|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统设计与实践 | | | 成 绩 |  | 教师签名 |  |
| 实验名称 | **多进程与进程调度** | | | 实验序号 | 8 | 实验日期 |  |
| 姓 名 | 张子航 | 学 号 | 2021302181026 | | | | 组长 |
| 姓 名 | 辜汝曦 | 学 号 | 2021302141194 | | | | 组员 |
| 姓 名 | 杨馨悦 | 学 号 | 2021302181212 | | | | 组员 |
| 姓 名 | 赵敏 | 学 号 | 2021302181215 | | | | 组员 |

**《操作系统设计与实践》实验报告**

# 实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容等）

## 实验目的

* 掌握多进程的实现机制
* 掌握minix的中断重入处理过程
* 掌握系统调用、带优先级的进程调度实现方法

## 参考资料

《Orange'S：一个操作系统的实现》第六章6.4、6.5、6.6

## 实验要求

* 扩展单进程到多进程
* 扩展中断处理程序支持进程之间的切换
* 实现一个获取时钟ticks的系统调用
* 进程调度问题，弄清楚实现调度的基本思路

# 实验环境及实验步骤

（列出本次实验所使用的软件、工具；简要概括实验步骤）

## 实验环境

虚拟机工具：VMWare Workstation 16

虚拟机版本：Ubuntu 14.04.6(内存4GB，硬盘40GB，双核处理器)

开发与调试工具：bochs 2.6.8

# 实验过程分析

（详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。）

## 在单进程的基础上如何实现扩展实现多进程？

在单进程的基础上扩展多进程，需要做到多个进程的实例化，以及进程之间的切换。

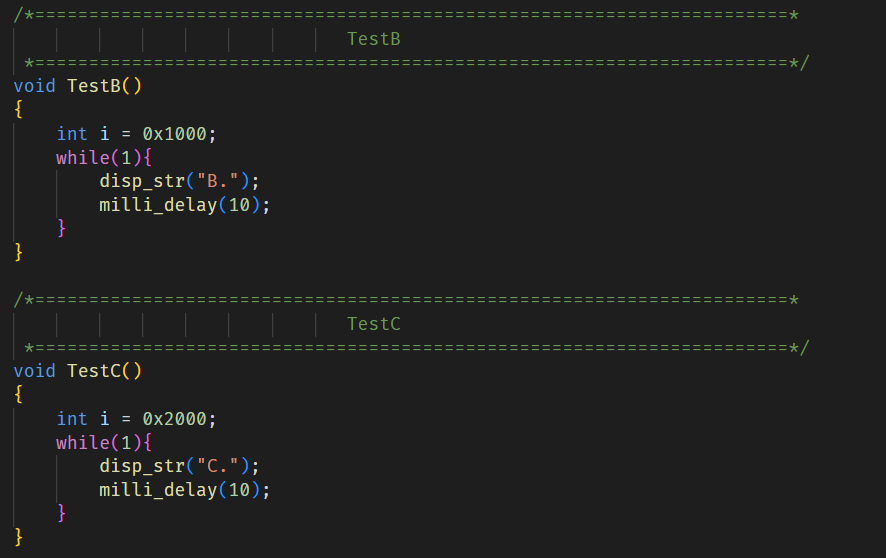
对于前者，首先，我们需要创建一个task\_table来管理多个进程，使得进程可以有序地装载到进程表中；对于不同的进程，我们需要定义并实现进程对应的功能，如打印不同的字符等；接下来，我们设置每一个进程的PCB、LDT等项，并为添加的进程分配STACK\_SIZE；还有一些宏也需要改变，譬如NR\_TASKS。

对于后者，我们可以通过时钟中断来实现进程切换。在时钟中断之前，程序保存现场；之后切换到内核栈，在内核栈中我们进行进程切换，也就是将p\_proc\_ready赋予进程队列中的下一个进程；之后，恢复之前保存的现场。

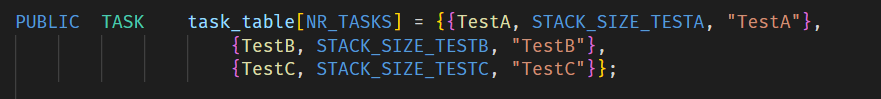
我们以实验代码来分析多进程相对于单进程扩展的部分：

* 添加新的进程函数：

在main.c中，程序添加了两个新的进程体TestB与TestC：

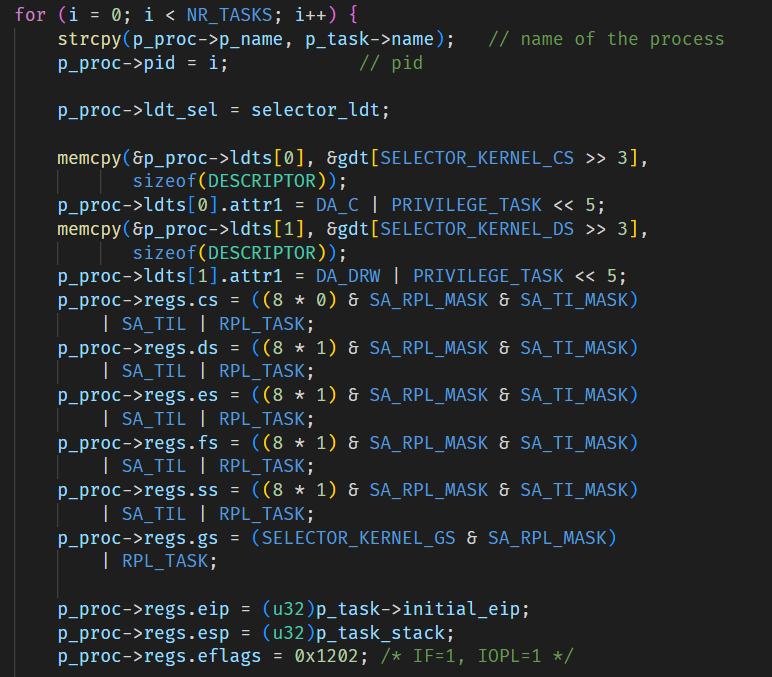


* 创建一张进程表，将TestA、TestB与TestC加载进来：



其中，NR\_TASKS为3。

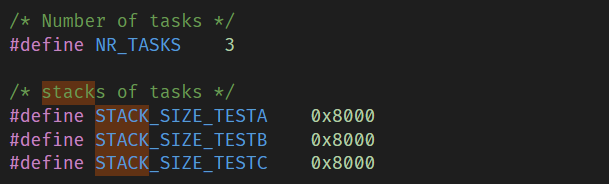
* 将三个进程分别进行PCB初始化：



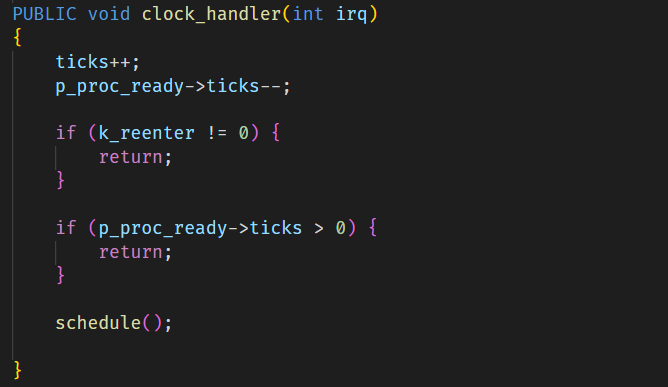
* 对每个进程设置LDT：

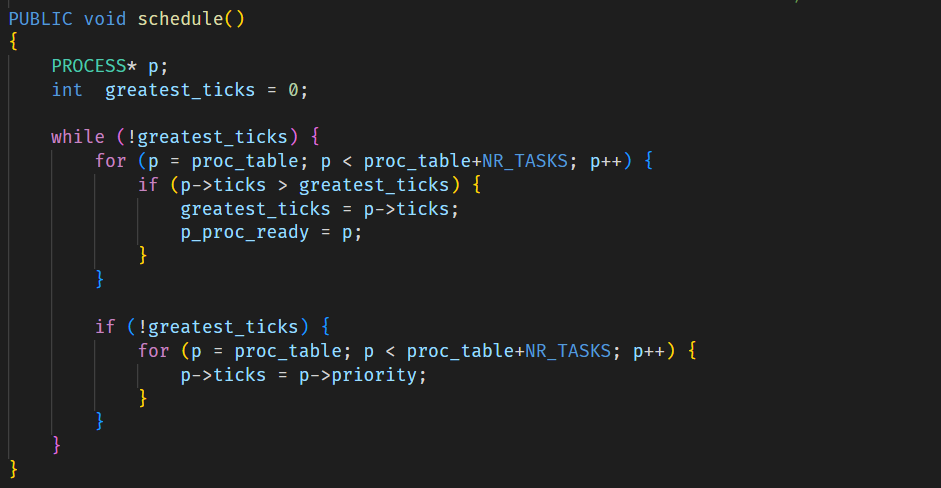


* 为每个进程设置堆栈段：



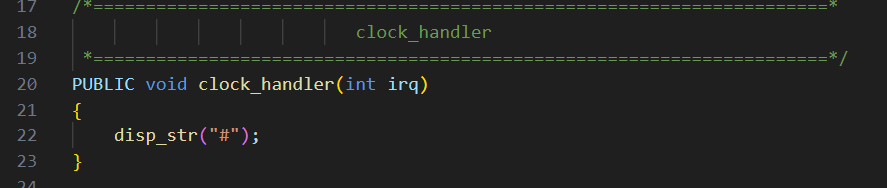
* 修改clock\_handler，使其支持多进程，具体来说，也就是增加schedule调度函数，将p\_proc\_ready赋值给下一个进程。在该部分，我们可以自由定义我们的调度方案，如FIFO、时间片轮转等：



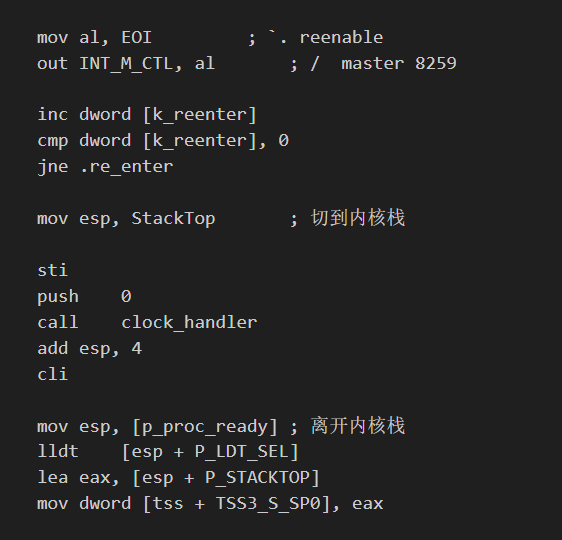


## 如何修改时钟中断来支持多进程管理，画出新的流程图。

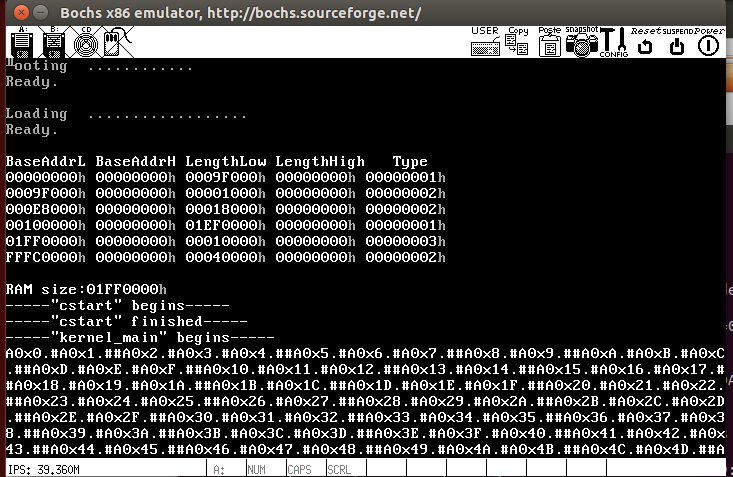
1. 创建一个 clock.c 。目前 clock.c 里面只有一个函数，且只有一个打印字符 # 的语句：



