Lab 4 调度

宁晨然 17307130178

前言：本次实验花了很多很多功夫和精力！并且自我思考了很多东西！

本实验报告能充分体现出来！希望助教好好浏览~

1.实验目的

- 理解操作系统的调度管理机制
- 熟悉 `xv6` 的系统调度器框架，以及缺省的 `Round-Robin` 调度算法
- 实现 `Priority based scheduling` 调度算法来替换缺省的调度算法

2.实验结果

2.1 RR

`n = 16`

```
RRStatistics test
Input n please:
16
Process 4:created 188 dead 288 rutime 25, retime 65, sltime 0
Process 7:created 201 dead 293 rutime 14, retime 75, sltime 0
Process 6:created 199 dead 303 rutime 16, retime 79, sltime 0
Process 9:created 202 dead 303 rutime 14, retime 79, sltime 0
Process 5:created 198 dead 310 rutime 19, retime 88, sltime 0
Process 8:created 202 dead 310 rutime 17, retime 86, sltime 0
Process 12:created 215 dead 319 rutime 15, retime 85, sltime 0
Process 13:created 215 dead 320 rutime 14, retime 86, sltime 0
Process 11:created 209 dead 323 rutime 15, retime 96, sltime 0
Process 10:created 207 dead 327 rutime 19, retime 100, sltime 0
Process 16:created 230 dead 329 rutime 15, retime 82, sltime 0
Process 18:created 240 dead 331 rutime 14, retime 76, sltime 0
Process 14:created 218 dead 333 rutime 18, retime 95, sltime 0
Process 15:created 226 dead 336 rutime 20, retime 89, sltime 0
Process 17:created 230 dead 337 rutime 20, retime 87, sltime 0
Process 19:created 245 dead 344 rutime 20, retime 78, sltime 0

TEST1: 1 parent, 16 children
Total: rutime 275, retime 1346, sltime 0
Average time: 101
```

2.2 PB

`4*n = 16`

```
$ PB
PBStatistics test
Input n please:
4
Process 4:created 377 dead 415 No.1 rutime 24, retime 14, sltime 0
Process 8:created 393 dead 423 No.1 rutime 20, retime 9, sltime 0
Process 12:created 397 dead 432 No.1 rutime 17, retime 18, sltime 0
Process 16:created 400 dead 449 No.1 rutime 25, retime 23, sltime 0
Process 5:created 391 dead 450 No.2 rutime 18, retime 41, sltime 0
Process 13:created 397 dead 463 No.2 rutime 13, retime 53, sltime 0
Process 9:created 394 dead 475 No.2 rutime 26, retime 54, sltime 0
Process 17:created 401 dead 479 No.2 rutime 16, retime 62, sltime 0
Process 6:created 392 dead 494 No.3 rutime 19, retime 82, sltime 0
Process 10:created 395 dead 500 No.3 rutime 21, retime 84, sltime 0
Process 14:created 398 dead 509 No.3 rutime 15, retime 95, sltime 0
Process 18:created 401 dead 523 No.3 rutime 23, retime 99, sltime 0
Process 7:created 392 dead 534 No.4 rutime 24, retime 117, sltime 0
Process 11:created 395 dead 539 No.4 rutime 16, retime 128, sltime 0
Process 19:created 402 dead 556 No.4 rutime 17, retime 137, sltime 0
Process 15:created 399 dead 561 No.4 rutime 27, retime 134, sltime 0

TEST: 1 parent, 16 children
Priority queue 1:
Total: rutime 86, retime 64, sltime 0
Average time: 37
Priority queue 2:
Total: rutime 73, retime 210, sltime 0
Average time: 70
Priority queue 3:
Total: rutime 78, retime 360, sltime 0
Average time: 109
Priority queue 4:
Total: rutime 84, retime 516, sltime 0
Average time: 150
Final average: 91
```

3.实验过程

3.1 TODO 1 阅读代码

阅读`proc.c`中以下函数，分析函数的原理并解释函数的作用：

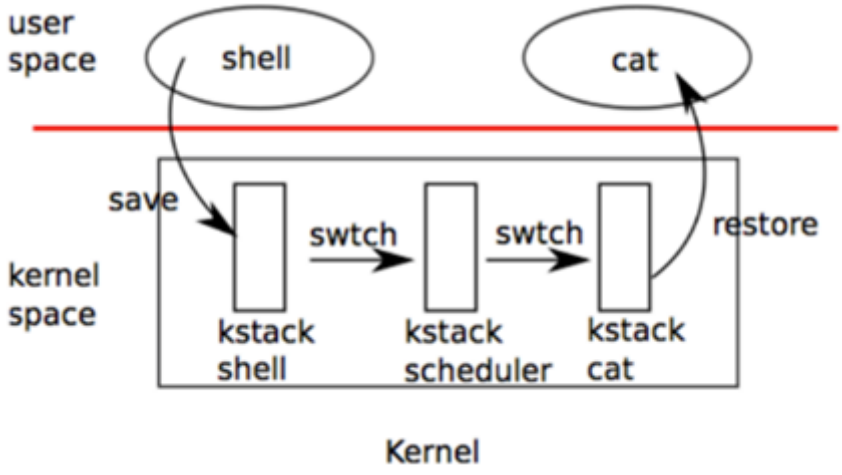
总体解释：`proc.c`中存储的是关于进程的函数

3.1.1 `sched()`

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- **作用：**调度程序切换上下文
- **进入条件：**

    - 只持有进程表锁
    - 当前进程状态非`RUNNING`
    - 不可打断状态
- **实现：**
    1. **检查：**`myproc()`获取当前进程。检查是否具有`ptable`锁、只获取了`ptable`锁（即深度为1）、进程未在运行状态、不能被中断。
    2. **保存：**保存当前`intena`
    3. **切换上下文：**将当前寄存器的值保存在栈中，切换到新栈区后，从栈中读出原来存放的寄存器的值。保存当前进程`p->context`寄存器值，将新的内容，即存放在`cpu`调度程序`scheduler()`中的值切换进来，实现上下文切换。
    4. **恢复`intena`：**需要保存这个值，因为`intena`是内核线程中的参数，而不是当前`CPU`参数，在`scheduler`的内核线程调度器中可能修改。



- **注意：**
    1. `xv6`永远不会直接从一个进程上下文切换为下一个进程，是通过中间的内核线程实现的。*如上图。*
        2. 内核调度器线程`swtch(&p->context,mycpu()->scheduler)`实现了将当前进程切换给内核调度器线程，然后`scheduler()`函数是一个轮转的查找是否有**就绪进程**，并且一直处于运转状态，初始化就绪进程后，在将上下文切换回给`p`。
        3. 进程表的锁总是由旧进程获得，新进程释放，为了保证进程切换正常进行。并且在`scheduler`的外层循环需要打开进程表的锁，为了防止`CPU`闲置占用锁，导致死锁。而且`scheduler()`打开了中断，也是为了防止进程等待IO的时候导致的死锁而无法中断。

3.1.2 `yield()`

实质即`sched()`的复杂版本，在调度`sched()`之前满足了调度程序的两个条件，获得进程表锁且已经修改了当前进程为就绪状态。

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- **作用：**放弃当前时间切片运行的进程

- **进入条件**：时间切片结束，需要切换下一个进程
- **实现**：
    1. 请求进程表锁
    2. 将当前进程状态切换成就绪
    3. 上下文切换
    4. 释放进程表锁

