

# 南京航空航天大学

## 操作系统实践

Proj2

班级：1618001

学号：161840230

姓名：王可

# 一、 题目简述

## 1. 线程支持(60%)

修改xv6系统使得系统支持多线程，任务分为两部分，第一部分为库函数，第二部分为系统调用（为使得编程和阅读方便，我在proc.c 文件中实现三个用户函数，使用系统调用包装）。

### 1.1 库函数

根据pthread库，实现下列三个函数：

```
1 // 1
2 int xthread_create(int *tid, void *(*start_routine)(void *), void *arg);
3 // 2
4 void xthread_exit(void *ret_val_p);
5 // 3
6 void xthread_join(int tid, void ** retval);
```

#### (1) 创建线程

该函数主要用来控制线程的创建，返回成功与否，主要功能有获得线程tid（本实验中用pid代替），使线程执行函数设置为start\_routine，并传入参数arg，需要考虑函数主动退出和运行完成被动退出时返回值的获取问题。

#### (2) 等待线程

此函数功能与wait()函数类似，调用者将等待线程号为tid的线程执行完成并将返回值传回，再继续工作，考虑参考wait，使用sleep与wakeup机制完成函数。

#### (3) 退出线程

函数主要目标是主动结束线程，获得返回值（退出码）以供其他线程获得（调用join函数）。

## 1.2 系统调用

实现以下三个系统调用辅助完成：

```
1 // 1
2 int clone(void *(*fn)(void *), void *stack, void *arg);
3 // 2
4 void join(int tid, void **ret_p, void **stack);
5 // 3
6 void thread_exit(void *ret);
```

#### (1) clone

- clone 使得新的线程共享创建者的地址空间，可以继续使用PCB数据结构；
- 通过设置新线程的初试状态trapframe使得线程运行函数fn；
- 通过传入create函数malloc获得的地址空间，设置用户栈(注意malloc返回空间低地址)。

## (2) join

- 通过join调用完成等待，并将终止线程的返回值存入参数，将用户栈释放（通过free低地址的方式）。

## (3) thread\_exit

- 主动终止线程，需要记录返回值

## 1.3 约定

- 主线程才执行clone（不需要考虑多层父子关系）；
- 主线程调用exit不调用thread\_exit（不需要判断是主线程还是子线程）；
- 主线程调用exit时，终止所有未终止线程并释放资源；
- 主线程才会fork
- PCB不可以添加太多字段

---

## 2. 优先级调度(40%)

在xv6原生的时间片轮转调度算法的基础上实现一个优先级调度算法：

- 线程具有三个优先级：1最高，3最低；
- 线程在创建时默认优先级为2；
- 调度时优先调度优先级最高的进程
- 同一优先级的多个进程采用轮转调度

实现两个系统调用来辅助完成任务：

```
1 | int enable_sched_display(int i);  
2 | int set_priority(int pid, int prior);
```

---

# 二、实验过程(1)

## 1. 线程支持

### 1.1 全局考虑

1. 线程使用PCB数据结构，将tid存入pid字段，创建者为其parent，pgdir等字段均继承自其创建者。起初我以为用户的内核栈也应公用地址空间，因此自己重新实现了allocproc函数，并未使用内置的kalloc函数。然而实现时才考虑到上下文及初始化等问题，重新使用allocproc分配内存空间。既然使用PCB，那么自然也是存入ptable数组中，即最多产生64个进程（线程）。
2. 创建时，主线程并不主动放弃时间片，只有等到join的时候才会进入sleep，等待子线程完成工作并唤醒主线程。
3. 通过阅读源码可知，线程自身结束后（无论是主动结束还是被动结束），均进入僵尸状态，只有在主线程调用exit或者其他线程调用join时才会被彻底释放，因此有可能存在同时几个线程结束并产生返回值的情况，因此，每一个线程都需要一个独立的存放返回值的空间，而不可以共用一个（这容易导致先结束的进程返回值被覆盖）。因此，设置一个PCB字段存放返回值是比较方便的选择。

