

## Lab2 Nachos 进程管理与调度

唐凯成, PB16001695

### 一、实验目的

1、掌握并运用有关进程管理与同步的相关知识，并在 Nachos 中实现以下三个系统调用：

(1) `int Fork ()`: 创建一个子进程，在父进程中返回子进程的 `id`，在子进程中返回 `0`。

(2) `void Exec (char *exec_name)`: 载入并执行其他可执行程序

(3) `void Join (int child_id)`: 父进程等待某个子进程执行结束

2、理解掌握有关进程调度的相关知识，实现 Nachos 内的进程优先级调度。

3、针对 Nachos 编写一个简单的 shell 界面。

### 二、实验过程

1、实现有关进程管理的系统调用

(1) 添加三个系统调用的预处理接口：

对于本次所需添加的系统调用，由于整体为代码填空，故无需自己实现系统调用号、系统调用接口、进入内核系统调用的接口等，只需要简单修改中断入口的处理函数，并加入实现系统调用的函数即可。特别的，由于本次实验中所涉及的类中有部分成员函数在实际使用中仍需调整，故具体实验中也有少量对成员函数的修改。

(2) 系统调用的具体实现：

1) Fork 系统调用

a) 修改中断入口处理函数：

对于 Fork 的中断入口，由设计只需调用 `Sysfork` 得到子程序 `pid` 的返回值即可，在之后调用 `PC` 指针向前移动的 `PCPlusPlus()` 函数即可完成。

b) 调整预设类里的成员函数：

在具体实验前，需要了解有关进程的类函数 `Thread`，文件为 `Thread.cc`，从而了解进程类的变量及成员函数。关于使用到的成员函数，将在具体步骤中再详细说明。对于 Fork 调用，由于需要返回进程的 `tid`，故在 `Thread.h` 中加入 `int tid`，然后在其构造函数 `Thread(char *threadName)` 中加入以下语句，用以实现对每个进程随机生成一个小于 5000 且不重复的 `tid`：

```
-   tid=5001;
while(tid>5000)
{
-   tid = random() % 5000; //随机赋值给tid，考虑最多同时运行5000个进程，tid<5000
-   while(kernel->threadMap.count(tid)!=0){ tid++;} //确保新生成的tid为第一次出现
} //循环确保最终生成的tid小于等于5000
kernel->addThread(tid, this); //在进程键值对中添加当前进程
parent = NULL; //初始化时默认无父进程
```

最后加入对 `tid` 操作的成员函数 `getTid()`，定义为：

```
int Thread::getTid(){return tid; }
```

原来 Nachos 中进程类中没有指向父进程的指针，为了方便我们在子进程与父进程间切换，此处也应补上父进程指针，在 `Thread.h` 的 `Thread` 类定义中加入如下定义：

```
Thread *parent; //即为一个指向 Thread 类的指针
```

另外还要再析构函数中完成对进程的移除，以及对父进程指针的清空。

## c) 在内核中实现系统调用函数:

该系统调用的实现主要分为 2 块, 先是完成子进程的创建和相关资源的分配, 这部分在函数 SysFork() 中实现, 再是在子进程内的操作, 在函数 forked(int arg) 中实现。

另外在开始需要注意的一点是, 在 kernel 类中定义了: Thread \*currentThread, 即当前进程的类已由 currentThread 指针来表示, 可在任何时候调用该指针来获取当前进程的信息。

下面分别描述实现两个函数的思路:

