**西北工业大学 操作系统实验 实验报告**

班号：**10012001**  姓名： 张俊宁 学号： 2020302766

实验日期： 2022.11.06 实验名称： 线程调度的优化

**一、实验目的**

掌握GeekOS系统的线程调度算法，实现线程调度的优化。

**二、实验要求**

1. 按照实验讲义P146页中的设计要求，增加线程调度算法的选择策略，使系统可以在时间片轮转调度和四级反馈队列调度之间切换，并且实现四级反馈队列调度算法，给出关键函数的代码以及实验结果。

2.回答问题：在MLFQ算法中，如果为不同队列的线程设置不同的时间片，如何实现？代码要做哪些修改？第十章第1、2题.

1. **实验过程及结果**

在项目2 的基础上，增加调度算法的选择策略，使系统可以在时间片轮转调度和四级反馈队列调度之间切换。实现四级反馈队列调度，主要是修改kthread.c 里面的Schedule()和Get\_Next\_Runnable()函数。

(1)实现src/geekos/syscall.c 文件中的Sys\_SetSchedulingPolicy 系统调用，它的功能是设置系统采用的何种进程调度策略；

(2)实现 src/geekos/syscall.c 文件中的 Sys\_GetTimeOfDay 系统调用，它的功能是获取全局变量g\_numTicks 的值；

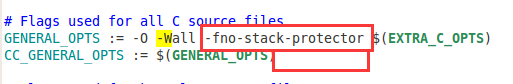
在kthread.c 中：

(3)实现函数Change\_Scheduling\_Policy()，具体实现不同调度算法的转换。

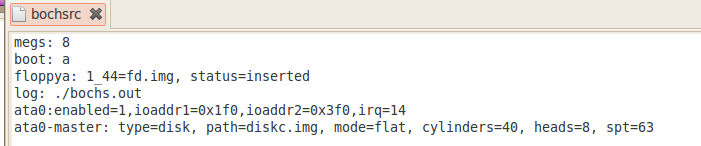
(4)实现函数Get\_Next\_Runnable(),找到优先级最高的进程作为当前运行进程，并将其从准备运行进程队列中取出。

project3的实验是在project2的基础上完成的，所以将project2中的user.c、userseg.c、syscall.c、kthread.c源代码拷贝到project3中。在project2的基础上，增加调度算法的选择策略，使系统可以在时间片轮转调度和四级反馈队列调度之间切换。实现四级反馈队列调度，主要是修改kthread.c 里面的Schedule()和Get\_Next\_Runnable()函数。

修改makefile文件：



拷贝project2中的bochsrc：



（1）Schedule函数的修改：

void Schedule(void)

{

struct Kernel\_Thread\* runnable;

/\*确保中断真的被禁用\*/

KASSERT(!Interrupts\_Enabled());

/\*不应禁用抢占\*/

KASSERT(!g\_preemptionDisabled);

/\*从运行队列中获取下一个运行的线程\*/

runnable = Get\_Next\_Runnable();

／\*

\*激活新线程，保存当前线程的上下文。

\*最终，这个线程将被重新激活和Switch\_To\_Thread()

\*将"return"，然后Schedule()将返回到任何位置

\*／

Switch\_To\_Thread(runnable);

}

（2）Get\_Next\_Runnable()的修改：

struct Kernel\_Thread\* Get\_Next\_Runnable(void)

{

/\*从最高优先级的运行队列中找到最好的线程\*/

KASSERT(g\_curSchedulingPolicy == ROUND\_ROBIN ||

g\_curSchedulingPolicy == MULTILEVEL\_FEEDBACK);

/\* 查找下一个被调度的线程 \*/

struct Kernel\_Thread\* best = NULL;

if (g\_curSchedulingPolicy == ROUND\_ROBIN)

{

/\* 轮询调度策略：只需要从 Q0 队列找优先级最高的线程取出 \*/

best = Find\_Best(&s\_runQueue[0]);

/\* 如果找到了符合条件的线程则将其从队列中移出 \*/

if (best != NULL)

Remove\_Thread(&s\_runQueue[0], best);

}

else

{

int i;

for (i = 0; i < MAX\_QUEUE\_LEVEL; i++)

{

/\* 从最高层队列依次向下查找本层队列中最靠近队首的线程，

如果找到则不再向下继续查找 \*/

best = Get\_Front\_Of\_Thread\_Queue(&s\_runQueue[i]);

if (best != NULL)

{

Remove\_Thread(&s\_runQueue[i], best);

break;

}

}

}

/\* 如果当前没有可执行进程，则至少应该找到 Idle 线程 \*/

KASSERT(best != NULL);

return best; }

2、

（1）实现 src/geekos/syscall.c 文件中的 Sys\_SetSchedulingPolicy 系统调用：

static int Sys\_SetSchedulingPolicy(struct Interrupt\_State\* state)

{

/\* 如果输入的优先级调度方法参数无效(非 0 或 1)则返回错误 \*/

if (state->ebx != ROUND\_ROBIN && state->ebx != MULTILEVEL\_FEEDBACK)

{

Print("Error! Scheduling Policy should be RR or MLF\n");

return -1;

}

/\* 如果输入的时间片参数不在[1, 100]之间则返回错误 \*/

if (state->ecx < 1 || state->ecx > 100)

{

Print("Error! Quantum should be in the range of [1, 100]\n");

return -1;

}

int res = Chang\_Scheduling\_Policy(state->ebx, state->ecx);

return res;

}

（2）实现 src/geekos/syscall.c 文件中的 Sys\_GetTimeOfDay 系统调用：

static int Sys\_GetTimeOfDay(struct Interrupt\_State\* state)

{

/\* 直接返回全局变量 g\_numTicks 的值 \*/

return g\_numTicks;

}

／\*

创建一个信号量。

\*参数:

\* state->ebx -信号量名称的用户地址

\* state->ecx -信号量名称的长度

\* state->edx -初始信号量计数

\*返回:全局信号量id

\*／

3、

（1）int Chang\_Scheduling\_Policy函数的实现：

int Chang\_Scheduling\_Policy(int policy, int quantum)

{

/\* 如果调度策略不同，则修改线程队列 \*/

if (policy != g\_curSchedulingPolicy)

{

/\* MLF -> RR \*/

if (policy == ROUND\_ROBIN)