4. 然而由于分数限制只能开辟一个新的字段，因此这个字段要尽可能多加利用。每个子线程自己拥有的资源只有一个用户栈。而栈由于仅由esp指针控制，具有较大的局限性，而若标记其起始或者终止位置，则可以将其化为类似数组的，可以随机存储访问的数据结构，大大提高其利用效率和便利性。因此选择在PCB中添加stack\_top字段（实际上是分配的低地址，取名top是因为栈自高向低的特性使然）。

## 1.2 创建线程

### 1.2.1 主要思路

调用allocproc函数获得PCB并分配内核栈，将相关的PCB字段由创建者继承，并根据malloc分配一个用户栈，将stack\_top字段赋值，并在栈顶端导入参数和返回地址(0xffffffff)，并初始化eip和esp指针。最后将子线程的状态设为RUNNABLE可运行。

接下来讨论返回地址问题，根据试验任务书提醒，返回地址设置为0xffffffff可以触发PAGE FAULT自陷，通过trap函数进行处理，中断号为14。实现一简单函数辅助完成被动退出后线程的处理任务如下。首先从trapframe的eax寄存器中将返回值取出，放入ret中，并存到用户栈空间底部（我认为此处可以视作无法到达，或者如果到达了，也可以在最后时间覆盖，不做使用）。随后，将该线程状态置为僵尸，唤醒其父进程，然后关闭线程资源并进入调度。

```
1 // in trap.c
2 case T_PGFLT:    // 14 PAGE FAULT
3 {
4     int ret = 0;
5     ret = myproc()->tf->eax;
6     int *ret_temp;
7     ret_temp = (int*)myproc()->stack_top;
8     *ret_temp = (int)ret;
9     myproc()->state = ZOMBIE;
10    if(myproc()->parent->state == SLEEPING)
11        myproc()->parent->state = RUNNABLE;
12    sched_return();
13    break;
14 }
15
16 // in proc.c
17 void
18 sched_return(void){
19     struct proc *curproc = myproc();
20     int fd;
21
22     //close all open files
23     for(fd = 0; fd < NOFILE; fd++){
24         if(curproc->ofile[fd]){
25             fclose(curproc->ofile[fd]);
26             curproc->ofile[fd] = 0;
27         }
28     }
29     acquire(&ptable.lock);
30     sched();
31 }
```

### 1.2.2 实现代码

```
1 // in xthread.c
2 int xthread_create(int * tid, void * (* start_routine)(void *), void * arg)
3 {
4     // add your implementation here ...
5     int flag;
6     void *stack;
7     stack = (void *)malloc(4096);
8     flag = clone(start_routine, (void *)((int)stack+4096), arg);
9     if(flag == -1){
10         return -1;
11     }
12     *tid = flag;
13     return 1;
14 }
15 // in proc.c
16 int clone(void* (*fn)(void *), void *stack, void *arg){
17     int i, pid;
18     struct proc *curproc = myproc();
19
20     struct proc *p;
21     if((p = allocproc()) == 0){
22         return -1;
23     }
24     p->sz = curproc->sz;
25     p->pgdir = curproc->pgdir;
26     p->parent = curproc;
27     p->stack_top = stack-4096;
28     *p->tf = *curproc->tf;
29
30     //clear %eax so that return 0
31     //np->tf->eax = 0;
32     stack -= 4;
33     *((int *)stack) = (int)arg;
34     stack -= 4;
35     *((int *)stack) = 0xffffffff;
36     p->tf->eip = (uint)fn;
37     p->tf->esp = (uint)stack;
38
39     //similar fork()
40     for(i = 0; i < NOFILE; i++){
41         if(curproc->ofile[i]){
42             p->ofile[i] = filedup(curproc->ofile[i]);
43         }
44     }
45     p->cwd = idup(curproc->cwd);
46
47     safestrcpy(p->name, curproc->name, sizeof(curproc->name));
48
49     pid = p->pid;
50     acquire(&ptable.lock);
51     p->state = RUNNABLE;
52     release(&ptable.lock);
53     return pid;
54 }
```