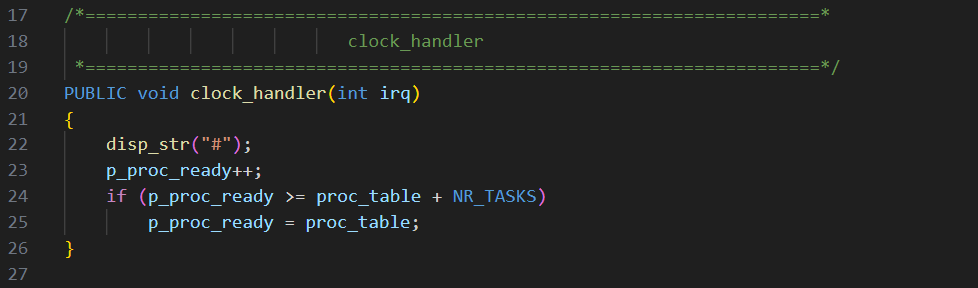
1. 在kernel.asm的时钟中断例程中调用这个函数：



1. 修改Makefile。 make ，运行。图中打印的字符 # 说明我们刚刚所增加的代码已经在正确运行。

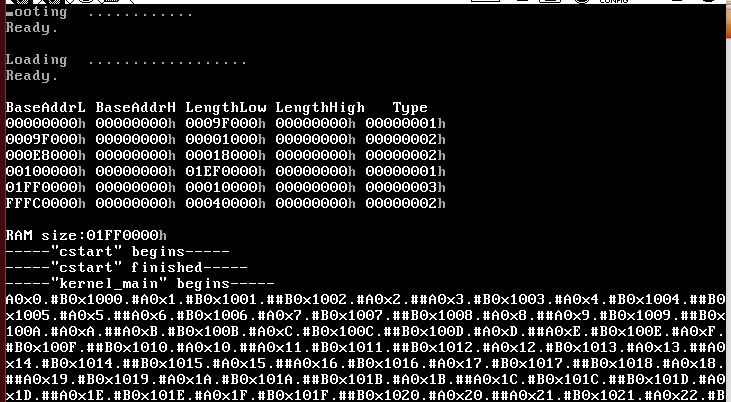


1. 进行进程切换，修改clock.c为：



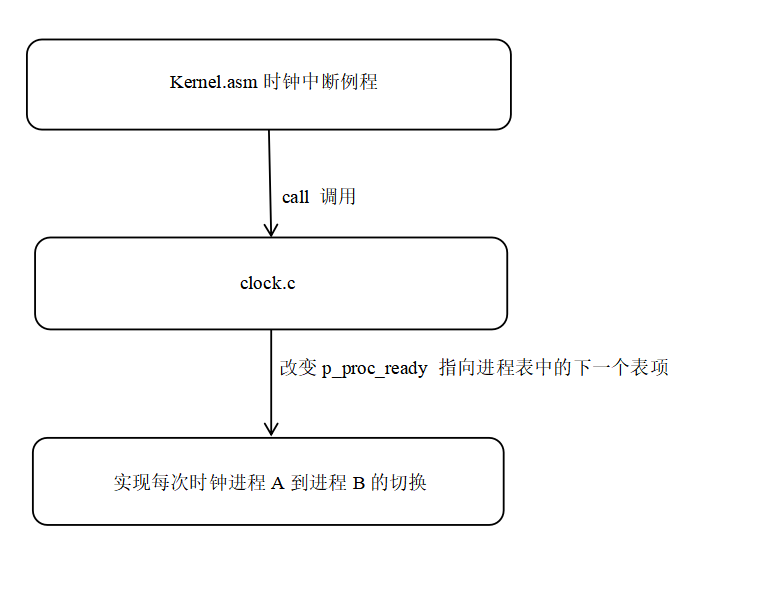
每一次我们让 p\_proc\_ready 指向进程表中的下一个表项，如果切换前已经到达进程表结尾则回到第一个表项。

1. 运行，完成了进程切换：



我们看到了交替出现的 A 和 B ，还有各自不断增加的数字。这表明我们的第二个进程运行成功了，我们已经成功实现了多进程。

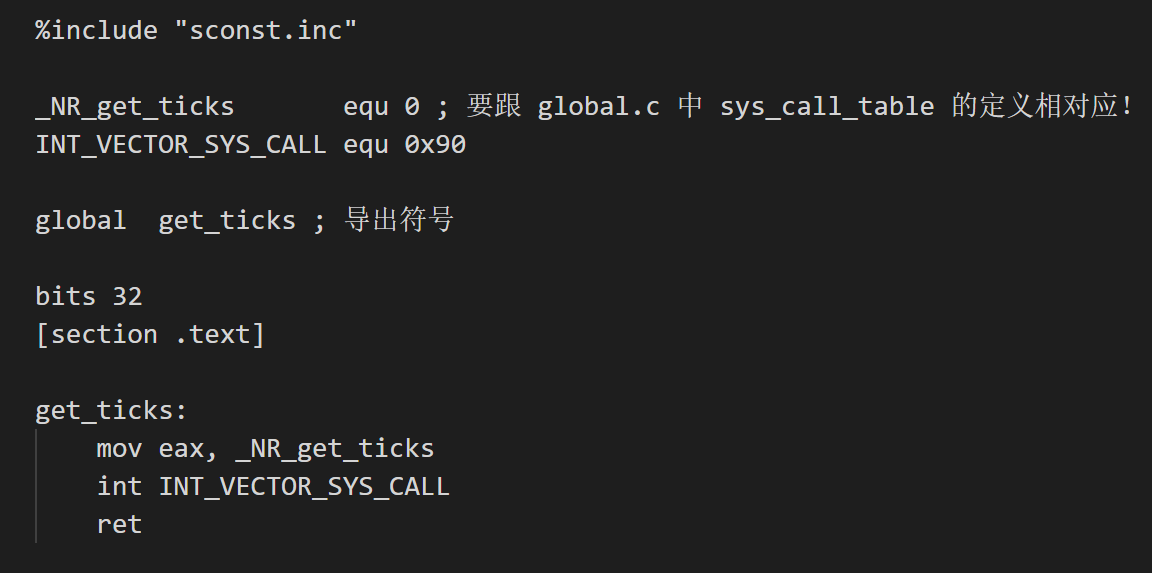
1. 流程图:



## 实现一个获取时钟ticks的系统调用

我们将实现一个 int get\_ticks( ) 的函数，用来得到当前总共发生了多少次时钟中断。设置一个全局变量 ticks ，每发生一次时钟中断就加一。进程可以随时通过这个系统中断得到 ticks

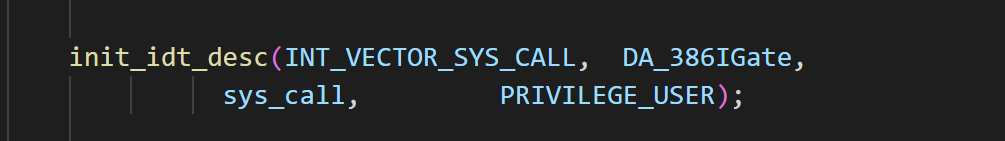
使用中断来实现系统调用。我们的get\_ticks()如下：



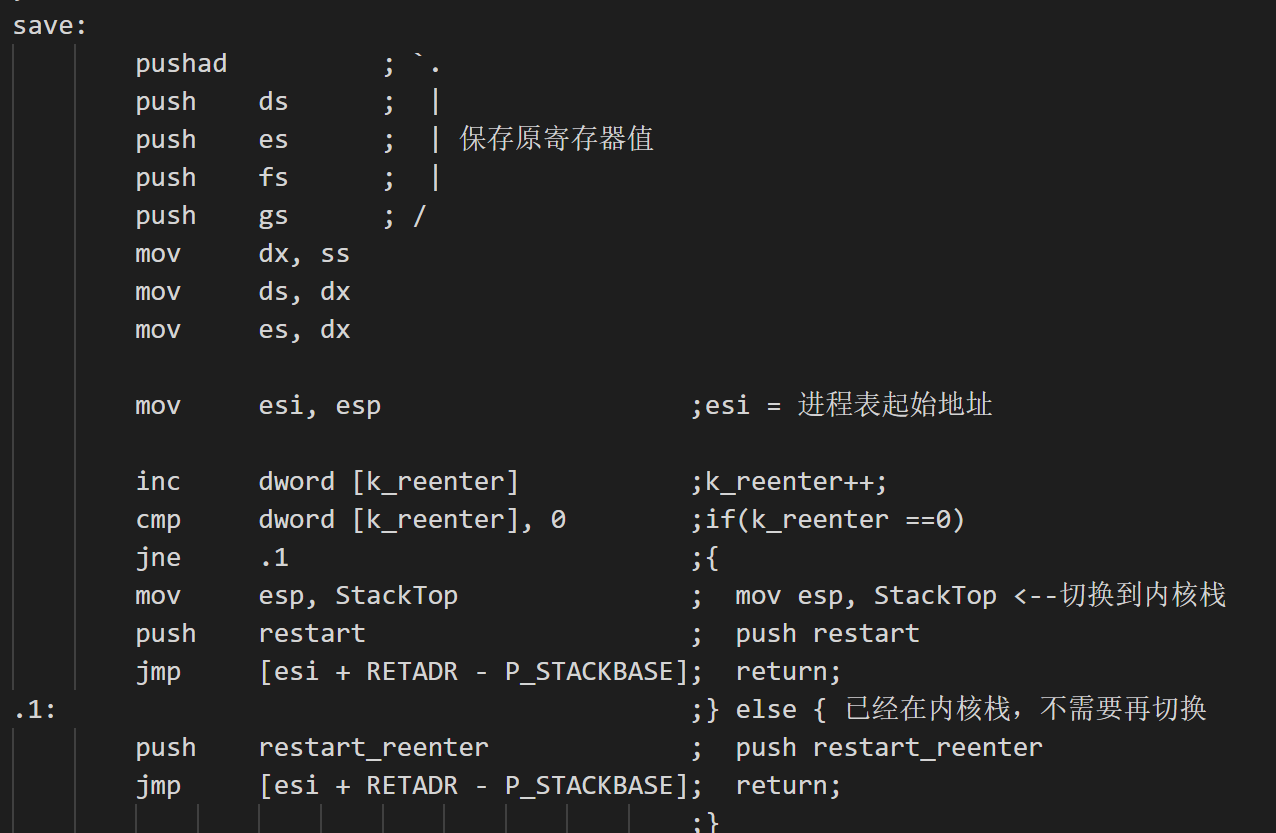
将eax的值赋予\_NR\_get\_ticks ，OS看到当前的eax是NR\_get\_ticks，就知道问题是"请问当前的ticks是多少"；