{

/\* 从最后一个线程队列(此处为 Q3)开始将其中的所有线程依次移动到前一个队列，

直到所有线程都移动到 Q0 队列 \*/

int i;

for (i = MAX\_QUEUE\_LEVEL - 1; i > 0; i--)

Append\_Thread\_Queue(&s\_runQueue[i - 1], &s\_runQueue[i]);

}

/\* RR -> MLF \*/

else

{

/\* 判断 Idle(空闲)线程是否在 Q0 队列 \*/

if (Is\_Member\_Of\_Thread\_Queue(&s\_runQueue[0], IdleThread))

{

/\* 将 Idle 线程从 Q0 队列移出 \*/

Remove\_Thread(&s\_runQueue[0], IdleThread);

/\* 将 Idle 线程加入到最后一个队列(此处为 Q3) \*/

Enqueue\_Thread(&s\_runQueue[MAX\_QUEUE\_LEVEL - 1], IdleThread);

}

}

/\* 保存原来的调度策略 \*/

g\_preSchedulingPolicy = g\_curSchedulingPolicy;

/\* 将全局变量设置为对应的输入值 \*/

g\_curSchedulingPolicy = policy;

Print("g\_schedulingPolicy = %d\n", g\_curSchedulingPolicy);

}

g\_Quantum = quantum;

Print("g\_Quantum = %d\n", g\_Quantum);

return 0;

}

（2）Get\_Next\_Runnable()函数的实现：

Get\_Next\_Runnable( )

struct Kernel\_Thread \*Get\_Next\_Runnable(void)

{

//Print\_Queues();

KASSERT(g\_curSchedulingPolicy == ROUND\_ROBIN ||g\_curSchedulingPolicy == MULTILEVEL\_FEEDBACK);

struct Kernel\_Thread \*best = NULL;//查找下一个被调度的线程

if (g\_curSchedulingPolicy == ROUND\_ROBIN){//轮询调度策略：只需要从 Q0 队列找优先级最高的线程取出

best = Find\_Best(&s\_runQueue[0]);//如果找到了符合条件的线程则将其从队列中移出

if (best != NULL){

Remove\_Thread(&s\_runQueue[0], best);

}

}

else{

int i;

for (i = 0; i < MAX\_QUEUE\_LEVEL; i++){//从最高层队列依次向下查找本层队列中最靠近队首的线程，如果找到则不再向下继续查找

best = Get\_Front\_Of\_Thread\_Queue(&s\_runQueue[i]);

if (best != NULL){

Remove\_Thread(&s\_runQueue[i], best);

break;

}

}

}

KASSERT(best != NULL);//如果当前没有可执行进程，则至少应该找到 Idle 线程

return best;

Print("Scheduling %x\n", best);

}

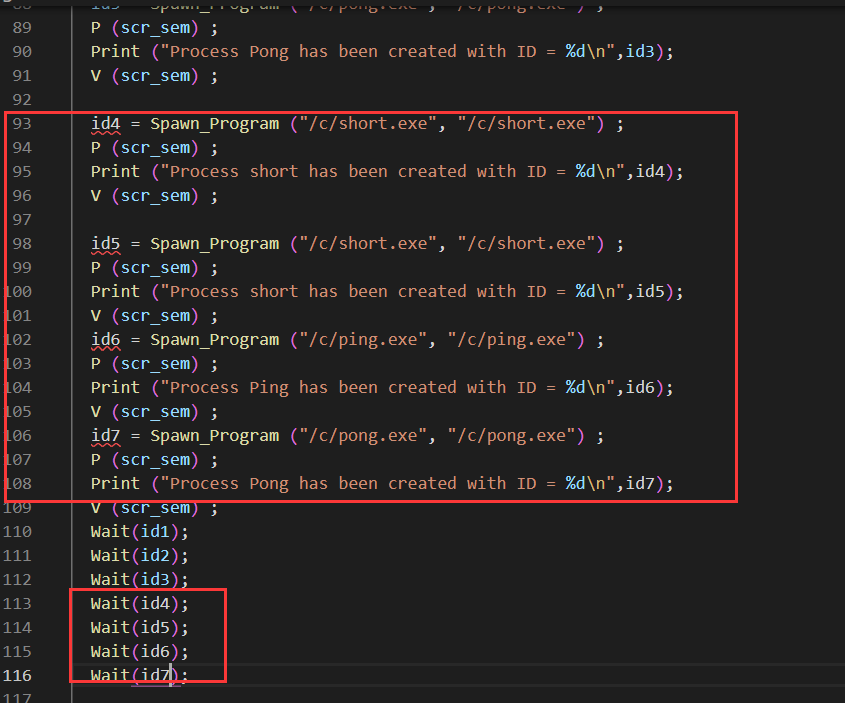
struct Kernel\_Thread\* Get\_Current(void)

{

return g\_currentThread;

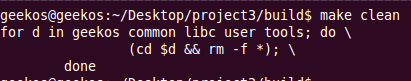
}

workload 首先设置内核调度算法，然后启动long、ping、pong、short、short、ping、pong一共7个进程。通过观察每个时间片内运行的进程的pid，可以分析不同进程调度策略对进程调度的影响。