## 1.3 等待线程

### 1.3.1 主要思路

对于一个线程是否满足等待要求，有以下两点：

- 该线程存在可以等待的其他线程（根据测试样例中，不存在调用兄弟线程的情况，即存在孩子线程）
- 该线程在等待结束后可以继续执行，即处于正常运行状态（否则没有等待的意义）

因此设置haveKid和使用killed字段进行判断join调用的成功与否。

而函数具体逻辑为：

- 对所有PCB进行扫描，找到自己的目标孩子并且已经执行完毕的线程，将其资源释放，状态改为UNUSED释放PCB，并从stack\_top位置取得返回值，将返回值置入参数地址中。若没有找到符合条件的线程，调用sleep函数进入睡眠状态等待孩子将自己唤醒。而若唤醒自己的不是目标孩子，则该线程会继续陷入睡眠直到目标执行完毕为止。

### 1.3.2 实现代码

```
1  int join(int tid, void **ret_p, void **stack){
2      struct proc *curproc = myproc();
3      struct proc *p;
4      int havekid, pid;
5      acquire(&ptable.lock);
6      for(;;){
7          havekid = 0;
8          for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
9              if(p->parent != curproc){
10                 continue;
11             }
12             havekid = 1;
13             if(p->state == ZOMBIE && p->pid == tid){
14                 pid = p->pid;
15                 kfree(p->kstack);
16                 p->kstack = 0;
17                 p->pid = 0;
18                 p->parent = 0;
19                 p->name[0] = 0;
20                 p->killed = 0;
21                 p->state = UNUSED;
22                 *stack = p->stack_top;
23                 *ret_p = *(void**)p->stack_top;
24                 release(&ptable.lock);
25                 return pid;
26             }
27         }
28         if(!havekid || curproc->killed){
29             release(&ptable.lock);
30             return -1;
31         }
32         sleep(curproc, &ptable.lock);
33     }
34     return 0;
35 }
```

## 1.4 退出线程

### 1.4.1 主要思路

大体退出思路与被动退出线程大同小异，关闭文件等释放所有文件资源，并将返回值置入栈底部位置，将栈低地址返回给库函数进行空间释放（使用free函数），最后将线程状态设置为僵尸，进入调度。

### 1.4.2 实现代码

```
1 // xthread.c
2 void xthread_join(int tid, void ** retval)
3 {
4     // add your implementation here ...
5     void *stack;
6     join(tid, retval, &stack);
7     free(stack);
8
9 }
10
11 // proc.c
12 int
13 thread_exit(void *ret)
14 {
15     struct proc *curproc = myproc();
16     int fd;
17
18     if(curproc == initproc){
19         panic("init exiting");
20     }
21
22     //close files
23     for(fd = 0; fd < NOFILE; fd++){
24         if(curproc->ofile[fd]){
25             fileclose(curproc->ofile[fd]);
26             curproc->ofile[fd] = 0;
27         }
28     }
29
30     begin_op();
31     iput(curproc->cwd);
32     end_op();
33     curproc->cwd = 0;
34     acquire(&ptable.lock);
35
36     //Take the test file into account, the thread that call the function
37     // join must be the main thread ( the parent process ), so we just need to
38     // wake up the parent process
39     wakeup1(curproc->parent);
40
41     *(int*)curproc->stack_top = (int)ret;
42     curproc->state = ZOMBIE;
43     sched();
44     panic("zombie exit");
45 }
```