系统中断号设为0x90，只要不和原来的中断号重复即可；

接着在init\_prot()中，定义INT\_VECTOR\_SYS\_CALL对应的中断门，将这号中断和sys\_call对应起：

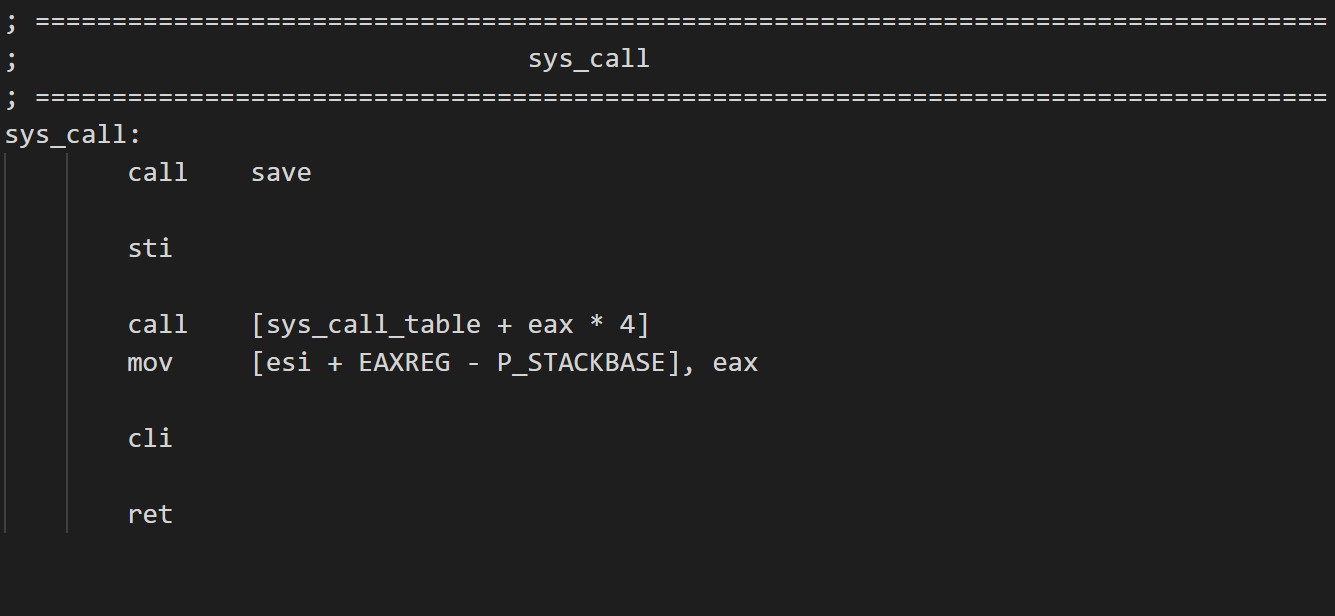


因为在之前eax的值被赋予了NR\_get\_ticks，所以save中不能再继续使用该寄存器，所以将 eax 全部改为esi：

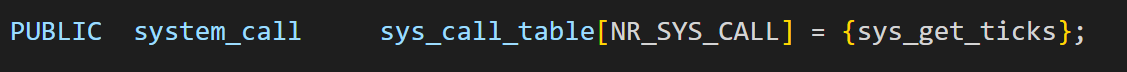


写的sys\_call如下， call [sys\_call\_table+eax\*4]（调用的是 sys\_call\_table[eax] ）。sys\_call\_table是一个函数指针数组，每个成员都指向一个函数，用于处理对应的系统调用。

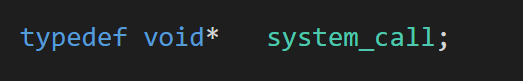
 



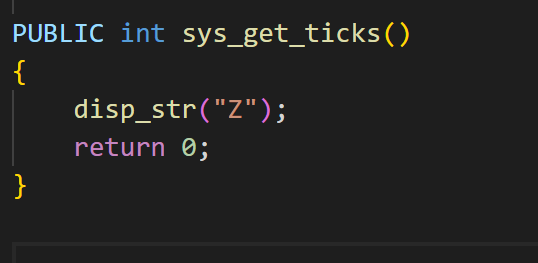
sys\_call\_table 定义在 global.c 中：



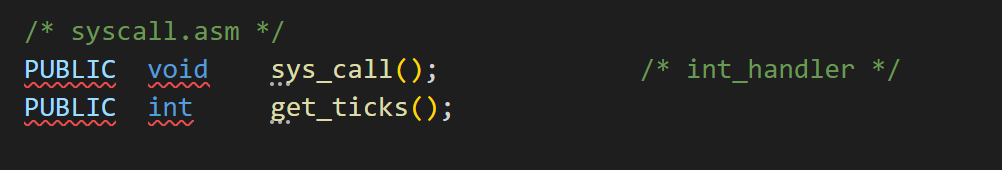
system\_call 定义在 type.h 中：



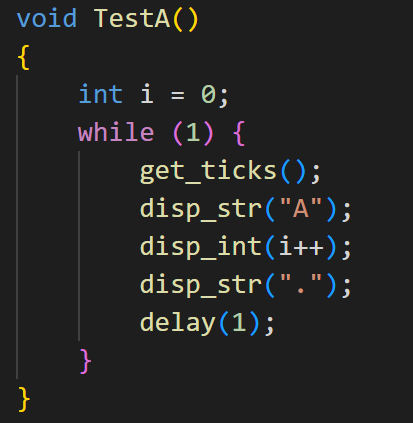
前面eax已被赋值为NR\_get\_ticks，而sys\_call\_table[0]已经初始化为sys\_get\_ticks，所以 call [sys\_call\_table+eax\*4]这一句调用的便是sys\_get\_ticks 。由于ticks与进程相关，单独建立一个文件 proc.c ，把 sys\_get\_ticks 放在里面。该函数打印一个字符z后就返回：