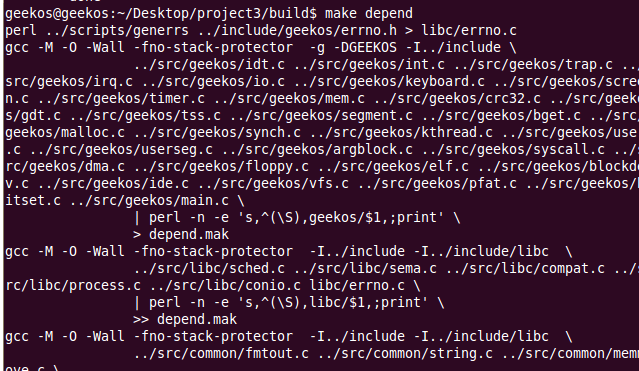


编译运行：

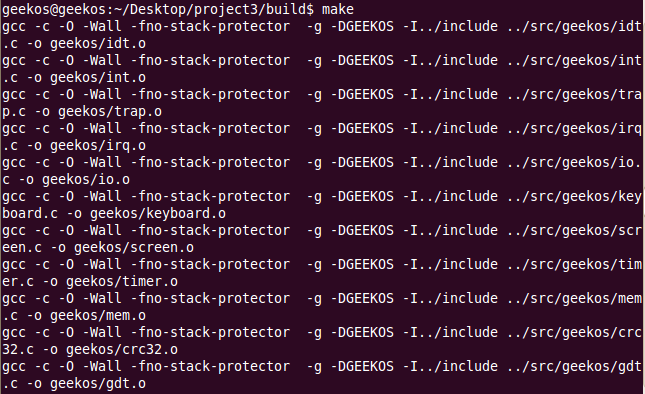
输入指令make clean将无关的.o文件删除：



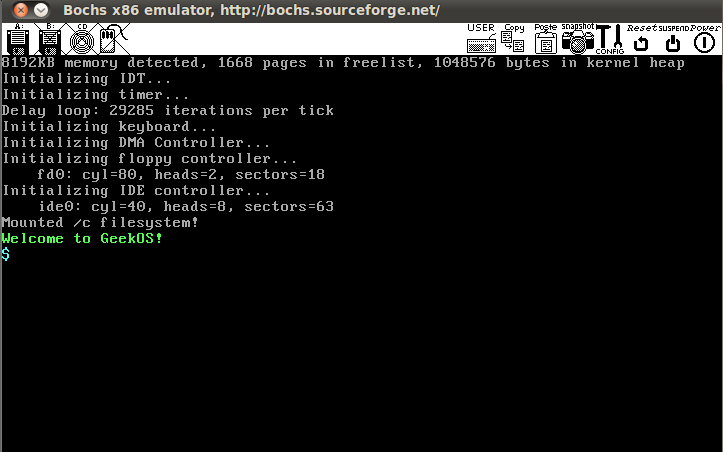
输入指令make depend：

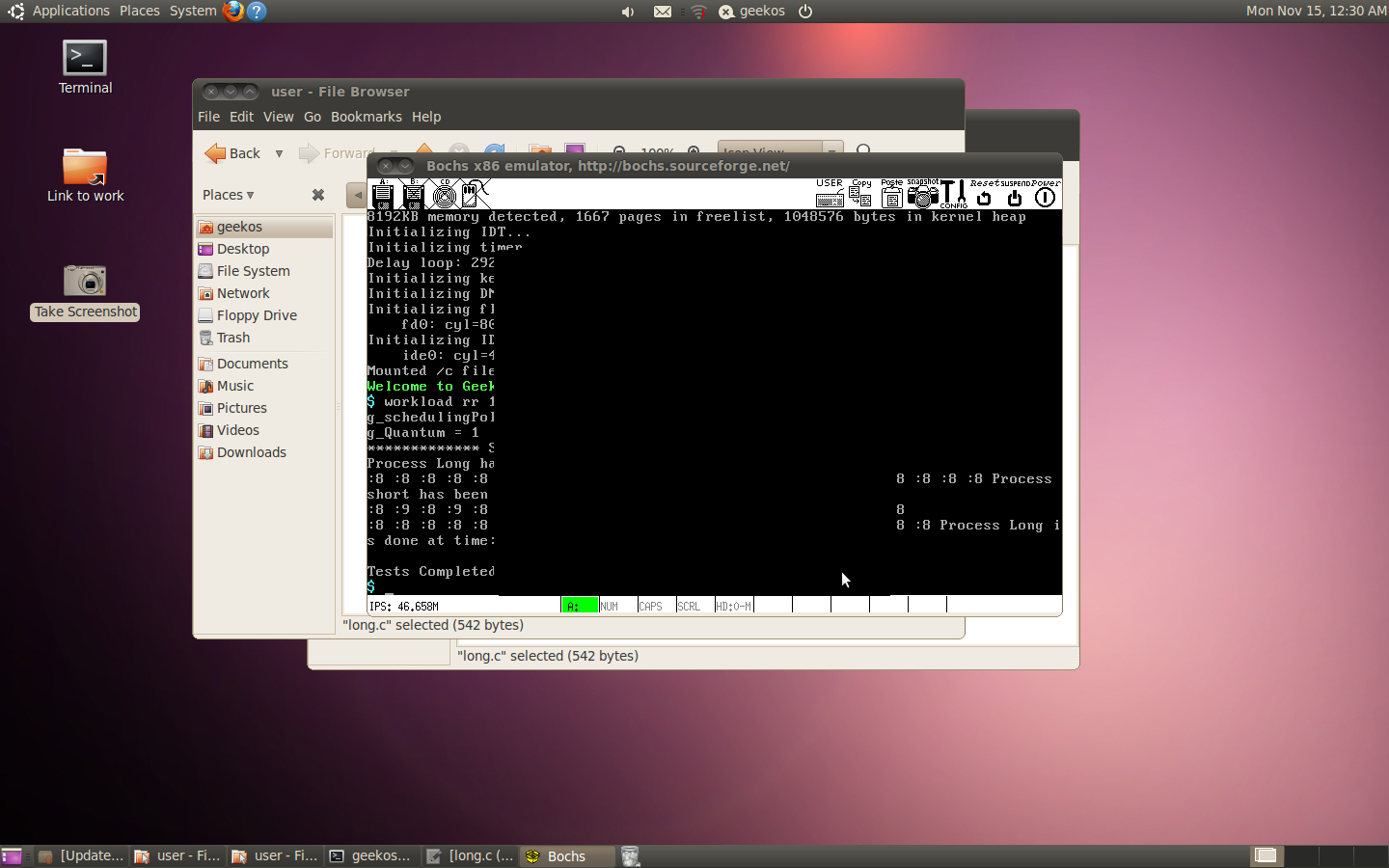


输入指令make编译所有程序并制作软盘：



运行bochs虚拟机：



****

测试：

使用用户程序workload 对调度策略的效果进行测试。workload 及其他测试程序的代码位于./src/user 目录，阅读其代码可知workload 程序接受两个参数，第一个参数指定内核调度算法rr 或mlf，第二个参数指定时间片长度。

