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

实验设计报告

开课学期：2024 年秋季

课程名称：操作系统

实验名称：进程与系统调用

实验性质：课内实验

实验时间：10 月 17 日地点：T2507

学生班级：22 级 4 班

学生学号：220110430

学生姓名：吴梓滔

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2024 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？

syscall.c 中将所有的系统调用处理函数指针存成一个 syscalls 数组，syscall()从寄存器 a7 中获取系统调用号 num，通过 syscalls[num]()调用指定的系统调用处理函数。例如用户态调用 fork(),运行到达 syscall()时 a7 寄存器已经设置成了相应的系统调用号 1，于是获取 num=1,syscalls[1]就是 sys_fork，通过 syscalls[1]()就完成了 sys_fork 的调用。系统调用的返回值是一个 uint64 型，会直接存放到寄存器 a0。

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。

fetchaddr(),fetchstr(),argraw(),argint(),argaddr(),argstr()用于向系统调用传递参数。

argraw()函数读入一个整型 n，从 n 指定的寄存器中返回数据。n 可以指定为 0 到 5，从 a0 到 a5 寄存器中返回数据。用户调用时存放参数的顺序是一致的，所以 n 就代表传入的第 n 个参数，argraw()即返回传入的第 n 个参数。系统调用的处理函数不会直接调用 argraw，而是调用 argint()或者 argaddr()，这两个函数中会调用 argraw()。

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，定义为 struct proc proc[NPROC];

进程控制块中 state 指示进程状态，定义为 enum procstate state;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？）

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

```
while (getcmd(buf, sizeof(buf)) >= 0) {
    ……//一些判断
    if (fork1() == 0) runcmd(parsecmd(buf));
    wait(0, 0);
  }
```

sh 关键代码如上所示，getcmd 里面输出了“$ ”。每次 sh 创建的子进程退出后，就会输出一次”$ ”。图中”$”的稳定出现是因为 exittest 进程（pid=3）一定会早于进程 4，5，6 退出，而它退出的时候 sh 就会输出“$ ”。

exittest 的运行过程：用户输入 exittest 并回车后，因为 sh（pid=2）通过 fork 创建了子进程 3，子进程 3 执行 exec 运行了 exittest 程序后，sh 运行 wait(0,0)调用，将会等待一个子进程退出并将它回收。exittest 的执行就是创建子进程并用 exit(0)退出，且让父进程先于三个子进程退出。

因此，sh（pid=2）创建的进程 3 创建了子进程 456 后先退出，并且输出了前 4 条输出；此时 sh 正运行到 wait(0,0)语句，会将它回收，随后重新开始一次等待输入命令的循环，getcmd 的调用将会输出一次“$ ”。在这之后，进程 4，5，6 才会退出并输出信息。因此，进程 3 推出并被回收后，会输出“$ ”，三个子进程的输出信息一定在这之后输出。

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？

```
$ yieldtest
yield test
pareCChhiilldd  wwiitthh  PPInDI Dt 9  y1b0e gbiengsi ntso  trou nr
uiproc 9 exit, parent pid 8, name yieldtest, state run
ne
lCproc 10 exit, parent pid 8, name yieldtest, state run
d
hSave the context of the process to the memory region from address 0x0000000080012098 to 0x
0000000080012108
Current running process pid is 8 and user pc is 0x00000000000003e2
ilpanic: release
```

多核运行的结果

在 CPU=1 的情况下，同时只有一个进程在进行，运行的过程是 yieldtest（pid=3）被创建，进程 3 创建 3 个子进程，进程 3 继续运行，输出"parent yield\n"，调用 yield(),随

后内核执行 sys_yield(),输出进程 3 的一些信息，交给调度器。调度器接下来能调度到的 3 个进程一定是进程 3 创建的 3 个子进程。子进程 4，5，6 将依次运行并退出，依次输出相关信息。3 个进程均退出后，又会调度到进程 3，再进行一次输出。这个过程会固定按照这样的顺序依次进行，得到理想的输出。

因此可以得出，yieldtest 要得到理想的输出，父进程和三个子进程一定要有顺序依次执行。

在多核的情况下，父进程 3 创建了 3 个子进程后，其他的核心立刻可以调度子进程，让子进程开始执行，而不是像单核心一样依次执行。此时四个进程会有同时执行，同时输出的情况，因此就会得到乱码。如上图输出，有“cchhiilldd wwiitthh”的信息，显然是两个子进程的开始执行的信息在同时输出。