## 1.5 回答问题

- 在你的设计中，struct proc增加了几个字节？（优先级调度部分增加的不算）

在PCB中，我增加了4个字节 stack\_top（一个Int型字段），该字段既可以用来表示应该释放的栈地址空间，也可以用于随机访问栈地址空间，同时，还用于存放了返回值。

## 2. 优先级调度

实现优先级调度的主要思路为，对于所有PCB列表，按照优先级由高到低的顺序遍历，并在观察到更高优先级的线程时，从该优先级重新遍历，以确保高优先级的任务可以优先执行。

同时，在proc.c 定义一个全局变量用于控制sched()函数是否输出调度信息。以此控制测试的实现。实现代码如下：

```
1 // proc.c
2 int display_enabled = 0;
3
4 int
5 set_priority(int pid, int prior)
6 {
7     struct proc *p;
8
9     acquire(&ptable.lock); // to ensure the setter's success, hold the lock
10    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
11        if(p->pid != pid)
12            continue;
13        p->priority = prior;
14        release(&ptable.lock);
15        return -1;
16    }
17    release(&ptable.lock);
18    return pid;
19 }
20
21 int
22 enable_sched_display(int i)
23 {
24     display_enabled = i;
25     return display_enabled;
26 }
27
28 void
29 scheduler(void)
30 {
31     struct proc *p;
32     struct cpu *c = mycpu();
33     c->proc = 0;
34     int flag = 1;
35     for(;;){
36         // Enable interrupts on this processor.
37         sti();
38
39         // Loop over process table looking for process to run.
40         acquire(&ptable.lock);
41         for(int i = 1; i <= 3;){
42             flag = 1;
```



```

43     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
44         if(p->state != RUNNABLE || p->priority > i)
45             continue;
46         if(p->state == RUNNABLE && p->priority < i){
47             i = p->priority;
48             flag = 0;
49             break;
50         }
51
52         // Switch to chosen process. It is the process's job
53         // to release ptable.lock and then reacquire it
54         // before jumping back to us.
55         c->proc = p;
56         switchvm(p);
57         p->state = RUNNING;
58
59         swtch(&(c->scheduler), p->context);
60         switchkvm();
61
62         // Process is done running for now.
63         // It should have changed its p->state before coming back.
64         c->proc = 0;
65     }
66     if(flag) i++;
67 }
68
69 release(&ptable.lock);
70
71 }
72 }
73
74
75 void
76 sched(void)
77 {
78     int intena;
79     struct proc *p = myproc();
80     if(display_enabled)
81         cprintf("%d - ", p->pid);
82     //.....
83 }

```

####

父进程的优先级为1，为何有时优先级低的子进程会先于它执行？父进程似乎周期性出现在打印列表中，为什么？（你需要阅读schedtest.c）set\_priority系统调用会否和scheduler函数发生竞争条件（race condition）？你是如何解决的？

