户进程的相应寄存器 an 的值。当 n 的值为 0，1，2，3，4，5 时分别返回对应寄存器 an 的值，否则进行 panic。

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？

进程控制块存储在 proc 数组中。进程控制块的 state 成员指示了进程的状态。一共有 UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE 五种状态。

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？）

原因：sh.c 中，执行 getcmd 函数即在获取命令提示符时打印出$符号。观察 exittest.c 的 exittest 函数，for 循环创建 3 个子进程并且让它们睡眠 10 个单位之后再 exit。for 循环之后 sleep(1)，再执行 exit(0)。由于子进程睡眠时间较长，故 for 循环、sleep(1)、exit(0)会在所有子进程仍在睡眠时就执行完，故 exittest 会在所有子进程睡眠时就执行完。执行完 exittest 命令后需要获取命令提示符，即执行 getcmd 函数，所以会打印出$符号。再过一段时间，子进程醒来并 exit，会打印相关信息。由于 exittest 命令总是在子进程仍睡眠时就执行完，所以子进程输出之前会稳定地出现一个$符号。

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？）

当 CPU 数量大于 1 时，为多核心调度，允许多个 CPU 同时执行多个进程。在 yieldtest 中，不同的进程需要用 printf 打印出相关信息，printf 是一个字符一个字符进行输出的，当一个进程需要打印的语句的所有字符还没有输出完，但其他进程同时进行了输出时，它们输出的结果相互混合，于是呈现出乱码。当 CPU 数量为 1 时，为单核心调度，同时只能执行一个进程，因此每个进程可以完整地执行完 printf，输出结果不会相互混合，所以不会出现乱码。

二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写

任务一详细设计：
1.任务分析
(1)exit 函数的整体流程：
首先关闭所有打开的文件，然后唤醒初始进程，保存当前进程的父进程指针，然后将当前进程的所有子进程的父进程指针更改成 initproc，再唤醒当前进程的父进程，更改当前进程状态，最后进入调度器等待被回收。
(2)任务要求：进程在退出时打印自己的父进程和子进程的信息。
(3)分析父进程打印的时机：
根据指导书对打印信息的提示，需要获得进程指针才能打印其信息，所以必须获得父进程的指针，才能打印父进程的相关信息。所以选择在保存父进程指针 original_parent 并且获得父进程指针的锁之后，利用父进程指针 original_parent 打印父进程信息。
(4)分析子进程打印的时机：
执行 reparent 函数时，将当前进程的所有子进程的父进程指针改成 initproc，而 initproc 可能还有其他的子进程，因此如果在执行 reparent 函数之后查询 initproc 的所有子进程，是无法辨认其是否为原来进程的子进程的，即原来进程的子进程信息丢失了。所以选择在获得当前进程的锁之后，执行 reparent 函数之前打印当前进程的子进程信息。
(5)经过以上分析，进行代码的编写。

2.首先在 proc.c 文件的 exit 函数里定义好进程的五种状态及其小写表示，便于打印

```
enum state { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
char *s[]={"unused","sleep","runble","run","zombie"};
```

假设打印时该进程的状态为 state，则应打印 s[state]。

3.在获取父进程的锁之后打印父进程的信息

```
338   // we need the parent's lock in order to wake it up from wait().
339   // the parent-then-child rule says we have to lock it first.
340   acquire(&original_parent->lock);
341
342   //打印父进程信息
343   exit_info("proc %d exit, parent pid %d, name %s, state %s\n",p->pid,original_parent->pid,original_parent->name,s[original_parent->state]);
```

4.在获取当前进程的锁之后，执行 reparent 函数之前打印子进程的信息
这里获取所有子进程的办法参考了 reparent 函数的做法。

```
acquire(&p->lock);

//打印当前进程p的子进程信息
struct proc *pchild;
int child_cnt=0;
for (pchild = proc; pchild < &proc[NPROC]; pchild++) {
  if (pchild->parent == p) {
    exit_info("proc %d exit, child %d, pid %d, name %s, state %s\n",p->pid,child_cnt,pchild->pid,pchild->name,s[pchild->state]);
    child_cnt++;
  }
}

// Give any children to init.
reparent(p);
```

任务二详细设计：
1.任务分析
分析阻塞的 wait 系统调用流程，首先调用 sysproc.c 的 sys_wait 函数，然后在 sys_wait 里面调用 proc.c 的 wait 函数，当前进程睡觉。分析非阻塞的 wait 系统调用的流程，首先调用 sysproc.c 的 sys_wait 函数，然后在 sys_wait 里面调用 proc.c 的 wait 函数，判断是否阻塞，若非阻塞，则直接返回-1，否则当前进程睡觉。
因此，将阻塞的 wait 系统调用修改为非阻塞的，需要添加参数 flags 来说明是否非阻塞。
2.修改 defs.h 里的 wait 函数，增加参数 flags

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

3.修改 proc.c 中的 wait 函数，增加参数 flags，并且在 sleep 之前增加一个判断，当 flags 为 1 时，不进行 sleep，而是解锁之后直接返回-1，实现非阻塞逻辑。

```
//flas为1时实现非阻塞逻辑
if(flags==1){
  //返回之前需要解锁
  release(&p->lock);
  //返回-1
  return -1;
}

// Wait for a child to exit.
sleep(p, &p->lock);  // DOC: wait-sleep
```

4.修改 sysproc.c 中的 sys_wait 函数，在内部添加一个变量 flags，利用 argint 函数获取用户态传入的参数并将其赋给 flags，最后再调用修改后的 wait 函数。
获取 flags 参数参考了上一行的 argaddr，由于 flags 是 int 类型，需要调用 argint。

```
uint64 sys_wait(void) {
  uint64 p;
  int flags;
  if (argaddr(0, &p) < 0) return -1;
  //需要获取用户态传入的新参数
  if (argint(1,&flags)<0) return -1;
  return wait(p,flags);
}
```

任务三详细设计：

1.任务分析

在调用 yield 系统调用时，打印用户态陷入内核时对应的 PC 值并且将当前进程挂起。

2.在各个文件中添加：

(1)在 Makefile 里面加上 `$U/_yieldtest\`

(2)在 syscall.h 里面加上 `#define SYS_yield  23`，其中 23 是其进程号

(3)在 syscall.c 里面加上 `extern uint64 sys_yield(void);`，

在函数指针数组 syscalls 里添加`[SYS_yield]sys_yield,`

(4)在 user.h 里面加上 `int yield(void);`

(5)在 uyus.pl 里面加上 `entry("yield");`

3.在 sysproc.c 里添加一个 sys_yield 函数：

首先调用 myproc 函数来获得当前 CPU 上运行进程的指针，将其赋给指针变量 p，再将 p->trapframe->epc 即用户态陷入内核时对应的 PC 值赋给 pc，然后进行打印。打印完之后调用内核态已经实现的 yield 函数挂起当前进程。

```
uint64 sys_yield(void){
  //获取用户上下文保存的PC值并打印
  //根据proc.h, 需要epc的值
  uint64 pc;
  struct proc *p = myproc();
  pc=p->trapframe->epc;
  printf("start to yield, user pc %p\n", pc);
  //将当前进程挂起
  yield();

  return 0;
}
```

三、 实验结果截图

```
$ 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 (1.7s)
== Test yield test ==
$ make qemu-gdb CPUS=1 LAB_SYSCALL_TEST=on
yield test: OK (0.7s)
== Test time ==
time: OK
Score: 100/100
210010101@comp2:~/xv6-oslab23-hitsz$
```