## Int SysFork():

1. 创建子进程 t: 直接申明 Thread\* child, 即一个新的类指针并指向 new Thread。
2. 为 t 创建进程空间: 进程空间在该系统中由 AddrSpace 类来定义, 而在 Thread 类中定义了一个 AddrSpace 类指针 \*space, 为该指针用 new 产生一个 AddrSpace 大小的空间即可。
3. 将父进程指针指向当前进程: 只需将 parent 指针等于 currentThread 即可。
4. 将当前进程(父进程)的内存空间拷贝给 t: 调用 AddrSpace 类中 copyMemory(AddrSpace \*dst, AddrSpace \*src) 来实现即可, 代入的参数为两个进程类的 space 变量。
5. 保存当前寄存器的状态到 t: 调用 Thread 类中的 SaveUserState() 即可。
6. 给予进程分配栈空间: 调用 Thread 类中 Fork 函数, 注意代入变量为一个 void 型函数及一个返回值(此处设置为 0), 代入函数 forked 即为子进程内操作。
7. 进行进程切换: 调用 Thread 类中 Yield() 函数, 具体实现方式在进程调度中说明。
8. 返回子进程的 tid: 调用 Thread 类中修改好的 getTid() 即可。

## void forked(int arg):

1. 将自己的寄存器状态恢复(到机器): 对当前进程 currentThread 调用 RestoreUserState()。
2. 将自己的进程空间恢复(到机器): 对当前进程 currentThread 调用 RestoreState()。
3. 写入返回值 0: 调用 machine 类函数 WriteRegister(2,arg), 写回 arg (已定义为 0)。
4. PC 指向下一条指令: 调用 machine 类函数 PCplusplus()。
5. 开始执行用户空间的代码: 调用 machine 类函数 Run()。

如此即实现了 Fork 的系统调用。

## 2) Exec 系统调用

## a) 修改中断入口处理函数:

采用类似于实验 1 中的方法, 采用循环读入所要执行的文件名, 并将文件名代入具体的实现函数 SysExec(ExecName) 中执行下一步操作即可。

## b) 在内核中实现系统调用函数:

## void SysExec(char \*fileName)

1. 设定新产生的子进程名: 对当前进程 currentThread 调用 setName 成员函数即可。
  2. 重置进程空间, 清空内存: 对当前进程 currentThread 的 space 变量分别调用 reset() (清空内存) 和 RestoreState() (恢复存储空间)。
  3. 根据文件名将程序加载到内存: 对当前进程 currentThread 的 space 调用 Load 函数。
  4. 若加载成功则执行: 加载成功时返回 true, 继续对 space 调用 Execute() 函数。
  5. 若不成功交给 Linux 上处理: 失败时返回 false, 用 system(filename) 交给 Linux 处理。
  6. 结束当前进程: 对当前进程 currentThread 调用 Finish() 函数。
- c) 调整预设类里的成员函数 Thread::Finish():

对于 Thread 类中的 Finish()函数，用于实现在子进程结束时对信号量的调整，抬高信号量以允许其他进程执行。修改时主要加入如下操作：

1.判断该进程是否有父进程，有父进程时继续。

2.新建信号量指针：信号量类为 Semaphore，新建指针后调用进程 parent 中的成员函数 joinSemMap\_getSemByID(tid)来获得其父进程的信号量。

3.提升信号量：调用 Semaphore 类中的 V()来实现。

如此即实现了 Exec 的系统调用。

### 3) Join 系统调用

a) 修改中断入口处理函数：

对于 Join 系统调用，和之前中断入口略有不同，没有再专门编写系统调用函数，而是直接在 Thread 类的成员函数中添加 join，故在此处对当前进程 currentThread 调用 join 函数。

b) 在 Thread 类的成员函数中添加 join：

void Thread::join(int tid):

1.根据 id 获取子进程：调用 kernel 类的 getThreadByID(tid)函数，当没有子进程时会返回 NULL，在返回值不为 NULL 时进行以下操作。

2.新建一个初值为 0 的信号量：新建一个 Semaphore 类的指针并在 new 的时候设置初始值为 0 即可。

3.将子进程 id 和该信号量插入 joinSemMap：对当前进程 currentThread 调用 joinSemMap\_insert(int tid, Semaphore \*sem)即可完成插入。