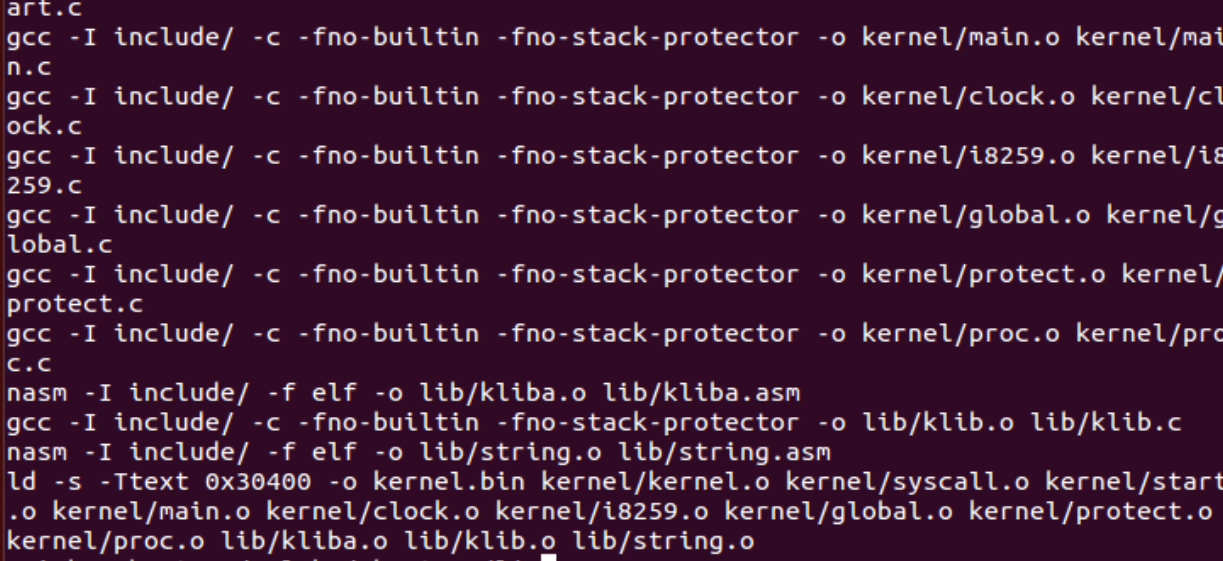
下面在proto.h中添加函数声明：



现在在进程中添加调用get\_ticks 的代码：在TestA中添加如下语句



修改Makefile（增加了一个文件proc.c），make

并运行，结果如图所示：



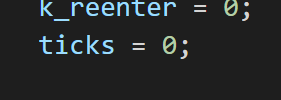
这里，加号出现在'A'的前面，所以第一个系统调用成功了

接着改进函数sys\_get\_ticks( )，实现真正的效果，返回当前的 ticks

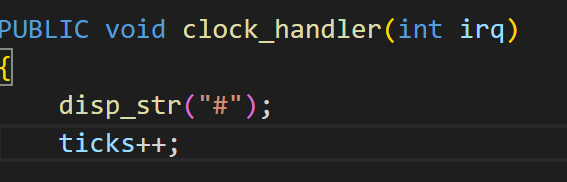
先在 global.h 中定义全局变量：



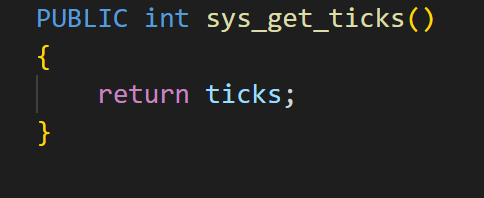
在 main.c 中初始化ticks：



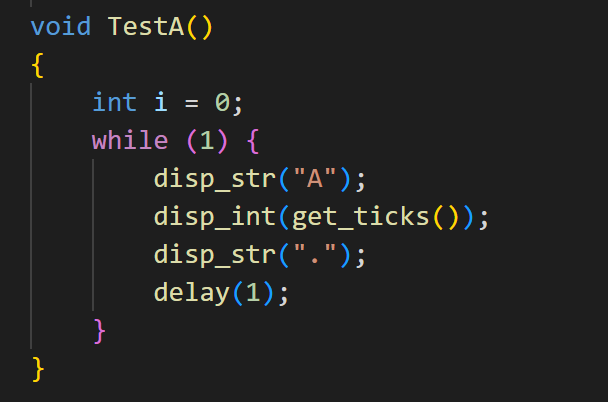
在clock\_handler(int irq)中添加：



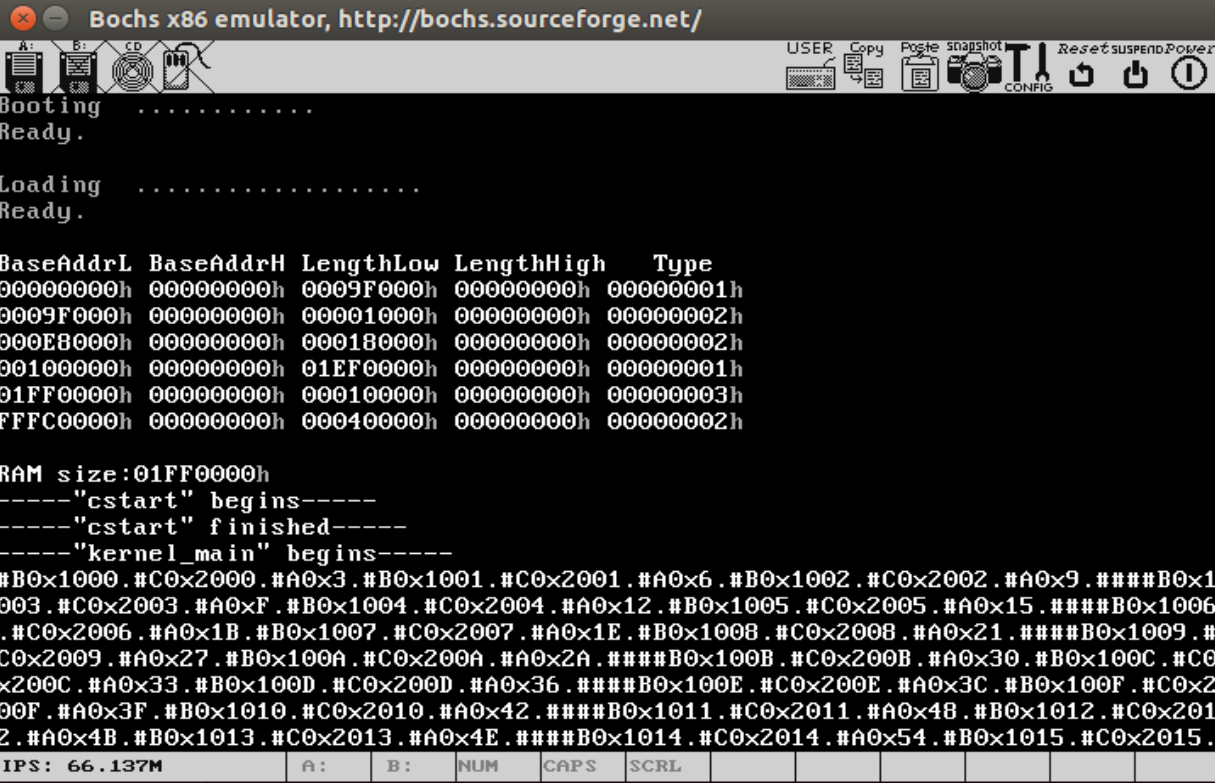
修改sys\_get\_ticks()：



最后修改 TestA()，打印当前的ticks：



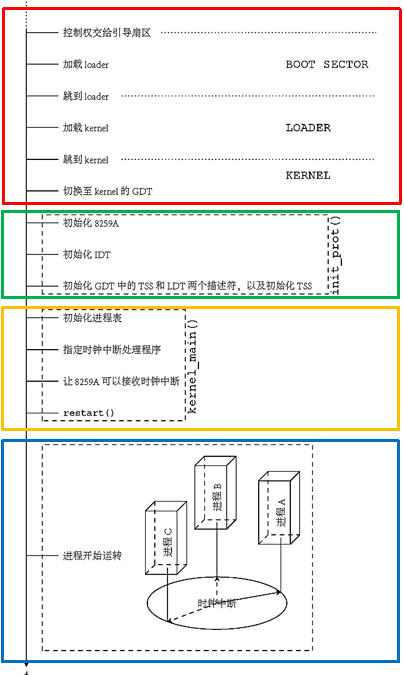
实验结果如下：



发现每次打印的“#”刚好是差值

## 进程调度的框架是怎样的？优先级调度如何实现？

1. 多进程调度框架包含以下几个方面：如何实现引导扇区、Loader加载、Kernel加载、转换控制权给Kernel、初始化中断控制器、初始化进程管理模块、多进程调度



1. 优先级调度的实现方法：对导读算法稍作修改

过去，我们在发生时钟中断选择下一个执行的进程时，直接选择进程表中的下一个进程，这种时间片轮转的方式给了每个进程均等的机会。我们现在不再给每个进程以相等的机会了。具体的方法是，给每一个进程都添加一个变量（可以放在进程表中），在一段时间的开头，这个变量的值有大有小，进程每获得一个运行周期，这个变量就减1，当减到0时，此进程就不再获得执行的机会，直到所有进程的变量都减到0为止。这样，每个进程获得的执行时间就不一样了。

## 基础题：在已有实验代码基础上，实现分时的进程调度和事件触发的进程调度。

### 三个用户进程A、B、C功能分别是打印出“hello,I am proc A/B/C”

### 系统启动先运行A，然后通过键盘中断切换到B运行；B运行10个时钟周期后，切换到C；然后通过自定义的系统调用切换回A。

### 实现思路一：

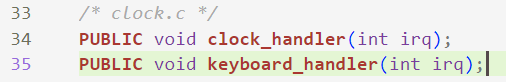
注：以下代码是基于chapter6/o的代码修改而来的

* 步骤一：添加键盘中断

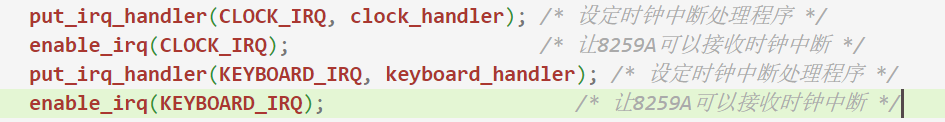
1. 首先，修改clock.c文件，添加键盘中断函数。在时钟中断函数下，平行添加一个键盘中断函数，具体代码先从时钟中断处移植：



另外，在proto.h中添加函数声明：

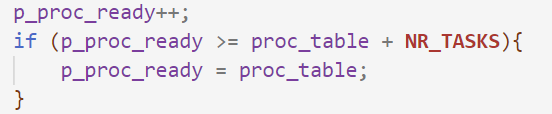


1. 打开键盘中断：修改main.c文件，在打开时钟中断处，并行添加类似的函数调用，并传入已经定义的键盘中断和上一步写好的键盘中断处理函数：



1. 暂时取消时钟中断切换进程的功能：修改clock.c文件

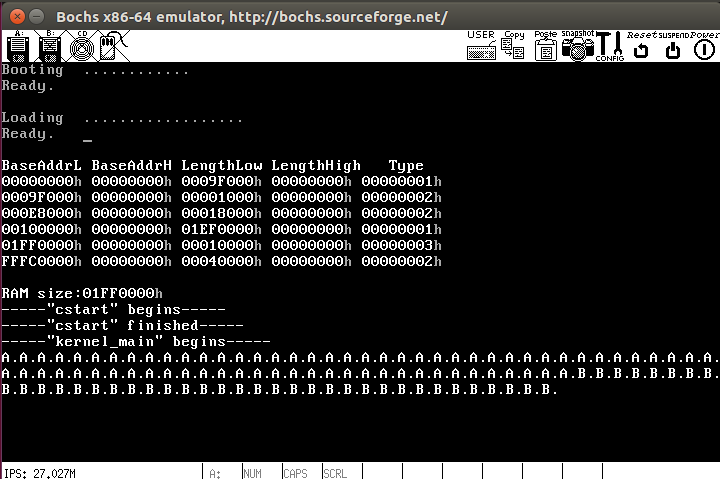
中断切换进程的功能在中断处理函数中，其具体操作方式是将指示进程的指针自增来指向下一个进程，并检测是否已经达到进程表末尾，若达到则返回第一个进程执行，这段代码如下：



由于我们要暂时关闭时钟中断对进程切换的影响，因此在时钟中断处理函数中将这一段函数注释掉，在键盘中断中仍然保留，修改后的时钟中断与键盘中断如下：

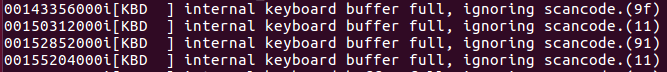
|  |  |
| --- | --- |
|  |  |

截止目前，我们已经可以将原本的时钟中断控制进程切换修改为键盘中断控制进程切换，此时我们已经能够通过键盘输入实现之前的效果，运行如下：

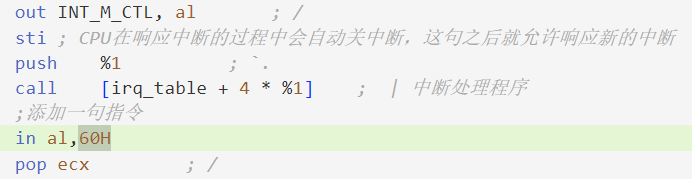


1. 读取键盘缓冲区

但是我们却遇到了问题，目前我们能够通过键盘输入实现A->B切换，但无法执行再一次的切换，观察发现，如果连续按多次键盘，bochs会显示keyboard buffer overflow，即键盘缓冲区溢出：

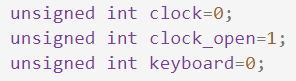


看来每次键盘输入后，都需要通过in al,60H命令来读一下缓冲区，才能继续接受下一次输入，因此我们把这段代码加到中断处理汇编代码处，加入读缓冲区操作后，就再也不会出现键盘中断只能执行一次和缓冲区溢出的情况了，修改kernel/kernel.asm文件：



1. 定义中断控制全局变量

定义了三个全局变量用于控制键盘中断和时钟中断的开启和关闭，修改clock.c文件，



keyboard和clock初值设置为0，clock\_open设置为1，keyboard=0时允许键盘中断切换进程，clock\_open=0时允许时钟中断切换进程，clock变量用于时钟中断计数（用于B进程转C）。

在键盘中断处理函数中，检测当前的keyboard值，此值为0时令键盘中断切换进程；键盘中断执行一次后，令keyboard自增，即仅允许键盘中断切换一次进程，并将clock\_open置为0，即开始允许时钟中断切换进程。

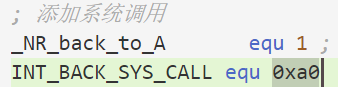
在时钟中断处理函数中也进行判定，当clock\_open==0时开始对时钟中断进行计数，计数达到10时进行进程切换（进程B切换至进程C），这里用1000表示，运行效果更明显。

|  |  |
| --- | --- |
|  |  |

至此，我们完成了由A通过键盘中断切换至B执行，十个时钟周期后又由B切换至C执行的过程。接下来是要添加一个系统调用。

* 步骤二：添加系统调用

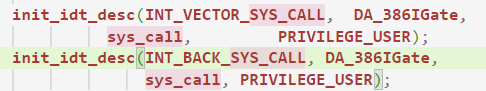
1. 首先，在syscall.asm中添加系统调用的基本定义：



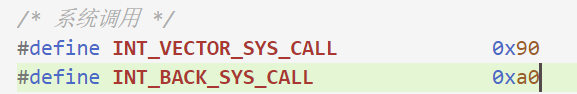
\_NR\_back\_to\_A是对应的系统调用号，INT\_BACK\_SYS\_CALL是自己定义的端口号，不与之前的值重合即可，接下来写函数back\_to\_A，这段函数就是将系统调用编号和端口设置为当前函数，之后导出符号：

|  |  |
| --- | --- |
|  |  |

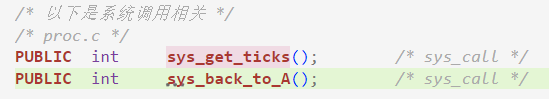
1. 为了真正将调用实现，需要定义这个系统调用的中断门，在protect.c中做如下修改：



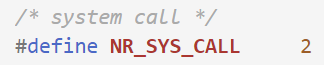
1. 此处使用到的INT\_BACK\_SYS\_CALL需要在protect.h文件进行预定义：



1. 还需要在proto.h声明sys\_back\_to\_A函数：



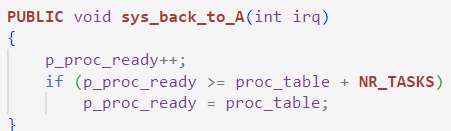
1. 添加了一个系统调用函数，所以我们需要在 include/const.h 中 修改 NR\_SYS\_CALL 个数：