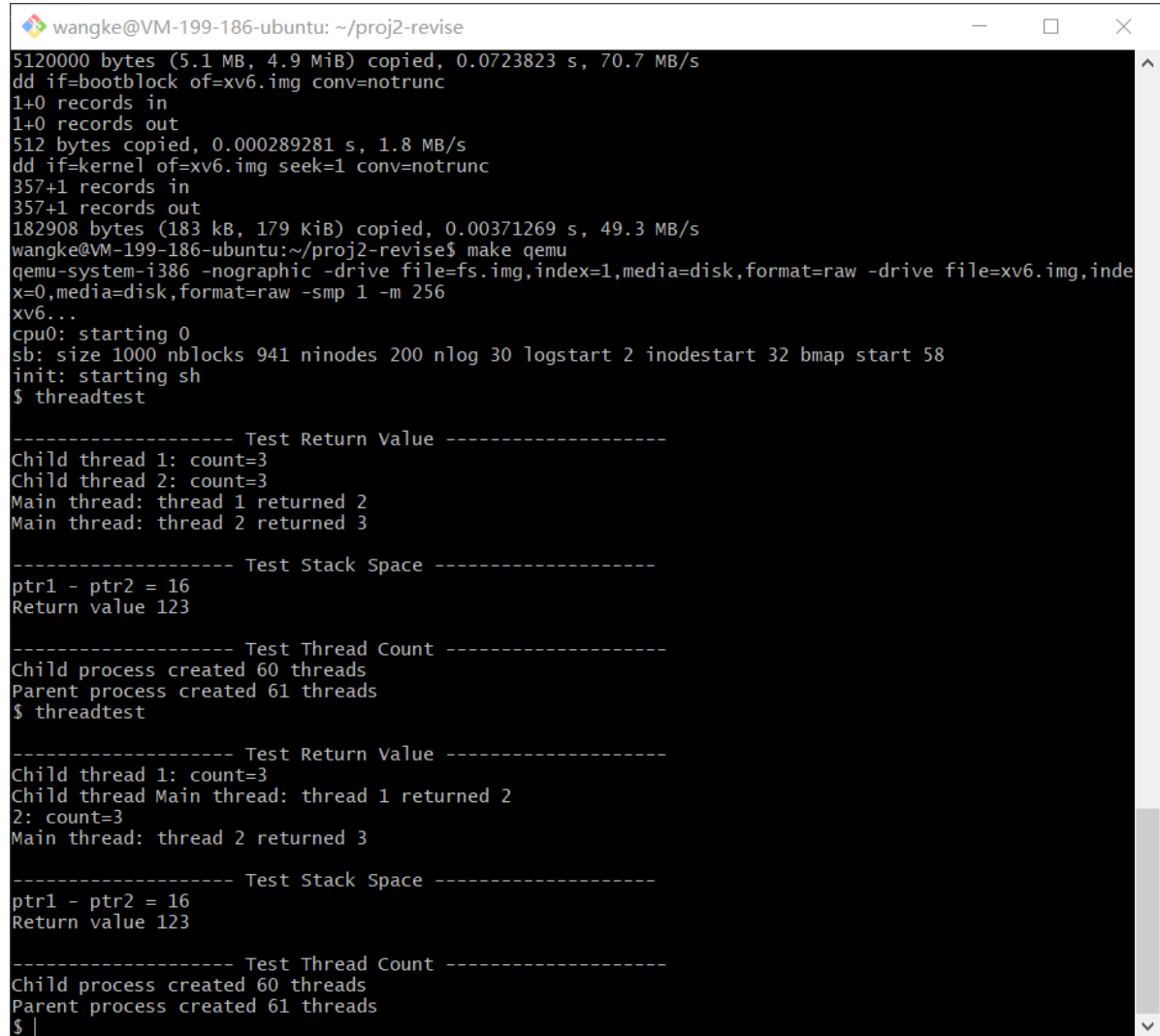
1. 在测试函数中，进行调度时，父进程循环7次调用wait()函数，进入sleeping状态，而调度函数只调度状态为RUNNABLE的进程。因此此时若没有与父进程优先级相同的进程，则调度算法将会将时间片交给低优先级的子进程，从而产生优先级低的子进程先于父进程执行的情况。
2. 父进程周期性出现在打印列表中也是出于上面解释的原因，当父进程执行wait()后，任一执行完成的子进程都将唤醒它，而因为父进程优先级高，被唤醒后又会很快进入wait()，因此会周期性出现在打印列表中。
3. 会发生竞争，因为两个函数都会访问ptable，并且对ptable中的信息进行修改（状态和优先级），若同时操作有可能出现数据丢失。我仿照其他使用ptable的函数，在进入set\_priority循环前对