4.等待该信号量：调用 Semaphore 类的 P()来实现。

5.将该信号量从 joinSemMap 中移除：对当前进程 currentThread 调用 joinSemMap\_remove 函数即可完成移除。

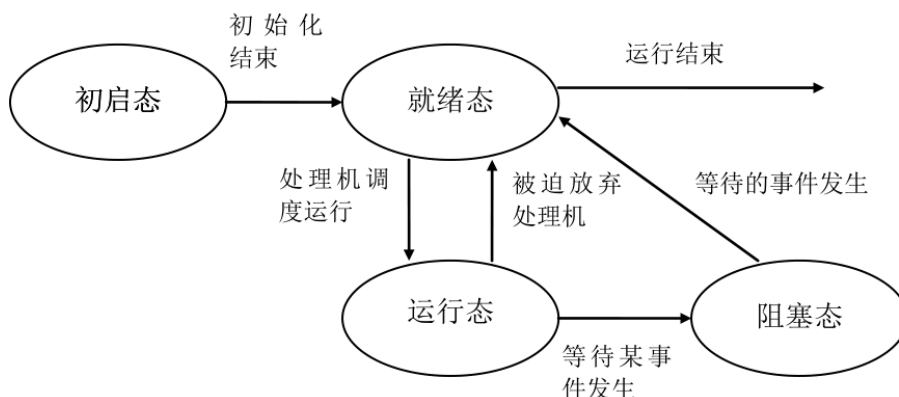
另外对于子进程的信号量处理已在前面 Fork 系统调用中的 Finish()函数中完成修改，实现了每次父进程等待子进程时 down 信号量，子进程结束时 up 信号量的循环。

如此即实现了 Join 的系统调用。

## 2、实现 Nachos 内的进程优先级调度：

### (1) 对于 Nachos 中进程调度机制的理解

对于 Nachos 中的进程，在以下四个状态循环：



而对于进程调度，Nachos 中默认使用 RR 轮转法，即在每个时间片内执行进程，在进程结束或时间片终止时强制进行进程切换，具体实现可在 Scheduler.cc 中找到。主要设计思路为在进程调度类里有一个就绪进程队列，每次进行进程切换时，都对就绪队列首取一个进程运行，如果之前的进程没运行完，就将其重新放回就绪队列尾部等待下一轮运行。

对于进程切换的实现函数即为之前 Thread 类中的 Yield() 函数，其在每个时间片的末尾被调用。对于时间片的操作函数在中断类 Interrupt 中定义了一个 OneTick() 函数来实现，每执行一个 OneTick() 函数，即代表完成了一个时间片。

而对于时间片的循环调用，在 mipssim.cc 中定义，由 machine 类中的 Run 函数来实现，在 Run() 函数中有一个无限 for 循环，只要不满足中断结束条件就一直循环调用 OneTick() 函数，如此实现完整的 RR 轮转调度。

### (2) 静态优先级调度的实现

对于助教已经实现的静态优先级调度，主要由原来的 RR 轮转修改而来：

在原来的进程类中添加变量 priority，代表每个变量的优先级，并在初始化时随机为每个进程生成一个 200+ 的数来存入优先级中。再对本来的调度类的进程就绪队列进行修改，将就绪队列改为对优先级自动排序的有序队列，优先级高的进程在队列首部。故每当有进程进入就绪队列时，会直接完成就绪队列对于优先级的排序，此时再选择队列首的进程运行，即为选择优先级最高的进程运行，如此即实现了静态优先级调度。

### (3) 动态优先级调度的实现

对于动态优先级的实现，只需在静态的基础上加入两部分即可：所有就绪进程的优先级自提升和当前运行进程的优先级下降。在实际编写中规定调整方法为：

- 调整所有就绪进程的优先级，计算公式为

$\text{priority} = \text{priority} + \text{AdaptPace}$ （定义为 2）

- 调整当前进程优先级，计算公式为

$\text{priority} = \text{priority} - (\text{当前系统时间} - \text{lastSwitchTick}) / 100$