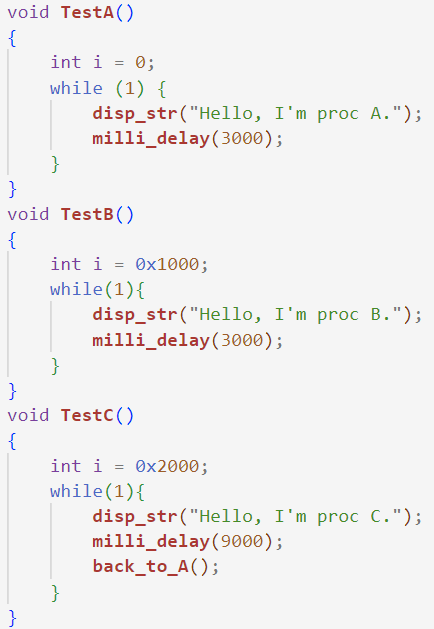
1. 还有一个重要步骤是要在系统调用“数组”中添加属于我们系统调用的一项，这个定义在global.c文件中：



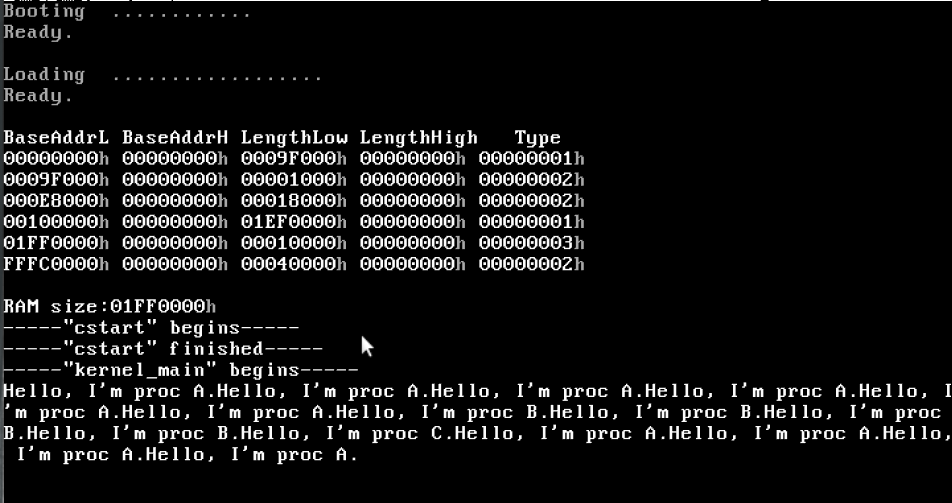
1. 添加一项后，再修改真正的系统调用执行函数，这个函数定义在proc.c中，我们将中断处理函数中切换进程的代码一直到这个函数体中就可以了：



1. 最终，在main.c进程函数体中，修改TestA、TestB和TestC函数以打印指定字符串，修改延迟以便显示，并在程序C执行这个系统调用：



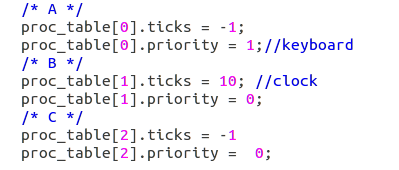
最终执行结果，连续输出某一句话就说明在持续执行某进程体，可见，A切换为B，过一段时间（运行时我设置成了1000个时钟周期，方便观察）切换为C，然后很快自动切换回A：

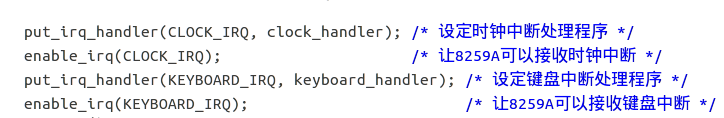


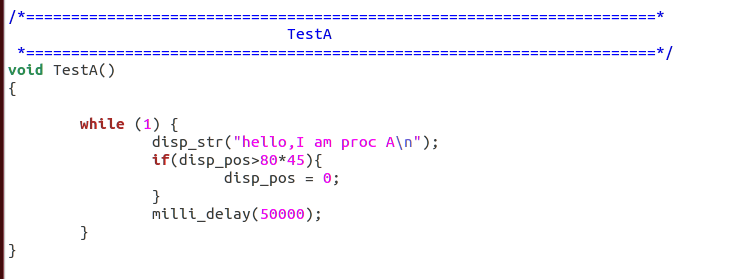
### 实现思路二：

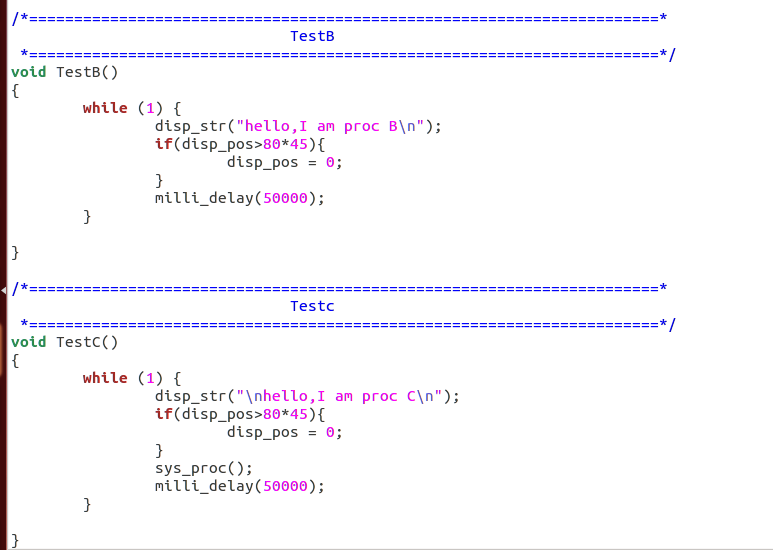
注：以下代码是基于chapter6/i的代码修改而来

1、修改main.c，添加A、B、C进程代码和相关设置

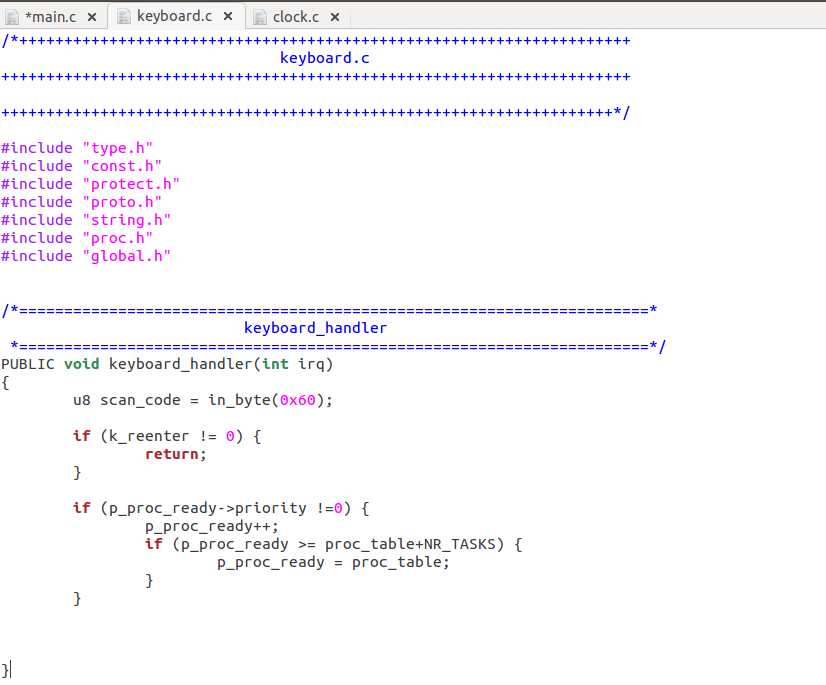






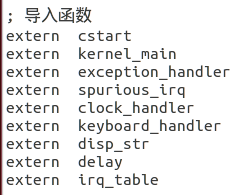


2、创建keyboard.c，编写keyboard\_handler函数以实现键盘中断



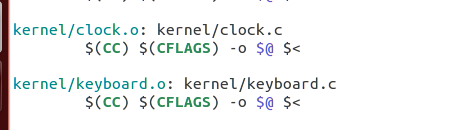
首先用in\_byte读取扫描码（防止输入缓冲区溢出）。然后判断k\_reenter来防止中断重入（因为同时开了时钟中断）。然后判断当前进程的priority属性是否等于0，根据前面设置的优先级，这样只有A进程能够接受键盘中断的切换进程的操作。

3、修改kernel.asm，添加函数keyboard\_handler的导入



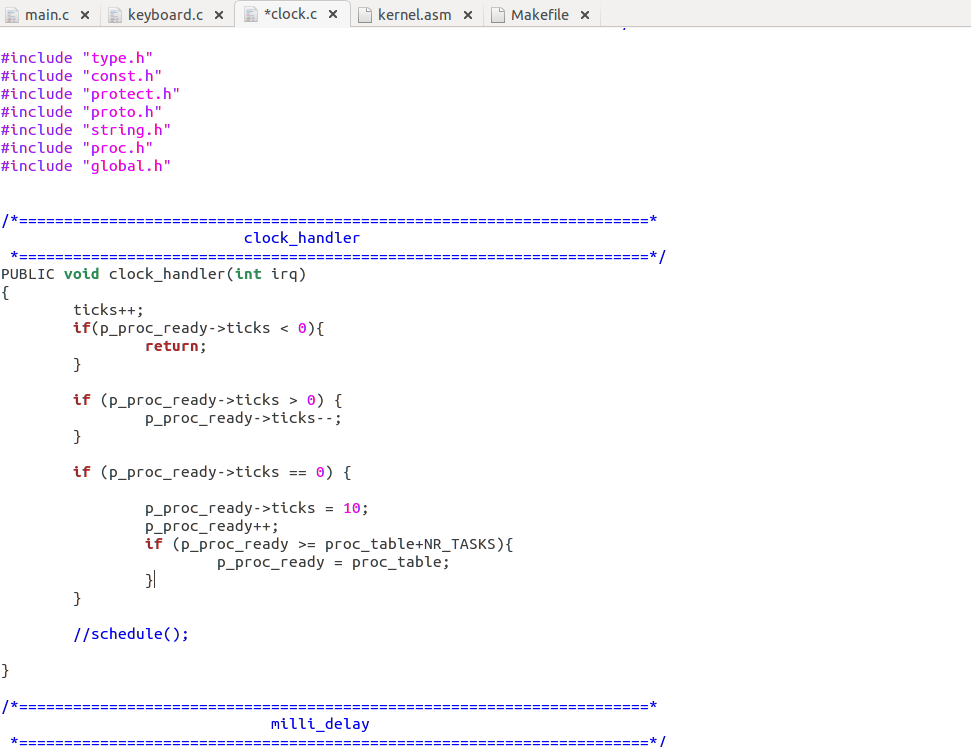
4、修改makefile文件





这样，就完成了一个键盘中断的添加，并能从A进程通过键盘中断切换到B进程。

5、在clock.c文件中，修改时钟中断的处理函数

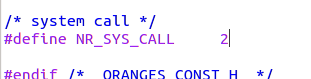


首先全局变量ticks++，这是为延时函数准备的。

然后，对于当前进程的ticks的判断与前面键盘中断代码中的优先级的判断是类似的。B进程在初始化时设置的时间片为10，而其它进程的时间片设置为-1。当本进程时间片不为0时，则将时间片减1，相当于运行了一个时钟周期；当时间片为0时，重新赋予当前进程10个时间片，并且切换到下一个进程。