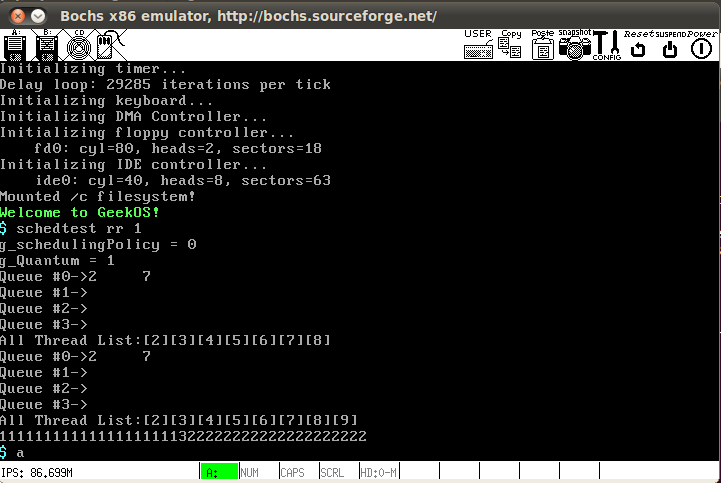
workload 首先设置内核调度算法，然后启动long、ping 和pong 三个用时不同的线程，这四个线程都会输出其运行时间，通过观察运行时间的变化可以分析不同调度算法，不同时间片对线程调度的影响。因为这四个线程都要在屏幕上输出，为避免冲突，使用“screen”信号量进行互斥。

workload的主要流程是：创建一个长作业，再创建一个短作业，等待短作业结束后，在创建下一个短作业，如此循环，直至所有短作业都被创建和运行结束，然后等待长作业结束。