两者分别在 Scheduler::flushPriority() 与 FindNextToRun() 中实现。

由于只是简单的调用和加减，此处就不再复制代码。

特别的，在进程切换时还需考虑一些条件，如进程切换的时间间隔是否过小和就绪队列是否为空，另外如果一个程序执行完已经为 BLOCKED 状态则强制进行切换：

```
if(Tick<MinSwitchPace) return NULL;
// 间隔太小，返回 NULL（避免过分频繁地调度）
if(readyList->IsEmpty()) return NULL;
// 就绪队列为空，返回 NULL（没有就绪进程可以调度）
Maxpri=readyList->Front()->getPriority();
if((Maxpri>currentpri) || (kernel->currentThread->getStatus() ==BLOCKED))
// 找到优先级最高的就绪态进程t
{
    lastSwitchTick = kernel->stats->totalTicks;
    return readyList->RemoveFront();
}
else return NULL;
```

### 3. 一个简单 shell 的实现

对于 shell 的实现基本已由助教完成，相关命令的分割和操作界面的输出均已完成设计，只需要加入关键部分的进程创建等指令即可，调用之前写好的几个系统调用如下：

```
for (i=0;i<cmdNum;i++)
{
    childID[i]=Fork();
    if(childID[i]==0)
        Exec(cmdLine[i]);
}
for (i=0;i<cmdNum;i++)
{
    if(childID[i]!=0)
        Join(childID[i]);
}
```

## 三、实验结果

(1) 相关系统调用的运行结果截图：

```
tkc@ubuntu:~/nachos/lab2-nachos-stu/code/test$ ../build.linux/nachos -x fork

1. init var = "parent"
2. i am child. i change var = "child"
3. i am parent. my var = "parent"

Machine halting!

Ticks: total 335258, idle 90, system 33560, user 301608
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

```
tkc@ubuntu:~/nachos/lab2-nachos-stu/code/test$ ../build.linux/nachos -x exec

1. i am child. i am runing 'add'.
   42 + 23 = 65
2. i am parent. i finished after my child

Machine halting!

Ticks: total 1823724, idle 90, system 182410, user 1641224
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

```
tkc@ubuntu:~/nachos/lab2-nachos-stu/code/test$ ../build.linux/nachos -x join

1. i am parent. i am waitting my childID=3335
2. i am child. i am runing 'add'. please wait...
   42 + 23 = 65
3. parent finished

Machine halting!

Ticks: total 2448078, idle 90, system 244850, user 2203138
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

(2) shell 运行结果的截图

```
tkc@ubuntu:~/nachos/lab2-nachos-stu/code/test$ ../build.linux/nachos -x shell

nachos >> ./add
42 + 23 = 65

nachos >> ps
Unable to open file ps
exec cmd by Linux
  PID TTY          TIME CMD
 15650 pts/5        00:00:00 bash
 16670 pts/5        00:00:00 nachos
 19413 pts/5        00:00:00 nachos
 19460 pts/5        00:00:00 nachos
 19516 pts/5        00:00:00 nachos
 25796 pts/5        00:00:00 nachos
 25815 pts/5        00:00:00 sh
 25816 pts/5        00:00:00 ps

nachos >> ./2;./2;./add;./2;./2
42 + 23 = 65
i am testshell
i am testshell
i am testshell
i am testshell

nachos >>
```

特别的，图中 ps 文件不为 nachos 中已经预设好的文件，故将该命令交给宿主机 Linux 执行，输出中也加入了“exec cmd by Linux”。另外对于底下并行输出结果的分析见下一点。

(3) 优先级调度文件结果截图：

节选了最初的 add 执行的相关部分和最后多个进程并行中进程切换到 add 的相关部分。

```

-----one switch-----
running: tid=3335      name=Forked thread  status=RUNNING  pri=245

Ready list contents:
  tid=886      name=main      status=READY  pri=173
|
-----one switch-----
running: tid=3335      name=./add      status=BLOCKED  pri=245

Ready list contents:
  tid=886      name=main      status=READY  pri=175

-----one switch-----
running: tid=886      name=main      status=RUNNING  pri=103