这样，就完成从进程B运行十个时间片然后切换到进程C的过程，接下来需要添加一个系统调用，使进程C切换回进程A，使得整个程序循环运行。

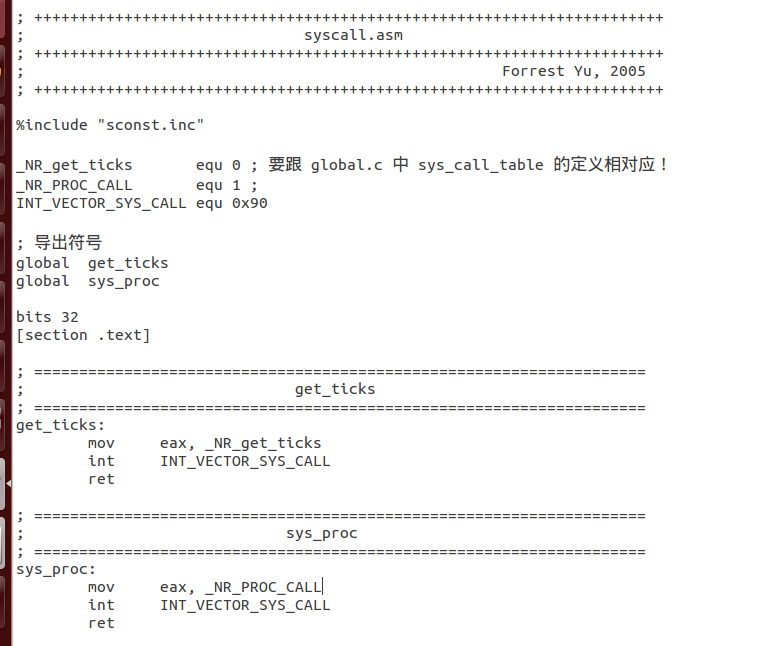
6、因为要增加一个系统调用，在const.h中修改系统调用号的总数：



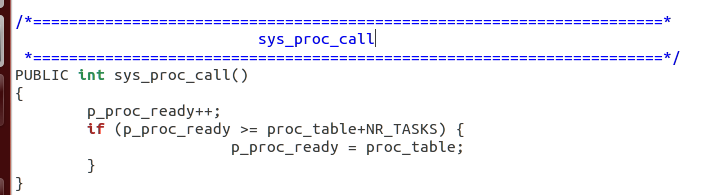
7、在global.c中修改系统调用表：

IMG_264

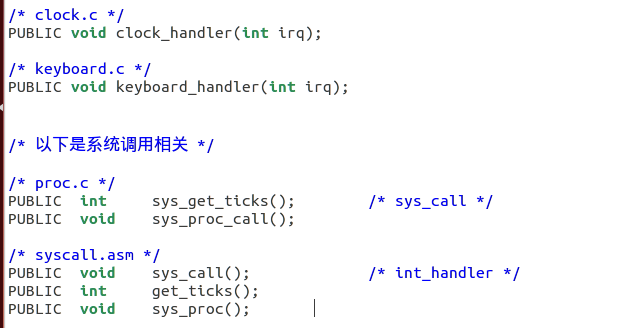
8、在syscall.asm添加对应的系统调用号的对应处理函数sys\_proc



9、在proc.c添加sys\_proc\_call来切换进程：

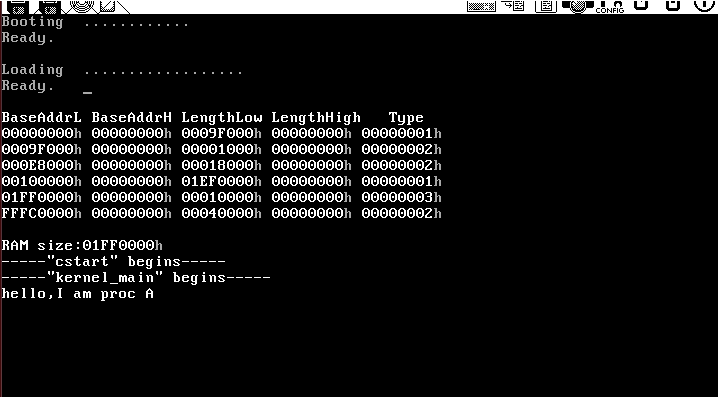


10、在proto.h声明相应函数：

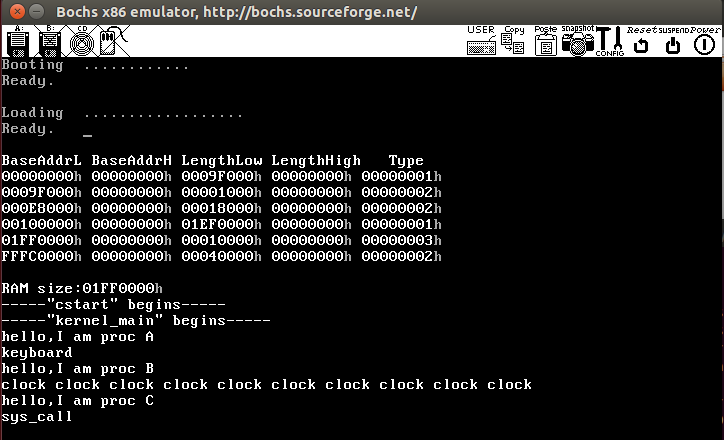


**实现效果：（添加了打印字符以观察键盘中断发生，时间片减少，以及系统调用发生）**

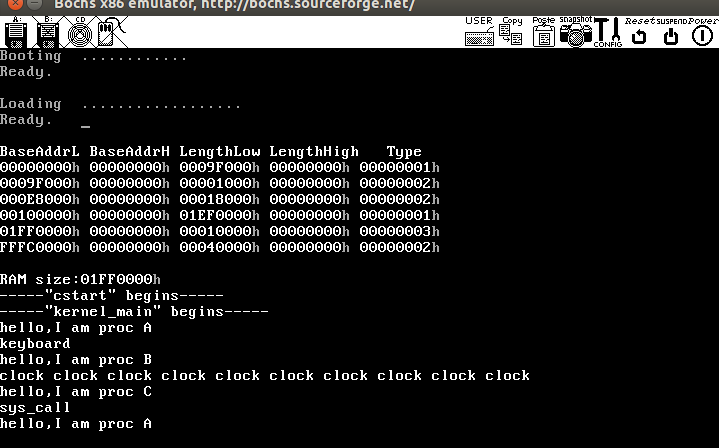
**开始时运行进程A：**

****

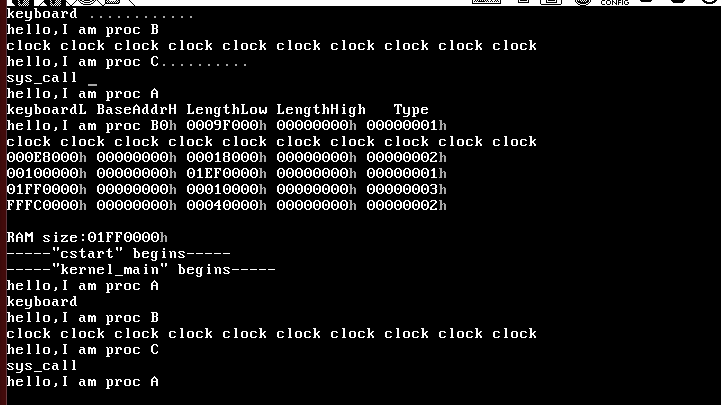
**按下任意键，运行进程B，十个时钟后（打印了clock体现），运行进程C**

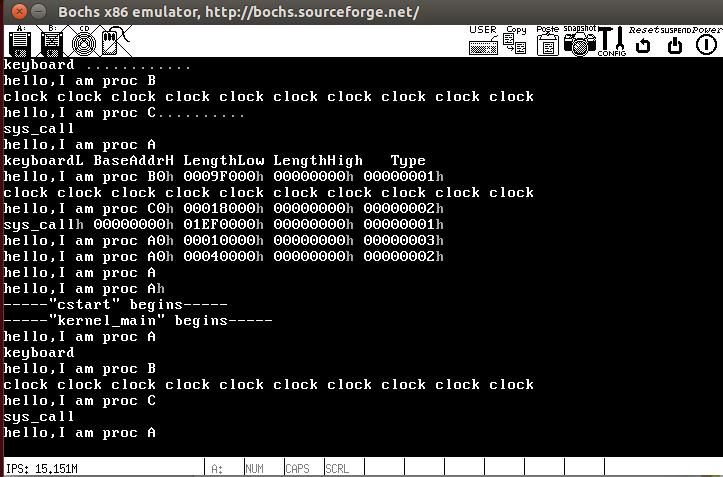


**通过自定义的系统调用，返回进程A：**



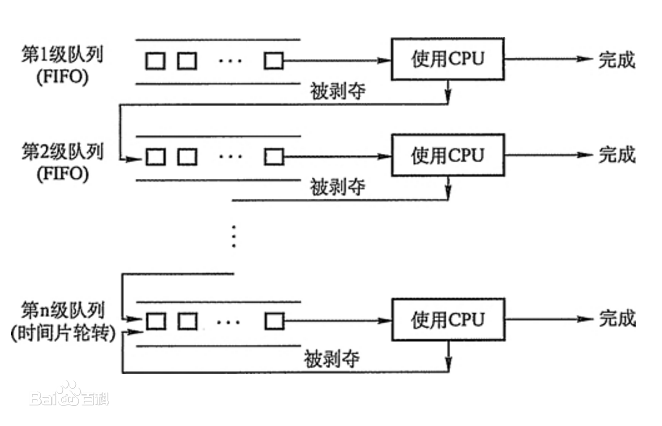
**这三个过程可以循环进行：**





## 选做题：修改课本的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度多个任务。

多级反馈队列调度算法是一种CPU处理机调度算法，它既可以使高优先级的作业得到响应、又可以使短作业迅速完成。UNIX系统使用了该调度算法。

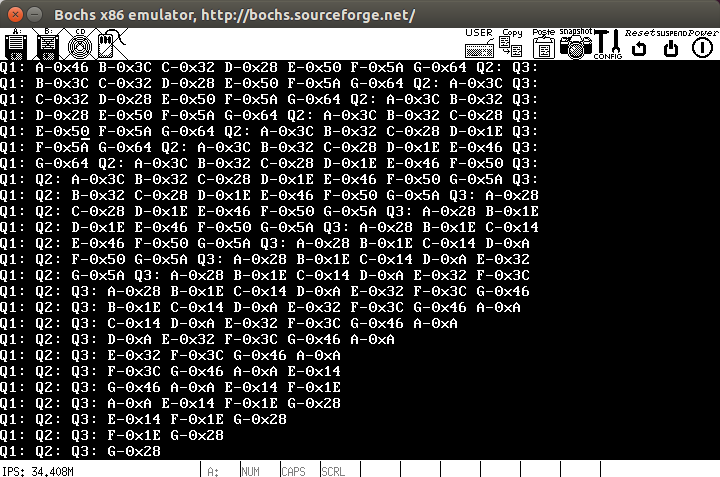


算法过程如下：

* 设置三个进程队列（本实验设置，也可以为多个）Q1、Q2、Q3，优先级Q1 > Q2 > Q3，而进程队列的时间片大小Q1 < Q2 < Q3。除了最低优先级的队列使用时间片轮转算法，其他队列均使用FIFO算法；
* 对于CPU而言，CPU优先调度高优先级队列中的进程，当高优先级队列中进程数为0，CPU转而调用较低优先级队列中的进程：也就是，当Q1为空时，CPU调度Q2中的进程，当Q1和Q2为空时，CPU调度Q3中的进程。注意，假如CPU在处理Q3的过程中，Q1新加入了一个进程，那么CPU会将正在处理的进程放在Q3队尾，转而处理Q1中的进程；
* 对于进程而言，刚到达的进程会进入最高优先级的队列，如果在对应时间片内进程没有执行完毕，进程会进入较低优先级的队列；长此以往，直到进入最低优先级的Q3队列，然后进程会一直处于Q3中直到被执行完毕。