使用不同的测试参数进行测试，在实验报告中对测试结果进行分析。如：

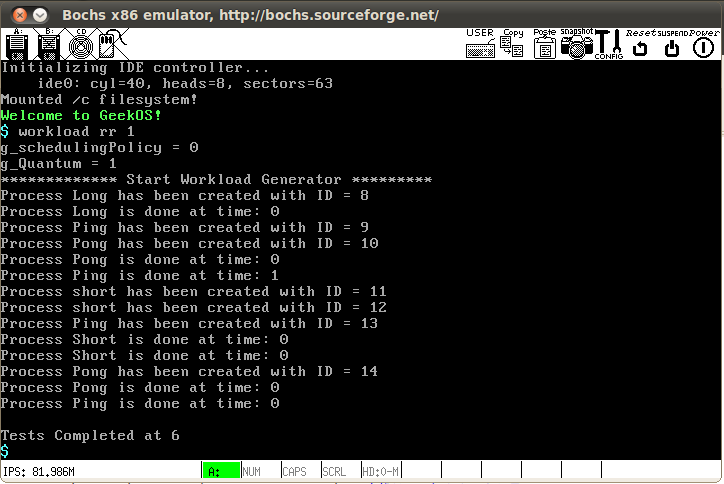
系统调入新的进程运行后，时间片轮转结束，继续调度新的进程。

运行结果：

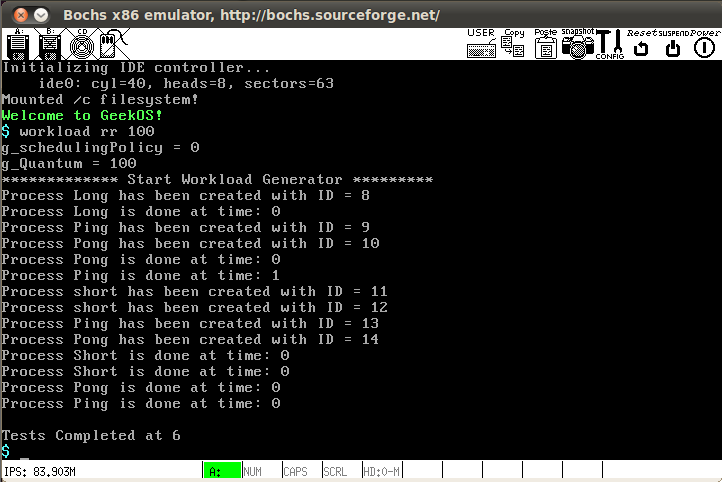


使用不同的测试参数进行测试，在实验报告中对测试结果进行分析。如：

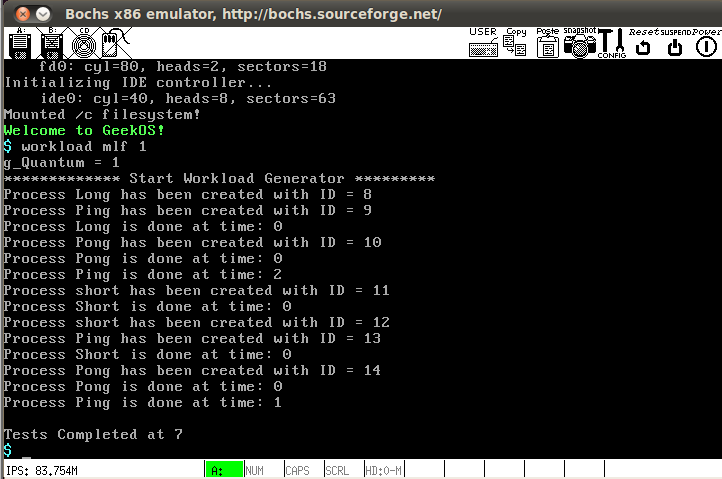
* workload rr 1



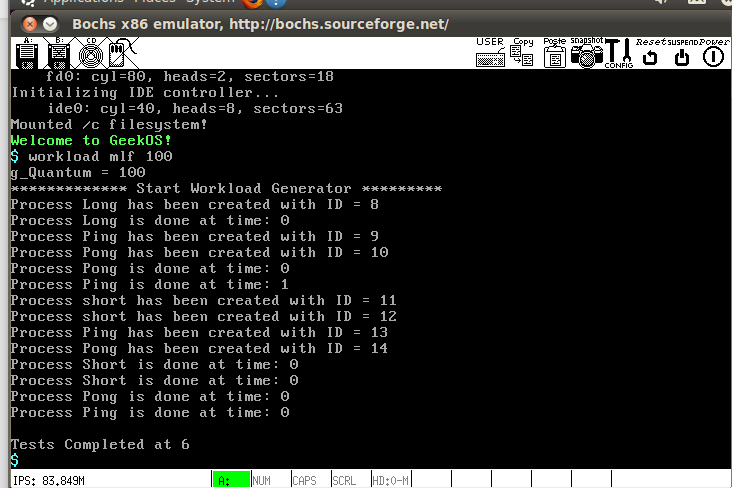
* workload rr 100



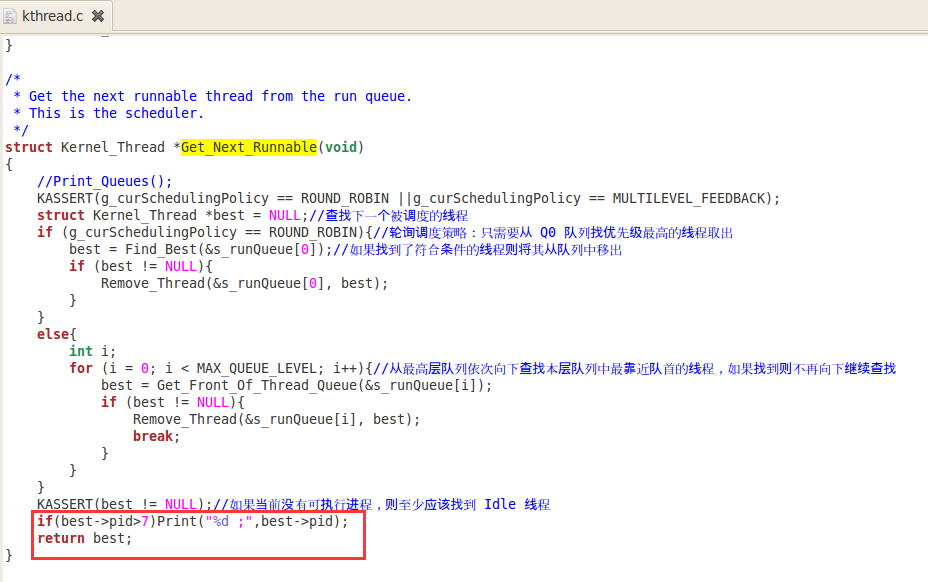
* workload mlf 1



* workload mlf 100

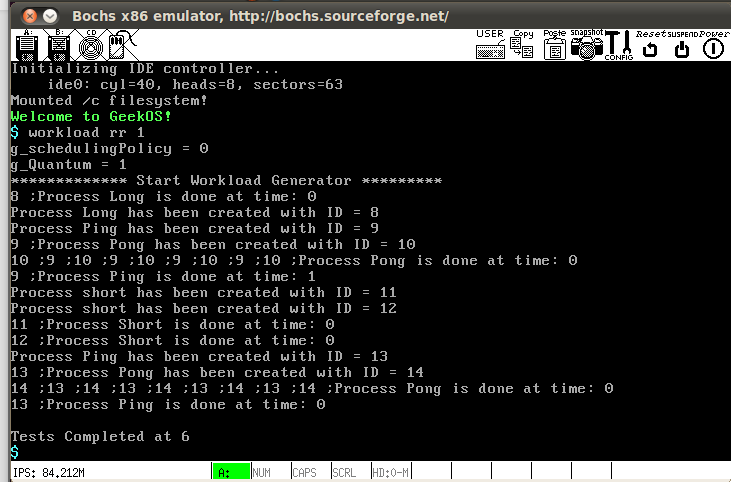


为了查看当前运行的进程，在调度进程时打印出当前进程pid，对Get\_Next\_Runnable做修改：因为pid小于等于6的进程是内核态进程，pid=7的是workload，所以只打印pid大于7的进程。