Ready list contents:
  tid=492      name=Forked thread  status=READY  pri=238

-----one switch-----
running: tid=492      name=ps      status=BLOCKED  pri=238

Ready list contents:
  tid=886      name=main      status=READY  pri=105

-----one switch-----
running: tid=59  name=./add      status=RUNNING  pri=242

Ready list contents:
  tid=886      name=main      status=READY  pri=75
  tid=27  name=./2      status=READY  pri=65
  tid=1421     name=./2      status=READY  pri=62|

-----one switch-----
running: tid=59  name=./add      status=BLOCKED  pri=242

Ready list contents:
  tid=886      name=main      status=READY  pri=77
  tid=27  name=./2      status=READY  pri=67
  tid=1421     name=./2      status=READY  pri=64

-----one switch-----
running: tid=886      name=main      status=RUNNING  pri=77

Ready list contents:
  tid=3926     name=Forked thread  status=READY  pri=215
  tid=27  name=./2      status=READY  pri=69
  tid=1421     name=./2      status=READY  pri=66

-----one switch-----
running: tid=3926     name=Forked thread  status=RUNNING  pri=215

Ready list contents:
  tid=886      name=main      status=READY  pri=79
  tid=27  name=./2      status=READY  pri=71
  tid=1421     name=./2      status=READY  pri=68

```

对于该运行结果可见已经实现动态的优先级调度：对于 2 程序由于其内部有一个 1000\*1000 的循环，故需要很多个时间片来完成其执行，而 add 只需要一个时间片即可，故在动态的优先级调度中 add 的进程在生成时的优先级远高于已经在进程中运行了一段时间的 2 个 2 程序，故会立刻被置顶运行，而它仅需要一个时间片就可以完成运行，故运行后立刻输出，因而 add 的输出结果在最前面，其余 2 程序的输出结果则靠后。

#### 四、实验中的问题与分析

##### 1、对于各个类中函数理解不全的问题

对于很多类中的成员函数相互关联相互交叉，很难一次性看懂所有类的成员函数，故在实验前半小时对很多调用都为一知半解；在实验后半小时，额外阅读了 `machine` 类、`List` 类和 `Interrupt` 类等，才完成了对进程调度的理解。

##### 2、初步完成系统调用时，对于执行 `join` 测试程序时出现以下问题：

```
tkc@ubuntu:~/nachos/lab2-nachos-stu/code/test$ ../build.linux/nachos -x join
1. i am parent. i am waitting my childID=3335
2. i am child. i am runing 'add'. please wait...
   42 + 23 = 65
```

##### 解决方案：

对于该问题经过多次尝试后发现是子进程在结束后没有提升信号量使得父进程可以继续运行，找到子进程结束时调用的 `Thread` 中的 `Finish()` 函数，在其中添加相关信号量提升的语句（在前面实验实现阶段已说明），使得信号量 `value` 重新等于 0，父进程可以结束。

##### 3、shell 完成后运行 `./2` 指令出现异常：

```
nachos >> ./2
nachos >>
nachos >>
nachos >>
nachos >>
nachos >>
nachos >> i am testshell
```

##### 解决方法：

对于该问题采用了许多方法尝试解决，后来发现实验 1 中的 `join` 等测试函数也出现了相应的问题如下：

```
nachos >> ./join
nachos >>
1. i am parent. i am waitting my childID=59
3. parent finished
```

和实验 1 时成功通过的代码对比，发现只是因为在本来定义的 `join` 中加入了一句判断语句：`if (Threadchild!=NULL)`，本来没有该判断直接执行后续语句：