ptable加锁，函数结束时再释放锁。

## 三、实验结果

### 1. 线程支持

在xv6中连续输入两次threadtest，按要求输出了指定内容：



```
wangke@VM-199-186-ubuntu: ~/proj2-revise
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0723823 s, 70.7 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000289281 s, 1.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
357+1 records in
357+1 records out
182908 bytes (183 kB, 179 KiB) copied, 0.00371269 s, 49.3 MB/s
wangke@VM-199-186-ubuntu:~/proj2-revise$ make qemu
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ threadtest

----- Test Return Value -----
Child thread 1: count=3
Child thread 2: count=3
Main thread: thread 1 returned 2
Main thread: thread 2 returned 3

----- Test Stack Space -----
ptr1 - ptr2 = 16
Return value 123

----- Test Thread Count -----
Child process created 60 threads
Parent process created 61 threads
$ threadtest

----- Test Return Value -----
Child thread 1: count=3
Child thread Main thread: thread 1 returned 2
2: count=3
Main thread: thread 2 returned 3

----- Test Stack Space -----
ptr1 - ptr2 = 16
Return value 123

----- Test Thread Count -----
Child process created 60 threads
Parent process created 61 threads
$ |
```

### 3. 优先级调度

在xv6中输入schedtest，输出以下信息：

```
wangke@VM-199-186-ubuntu: ~/proj2-revise
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -dr
ive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ schedtest
=====
Parent (pid=3, prior=1)
Child (pid=4, prior=1) created!
Child (pid=5, prior=2) created!
Child (pid=6, prior=3) created!
Child (pid=7, prior=1) created!
Child (pid=8, prior=2) created!
Child (pid=9, prior=3) created!
Child (pid=10, prior=2) created!
=====
3 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 3 - 4 - 3 - 4 - 3 - 3 - 5 - 8 - 10 - 5 - 8 -
10 - 5 - 8 - 10 - 5 - 8 - 10 - 5 - 8 - 10 - 3 - 5 - 10 - 3 - 6 - 9 - 6 - 9 - 6
- 9 - 6 - 9 - 6 - 9 - 3 - 6 - 6 - 3 -
$
$
```

## 四、实验心得

本次实验对进程和线程的内存空间理解更加深刻，起初希望使用纯使用内核栈的方式进行实现，达到完全不修改PCB的目的，但是由于时间问题，最终没能完美地实现，因此更改为在PCB中增加一个字段 `stack_top`。而在整个实验中，给我印象比较深的其实是过程中对实验不要求的内容的畅想，比如实验限制非主线程不会再调用 `thread_create()` 函数，那么如果其他线程也能创建会是什么样的情况？或者说，将PCB和TCB分开，将TCB放入PCB中，是不是能扩展系统的能力等等，这些拓展的想法使得我更加思考深入的内容。

需要进行一点说明的是，这一次实验的代码我参考了部分林新智同学的实现，虽然整体思路都是我和他共同思考讨论得出的。我本来意图实现另一套思路，在PCB里不添加任何字段，全部使用父线程的内核栈完成，在context下先使用一个 `Int` 型存储现存的子线程数量，再用 `64*3*sizeof(Int)` 的大小存储大小为64的结构体数组，用来存储子线程的TID，返回类型（主动退出为0，被动退出为1），和返回值。通过这种方式，不仅父线程可以随时取到返回值和自己的子线程，其他线程对兄弟线程也有访问和查询的能力。并且，可以通过这样的方式实现递归 `exit()`，只要对每个子线程的tid，找到其内核栈，就能再递归查找到其子线程，从而进行树形搜索资源释放。同时，由于PCB中有parent字段，这样的线程树还是一棵双向树，更容易实现线程之间的联系。在实现过程中，测试点1和测试点2都顺利完成，但测试点3中，却遇到了无法创建60个线程就遭遇各种trap, panic的状况，努力查找bug后也仅仅发现不存储子线程数量可以使得线程全部创建，但是这是因噎废食的方案，并不能解决问题，由于时间的限制，在gdb调试手段的协助下我也未能完成所有的测试。因此我转而使用了和林新智同学一样的实现方法，未来有机会的话再尝试进行debug。