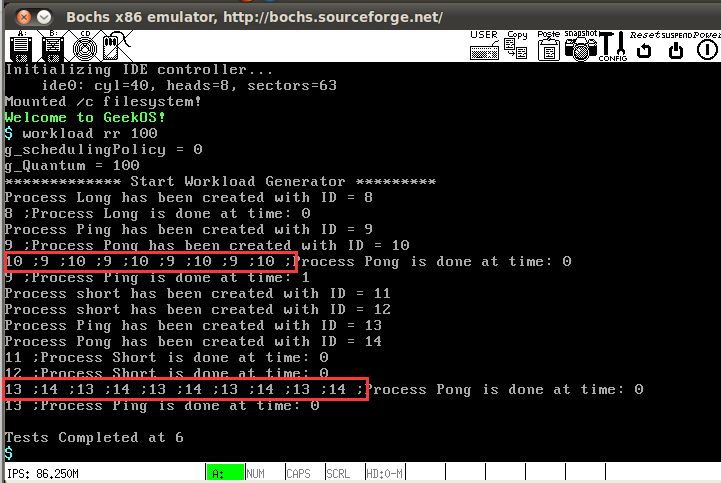


然后运行程序，使用不同的测试参数进行测试，在实验报告中对测试结果进行分析。如：

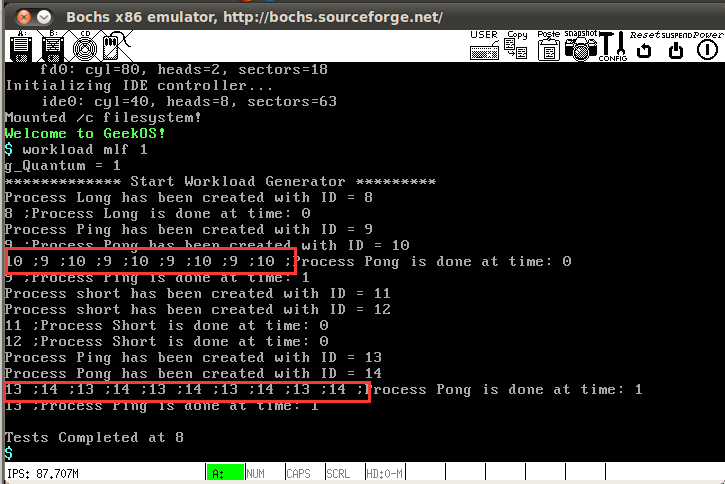
* workload rr 1



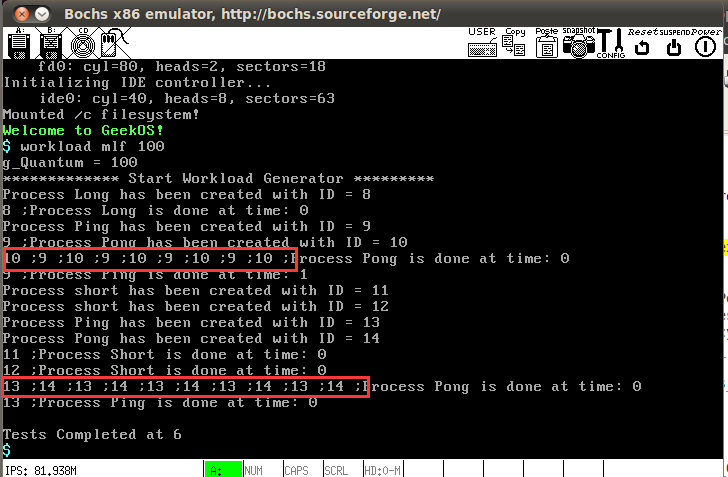
* workload rr 100

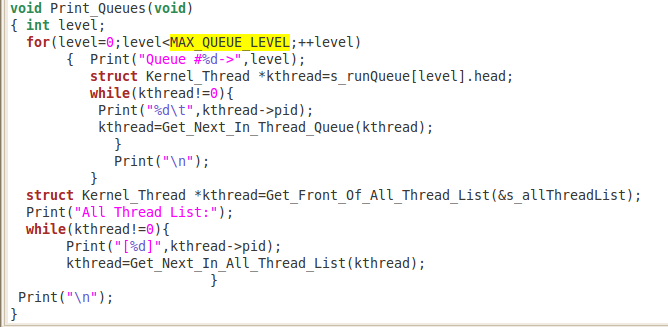


* workload mlf 1

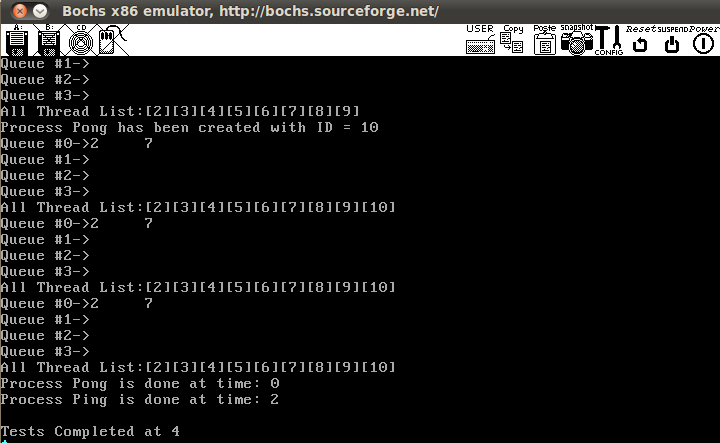


* workload mlf 100

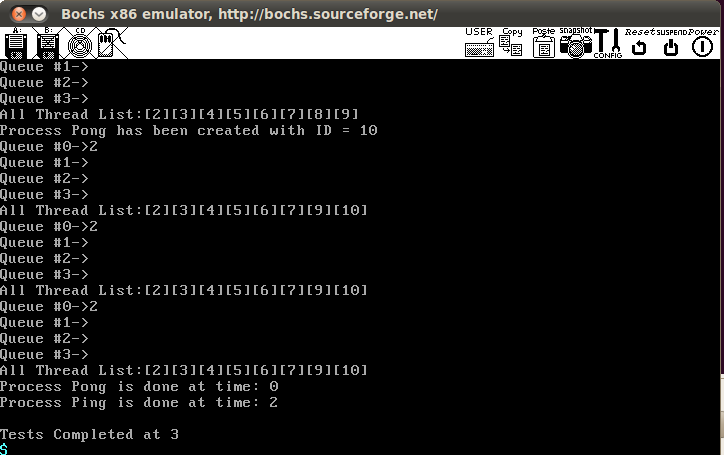


为了方便查看实验过程，添加Print\_Queues()函数：  


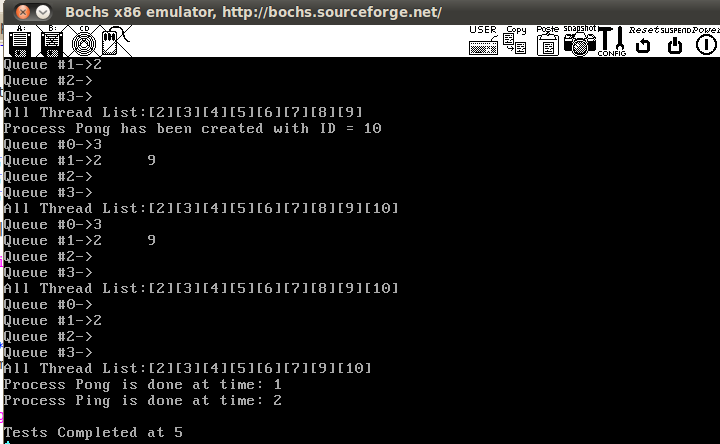
* workload rr 1



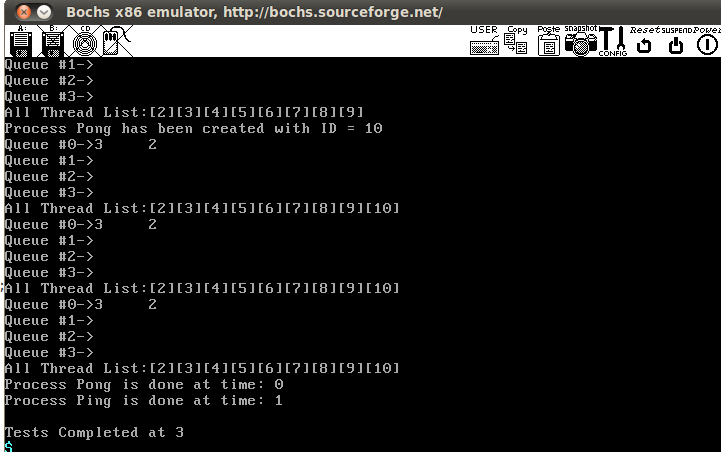
* workload rr 100



* workload mlf 1



* workload mlf 100

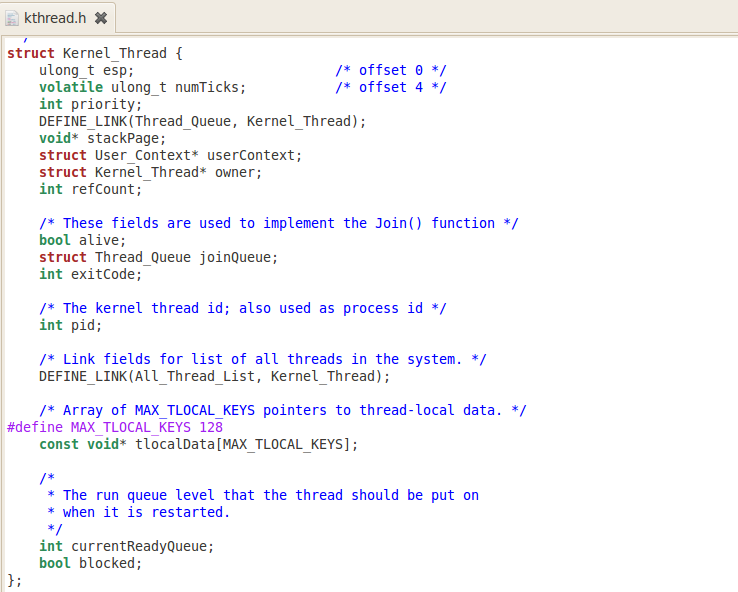


从上面的测试结果来看，这两种调度算法都是基于时间片轮转，输出2和输出1的的数量基本一样多（多队列优先级调度算法的时间片都是一样的）。进程的周转时间还包括进程切换的时间。同样的调度算法，同样的时间片，程序执行的周转时间可能不一样（进程切换花费的时间可能不一样）。同时可以看到mlf调度算法的的进程分布在不同的队列中，而rr调度算法的进程都在Queue0中。