基于此，该部分会根据实验代码中的chapter6/r/进行修改，以实现多级反馈队列调度算法。我们实现的多级反馈调度算法具有三个优先级队列，其时间片大小按照优先级从高到低，为10、20、30，单位为tick。每一个队列的调度算法如上所示。我们最终实现了7个进程的调度，其需要的时间分别为：A-70、B-60、C-50、D-40、E-80、F-90、G-100，单位也为tick。我们首先展示最终调度算法的结果，然后介绍我们实现调度算法的细节。

具体调度细节如下所示，开始时，七个进程同时到达Q1队列，并依次进入Q1。每一个进程的剩余时间用十六进制的方式进行呈现。第一行，A的剩余时间为0x46=70，经过Q1的一个大小为10 tick的时间片，A的剩余时间为0x3C=60，并进入Q2队列。之后，当Q1队列被清空时，A再次被调度，经过大小为20 tick的时间片后，被送入Q3队列，此时剩余时间为0x28=40。可以看出，屏幕输出的结果符合上述多级反馈队列调度的特征。



接下来，我们介绍多级反馈调度算法的流程。

比起实际上构造出三个队列，我们认为，在进程结构体中增加一些字段来模拟队列更为简便。我们在/include/proc.h的s\_proc(Process)结构体中添加如下字段：

**int queue; // 该进程处于哪一个队列**

**int pos; // 该进程处于队列中哪一个位置**

**int ft; // Q1中的剩余tick数，初始化为Q1时间片大小**

**int st; // Q2中的剩余tick数，初始化为Q2时间片大小**

**int tt; // Q3中的剩余tick数，初始化为Q3时间片大小**

其中，ft、st、tt看似有三项需要考虑，其实因为一个进程一个时间段内只能处于一个队列中，因此其中只有一项会被使用。queue=1，即进程在Q1时，我们只会考虑ft，而不考虑st和tt。当进程转到Q2时，我们也会再次对st进行初始化。

该结构体代表了一个进程。具体来说，它会在/kernel/main.c文件中被实例化为一张进程表，也就是proc\_table：

**PUBLIC PROCESS proc\_table[NR\_TASKS];**

**PROCESS\* p\_proc = proc\_table;**

其中，NR\_TASKS为7，表示该进程表中存在7个进程。进程名称为：

**PUBLIC TASK task\_table[NR\_TASKS] =**

**{{TestA, STACK\_SIZE\_TESTA, "A"},**

**{TestB, STACK\_SIZE\_TESTB, "B"},**

**{TestC, STACK\_SIZE\_TESTC, "C"},**

**{TestD, STACK\_SIZE\_TESTD, "D"},**

**{TestE, STACK\_SIZE\_TESTE, "E"},**

**{TestF, STACK\_SIZE\_TESTF, "F"},**

**{TestG, STACK\_SIZE\_TESTG, "G"}};**

我们分别对这些进程赋值，具体代码为：

**for (i = 0; i < NR\_TASKS; i++) {**

**p\_proc->pid = i;**

**p\_proc->queue = 1; //进程一开始进入Q1**

**p\_proc->pos = i; // 确保进程依次进入Q1**

**p\_proc->ft = 10;**

**p\_proc->st = 20;**

**p\_proc->tt = 30;**

**}**

表示该进程最开始被送入队列Q1，所有进程在Q1队列中的位置依次增大，初始在Q1、Q2、Q3中的时间片大小为10、20、30。

为了更好对队列进行操作，我们还初始化一些全局变量，它们代表每个队列中的进程数、以及每个队列中的时间片大小。

**first\_num = 7;**

**second\_num = 0;**

**third\_num = 0;**

**first\_ticks = 10;**

**second\_ticks = 20;**

**third\_ticks = 30;**

之后，我们为进程表中的每一个进程赋予一个任务时间，也就是该进程需要的tick总量。

**proc\_table[0].ticks = proc\_table[0].priority = 70;**

**proc\_table[1].ticks = proc\_table[1].priority = 60;**

**proc\_table[2].ticks = proc\_table[2].priority = 50;**

**proc\_table[3].ticks = proc\_table[3].priority = 40;**

**proc\_table[4].ticks = proc\_table[4].priority = 80;**

**proc\_table[5].ticks = proc\_table[5].priority = 90;**

**proc\_table[6].ticks = proc\_table[6].priority = 100;**

然后，将proc\_table赋给p\_proc\_ready，也就是将第一个进程置于Q1队首，进程调度的准备工作就做好了。

进程调度的核心在于clock\_handler以及schedule，具体来说，每一次首先调用时钟中断，如果此时发生了中断重入，也就是k\_reenter不为0，则直接退出。

否则，首先检查当前进程是否已经用完了其所在队列的一个时间片，以及当前进程是否已经执行完毕，当进行消耗所有的时间片、或是已经执行完毕时，我们必须调度下一个进程。我们使用三个if语句来依次检查三个队列，目的只是为了找出进程所处的是哪一个队列，之后在队列中再进行上述两个判断。假设进程处于Q1队列，p\_proc\_ready->ft > 0判断的是当前进程在当前队列中的时间片大小，p\_proc\_ready->ticks判断的是当前进程的剩余时间。假如进程剩余时间片充足、且没有执行完毕，则执行一个tick的时间，也就是将进程总时间与进程在队列中的剩余时间片分别减1。否则，我们需要调用scheduler来调度下一个进程。

下面的代码描述的即是上述过程。

**if (p\_proc\_ready->ft > 0 && p\_proc\_ready->queue == 1) {**

**if (p\_proc\_ready->ticks > 0) {**

**p\_proc\_ready->ft--; // 对队列而言，进程又使用了一点点时间片**

**p\_proc\_ready->ticks--; // 对进程而言，又完成了一点点任务**

**return;**

**}**

**}**

**if (p\_proc\_ready->st > 0 && p\_proc\_ready->queue == 2) {**

**if (p\_proc\_ready->ticks > 0) {**

**p\_proc\_ready->st--;**

**p\_proc\_ready->ticks--;**

**return;**

**}**

**}**

**if (p\_proc\_ready->tt > 0 && p\_proc\_ready->queue == 3) {**

**if (p\_proc\_ready->ticks > 0) {**

**p\_proc\_ready->tt--;**

**p\_proc\_ready->ticks--;**

**return;**

**}**

**}**

**schedule();**

当if判断全部出错，我们则需要即刻调度下一个进程。同时，如果上一个进程没有执行完毕，我们就需要将该进程添加到低优先级的队列中；如果已经执行完毕，我们就需要删除该进程。添加与删除同样需要我们编写对应函数。

在delete\_proc操作中，我们首先将对应队列中的进程数减1，然后将上一个进程之后的所有进程向前移动1位；下面代码所展示的add\_proc操作不是必要的，因为它代表一个全新进程的加入，其功能是直接将进程放入Q1中，但我们所有的进程都是同一时间到达的。对于时间片用完的进程，我们在schedule函数本身进行添加操作。

**PRIVATE void delete\_proc() {**

**int queue = p\_proc\_ready->queue;**

**if (queue == 1) { // 把前一个进程丢掉**

**first\_num--;**

**} else if (queue == 2) {**

**second\_num--;**

**} else { // if (queue == 3)**

**third\_num--;**

**}**

**p\_proc\_ready->queue = 0; // 找到前一个进程的队列，每一个向前移动一位**

**PROCESS\* p;**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->queue == queue) p->pos--;**

**}**

**}**

**PRIVATE void add\_proc() {**

**p\_proc\_ready->ticks = p\_proc\_ready->priority;**

**p\_proc\_ready->ft = first\_ticks;**

**p\_proc\_ready->st = second\_ticks;**

**p\_proc\_ready->tt = third\_ticks;**

**p\_proc\_ready->queue = 1;**

**p\_proc\_ready->pos = first\_num;**

**first\_num++;**

**}**

schedule函数本质上来说就是将p\_proc\_ready易主。首先，我们会检测该进程是否已经执行完毕：如果执行完毕，直接将该进程从队列中删除；否则，将其移入下一级队列，增加下一级队列的进程数。然后，选取本级队列中队首的进程（如果不考虑抢占）作为即将被调度的进程赋给p\_proc\_ready，进行之后的调度工作。更准确地说，删除或移动进程之后，算法会再次从Q1检查到Q3，如果Q1有pos为0的进程，将其赋给p\_proc\_ready；如果没有，就再检查Q2、Q3中pos为0的进程。

为了考虑到调度过程的可视化，每一次调用schedule函数，我们都会调用一次disp\_queue函数进行队列信息的展示。

**PUBLIC void schedule()**

**{**

**if (p\_proc\_ready->ticks == 0) { // 进程执行完毕，直接删除**

**delete\_proc();**

**p\_proc\_ready->out\_ticks = get\_ticks();**

**finish\_proc\_num++;**

**} else if (p\_proc\_ready->queue == 1) { // 进程没有执行完毕，移动**

**delete\_proc();**

**p\_proc\_ready->queue = 2;**

**p\_proc\_ready->pos = second\_num;**

**second\_num++;**

**} else if (p\_proc\_ready->queue == 2) {**

**delete\_proc();**

**p\_proc\_ready->queue = 3;**

**p\_proc\_ready->pos = third\_num;**

**third\_num++;**

**} else {**

**delete\_proc();**

**p\_proc\_ready->queue = 3;**

**p\_proc\_ready->pos = third\_num;**

**third\_num++;**

**p\_proc\_ready->tt = third\_ticks;**

**}**

**if (finish\_proc\_num < NR\_TASKS) { // 打印队列，方便展示**

**disp\_q ();**

**}**

**PROCESS\* p; // 从Q1找到Q3，找pos=0的进程**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == 0 && p->queue == 1) {**

**p\_proc\_ready = p;**

**return;**

**}**

**}**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == 0 && p->queue == 2) {**

**p\_proc\_ready = p;**

**return;**

**}**

**}**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == 0 && p->queue == 3) {**

**p\_proc\_ready = p;**

**return;**

**}**

**}**

**}**

disp\_queue函数如下，我们分为Q1、Q2、Q3三个队列分别打印，首先，打印队列中每一个进程名称，然后是进程的p\_ticks，也就是剩余时间。每一次打印都会占据终端的一行。bochs终端大小为25\*80，我们首先打印25\*80个空格，意在清空终端的所有信息，然后从第一行开始打印信息；每次调用都会新起一行打印新的信息。