3.1.3 `sleep()`

睡眠操作是当前进程转换为睡眠状态并且调用调度程序释放CPU。需要先获取进程表锁，才能执行睡眠操作，并且由于获取了锁，所以满足了`sched()`的条件，而不是使用yield调度。

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  p->chan = 0;
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

- **作用**：进程由于等待某个条件而让出CPU
- **进入条件**：需要休眠进程
- **实现**：
    1. 检查是否有进程且持有锁；如果锁非进程表锁，申请进程表锁并释放当前锁。
    2. 当前进程休眠，放在休眠链中。
    3. 进程的`chan`值非0即在休眠链`chan`中，若为0则不在。
    4. 切换上下文，切换出来的进程退出休眠链。
    5. 如果之前释放了锁，再次请求锁。

3.1.4 `wakeup()`

```
static void
wakeup1(void *chan)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

```
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

- **作用**：唤醒所有链上的进程
- **实现**：
    1. 获取进程表锁
    2. 遍历进程表，若进程在`chan`上睡眠则切换状态为就绪状态。
    3. 释放进程表锁
- **注意**：实质是比对进程表中每个进程的`chan`值是否为所需`chan`值。

3.1.5 `wait()`

- **作用**：等待子进程退出
- **进入条件**：父进程调用
- **实现**：
    1. 获取进程表锁
    2. 遍历进程表，若有子进程，且子进程处于僵尸状态，则释放子进程所有数据，并且释放进程表锁返回子进程`pid`
    3. 若无子进程则释放锁后返回-1，且将当前进程睡眠。

3.2 TODO 2 统计RR调度情况

RR调度是分时系统时间切片调度，每个时间切片分配给一个进程后，按照轮转法，调度程序会调度下一个进程。

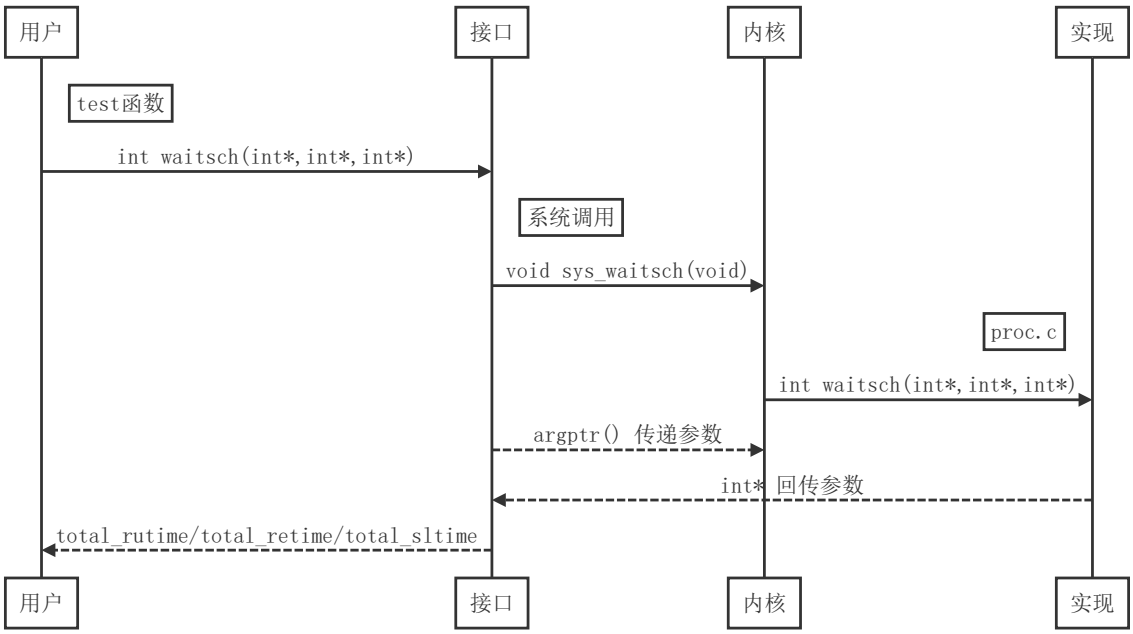
设计思路：

- 使用`ticks`时间切片为时间单位计算进程各状态时间
- 在每次时间中断时遍历进程表更新各进程状态时间
- 子进程死亡时输出子进程各状态信息
- 定义`waitsch()`系统调用和用户接口，传出参数为各状态总时间

3.2.1 添加系统调用`waitsch()`

实现思路：

- `struct proc`结构体改写
- 系统调用的添加
- 流程图如下



3.2.1.1 `struct proc`结构体改写

需要记录创建、运行、就绪、休眠时间，故每个进程都应该有此值。

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int ctime;                  // create time
  int rutime;                // running time
  int retime;                  // runnable time
  int sltime;                  // sleep time
  int priority;                // priority
};
```

3.2.1.2 系统调用的添加

系统调用添加需要定义系统调用的函数并封装，定义用户接口。

添加系统调用在`Lab2`中获取父进程`pid`的函数`get_parentid()`中实现过，需要的操作如下：

```
1. syscall.h:   definition
    #define SYS_waitsch 23
2. syscall.c:   declaration
    extern int sys_waitsch(void);
    static int (*syscalls[])(void) = {
```

```
        ......
        [SYS_waitsch] sys_waitsch,
    };
3. sysproc.c:   implementation
    int sys_waitsch(void)
    {
        int* total_rutime,* total_retime,* total_sltime;
        argptr(0, (void*)&total_rutime, 4);
        argptr(1, (void*)&total_retime, 4);
        argptr(2, (void*)&total_sltime, 4);
        return waitsch(total_rutime, total_retime, total_sltime);
    }
```

前面三个步骤是定义宏、声明、并且写好系统调用函数的接口。具体`sys_waitsch()`的实现，并不需要放在`sysproc.c`当中，因为函数体比较复杂。由于`waitsch()`的用户程序会传入三个`int*`参数，所以需要在`sys_waitsch()`中先获取到这三个参数，根据`xv6`手册，可知道获取指针参数的方法为：

```
int argptr(int n, char **pp, int size)
```

其中`n`是第几个参数，`pp`是需要写入参数的变量，`size`是参数的大小。获得参数后，就可以将具体实现转交给`proc.c`中的`int waitsch(int *rutime, int *retime, int *sltime)`。

初始化，在`allocproc()`中，因为每次新建一个进程的时候都会在这个函数中初始化。

```
4. proc.c:      real implementation
    int waitsch(int *rutime, int *retime, int *sltime)
    {
      struct proc *p;
      int havekids, pid;
      struct proc *curproc = myproc();
      acquire(&ptable.lock);
      for(;;){
        // Scan through table looking for exited children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          if(p->parent != curproc)
            continue;
          havekids = 1;
          if(p->state == ZOMBIE){
            // Found one.
            pid = p->pid;
            kfree(p->kstack);
            p->kstack = 0;
            freevm(p->pgdir);
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            p->state = UNUSED;
            // counting time for total process
            *rutime += p->rutime;
            *retime += p->retime;
            *sltime += p->sltime;
            cprintf("Process %d:",pid);
            cprintf("created %d ",p->ctime);
            acquire(&tickslock);
```