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

exit()实验任务要求在 exit()已经实现好的代码中，添加一处代码，打印出当前进程的父进程和全部子进程的信息。问题的关键在于找到一处适合进行输出的位置。

打印父进程的相关信息，需要打印出：当前进程 pid，父进程的 pid、名称、状态。由于进程 pid、进程状态都是要求在加锁条件下才可以使用的。因此，必须在获取到当前进程锁、父进程锁的情况下，才可以进行输出。因此，在源代码中获取到父进程和当前进程锁的后面进行输出。

```
acquire(&original_parent->lock);      // 此处获取到父进程锁
acquire(&p->lock);
//…… 在此处添加代码，下一行就是添加的代码
exit_info("proc %d exit, parent pid %d, name %s, state %s\n", p->pid, original_parent->pid, original_parent->name, states_names[original_parent->state]);
```

使用 exit_info 语句打印出父进程代码。由于 PCB 中，state 是 5 种状态名的枚举，为了进行状态名的输出，还专门定义了一个字符串数组 states_names，通过枚举量 state 作为下标能够得到状态名。

与父进程的输出同理，为了输出子进程信息，必须获取子进程的锁。为了遍历所有的子进程并依次输出信息，需要设计一种遍历子进程的方法。这种遍历方式在 reparent(p)中已经实现好了，因此参考它的实现，用 for 循环遍历进程数组，每个进程都判断是不是当前进程的子进程，如果是就做操作，不是就略过。如下面代码所示

```
for (pp = proc; pp < &proc[NPROC]; pp++) {
    if (pp->parent == p) {
```

```
            acquire(&pp->lock);
            // 修改此处代码进行输出
            release(&pp->lock);
        }
    }
```

在判断 pp 的父亲是 p 时，不应持有 pp 的锁，否则可能会有死锁。因此在确定 pp 的父亲是 p 后，再获取 pp 的锁，进行输出，输出完立刻释放锁。

在这段代码结束后，就完成了父进程和子进程信息的输出，随后紧跟着原本的 reparent(p);将子进程的父进程都设置初始进程。我们添加的代码全部位于获取当前进程锁和 reparent 的中间。

任务二

wait()任务要求更改 wait()系统调用的定义，增加一个输入即阻塞选项。对内核态进行更改时，需要同时修改 proc.c 中的 wait()函数，sysproc.c 中的 sys_wait()函数。

由于用户态已经修改好了，用户态传入的第二个参数为 0 或 1，可以直接在 sys_wait()函数中用 argint 来获取。修改后的 sys_wait()如下：

```
    uint64 p;
    int flag;  // lab2-2 传一个新参数 flag 从寄存器获得
    if (argaddr(0, &p) < 0) return -1;
    if (argint(1, &flag) < 0) return -1;
    return wait(p, flag);
```

修改 wait()函数声明，多读入一个 int flag 参数。由于非阻塞的情况在没有孩子处于僵尸状态时也不会等，所以在遍历进程后应当退出 wait 调用，执行后续任务。原本代码在遍历进程后，若没有孩子，会选择退出 wait 调用，如果有孩子，会用 sleep 阻塞；在非阻塞情况下没有孩子需要回收，我们也需要退出调用，和阻塞情况下完全没有孩子需要的操作是一样的。因此，只需要增加 flag 为 1 的判断条件到原本结束调用的地方即可。

```
// 原本没有孩子，结束调用的地方
    if (!havekids || p->killed || flag) {
      release(&p->lock);
      return -1;
    }
```

随后，再更改内核代码中的 dfs.h，添加第二个 int 参数。就可以在用户态正常调用 wait 的非阻塞模式。

任务三

yield 实验任务要求实验一个 yield 系统调用的处理函数 sys_yield,而用户态的代码和系统调用函数 yield()都已经实现好了。

在 syscall.h 中新增 SYS_yield 系统调用号为 23，并在 syscall.c 的数组中添加 sys_yield，再到 sysproc.c 定义 sys_yield()函数。

sys_yield()函数首先需要完成两段信息的输出。因此，先调用 myproc()获取当前进程的 PCB，并进行两段输出。由于要打印 pid 等信息，输出信息的语句应当先获取锁，后面释

放锁。代码示意如下

```
// 1 获取当前正在执行的进程 PCB
  struct proc *p = myproc();
  acquire(&p->lock);  // 用 pid，应当获取锁
  // 2 打印 context 保存范围
  printf("Sa……s %p to %p\n", &p->context, &p->context + 1 );
  // 3 打印 pid 和 pc
  printf("C…s %d … %p\n", p->pid, p->trapframe->epc);
  release(&p->lock);  // 结束立即释放锁
```

