

操作系统原理与设计

实验二 Nachos 进程管理与调度

—— 简单 Nachos shell 的实现

必读：

根据实验反馈，考虑到同学们实际情况，对实验二做以下调整

1、本次实验形式主要为代码填空。

- 所有需要填空的部分助教都已经在代码中做了标记。
- 你必须且只能把代码写在助教圈定的范围以内。
- 且不能在任何地方添加任何头文件。
- 不能使用 C 或 C++ 的库函数（如 vfork, wait, exec 函数族等）

2、所有复杂操作助教都已经把它们封装成函数，你只需要调用即可。

- 本文第三部分加下划线的每个句子都对应一个被封装的函数。
- 你必须阅读代码，找到并使用它们。

3、本次分两阶段进行

- 第一阶段实验内容为：实现 fork、exec、join 系统调用
验收时间为：4 月 27 日晚上实验课（待老师决定）
- 第二阶段要求实现所有内容
验收时间为：5 月 11 日晚上实验课（待老师决定）

一、实验目的

1. 掌握进程管理与同步：实现 fork、exec、join 系统调用。
2. 掌握进程调度：实现优先级调度。

二、实验内容

运用理论课上学习的 `fork`、`exec`、`waitpid`/`join` 等系统调用的工作原理,在 Nachos 上实现进程的管理、同步与调度。主要包含以下几点:

1. 实现 `fork`、`exec`、`join` 系统调用。(进程管理、同步)
2. 实现进程优先级调度。(进程调度)
3. 编写一个简单的 Nachos shell。(具体运用)

下面对上述三点展开介绍:

2.1 实现 `fork`、`exec`、`join` 系统调用.

- **Fork:**
 - 系统调用 `fork` 在 nachos 中的函数原型是 `int Fork ();`
 - 该函数和 C 语言中的 `fork` 功能基本相同,即创建一个子进程。
 - 需要注意的是该函数返回两次,在父进程中返回子进程的 `id`,在子进程中返回 0.
- **Exec:**
 - 系统调用 `exec` 在 nachos 中的函数原型是 `void Exec (char *exec_name)`
 - 该函数用于载入并执行其他可执行程序。
 - 参数: `exec_name` 表示要运行的可执行程序的文件名
- **Join:**
 - 系统调用 `jion` 在 nachos 中的函数原型是 `void Join (int child_id);`
 - 该函数功能是父进程等待某个子进程执行结束,类似课上讲的 `waitpid` 函数:
 - 参数: 类似课上讲的 `waitpid` 函数

- **测试方法**

你可以使用以下程序测试你的 `Fork`、`Exec`、`Join` 系统调用是否正确

`Fork` 的测试程序: `test` 目录下的 `fork.c`

`Exec` 的测试程序: `test` 目录下的 `exec.c`

`Join` 的测试程序: `test` 目录下的 `join.c`

验收标准: 能正确运行上述三个测试

2.2 编写一个简单的 Nachos shell.

- Shell 简介:

shell 是用户和操作系统之间的接口。用户可以通过 shell 执行操作系统的指令。如 linux 中的 ls, cat, pwd 等指令。常见的 shell 有 Windows 的 PowerShell 和大多数 linux 系统默认使用的 bash shell.

- 实验内容:

补全 test/shell.c 程序, 实现以下功能

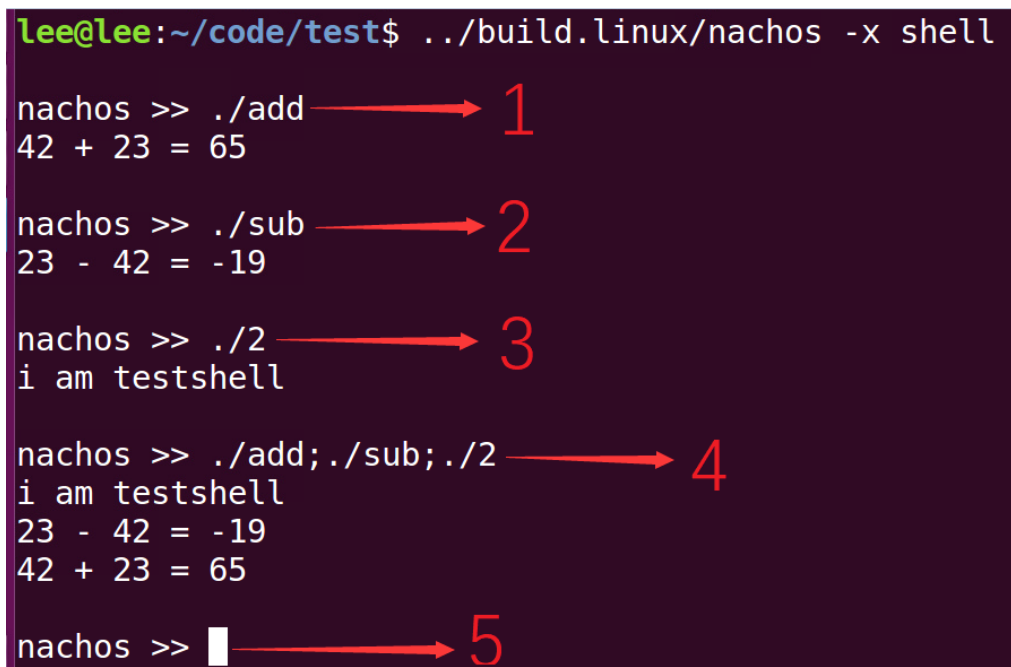
- 运行可执行文件. (如 ./add)
./add 的含义是运行当前目录下的可执行程序 add
- 并行执行多条指令. (如 ./add ; ./sub ; ./join ; ./fork)
指令间用英文分号隔开, 且假设中间无空格
- 指令运行结束后, 回到 shell, 继续接收用户输入, 循环执行

- 要求:

必须使用多进程的方式实现

- 父进程负责循环整个 shell, 有命令时交给子进程完成
- 执行多条指令时, 子进程数量 = 指令数量, 且每个子进程负责一条指令

- 最终效果:



```
lee@lee:~/code/test$ ../build.linux/nachos -x shell

nachos >> ./add → 1
42 + 23 = 65

nachos >> ./sub → 2
23 - 42 = -19

nachos >> ./2 → 3
i am testshell

nachos >> ./add;./sub;./2 → 4
i am testshell
23 - 42 = -19
42 + 23 = 65

nachos >> █ → 5
```

如上图：

在 test 目录下执行 “../build.linux/nachos -x shell” 运行你的 nachos shell

第一次输入 “./add”：执行名字为 add 的程序

第二次输入 “./sub”：执行名字为 sub 的程序

第三次输入 “./2”：执行名字为 2 的程序

第四次输入多条指令，“./add;./sub;./2”：并行执行 add、sub、2 三个程序。

上图中标号 5 处，每次命令执行结束就等待用户输入。

2.3 实现进程优先级调度.

- 背景：

- 当 shell 同时运行多条指令时，希望优先执行高优先级指令
- 进程调度决定当多进程处于就绪态时如何选择下一个运行的进程
- Nachos 目前只支持 RR 调度（轮流执行进程）