```
            int xtick = ticks;
            release(&tickslock);
            cprintf("dead %d ",xtick);
            #if (USE_RR!=1)
            cprintf("No.%d ",p->priority);
            #endif
            cprintf("rutime %d, retime %d, sltime %d\n",p->rutime,p->retime,p-
>sltime);
            release(&ptable.lock);
            return pid;
          }
        }
        // No point waiting if we don't have any children.
        if(!havekids || curproc->killed){
          release(&ptable.lock);
          return -1;
        }
        // Wait for children to exit.  (See wakeup1 call in proc_exit.)
        sleep(curproc, &ptable.lock);  //DOC: wait-sleep
      }
    }

    static struct proc *allocproc(void){
      //add this
      acquire(&tickslock);
      int xticks = ticks;
      release(&tickslock);
      p->ctime = xticks;
      p->rutime = p->retime = p->sltime = 0;
      p->priority = 0;
    }
```

`int waitsch(int *rutime, int *retime, int *sltime)`的函数实现即`wait()`的变体。修改如下：

- 每次检测到子进程退出为`ZOMBIE`状态时，都需要在`total`时间中加上该子进程的各时间，实现统计。
- 此时还要输出该子进程退出时的各状态。
- 死亡时间`dead time`就是当前时间切片`ticks`，注意要获得锁。
- 输出样例：`Process 3: created 32 dead 56 rutime 13 retime 11 sltime 0`

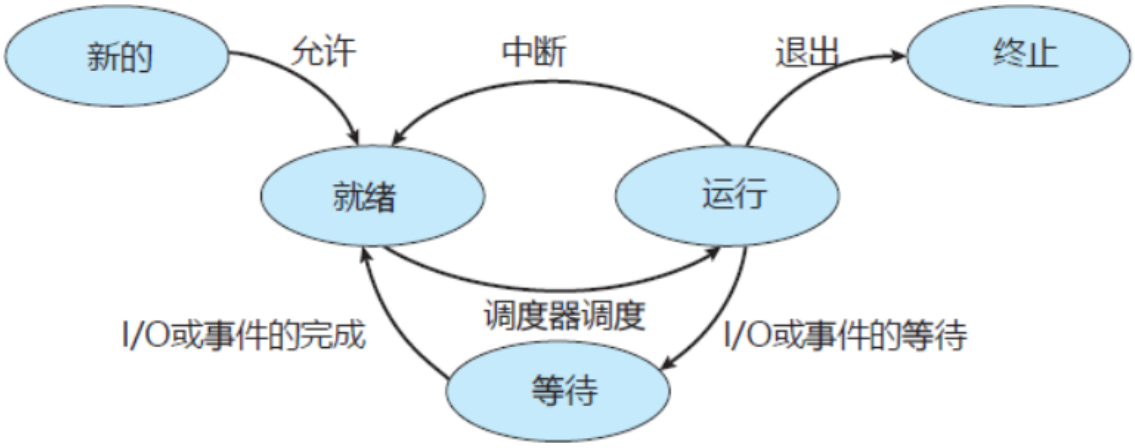
```
5. user.h
    int waitsch(int*, int*, int*);
6. usys.S
    SYSCALL(waitsch)
```

上面步骤定义了用户接口，用户应用程序使用`waitsch()`可以直接调用系统调用。

3.2.2 更新状态时间

阅读`trap.c`的`trap()`函数之后，结合`xv6`文档，可知系统有处理**系统调用、异常、中断**的机制——`trap()`

`trap()`是中断处理程序，在处理程序建立中断帧后，会立刻调用`trap()`。



> **xv 6文档中的trap()**
>
> `trap` 查看硬件中断号 `tf->trapno` 来判断自己为什么被调用以及应该做些什么。如果中断是 `T_SYSCALL`，`trap` 调用系统调用处理程序 `syscall` 。当检查完是否是系统调用，`trap` 会继续检查是否是硬件中断（我们会在下面讨论）。中断可能来自硬件设备的正常中断，也可能来自异常的、未预料到的硬件中断。
>
> 如果中断不是一个系统调用也不是一个硬件主动引发的中断，`trap` 就认为它是一个发生中断前的一段代码中的错误行为导致的中断（如除零错误）。如果产生中断的代码来自用户程序，`xv6` 就打印错误细节并且设置 `cp->killed` 使之待会被清除掉。
>
> 如果是内核程序正在执行，那就出现了一个内核错误：`trap` 打印错误细节并且调用 `panic`。

我们只需要关心两种情况对进程状态的影响：

- 系统调用：此时 `trap()` 会调用 `syscall()` 并根据 `myproc()->tf->eax` 的值判断调用的哪一个系统调用函数。`sys_call` 中可能调用的函数会修改当前进程的状态，例如 `exit()` 会将进程从 `RUNNING` 切换为 `ZOMBIE`。但综合考虑，如果每次调用系统调用的时候，或者调用某个特定的改变状态的函数的时候就计算进程的状态时间，会需要修改很多代码段，非常不方便；并且可知大规模运算本身需要的时间切片很多，所以不需要在系统调用的时候考虑状态切换。
- 时间中断：`trap()` 用 `T_IRQ0 + IRQ_TIMER` 表示一个时间片结束进入的陷入，此时必须统计此时的进程状态。又因为**进入时间中断的当前进程一定是 `RUNNING` 状态**，所以不能只针对当前进程累计状态时间，否则会只记录进程 `rutime`。

综上，进程只会在 `RUNNING` 进入时钟中断；近似估计每个时间切片，进程状态不变。

`trap()` 修改如下，`timecounting()` 定义在 `proc.c()` 中，且在 `defs.h` 声明。

```
1. trap.c
    void trap(struct trapframe *tf)
    {
      ...
      switch(tf->trapno){
      case T_IRQ0 + IRQ_TIMER:
      // try counting the time everytime trap into interrupt
        timecounting();
        if(cpuid() == 0){
          acquire(&tickslock);
          ticks++;
          wakeup(&ticks);     //wake up all the process on chan no.ticks
          release(&tickslock);
        }
        lapiceoi();
```

```
        break;
      ...
    }
2. proc.c
    // count time for process everytime interrupted
    void timecounting(void)
    {
      acquire(&tickslock);
      int xtick = ticks;
      release(&tickslock);
      acquire(&ptable.lock);
      struct proc* p;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        int living = xtick - p->ctime;
        int present = living - p->retime - p->rutime - p->sltime;
        switch(p->state){
          case RUNNABLE: p->retime += present;break;
          case RUNNING:  p->rutime += present;break;
          case SLEEPING: p->sltime += present;break;
          default:break;
        }
      }
      release(&ptable.lock);
      return;
    }
3. defs.h
    void            timecounting(void);
```

`timecounting()`函数实现细节

- 获取当前时间`ticks`
- 遍历`ptable`，以现在进程状态为准更新进程状态时间，对应状态`+ used_time`
- 由于进程创建后只可能是三个状态，所以有以下等式：

  `living_time = current_time - createtime`

  `living_time = retime + rutime + sltime + used_time`

3.2.3 测试程序

编写`RRStatistics.c`程序测试`RR`调度的算法，根据以上算法可以得到总时间并且输出子进程的各状态时间。

3.2.3.1 大规模运算

一开始测试的时候设置了循环多次的乘法运算发现每次时间切片一个都用不到就执行完了，后来找到了矩阵乘法的方法，并且定义全局变量会比局部变量计算量大得多，时间切片刚好合适用于观测。

```
// Test RR
#include "types.h"
#include "stat.h"
#include "user.h"