在结果展示中，因为调用次数过多，因此最后一部分的信息没有被完全打印；然而，前面部分的信息也足以了解到多级反馈队列的工作原理。

**PUBLIC void disp\_q() {**

**if (finish\_proc\_num == 7) return;**

**disp\_str("Q1: ");**

**for (int i = 0; i < first\_num; i++) {**

**PROCESS\* p;**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == i && p->queue == 1) {**

**disp\_str(p->p\_name);**

**disp\_str("-");**

**disp\_int(p->ticks);**

**disp\_str(" ");**

**}**

**}**

**}**

**disp\_str("Q2: ");**

**for (int i = 0; i < second\_num; i++) {**

**PROCESS\* p;**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == i && p->queue == 2) {**

**disp\_str(p->p\_name);**

**disp\_str("-");**

**disp\_int(p->ticks);**

**disp\_str(" ");**

**}**

**}**

**}**

**disp\_str("Q3: ");**

**for (int i = 0; i < third\_num; i++) {**

**PROCESS\* p;**

**for (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {**

**if (p->pos == i && p->queue == 3) {**

**disp\_str(p->p\_name);**

**disp\_str("-");**

**disp\_int(p->ticks);**

**disp\_str(" ");**

**}**

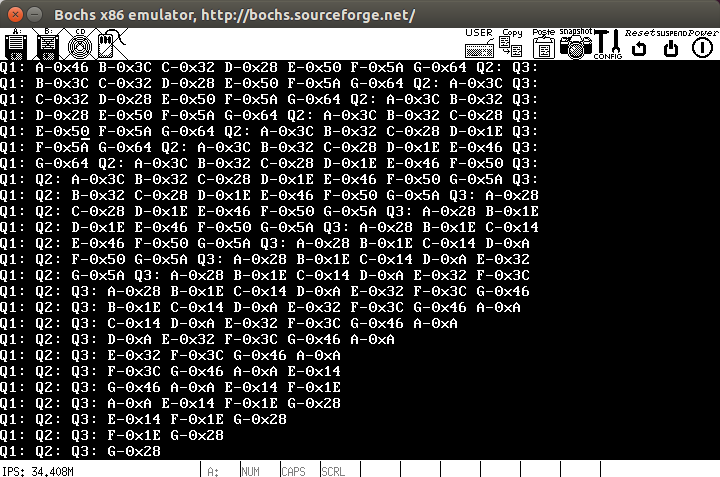
**}**

**}**

**disp\_str("\n");**

**}**

实验结果如下图所示：



# 实验结果总结

（对实验结果进行分析。并理论联系实际，思考并列出本实验对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。）

## 进程

### 进程的定义

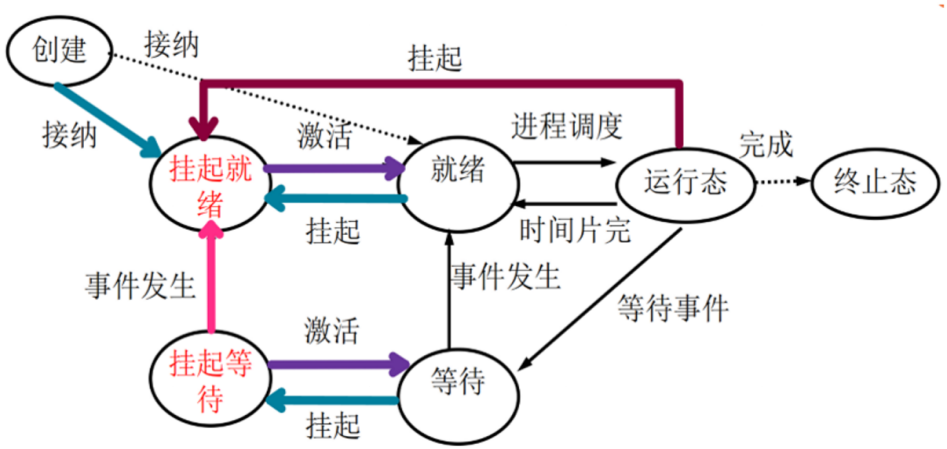
从理论角度看，进程是对正在运行的程序过程的抽象；从实现角度看，进程是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

### 进程的特征

1. 动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。
2. 并发性：任何进程都可以同其他进程一起并发执行
3. 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位
4. 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进

### 进程的状态

1. 就绪状态：进程已获得除处理器外的所需资源，等待分配处理器资源；只要分配了处理器进程就可执行。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由 I/O 操作完成而进入就绪状态时，排入高优先级队列。
2. 运行状态：进程占用处理器资源；处于此状态的进程的数目小于等于处理器的数目。在没有其他
3. 进程可以执行时 (如所有进程都在阻塞状态)，通常会自动执行系统的空闲进程。
4. 阻塞状态：由于进程等待某种条件（如 I/O 操作或进程同步），在条件满足之前无法继续执行。该事件发生前即使把处理机分配给该进程，也无法运行。
5. 进程状态转换图如下



## 进程表

内核负责管理维护所有进程，为了管理进程，内核在内核空间维护了一个称为进程表（Process Table）的数据结构，这个数据结构中记录了所有进程，每个进程在数据结构中都称为一个进程表项 （Process Table Entry）。进程表在进行上下文切换时，能够保存下在 CPU 中关于当前运行进程的一些重要寄存器信息。

因为在进程切换的过程中，esp 会分别指向不同的位置，如果入出栈操作不当容易造成数据被破坏，因此除了进程表，还要引入一个内核栈，esp 的位置出现在这3个不同的区域：

• 进程栈–进程运行时自身的堆栈

• 进程表–存储进程状态信息的数据结构

• 内核栈–进程调度模块运行时使用的堆栈

## 多进程切换

添加任务的主要步骤为：

• 在 task\_table 中增加一项

• 让 NR\_TASK 加 1

• 定义任务堆栈

• 修改 STACK\_SIZE\_TOTAL

• 添加新任务执行体的函数说明

## 系统调用

### 系统调用的概念

1. 由操作系统提供的功能，通常应用程序本身是无法实现的。例如对文件进行操作，应用程序必需通过系统调用才能做到，因为只有操作系统才具有直接管理外围设备的权限。又如进程或线程间的同步互斥操作，也必需经由操作系统对内核变量进行维护才能完成。
2. 应用程序的进程通常在 user 模式下运行，当它调用一个系统调用时，进程进入 kernel 模式，执行的是 kernel 内部的代码，从而具有执行特权指令的权限，完成特定的功能。换句话说，系统调用是应用程序主动进入操作系统内核的入口。
3. 系统调用主要过程为：用户调用系统调用的封装函数 -> 封装函数赋值 eax ，执行中断 -> 中断处理程序在 sys\_call\_table 数组中根据 eax 找到最终要执行的程序并执行。

### 实验中的 get\_ticks 函数（封装函数）

1. 用户使用的函数 sys\_call：此函数中以 0 作为系统调用号来请求系统调用，注意，不要将系统调用号和中断号混淆，所有的系统调用请求都是从 0x90 中断号进入的，进入之后统一执行 sys\_call中断处理函数，然后在此函数中再通过用户的系统调用请求号来执行相应的系统调用函数，所以相对于系统调用请求号，有系统调用函数表。
2. 0x90 中断处理程序：所有的系统调用都从这个中断号进入执行系统调用表里面的函数，Linux 是0x80，我们是 0x90！需要注意的是，上一节我们搭建的外部中断处理框架，可以应用到这里的系统调用陷阱，不同的是我们允许同时多个系统调用陷阱执行 sys\_call ，只是后面陷阱返回略有不同。

## 基于优先级的进程调度

实验中我们对之前的代码稍作修改使得三个进程的延时不同，相当于给了三个进程不同的优先级。

为了实现基于优先级的调度，给每一个进程都添加一个代表优先级的变量，进程每获得一个运行周期就-1，直到 0 时不再获得执行的机会，这样进程获得的执行时间就不一样了。

在拓展学习中，可以了解到 Minix 的进程调度方法。Minix 的进程调度也和任何多道程序操作系统的调度基本相同，也是主要运用轮转调度原理。Minix 分成 16 个优先级队列，驱动程序进程，服务器程序进程优先级排在前面，然后是用户进程，优先级最低的是 idle 列，调度顺序当然是先调度优先级高的队列。当然也可以使用 Nice 系统调用修改进程的优先级。

调度过程中除了优先级是调度决定因素，还有时间片也是调度决定因素。一般驱动程序进程，服务器进程的时间片都分配得很多，通常这些高优先级进程由于时间片都用不完，导致它们常常处于阻塞状态；而用户进程的时间片都分配得很少，当一个进程的时间片使用完成将被移动到队列尾部。当一个时钟中断来临，内核检查那些进程时间片使用完，或处于阻塞状态，然后把这些进程转移到队列尾部。

Minix 也是一个抢占式操作系统，当内核发现一个进程用完了时间片，上次运行的进程仍然是此进程，系统可认为这个进程卡在循环里面了，重而导致低优先级的进程无法运行征兆，系统将降低它的优先级。总之系统认为你卡在循环里，影响其它优先级较低的进程，系统就会降低你的优先级。反之，当一个进程时间片运行完了，但没有妨碍其它进程，系统将提高它的优先级。当一个进程没有用完时间片就变为阻塞，系统认为此进程进入了 I/O 阻塞，系统把此进程移到对尾，当此进程转为就绪状态，系统直接把此进程移动到对列首部，并分配上次所剩余时间片，这样做主要是为了立即响应用户，还有就是 I/O 操作因读写操作占用内存过多，系统想此进程使用完早点释放大量内存。

# 个人分工及心得体会

（每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。）

## 张子航

完成本次实验要解决的问题的第3题，本次实验要解决问题的第5题，并撰写实验报告的相应部分

本次实验，我主要实现了一个系统调用，了解到了如何在单进程的基础上如何实现扩展实现多进程，并通过修改时钟中断来支持多进程管理。学习到了如何去实现一个简单的系统调用，及其基本功能。认识到了进程调度的框架，也复习了优先级调度的实现方法，进程的实现过程等操作，对操作系统这门课程中多进程的知识有了更加深刻的理解与认识

## 辜汝曦

完成本次实验要解决的问题的第1题、第6题，并撰写实验报告的对应部分、以及实验目的及实验内容部分。

本次实验我主要完成了多级反馈队列调度算法。首先，我了解到进程切换的一系列数据结构与相关函数，包括进程表、进程堆栈以及进程的clock\_handler以及schedule函数；在这些知识的基础上，我再思考多级反馈队列的实现。如果真的构造出一个进程的队列，数据结构将会比较复杂，三个队列也难以维护，自己很难在编程时考虑周全。于是，我直接在进程的结构体中添加队列相关的属性，也就是queue和pos，在进行进程操作时，直接修改queue和pos的属性即可。通过本次实验，我对多级反馈队列的认识更加深刻，也锻炼了自己系统编程的能力。

## 杨馨悦

完成本次实验要解决的问题的第4题、第5题，并撰写实验报告中本次实验要解决的问题的第4题、第5题的实现思路一，以及实验结果总结部分。

本次实验我主要是在已有实验代码基础上，实现分时的进程调度和事件触发的进程调度。在此过程中我了解到在添加键盘中断和系统调用时，会涉及到多个.c文件和.h文件的修改，是牵一发而动全身的事情。通过本次实验，我掌握了多进程的实现机制，了解了系统调用的概念、进程调度框架和带优先级的进程调度实现方法，学会了通过代码来指挥进程的运行并根据进程的情况做调度，也了解了进程如何进行切换和使用中断等知识，从而进一步加深了对操作系统进程调度的了解。

## 赵敏

完成本次实验要解决的问题的第2题、第5题，并撰写实验报告中本次实验要解决的问题的第2题、第5题的实现思路二部分。

再本次实验过程中，我们学习了在单进程的基础上实现多进程，学会了使用时间中断、键盘中断和系统调用等方法实现进程切换。了解了进程调度中的多级反馈队列调度算法，与以往理论的学习再次实现了接轨。在调整代码框架，添加键盘中断和系统调用的过程中，我再次熟悉了整个多进程实现的流程，对于进程表的作用以及特权级的切换也增加了新的理解。在本次实验的基础上，也收获了在良好的代码框架上添加更多进程和实现多进程切换的方法。