```

Thread *Threadchild = kernel->getThreadByID(tid);
if (Threadchild!=NULL)
{
    Semaphore *sem = new Semaphore("create semaphore", 0);
    kernel->currentThread->joinSemMap_insert(tid,sem);
    sem->P();
    kernel->currentThread->joinSemMap_remove(tid);
    delete sem;
}

```

经过对比分析后可知，由于加入的 if 语句在明明有子程序时得到的 Threadchild 仍为 NULL，因此导致判断后跳过后续信号量处理语句而导致子程序的运行错误。此时才将错误锁定在之前初始化时的 getThreadByID(tid)函数上，之前助教提问时也提到了该函数的用处，该函数本来用于由进程的 tid 返回进程指针，以此来在父进程中找到子进程的指针，但若不添加后面的这个 if 判断语句，得到子进程的指针也没有实际意义，因此才在实验前半结束后添加了这个判断语句，原意为当该子进程存在时，父进程才进行以下操作，不存在时则无需执行 join 函数。不过如果不加 if 语句，由于之前父进程执行时子进程都存在，所以不会出现无需执行 join 函数的情况，故不加 if 时程序也可以运行通过。

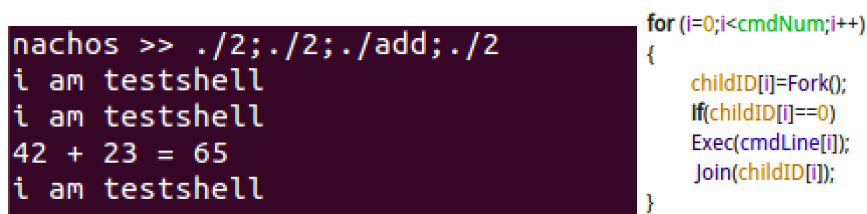
之后再进一步考虑 getThreadByID(tid)的出错原因，阅读函数源码后发现该函数是从键值对 threadMap 中根据 tid 查找进程，而在 Fork 调用时并未把子进程的 tid 和进程指针放入 threadMap 键值对中，因此导致在该键值对中搜索子进程时总会返回 NULL。因此只要是在子进程新建时把子进程的 tid 和进程指针插入 threadMap 中即可，故在 Thread 类的构造和析构函数中分别加入：

构造函数中加入：kernel->addThread(tid, this);

析构函数中加入：kernel->removeThread(tid);

其中 addThread 成员函数用于添加键值对，removeThread 用于移除键值对，如此即实现了子进程在 threadMap 键值对中的插入与删除，此时再在 join 中加入 if 语句则不会再出现问题，返回值也不为 NULL，运行结果正确。

4、在最初完成 shell 搭建时的多程序运行结果：



```

nachos >> ./2;./2;./add;./2
i am testshell
i am testshell
42 + 23 = 65
i am testshell

for (i=0;i<cmdNum;i++)
{
    childID[i]=Fork();
    if(childID[i]==0)
        Exec(cmdLine[i]);
    Join(childID[i]);
}

```

解决方法：

最初 shell 添加的代码为右上图，此时已完成进程动态优先级的相关代码，故结果中 add 的运行结果应最先输出，而此时仍按照命令输出，观察此时的进程调度文件如下：

```

-----one switch-----
running: tid=3335      name=../2      status=BLOCKED  pri=-121

Ready list contents:
  tid=886      name=main      status=READY    pri=195

-----one switch-----
running: tid=886      name=main      status=RUNNING  pri=119

Ready list contents:
  tid=492      name=Forked thread  status=READY    pri=238

-----one switch-----
running: tid=492      name=../add     status=BLOCKED  pri=238

Ready list contents:
  tid=886      name=main      status=READY    pri=121

-----one switch-----
running: tid=886      name=main      status=RUNNING  pri=17

Ready list contents:
  tid=1421     name=Forked thread  status=READY    pri=251

-----one switch-----
running: tid=1421     name=Forked thread  status=RUNNING  pri=251

Ready list contents:
  tid=886      name=main      status=READY    pri=19

-----one switch-----
running: tid=1421     name=../2      status=RUNNING  pri=250