#define N 1000
#define M 5
#define MAX_TRSIZE 300
```

```
int a[MAX_TRSIZE][MAX_TRSIZE];
int b[MAX_TRSIZE][MAX_TRSIZE];
int c[MAX_TRSIZE][MAX_TRSIZE];

void matrixmul(void){
    int i,j,k;
    int sum = 0;
    for(i = 0; i < MAX_TRSIZE; i++)
    {
        for(j = 0; j < MAX_TRSIZE; j++)
        {
            a[i][j]=  i*j;
            b[i][j]=  i+j;
            c[i][j]= 0;
        }
    }
    for(i = 0; i < MAX_TRSIZE; i++)
    {
        for(j = 0; j < MAX_TRSIZE; j++)
        {
            for(k = 0; k < MAX_TRSIZE; k ++)
            {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
```

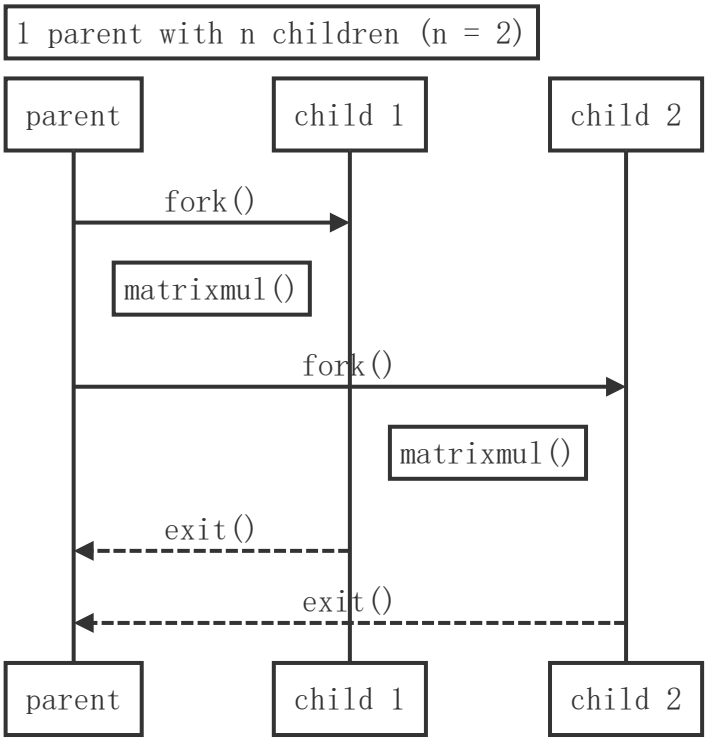
3.2.3.2 父进程创建子进程

题目要求父进程创建n个子进程，然后轮转计算大规模运算，我写了两种思路的创建方法。

`test1()`：**顺序创建**

如图所示，`test1`符合常理，1个父进程，创建n个子进程。由于`fork()`函数，父进程会创建子进程，子进程执行`matrixmul`后`exit`，父进程继续创建子进程，然后陷入n次的`waitsch()`，每次子进程死亡就会输出相关状态信息。**`RR`会在`n+1`个进程中轮转。**

这种创建的特点是没有`sltime`，所有进程只有有`RUNNING`和`RUNNABLE`两种状态。




```
//only one parent process,n children
int test1(int n)
{
  int i;
  int rutime,retime,sltime;
  rutime = retime = sltime = 0;
  for(i=0;i<n;i++){
      int pid = fork();
      if(pid < 0)
        break;
      if(pid == 0){
      matrixmul();
      exit();
      }
  }
  for(;i>0;i--){
    waitsch(&rutime,&retime,&sltime);
  }
  printf(1,"\nTEST1: 1 parent, %d children\n",n);
  return output(rutime,retime,sltime,n);
}
```

`test2`：金字塔创建

如图所示，`test2()`每个进程嵌套创建下一个进程，n个父进程和1个子进程。这样实行就可以使得金字塔底端的父进程在`waitsch()`进入休眠，所以拥有了`sltime`，其实根据实验结果和图示可知`sltime`也是金字塔式的。

`test1()`观察到，没有`sltime`，是因为所有进程不是在运行，就是在就绪队列中等待轮转，除了最初的创建的那个父进程外其他进程都不可能进入`sleep`，因为只有当有子进程未运行结束的时候，才会进入`sleep`。所以建立这样一个金字塔式嵌套生成的`test2`，原因如下：

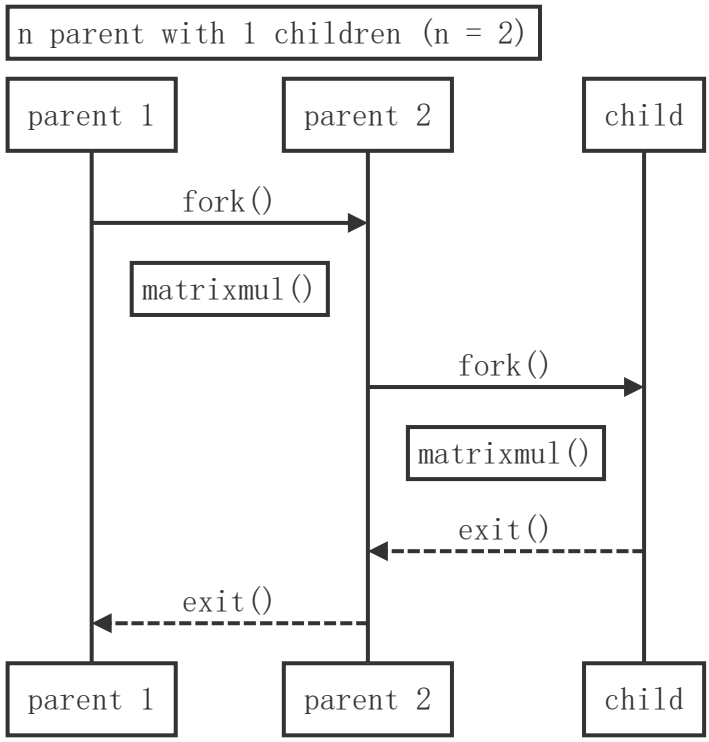
- 测试`sleep_time`计算正常

- 验证进入`sleep`状态的猜想正确

并且可知这种创建（下图）的特点是：

- 每个（中间）进程既是父进程又是子进程，一共n个父进程，1个子进程
- **就绪队列一直只有一个进程**
- 没有轮转时间

无轮转是因为每次就绪队列只有一个进程，相当于顺序执行，按照最里层的子进程不停执行。所以后面分析`test2`的平均轮转时间无意义，只需要看它的`sltime`特点。




```
//n parent
void test2(int n)
{
  int i;
  int rutime,retime,sltime;
  rutime = retime = sltime = 0;
  for(i=0;i<n;){
      int pid = fork();
      if(pid < 0)
        break;
      if(pid == 0){
        i++;
        matrixmul();
        if(i==N) exit();
        continue;
      }
      else{
        waitsch(&rutime,&retime,&sltime);
        if(i==0)
        {
            printf(1,"\nTEST2: %d parent\n",n);
            output(rutime,retime,sltime,n);
        }
```

```
        i--;
      }
      exit();
  }
}
```

3.2.3.3 其他代码

```
char buf[100];
int input(int n)
{
  int i,m=0;
  for(i=0;i<n-1;i++)
    m = buf[i]-'0'+10*m;
  return m;
}

int output(int rutime,int retime,int sltime,int n)
{
  printf(1,"Total: rutime %d, retime %d, sltime %d\n",rutime,retime,sltime);
  int average = (rutime+retime+sltime)/n;
  printf(1,"Average time: %d \n",average);
  return average;
}

void
RRStatistics()
{
  printf(1,"RRStatistics test\n");
  printf(1,"Input n please:\n");
  int n = read(0, buf, sizeof(buf));
  n = input(n);
  test1(n);
}

int
main(void)
{
  RRStatistics();
  exit();
}
```

在`Makefile`中添加用户程序`RR`，过程比较简单不予赘述。