- 实验内容：

动态优先级调度

流程如下：

- 创建进程时，为其赋予优先级
- 进程调度切换上下文时：
降低当前进程优先级， 增加所有就绪进程的优先级；
切换到当前优先级最高的进程执行；

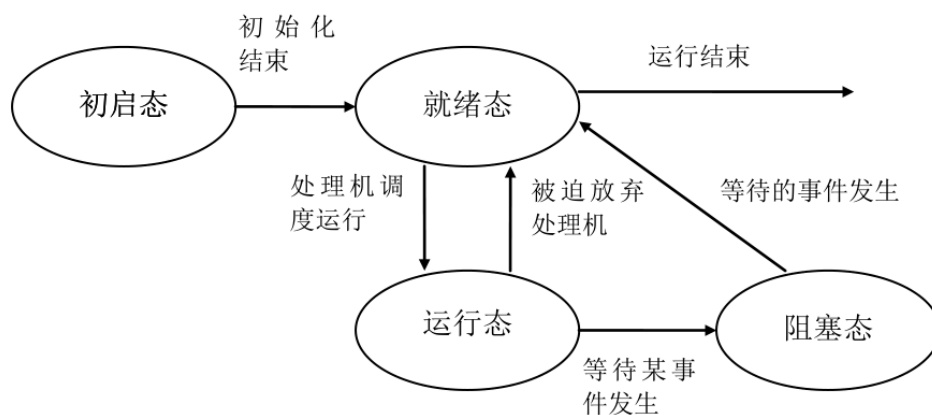
- 要求：

- 静态优先级调度：（基础）
为进程增加优先级，高优先级进程先执行（助教已实现）
- 动态优先级调度：（扩展升级）
在进程切换时动态调整进程优先级
（你必须看懂助教实现的静态优先级的代码，并基于此实现动态优先级）
验收标准：记录每次调度的结果，能看到每次都是高优先级优先执行。

三、详细培训

说明：

- Nachos 目前不区分进程和线程，两者完全相同，nachos 中的“进程”和“线程”都等价于理论课上讲述的“进程”。所以在一些 nachos 的资料中和本次实验中你可能会看到“进程”、“线程”两个词混用的情况（此时不要疑惑）。
- nachos 的进程只有四种状态，它们的关系如下图所示。它们的关系与理论课上的讲述的基本一致，不同之处在于 nachos 进程没有终止态。当 nachos 进程需要终止时，它会调用某个函数（请你阅读代码找到它）将自己置为结束，下一个执行的进程会销毁它的内存空间。



3.0 搭建代码环境

- 将提供的 lab2-nachos-stu.tar.gz 复制到实验一的 nachos 目录下。（该目录下必须有一个名为 usr 的文件夹，如果没有 usr 文件夹说明你没有实验环境，那么你首先需要按照实验一的培训文档搭建环境）
- 打开终端，进入 nachos 目录，顺序执行以下命令：
 - `tar -zxvf lab2-nachos-stu.tar.gz`
 - `cd lab2-nachos-stu`
 - `cd coff2coff`
 - `make`
 - `cd ../code/test`

- make
- cd ../build.linux
- make (若 make 出错, 则要先 make depend 再 make)

3.1 实现 fork、exec、join 系统调用.

● Fork

- Fork 的实现流程:
 - 创建子进程 t
 - 为 t 创建进程空间, 并将当前进程 (父进程) 的内存空间拷贝给 t (提示: 拷贝内存的函数是 AddrSpace::copyMemory)
 - 保存当前寄存器的状态到 t
 - 给予进程分配栈空间 (提示: 使用 Thread::Fork 函数)子进程:
 - 得到栈空间之后, 获得 CPU 时, 就将自己的寄存器状态恢复 (到机器)、进程空间恢复 (到机器)
 - 写入返回值 0
 - PC 指向下一条指令 (PC: programm counter)
 - 开始执行用户空间的代码 (提示: 使用 Machine 类中的某函数)
- 父进程返回子进程的 id
- 提示:
 - 为了能够在 Fork 执行时返回进程 id, 需要为进程增加 id 属性。助教提供的代码中已经定义了名为 tid 的 int 型变量表示进程 id, 你需要在 Thread 类对象创建时给 tid 赋予一个唯一的值。
 - Fork(Func,arg)函数的用法, 该函数定义在 Thread 类中, 且该类中同时提供了一个此时测试 Fork(Func,arg)的例子, 在终端进入 code/test 目录下执行 “../build.linux/nachos -K” 即可运行该例子。你可以参考这个例子理解 Fork(Func,arg) 函数的用法。

● Exec

- Exec 的主要执行流程:
 - 重置进程空间, 清空内存 (AddrSpace::reset)
 - 根据文件名将程序加载到内存

- 若加载成功则执行
- 若不成功，（使用 Thread 类中的某函数）结束当前进程
- 提示：
 - 关于“将程序加载到内存并执行”的如何实现，请参考 main.cc 中的加载和执行方式。

● Join:

- 需要的数据结构：
 - 系统中所有进程的集合：
 - 用类型为 `map<int, Thread *>` 名字为 `threadMap` 的 `map` 来表示。
`threadMap` 是一个 key-value 类型的集合，其中 key 是进程 id，value 是执行进程对象的指针。
 - 为了操作该数据结构，助教提供了三个函数用于向插入、删除和查找，它们分别是：
 - `void Kernel::addThread(int tid, Thread *thread)`
作用：向 `threadMap` 中插入一个 key-value 对。
 - `void Kernel::removeThread(int tid)`
作用：从 `threadMap` 中删除 key 为 tid 的键值对。
 - `Thread *Kernel::getThreadByID(int tid)`
作用：获取 `threadMap` 中 key 为 id 的键值对。
 - 被父进程 join 的所有子进程的集合：
 - 用类型为 `map<int, Semaphore *>` 名字为 `joinSemMap` 的 `map` 来表示。
`joinSemMap` 的 key 是被等待子进程的 id，value 是与该子进程对应的信号量。
 - 为了方便你编程，助教提供了操作 `joinSemMap` 的函数，分别是：
 - 插入：
`Thread::joinSemMap_insert(int tid, Semaphore *sem)`, 插入成功返回 1，不成功-1
 - 删除：
`Thread::joinSemMap_remove(int tid)` 删除成功返回 1，不成功-1
 - 根据 key 获取 value：
`Thread::joinSemMap_getSemByID(int tid)` 获取成功返回 1，不成功-1

- Join 的执行流程：
 - 父进程：
 - 根据 id 获取子进程。（若子进程已不存在则会获取失败，这说明了什么？）
 - 新建一个初值为 0 的信号量，将子进程 id 和该信号量插入 joinSemMap
 - 等待该信号量
 - 将该信号量从 joinSemMap 中移除
 - 子进程：
 - 当子进程运行结束时，增加/up 父进程中与该子进程对应的的那个信号量。（提示：为了获取该信号量，你可能会用到成员 `parent`，以及 `joinSemMap` 的查找）
- 提示：
 - Nachos 目前的进程类不能表示进程之间的父子关系，为了表示父子关系，你需要在进程类中增加一个 `Thread*` 类型的成员，命名为 `parent`，它表示当前进程的父亲。在父进程 Fork 子进程的时候给这个变量赋值。
 - 用于表示父子等待关系的该信号量的使用方式如下：
 - 初值为 0.
 - 若父进程需要等待子进程，则 `down` 该信号量（此时父进程可能被阻塞）。
 - 当子进程运行结束时，`up` 该信号量。

3.2 实现一个简单的 nachos shell.

- shell 的工作流程如下：
 - 获取输入
 - 解析获取指令
 - 循环执行每条指令
 - 等待指令运行结束
- 具体实现方式：
 - 从控制台获取终端的用户输入，结果保存在 `buff` （助教已提供）
 - 分割指令，分割后的指令保存在 `cmdLine` 中（助教已提供）

- 循环执行每条指令，对于每条指令新建一个子进程执行
- 父进程等待所有子进程至执行完毕
- 循环上述步骤

3.3 实现进程优先级调度

- 需要增加的数据项：

在 Scheduler 类中增加以下常数、成员变量和成员函数，它们的含义如下：

```

1  #define TimeTicks 200                //时间片大小
2  #define AdaptPace 2                 //优先级调整幅度
3  #define MinSwitchPace (TimeTicks / 4) //两次抢占调度之间的最小时间间隔
4
5  class Scheduler
6  {
7      ... //原有的属性和方法
8      public :
9          void FlushPriority(); //调整所有就绪线程 的优先级
10     private :
11         int lastSwitchTick; //记录上次线程切换的时间
12 }

```

- 实现方法

- 动态调整优先级的操作方式如下：

在进程调度时：

- 调整当前进程优先级，计算公式为

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

提示：可用 kernel 中 stats 类对象 totalTick 成员的值代表当前系统时间

- 调整所有就绪进程的优先级，计算公式为

$$\text{priority} = \text{priority} + \text{AdaptPace}$$

- 具体操作方式如下：

现在，原来的成员函数 FindNextToRun 担起了优先调度的责任，新的调度流程如下：

- 按照上面的公式调整当前进程、所有就绪进程的优先级

- 打印当前进程状态。（助教已提供打印函数）
- 计算最近调度与现在的间隔
- 间隔太小，返回 NULL （避免过分频繁地调度）
- 就绪队列为空，返回 NULL （没有就绪进程可以调度）
- 找到优先级最高的就绪态进程 t
 - 如果 t 的优先级高于当前运行进程，
 - 将该进程从就绪队列移除，运行该进程
 - 设置最近调度事件
- 返回 NULL （继续运行当前进程）