Ready list contents:
  tid=886      name=main      status=READY    pri=21

```

此时可见就绪队列里始终只有一个进程，即多个程序不是并发执行的，故问题还是应该出在 `shell` 中，即要考虑如何使进程能并发执行。考虑到每个子进程如果在 `Fork` 后就执行父进程的 `Join`，则会导致父进程每完成一个子进程的运行才创建一个新的子进程，因此导致子进程间相互是不并行地。由此考虑一次性 `Fork` 多个子进程，在执行完后再一次性 `Join` 多个子进程，如此便能实现子进程的并行。修改后代码见实验过程 3，成功地实现了子进程的并发执行切同时输出了动态优先调度的结果。

5、对于进程切换问题的优化，在进行进程切换时出现以下情况：（见下页图片）  
解决方法：

该图中存在一进程切换中的严重问题，即 `add` 已经运行结束并被设置为 `BLOCKED` 状态，但由于其优先级远高于其他进程，故在每次切换进程进行优先级判断时，发现当前运行进程优先级大于就绪进程中优先级最高的，故继续运行该进程而不进行进程切换，等到下个时间片结束再重复该判断。然而对于该图片中的情况，进程已经运行完成后，由于每次判断完均不执行进程切换，导致该运行完的进程无法被 `Sleep` 函数调用而一直卡在运行进程中，直到其优先级不断降低直到其余就绪队列中进程优先级超过它时才发生进程切换。

为了处理该情况，只需要在判断运行中进程与就绪队列进程的优先级高低时加上条件：运行中进程状态不为 `BLOCKED` 即可，具体实现在实验步骤中已特意列出。如此也回答了本次进程调度实验的核心问题：进程切换发生在两种情况下：一是每个时间片截止时强制发生进程切换，挑选优先级最高的进程成为下一个执行的进程；二是当前运行程序已运行结束并被设置为 `BLOCKED` 时，也需进行进程切换执行就绪队列中进程。

```

-----one switch-----
running: tid=3926      name=./add      status=BLOCKED      pri=215

Ready list contents:
    tid=886      name=main      status=READY      pri=91
    tid=27  name=./2      status=READY      pri=89
    tid=59  name=./2      status=READY      pri=77

-----one switch-----
running: tid=3926      name=./add      status=BLOCKED      pri=214

Ready list contents:
    tid=886      name=main      status=READY      pri=93
    tid=27  name=./2      status=READY      pri=91
    tid=59  name=./2      status=READY      pri=79

-----one switch-----
running: tid=3926      name=./add      status=BLOCKED      pri=212

Ready list contents:
    tid=886      name=main      status=READY      pri=95
    tid=27  name=./2      status=READY      pri=93
    tid=59  name=./2      status=READY      pri=81

-----one switch-----
running: tid=3926      name=./add      status=BLOCKED      pri=209

Ready list contents:
    tid=886      name=main      status=READY      pri=97
    tid=27  name=./2      status=READY      pri=95
    tid=59  name=./2      status=READY      pri=83

```

## 五、实验总结与收获

本次实验中完成了对 Nachos 系统进程管理的一系列学习，从添加基础的系统调用到后面层次更高的优先级调度，整体难度比实验一中大很多，且需要阅读许多预设好的类成员函数。在实际尝试中花费了大量时间在不同类间交叉阅读成员函数，虽然 Nachos 操作系统的整体实现并不复杂，不过其各个类间的互相调用确实为我打开了新世界，第一次明白了 C++ 中类的强大作用及多个类间相互调用的灵活。

为了深入理解每个函数的作用，往往一个函数会穿插在 3-5 个类间，这种阅读体验虽然繁琐但给我一种奥妙无穷的感觉，简单的几行代码相互调用就组成了一个操作系统的雏形。在这个实验中我也真正感受到了进程调用 RR 轮转法的实际实现和各处的细微优化，大大增加了我对操作系统进程管理有关书本知识的了解。总得来说感觉一次实验收获了很多，希望以后这种课程也能多增设几个实验，而不只是再局限于书面知识的理解。

## 参考文献:

- [1] Abraham Silberschatz. 操作系统概念. 高等教育出版社, 2007.3.
- [2] nachos 操作系统课程设计报告  
<https://max.book118.com/html/2017/0104/79987092.shtm>
- [3] Nachos 中文教程
- [4] OS 实验手册-lab2