```
UPROGS=\
    ...
    _RR\
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c RR.c PB.c zombie.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

3.2.4 实验结果

`test1` n = 10

```
$ RR
RRStatistics test
Input n please:
10
Process 4:created 463 dead 585 rutime 44, retime 75, sltime 0
Process 5:created 483 dead 591 rutime 25, retime 76, sltime 0
Process 6:created 484 dead 593 rutime 24, retime 83, sltime 0
Process 8:created 489 dead 600 rutime 24, retime 86, sltime 0
Process 9:created 492 dead 602 rutime 23, retime 85, sltime 0
Process 7:created 487 dead 605 rutime 28, retime 89, sltime 0
Process 10:created 498 dead 609 rutime 25, retime 85, sltime 0
Process 11:created 502 dead 613 rutime 27, retime 84, sltime 0
Process 12:created 506 dead 613 rutime 25, retime 82, sltime 0
Process 13:created 515 dead 617 rutime 25, retime 76, sltime 0

TEST1: 1 parent, 10 children
Total: rutime 270, retime 821, sltime 0
Average time: 109
```

`test1` n = 20

```
RRStatistics test
Input n please:
20
Process 4:created 253 dead 435 rutime 49, retime 88, sltime 0
Process 5:created 277 dead 443 rutime 27, retime 98, sltime 0
Process 6:created 278 dead 443 rutime 18, retime 62, sltime 0
Process 7:created 281 dead 443 rutime 21, retime 82, sltime 0
Process 8:created 281 dead 450 rutime 21, retime 82, sltime 0
Process 9:created 286 dead 450 rutime 25, retime 110, sltime 0
Process 11:created 307 dead 464 rutime 24, retime 127, sltime 0
Process 10:created 290 dead 471 rutime 27, retime 147, sltime 0
Process 12:created 312 dead 471 rutime 23, retime 129, sltime 0
Process 14:created 331 dead 482 rutime 21, retime 124, sltime 0
Process 13:created 321 dead 492 rutime 25, retime 139, sltime 0
Process 15:created 342 dead 512 rutime 27, retime 140, sltime 0
Process 16:created 349 dead 518 rutime 27, retime 136, sltime 0
Process 17:created 355 dead 527 rutime 27, retime 142, sltime 0
Process 18:created 381 dead 533 rutime 28, retime 121, sltime 0
Process 23:created 428 dead 533 rutime 20, retime 85, sltime 0
Process 22:created 420 dead 540 rutime 24, retime 95, sltime 0
Process 20:created 402 dead 541 rutime 27, retime 112, sltime 0
Process 21:created 414 dead 541 rutime 26, retime 101, sltime 0
Process 19:created 390 dead 544 rutime 32, retime 121, sltime 0

TEST1: 1 parent, 20 children
Total: rutime 519, retime 2241, sltime 0
Average time: 138
```

`test1` n = 30

```
RRStatistics test
Input n please:
30
Process 4:created 176 dead 393 rutime 36, retime 65, sltime 0
Process 5:created 195 dead 409 rutime 18, retime 64, sltime 0
Process 6:created 196 dead 409 rutime 20, retime 93, sltime 0
Process 7:created 196 dead 409 rutime 16, retime 66, sltime 0
Process 8:created 199 dead 409 rutime 16, retime 90, sltime 0
Process 9:created 206 dead 410 rutime 15, retime 74, sltime 0
Process 10:created 206 dead 418 rutime 16, retime 102, sltime 0
Process 11:created 213 dead 418 rutime 13, retime 84, sltime 0
Process 12:created 219 dead 418 rutime 21, retime 136, sltime 0
Process 13:created 220 dead 418 rutime 19, retime 119, sltime 0
Process 14:created 229 dead 420 rutime 20, retime 142, sltime 0
Process 15:created 232 dead 420 rutime 17, retime 125, sltime 0
Process 19:created 271 dead 420 rutime 16, retime 126, sltime 0
Process 20:created 276 dead 420 rutime 15, retime 107, sltime 0
Process 21:created 288 dead 432 rutime 13, retime 95, sltime 0
Process 18:created 259 dead 439 rutime 21, retime 153, sltime 0
Process 23:created 304 dead 439 rutime 15, retime 114, sltime 0
Process 25:created 308 dead 445 rutime 13, retime 119, sltime 0
Process 16:created 241 dead 445 rutime 27, retime 176, sltime 0
Process 26:created 323 dead 451 rutime 15, retime 110, sltime 0
Process 22:created 288 dead 456 rutime 20, retime 143, sltime 0
Process 24:created 308 dead 460 rutime 21, retime 127, sltime 0
Process 17:created 245 dead 460 rutime 25, retime 189, sltime 0
Process 29:created 356 dead 470 rutime 15, retime 96, sltime 0
Process 27:created 337 dead 474 rutime 21, retime 115, sltime 0
Process 30:created 372 dead 476 rutime 17, retime 85, sltime 0
Process 32:created 375 dead 479 rutime 15, retime 88, sltime 0
Process 28:created 339 dead 481 rutime 21, retime 119, sltime 0
Process 31:created 372 dead 481 rutime 18, retime 90, sltime 0
Process 33:created 393 dead 493 rutime 23, retime 77, sltime 0

TEST1: 1 parent, 30 children
Total: rutime 558, retime 3289, sltime 0
Average time: 128
```

`test1` n = 40

```
TEST1: 1 parent, 40 children
Total: rutime 1015, retime 6261, sltime 0
Average time: 181
```

`test2`可以看出`sltime`是金字塔式的。

`test2` n = 10

```
RRStatistics test
Input n please:
10
Process 13:created 691 dead 748 rutime 15, retime 0, sltime 0
Process 12:created 659 dead 748 rutime 55, retime 34, sltime 0
Process 11:created 635 dead 748 rutime 42, retime 8, sltime 63
Process 10:created 614 dead 748 rutime 52, retime 6, sltime 76
Process 9:created 575 dead 748 rutime 53, retime 13, sltime 107
Process 8:created 551 dead 749 rutime 37, retime 6, sltime 155
Process 7:created 533 dead 749 rutime 46, retime 5, sltime 165
Process 6:created 504 dead 749 rutime 46, retime 9, sltime 190
Process 5:created 480 dead 749 rutime 44, retime 3, sltime 222
Process 4:created 426 dead 749 rutime 54, retime 33, sltime 236

TEST2: 10 parent
```