在上面的格式化输出中，由于输出 context 的保存地址范围，context 是 PCB 中定义的一个字段，保存在内存中，可以通过&获取地址，用%p 格式化输出。并且利用指针的性质，指针加上数字，就表示增加地址单位的大小。因此&p->context + 1 就是增加一个 context 的地址单位，能够获得地址的范围。而打印 pc 值时，p->trapframe->epc 其实是一个 uint64，因为是存储着一个地址，所以地址形式输出，也是用%p 进行输出。

随后需要找到下一个会调度到的进程并输出相关信息。为了找到下一个调度的进程，需要模拟调度器的工作方式。调度器是用 for 循环遍历全部进程，找到第一个 runnable 的进程来启动。调度器的 for 循环会从第一个进程开始。我们为了找到当前进程的下一个被调度到的进程，需要从当前进程开始，进行环形遍历。

在循环中，对于每个遍历到的进程，先获取锁，判断它的状态。如果是 runnable 的，说明可以进行输出。如果不是，则不输出。随后应释放锁，再到下一个进程。由于只需要找到接下来第一个调度的进程，和调度器类似，设置一个 found 标识量，初始为 0，在找到了下一个调度的进程后，将 found 设置为 1。并且判断只在 found 为 0 时进行，当 found 为 1，已经找到下一个进程，便不需要再找了。核心代码如下所示。

```
int found = 0;  // 设置一个标记
  for (np = p; np < &proc[NPROC]; np++) {
    if(!found) {
      acquire(&np->lock);
      if (np->state == RUNNABLE) {
        printf("Next…%d … %p\n", np->pid, np->trapframe->epc);
        found = 1;    // 找到后设置标记量
      }
      release(&np->lock);
    }
  }
```

从当前进程找到最后一个进程后，还没有遍历第一个进程到当前进程。为了实现环形遍历，还需要从第一个进程开始遍历至当前进程。因此，通过两个 for 循环，来实现从当前进程开始，环形遍历全部进程的过程。代码示意如下

```
  for (np = p; np < &proc[NPROC]; np++) {
  // 判断 np 是否为 runnable，如果是，做相应操作
  }
  // 实现环形遍历进程表，上面从 p 到结束，下面这个 for 从开始到 p
  if(!found) {  //如果已经找到，下一个循环就不用做了
    for (np = proc; np < p; np++) {
      // 判断 np 是否为 runnable，如果是，做相应操作
```

```
    }
  }
```

通过两个 for 循环，就可以完成查找和输出下一个调度到的进程信息。在输出完下一进程信息后，就可以调用 yield()，主动切换进程。就完成了 sys_yield 函数的设计。

之后再完成一些编译的添加，系统调用声明等步骤，就完成了全部任务。

三、 实验结果截图

请给出实验结果截图（包括在 xv6 中运行 exit、wait、yield 的结果截图，以及 make grade 命令结果截图）

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ exittest
exit test
[INFO] proc 3 exit, parent pid 2, name sh, state sleep
[INFO] proc 3 exit, child 0, pid 4, name child0, state runble
[INFO] proc 3 exit, child 1, pid 5, name child1, state runble
[INFO] proc 3 exit, child 2, pid 6, name child2, state runble
$ [INFO] proc 4 exit, parent pid 1, name init, state runble
[INFO] proc 5 exit, parent pid 1, name init, state sleep
[INFO] proc 6 exit, parent pid 1, name init, state run
```

exit 运行截图

```
220110430@comp7:~/xv6-oslab24-hitsz$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -
c -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,driv
-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ 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 sleep
child exited, pid 4
wait test OK
[INFO] proc 3 exit, parent pid 2, name sh, state sleep
```

wait 运行截图

```
220110430@comp7:~/xv6-oslab24-hitsz$ make qemu CPUS=1
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographi
c -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio
-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ yieldtest
yield test
parent yield
Save the context of the process to the memory region from address 0x0000000080012098 to 0x0
000000080012108
Current running process pid is 3 and user pc is 0x00000000000003e2
Next runnable process pid is 4 and user pc is 0x0000000000000332
Child with PID 4 begins to run
[INFO] proc 4 exit, parent pid 3, name yieldtest, state runble
Child with PID 5 begins to run
[INFO] proc 5 exit, parent pid 3, name yieldtest, state runble
Child with PID 6 begins to run
[INFO] proc 6 exit, parent pid 3, name yieldtest, state runble
parent yield finished
[INFO] proc 3 exit, parent pid 2, name sh, state sleep
```

yield 运行截图

```
make[1]: Leaving directory '/home/students/220110430/xv6-oslab24-hitsz'

$ make qemu-gdb LAB_SYSCALL_TEST=on
exit test: OK (6.1s)
== 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)
Score: 100/100
220110430@comp7:~/xv6-oslab24-hitsz$
```

make grade 命令截图