1. **实验分析**

**内核进程控制块：**

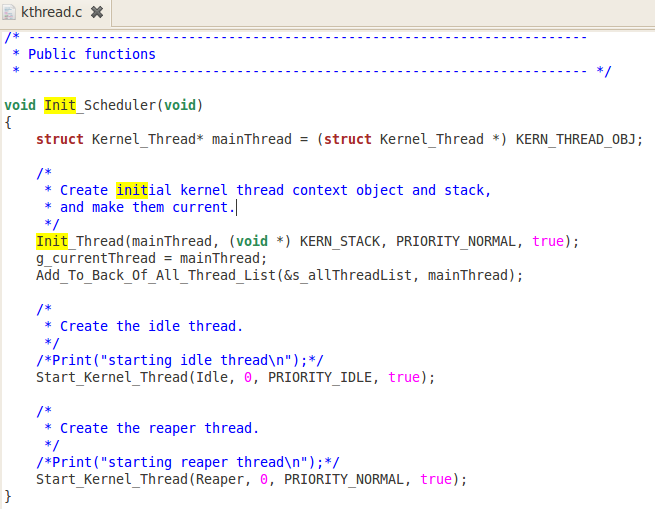
在include/kthread.h中定义：系统中每个内核进程有且只有一个进程控制块，进程控制块是用于记录进程状态及有关信息的数据结构。GeekOS操作系统中用数据结构Kernel\_Thread作为内核进程控制块，对系统中的进程信息、执行情况、控制信息等加以维护。



**GeekOS中最早的内核进程：**

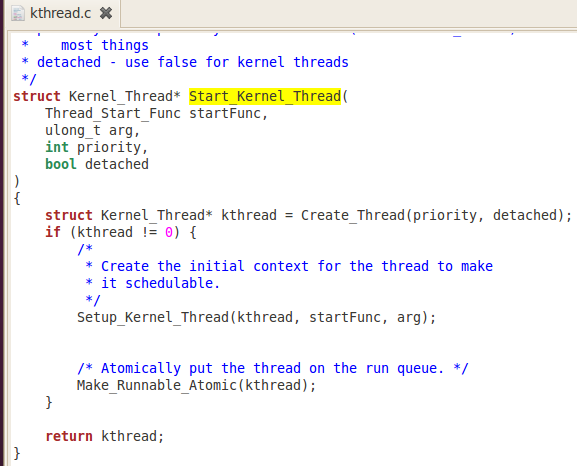
GeekOS系统最早创建的内核进程有Idle、Peaper和Main3个进程。在系统初始化时，Main函数调用了一系列初始化函数，Init\_Scheduler函数位于\src\geekos\kthread.c。它的作用：

* 初始化一个内核进程mainThread，并将该进程作为当前运行进程；
* 创建两个系统进程Idle和Reaper；
* Idle进程类似于Windows中的系统闲置进程，什么也不做，创建后就一直存在于系统中，它存在的唯一目的是保证准备运行队列中有可调度的进程。当系统没有可运行的进程时，CPU就运行Idle，一旦有其他准备运行的进程进入，Idle就会立即放弃CPU。
* Reaper负责消亡进程的善后工作，如释放消亡进程占用的资源，内存、堆栈等。



**内核进程对象：**

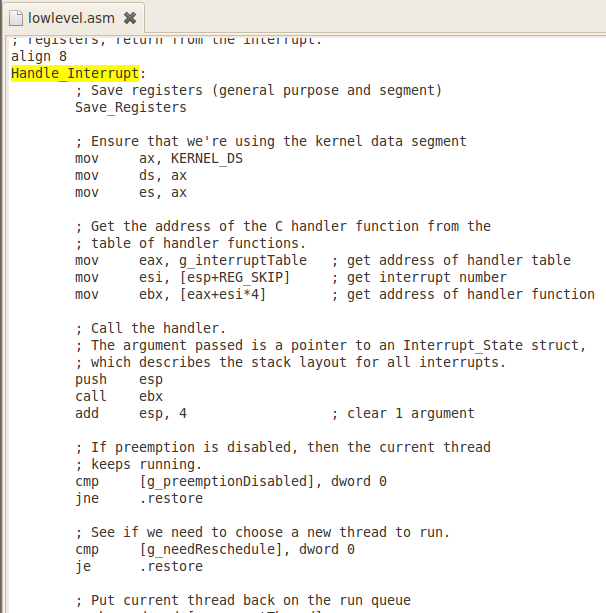
创建一个GeekOS内核进程需要调用Start\_Kernel\_Thread函数，其内部调用Creat\_Thread函数 ，该函数主要是创建内核进程对象，并调用Alloc\_Page函数为进程对象、进程内核堆栈各分配一页内存（若失败，返回0，同时释放内核控制块空间）。