`test2` n = 20

```
RRStatistics test
Input n please:
20
Process 23:created 813 dead 847 rutime 28, retime 5, sltime 0
Process 22:created 784 dead 847 rutime 40, retime 3, sltime 19
Process 21:created 729 dead 848 rutime 66, retime 13, sltime 39
Process 20:created 692 dead 848 rutime 53, retime 9, sltime 93
Process 19:created 663 dead 848 rutime 50, retime 4, sltime 130
Process 18:created 643 dead 849 rutime 47, retime 2, sltime 156
Process 17:created 609 dead 849 rutime 54, retime 8, sltime 177
Process 16:created 582 dead 849 rutime 40, retime 3, sltime 223
Process 15:created 550 dead 850 rutime 52, retime 10, sltime 237
Process 14:created 530 dead 851 rutime 46, retime 11, sltime 263
Process 13:created 505 dead 851 rutime 57, retime 13, sltime 275
Process 12:created 458 dead 851 rutime 61, retime 16, sltime 315
Process 11:created 420 dead 852 rutime 46, retime 7, sltime 378
Process 10:created 389 dead 852 rutime 51, retime 2, sltime 409
Process 9:created 355 dead 852 rutime 52, retime 9, sltime 435
Process 8:created 332 dead 853 rutime 49, retime 5, sltime 466
Process 7:created 296 dead 853 rutime 62, retime 6, sltime 488
Process 6:created 266 dead 853 rutime 52, retime 7, sltime 527
Process 5:created 236 dead 854 rutime 54, retime 2, sltime 561
Process 4:created 193 dead 854 rutime 56, retime 2, sltime 602

TEST2: 20 parent
```

`test2` n = 30

```
RRStatistics test
Input n please:
30
Process 33:created 822 dead 851 rutime 20, retime 9, sltime 0
Process 32:created 805 dead 851 rutime 33, retime 5, sltime 8
Process 31:created 778 dead 851 rutime 43, retime 14, sltime 16
Process 30:created 763 dead 851 rutime 30, retime 1, sltime 57
Process 29:created 748 dead 851 rutime 31, retime 3, sltime 69
Process 28:created 725 dead 851 rutime 33, retime 6, sltime 87
Process 27:created 711 dead 851 rutime 30, retime 1, sltime 109
Process 26:created 683 dead 851 rutime 40, retime 7, sltime 121
Process 25:created 656 dead 851 rutime 37, retime 4, sltime 154
Process 24:created 628 dead 851 rutime 35, retime 10, sltime 178
Process 23:created 611 dead 851 rutime 32, retime 2, sltime 206
Process 22:created 588 dead 851 rutime 44, retime 4, sltime 215
Process 21:created 575 dead 851 rutime 33, retime 1, sltime 242
Process 20:created 548 dead 851 rutime 36, retime 6, sltime 261
Process 19:created 531 dead 851 rutime 29, retime 2, sltime 289
Process 18:created 498 dead 851 rutime 37, retime 16, sltime 300
Process 17:created 480 dead 851 rutime 32, retime 1, sltime 338
Process 16:created 454 dead 851 rutime 42, retime 8, sltime 347
Process 15:created 435 dead 851 rutime 33, retime 2, sltime 381
Process 14:created 417 dead 851 rutime 34, retime 3, sltime 397
Process 13:created 382 dead 851 rutime 43, retime 9, sltime 417
Process 12:created 360 dead 851 rutime 34, retime 2, sltime 455
Process 11:created 343 dead 851 rutime 35, retime 2, sltime 471
Process 10:created 317 dead 851 rutime 33, retime 8, sltime 493
Process 9:created 304 dead 851 rutime 27, retime 3, sltime 517
Process 8:created 283 dead 851 rutime 34, retime 9, sltime 525
Process 7:created 263 dead 854 rutime 37, retime 10, sltime 544
Process 6:created 247 dead 854 rutime 29, retime 2, sltime 576
Process 5:created 231 dead 854 rutime 37, retime 2, sltime 584
Process 4:created 193 dead 854 rutime 50, retime 4, sltime 607

TEST2: 30 parent
```

3.3 TODO 3 统计PB调度情况

Priority based算法的核心是非抢占式优先级，对于相同优先级的就绪队列采用`FCFS`。在实现了RR的基础上实现PB算法简单很多。

设计思路：

- 每个时间切片更新进程状态时间
- `scheduler`判断优先级进行调度，`trap`中断时不需要`yield`
- `waitsch`传入参数由`int*`改成`int[]`（实质一样）
- 使用`#if #else #endif`的方法更新源代码

3.3.1 添加系统调用`setpriority()`

实现思路同 waitsch()同样，不予赘述。流程图也同理。

`struct proc`结构体在上面已经添加了`priority`，并且在`fork()`时，子进程继承父进程`priority`

```
1. proc.c
    int setpriority(int pid, int priority)
    {
      acquire(&ptable.lock);
      struct proc *p;
      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if (p->pid == pid)
        {
          p->priority = priority;
          release(&ptable.lock);
          return 0;
        }
      //fail to find pid
      release(&ptable.lock);
      return -1;
    }
    fork(){
        //add this
        np->priority = curproc->priority;
    }
2. sysproc.c
    int sys_setpriority(void)
    {
      int pid, priority;
      argint(0, &pid);
      argint(1, &priority);
      return setpriority(pid, priority);
    }
```

3.3.2 更新`scheduler/trap/waitsch`

由于调度算法更新为PB，所以需要修改三个地方。为了表示每次使用的调度算法，在`defs.h`中定义：

```
// use of RR/Priority
#define USE_RR 2
#define PB 4
```

3.3.2.1 scheduler更新

需要变化的地方是寻找下一个`RUNNABLE`进程的方法。原来的方法是不停的循环，只要找到一个就绪状态的进程就可以切换上下文。优先级队列就要求：

- 下一个就绪进程优先级最高
- 相同优先级进程要求最早到达

为了满足两个要求，则需要遍历一遍`ptable`，找到优先级最高且`ctime`最小的进程来切换上下文。

- 注意设置的优先级：数字越小，优先级越大。

```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;)
```

```
  {
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
// RR
#if (USE_RR == 1)
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
#else
    // find the most earliest process with high priority
    int least_pri = PB + 1;
    int ctime = 1000000000;
    struct proc *least_p = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE && p->priority < least_pri)
      {
        least_pri = p->priority;
        ctime = p->ctime;
        least_p = p;
      }
      else if (p->state == RUNNABLE && p->priority == least_pri && p->ctime <
ctime)
      {
        ctime = p->ctime;
        least_p = p;
      }
    }
    if (least_p)
    {
      p = least_p;
      // cprintf("Now: process %d with No.%d\n",p->pid,p->priority);
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      // Process is done running for now.
```

```
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
#endif
    release(&ptable.lock);
  }
}
```

3.3.2.2 trap更新

之前调用`trap`中断的时候需要`yield`，现在不需要这个强制调度调度程序了，只需要每次执行完进程后，`scheduler`自动切换到下一个优先级最高的进程。但是`trap`还是必要的，因为需要每个时间切片计算状态时间个数。

```
void trap(struct trapframe *tf)
{
    //change this
  #if (USE_RR==1)
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  #endif
}
```

3.3.2.3 waitsch更新

由于之前传入的参数是只指向一个变量的`int*`，此时改成指向`PB`个`int`的`int*`，即数组名。

所以需要修改传参过程。

```
1. sysproc.c
    int sys_waitsch(void)
    {
      int *total_rutime, *total_retime, *total_sltime;
    #if (USE_RR == 1)
      argptr(0, (void *)&total_rutime, 4);
      argptr(1, (void *)&total_retime, 4);
      argptr(2, (void *)&total_sltime, 4);
    #else
      argptr(0, (void *)&total_rutime, 4 * PB);
      argptr(1, (void *)&total_retime, 4 * PB);
      argptr(2, (void *)&total_sltime, 4 * PB);
    #endif
      return waitsch(total_rutime, total_retime, total_sltime);
    }
2. proc.c
    int waitsch()
    {
    //add this
#if (USE_RR == 1)
        // counting time for total process
        *rutime += p->rutime;
        *retime += p->retime;
        *sltime += p->sltime;
#else
        int q = p->priority;
        rutime[q - 1] += p->rutime;
```

```
        retime[q - 1] += p->retime;
        sltime[q - 1] += p->sltime;
#endif
#if (USE_RR != 1)
        cprintf("No.%d ", p->priority);
#endif
    }
```