● 提示：

- 就绪队列已经定义并实现，。但是首先你必须找到它在哪才能操作它。

成员函数 `FlushPriority()` 刷新所有就绪进程的优先级，其算法比较简单，这里就不多作介绍。（但是你需要实现它）

● 验收标准：

当你在 `nachos shell` 中执行多条指令（如 `./2;./2;./2`）之后，在 `test` 目录下会生成一个名为 `0schedule_info.txt` 的文件，该文件内保存了每次的调度信息。内容效果如下图：

```

50
51 -----one switch-----
52 running: tid=4  name=2          status=RUNNING  pri=222
53
54 Ready list contents:
55 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
56 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
57 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
58 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
59 -----one switch-----
60 running: tid=3  name=2          status=RUNNING  pri=224
61
62 Ready list contents:
63 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
64 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
65 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
66 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
67 -----one switch-----
68 running: tid=4  name=2          status=RUNNING  pri=223
69
70 Ready list contents:
71 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
72 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
73 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

```

图中展示了三次调度的快照，分别用红色大括号和数字 1、2、3 标注：

- 1**：第一次调度快照，显示 `running: tid=4 name=2 status=RUNNING pri=222`，就绪队列中有 `tid=3 name=2 status=READY pri=225` 和 `tid=1 name=main status=READY pri=148`。
- 2**：第二次调度快照，显示 `running: tid=3 name=2 status=RUNNING pri=224`，就绪队列中有 `tid=4 name=2 status=READY pri=224` 和 `tid=1 name=main status=READY pri=150`。
- 3**：第三次调度快照，显示 `running: tid=4 name=2 status=RUNNING pri=223`，就绪队列中有 `tid=3 name=2 status=READY pri=226` 和 `tid=1 name=main status=READY pri=152`。

第 52 行表示：当前正在运行的进程 id=4，名字=2，状态=RUNNING，优先级=222
第 54-56 行表示就绪队列中的所有的进程的信息。

第 59 行表示执行了一次调度。

从图中标红的第 1 部分到第 2 部分表示一次调度，可以看到在第 1 部分中 id=3 的进程优先级最高，于是在调度后的第 2 部分它被选中执行。而 4 号进程状态从 RUNNING 被置为 READY

说明：

问：如何才能把自己的调度结果输出到该文件的问题？

答：关于该文件你不需要做任何工作。只要在 FindNextToRun()函数中按照助教的要求补全代码即可在该文件中看到你的调度结果。

四、实验报告提交

- 报告提交截止时间待定
- 实验报告格式：

参考课程主页 <http://staff.ustc.edu.cn/~ykli/os/> 的作业模板

- 实验报告内容主要包含三部分：

- 1) 实现的主要步骤、核心代码解释说明、关键函数的功能与流程；**
- 2) 实验运行结果截图，并分析说明；
- 3) 实验过程中遇到的技术问题及解决方法。
- 4) 重要程度：3 > 1 > 2

(**说明：实现步骤、关键函数的功能与流程不能仅仅包含本实验手册中的步骤和函数，更不能直接拷贝。可以参考该手册，但是更重要的是介绍为了实现需求你自己阅读的那部分函数代码以及你自己的工作。)

- 提交方式：

实验报告+实验代码（包含整个 Nachos 工程）

将实验报告和实验代码一起打包为 zip 压缩包，命名方式“学号+姓名.zip”，提交到 ftp 服务器的“实验提交/实验 2”目录下，**放错位置无效。**

附件：

分数分配：

第一阶段：60 分

运行测试实例			回答问题		
Fork	Exec	Join	问题 1	问题 2	问题 3
10 分	10 分	10 分	10 分	10 分	10 分

第二阶段：40 分

能执行 shell	动态优先级调度	问题 1	问题 2
10 分	10 分	10 分	10 分