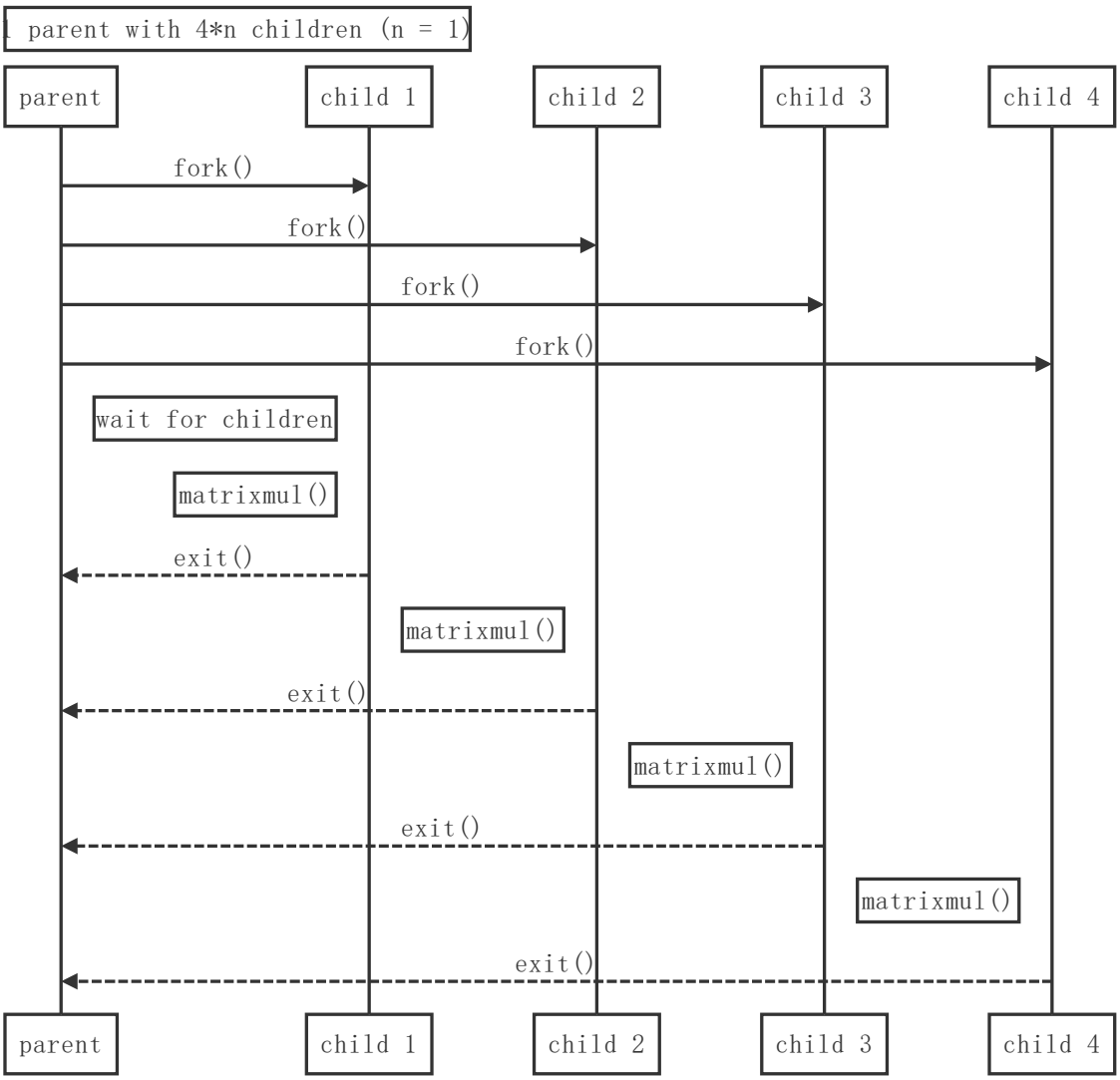
3.3.3 测试程序

编写`PBStatistics.c`的过程基本和`RR`一样。

`test`：优先队列

父进程一次性创建完所有子进程（`4n`个），然后按照优先队列1,2,3,4的顺序一次执行。

- 父进程一次性创建
- 优先队列中的队列是`FCFS`




```
void test(int n)
{
    int i;
    int rutime[4], retime[4], sltime[4];
    for (i = 0; i < n; i++)
    {
```

```
        int pid = fork();
        if (pid < 0)
            break;
        else if (pid == 0)
        {
            matrixmul();
            exit();
        }
        else
            setpriority(pid, pid % 4 + 1);
    }
    for (; i > 0; i--)
    {
        waitsch(rutime, retime, sltime);
    }
    printf(1, "\nTEST: 1 parent, %d children\n", n);
    int m, sum;
    for (m = sum = 0; m < 4; m++)
    {
        printf(1, "Priority queue %d:\n", m + 1);
        sum += output(rutime[m], retime[m], sltime[m], n / 4);
    }
    printf(1, "Final average: %d\n", sum / n);
}
```

3.3.4 实验结果

`test n = 2`

```
PBStatistics test
Input n please:
2
Process 4:created 233 dead 261 No.1 rutime 13, retime 11, sltime 0
Process 8:created 258 dead 275 No.1 rutime 13, retime 3, sltime 0
Process 5:created 244 dead 291 No.2 rutime 34, retime 13, sltime 0
Process 9:created 259 dead 299 No.2 rutime 24, retime 15, sltime 0
Process 6:created 248 dead 311 No.3 rutime 20, retime 43, sltime 0
Process 10:created 260 dead 327 No.3 rutime 28, retime 38, sltime 0
Process 7:created 256 dead 344 No.4 rutime 33, retime 55, sltime 0
Process 11:created 261 dead 356 No.4 rutime 28, retime 66, sltime 0

TEST: 1 parent, 8 children
Priority queue 1:
Total: rutime 26, retime 14, sltime 0
Average time: 20
Priority queue 2:
Total: rutime 58, retime 28, sltime 0
Average time: 43
Priority queue 3:
Total: rutime 48, retime 81, sltime 0
Average time: 64
Priority queue 4:
Total: rutime 61, retime 121, sltime 0
Average time: 91
Final average: 54
```

`test n = 4`

```
$ PB
PBStatistics test
Input n please:
4
Process 4:created 377 dead 415 No.1 rutime 24, retime 14, sltime 0
Process 8:created 393 dead 423 No.1 rutime 20, retime 9, sltime 0
Process 12:created 397 dead 432 No.1 rutime 17, retime 18, sltime 0
Process 16:created 400 dead 449 No.1 rutime 25, retime 23, sltime 0
Process 5:created 391 dead 450 No.2 rutime 18, retime 41, sltime 0
Process 13:created 397 dead 463 No.2 rutime 13, retime 53, sltime 0
Process 9:created 394 dead 475 No.2 rutime 26, retime 54, sltime 0
Process 17:created 401 dead 479 No.2 rutime 16, retime 62, sltime 0
Process 6:created 392 dead 494 No.3 rutime 19, retime 82, sltime 0
Process 10:created 395 dead 500 No.3 rutime 21, retime 84, sltime 0
Process 14:created 398 dead 509 No.3 rutime 15, retime 95, sltime 0
Process 18:created 401 dead 523 No.3 rutime 23, retime 99, sltime 0
Process 7:created 392 dead 534 No.4 rutime 24, retime 117, sltime 0
Process 11:created 395 dead 539 No.4 rutime 16, retime 128, sltime 0
Process 19:created 402 dead 556 No.4 rutime 17, retime 137, sltime 0
Process 15:created 399 dead 561 No.4 rutime 27, retime 134, sltime 0

TEST: 1 parent, 16 children
Priority queue 1:
Total: rutime 86, retime 64, sltime 0
Average time: 37
Priority queue 2:
Total: rutime 73, retime 210, sltime 0
Average time: 70
Priority queue 3:
Total: rutime 78, retime 360, sltime 0
Average time: 109
Priority queue 4:
Total: rutime 84, retime 516, sltime 0
Average time: 150
Final average: 91
```

`test n = 5`

```
PBStatistics test
Input n please:
5
Process 4:created 181 dead 224 No.1 rutime 13, retime 11, sltime 0
Process 8:created 194 dead 224 No.1 rutime 19, retime 11, sltime 0
Process 12:created 212 dead 237 No.1 rutime 13, retime 11, sltime 0
Process 20:created 218 dead 252 No.1 rutime 14, retime 19, sltime 0
Process 16:created 215 dead 258 No.1 rutime 34, retime 9, sltime 0
Process 5:created 192 dead 268 No.2 rutime 15, retime 60, sltime 0
Process 9:created 201 dead 281 No.2 rutime 23, retime 57, sltime 0
Process 13:created 213 dead 290 No.2 rutime 22, retime 54, sltime 0
Process 21:created 220 dead 305 No.2 rutime 15, retime 69, sltime 0
Process 17:created 216 dead 311 No.2 rutime 30, retime 65, sltime 0
Process 6:created 193 dead 325 No.3 rutime 19, retime 112, sltime 0
Process 14:created 214 dead 340 No.3 rutime 14, retime 111, sltime 0
Process 10:created 211 dead 341 No.3 rutime 30, retime 100, sltime 0
Process 18:created 217 dead 355 No.3 rutime 14, retime 123, sltime 0
Process 22:created 222 dead 364 No.3 rutime 23, retime 119, sltime 0
Process 11:created 211 dead 379 No.4 rutime 15, retime 153, sltime 0
Process 7:created 193 dead 380 No.4 rutime 25, retime 161, sltime 0
Process 19:created 218 dead 397 No.4 rutime 16, retime 162, sltime 0
Process 15:created 214 dead 410 No.4 rutime 31, retime 165, sltime 0
Process 23:created 222 dead 416 No.4 rutime 18, retime 175, sltime 0

TEST: 1 parent, 20 children
Priority queue 1:
Total: rutime 93, retime 61, sltime 0
Average time: 30
Priority queue 2:
Total: rutime 105, retime 305, sltime 0
Average time: 82
Priority queue 3:
Total: rutime 100, retime 565, sltime 0
Average time: 133
Priority queue 4:
Total: rutime 105, retime 816, sltime 0
Average time: 184
Final average: 107
```

`test n = 6`

```
TEST: 1 parent, 24 children
Priority queue 1:
Total: rutime 124, retime 114, sltime 0
Average time: 39
Priority queue 2:
Total: rutime 101, retime 441, sltime 0
Average time: 90
Priority queue 3:
Total: rutime 111, retime 773, sltime 0
Average time: 147
Priority queue 4:
Total: rutime 112, retime 1093, sltime 0
Average time: 200
Final average: 119
```

`test n = 10`

```
TEST: 1 parent, 40 children
Priority queue 1:
Total: rutime 242, retime 485, sltime 0
Average time: 72
Priority queue 2:
Total: rutime 229, retime 1660, sltime 0
Average time: 188
Priority queue 3:
Total: rutime 231, retime 2797, sltime 0
Average time: 302
Priority queue 4:
Total: rutime 251, retime 3957, sltime 0
Average time: 420
Final average: 246
```

4.实验解析

4.1 输出解释

算法	n	平均周转时间
RR - `test1`	10	109
	20	138
	30	128
	40	181
RR - `test2`	10	-
	20	-
	30	-
PB - `test`	2	54
	4	91
	5	107
	6	119
	10	246

4.1.1 RR

可以发现`test2`的方法中会有`sltime`，而`test1`没有，因为`test1`所有进程都会被轮转，而`test2`中处于底层的父进程会因为子进程没有返回而`sleep`等待子进程返回。

当然从流程图就可以知道`test1`与`test2`的区别。可以验证之前的猜想。

我还统计了`test2`就绪队列的大小变化，发现确实恒定不变，保持只有一个就绪程序。

```
1 processes runnable
1 processes runnable
1 processes runnable
1 processes runnable
1 processes runnable
```

- `test1`：n越大，平均周转时间越大
- `test2`：`sltime`呈现金字塔式，越早创建的进程`sltime`越大；`rutime`基本相同。

4.1.2 PB

可以发现进程是父进程`fork`完了所有子进程之后，子进程才开始执行，并且按照次序，按照流程，一个一个进行。并且先按照时间顺序执行完优先队列1后，再继续按照时间顺序执行优先队列2。

优先队列n+1的等待时间会比队列n长，所以队列平均周转时间就会变长。

这样的输出符合预期和流程图。

- n越大，平均周转时间越大。
- 优先级越小，优先队列平均周转时间越大。

4.2 RR & PB

两种算法各有优缺点：

算法	优点	缺点
RR	无饥饿	平均等待时间长
PB	高优先级进程优先	无穷阻塞，饥饿

比较RR&PB在相同子进程的情况下的运转情况，以平均周转时间为衡量标准：

n较小时，PB较好，n较大时，RR较好。

5.实验感想

本次实验是耗费精力最多的实验，感觉花了很多时间去琢磨和研究。

- 真正认识到了之前翻译代码的时候那几个函数的作用，比如`fork()`，之前就停留在返回值是`pid`的地步上，后来发现这个函数给用户程序使用还可以通过返回值控制父进程和子进程不同的操作。也感觉之前懵懵懂懂的函数理解，在这次实践当中彻底理解透了
- 学会了实现两种算法，也理解清楚了他们的实现细节。比如RR转换到`priority`，时间切片是否还需要用，一开始还一直在思考也需要用中断的机制去`yield`，后来才想起来书上根本不是这样的优先级调度，然后恍然大悟需要修改这些东西。
- 在`RR`上是真的学会了找函数声明、定义函数、`makefile`、用户接口等各方面的实现细节，也成功知道了定义系统调用函数、定义用户调用函数接口的实现过程。而且我使用的`vscode`确实可以非常方便的找到函数的声明、定义等信息。代码阅读能力大幅提升啊。
- 阅读透彻了`xv6`部分文档，一遇到问题就反复阅读，反复看代码，提高了很大的解决问题的能力呀！

- 花了很多心思设计`test`函数，因为没有`sltime`所以设计了`RR`的`test2`，因为大规模运算不够，又去找了很多参考的资料，测试了很多遍。
- 这也是第一次使用`markdown`写实验报告，看着美腻的实验报告成就感更高了。
- 缺点就是做的效率还需要提高！

6.参考资料

[1] https://github.com/yyd19981117/xv6-1909

[2] https://th0ar.gitbooks.io/xv6-chinese/content/content/chapter3.html