**进程调度：**

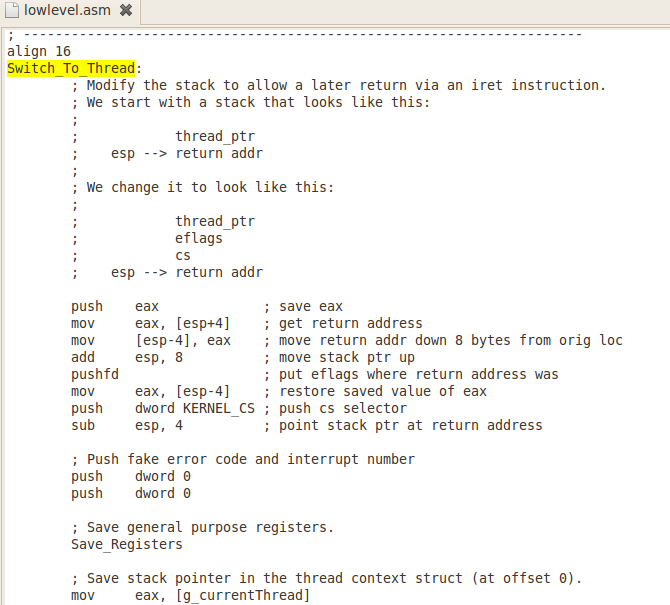
GeekOS在下述情况会发生进程切换：

* **时间片用完：**当时间片用完，进程切换在中断处理函数Handle\_Interrupt（在lowlevel.asm中定义）中完成



* 执行内核进程Idle
* 进程退出调用Exit函数
* 进程进入等待调用Wait函数

上面三种情况：函数内部都有调用Schedule()函数，其中又调用Switch\_To\_Thread函数（在lowlevel.asm中定义）



以上两个关键函数（Handle\_Interrupt&Switch\_To\_Thread）均在汇编文件中实现，最终执行一个iret指令中断返回，跳入目标进程。

**GeekOS进程调度策略：**

GeekOS的初始系统提供的进程调度是时间片轮转调度，所有准备运行进程（即Kernel\_Thread）都放在一个FIFO队列里面，进程调度时找优先级最高的进程投入运行。Get\_Next\_Runnable函数就是进程调度算法实现的地方，由\src\geekos\kthread.c文件中的Find\_Best函数在准备运行进程的队列（s\_runQueue指针指向）中查找，找优先级最高的。

**GeekOS 中的四级反馈队列调度策略实现：**

在GeekOS中实现四级反馈队列调度策略需要使用四个就绪队列代替原有的一个队列。实现思想是：给四个准备运行队列不同的优先级，优先级别标记为数字0~3 ，数字为0表示最高优先级，数字为3表示最低优先级。

首先，新创建的进程被入优先级最高的准备运行队列，即优先级为0的准备运行队列。若优先级为0的进程被调度后，在给定时间片内无法完成，那么进程就被移到下一优先级队列的尾部（即优先级为1的准备运行队列），以此类推，直到进程被放到优先级为3的队列为止。因此，要求运行时间较长的进程最终会被放入到优先级为3的准备运行队列。若进程被阻塞，每经过一个时间片，进程优先级都将增加1，所以当进程阻塞达连续的3个时间片后，又将升到最高优先级。进程调度总是优先调度优先级高的进程运行，当优先级为0 的进程队列为空时，进程调度从优先级为1 的队列选择进程调度。大家应该还记得系统中的空闲进程Idle ，在四级反馈队列调度中，该进程应始终放在优先级为3 的进程队列尾部，以便系统中没有其他可调度进程时就运行它。

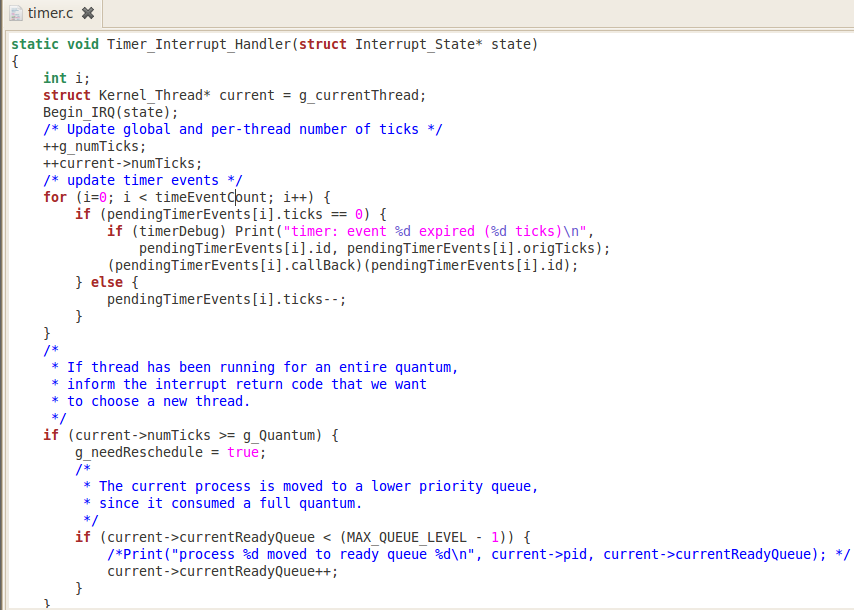
为实现四级队列，设计中首先要修改 s\_runQueue (在 src/GeekOS/kthread.c) 的定义，从原来的一个结构体改为一个4 元素的结构体数组，每一个结构体元素用于存放一个优先级队列的队首指针。在进程调度时，首先在最高优先级（优先级 0）的队列里面找，如果有进程存在，则调度它运行。如果没有，那就在次优先级的队列里面找，以此类推，直到找到一个进程投入运行为止。GeekOS 系统将Idle (系统空闲) 进程始终放在优先级为3 的进程队列末尾，且不允许移动到其他队列，以保证进程调度时一定能找到进程投入运行。

当实现四级反馈队列调度策略后，GeekOS 系统就拥有2 种进程调度策略，即单队列时间片轮转调度策略和四级反馈队列调度策略。那么系统究竟采用何种调度策略呢？调度策略的确定是通过系统调用Sys\_SetSchedulingPloicy 实现的，函数参数为指向Interrupt\_State的指针，使用其ebx 成员ecx 成员记录调度策略和时间片长度，其中，state‐>ebx 成员用于指定调度策略，它的取值为0 或1，值为0 代表系统采用时间片轮转调度策略，值为1 则代表系统采用四级反馈队列调度策略，若取其他值则出错。State‐>ecx 成员用于记录相应度策略下的时间片长度。时间片长度默认值是 4，用 DEFAULT\_MAX\_TICKS（在 timeer.c 中定义）表示。将g\_Quantum 作为一个可以由系统调用来设置的全局变量，以实现可变时间片。

**GeekOS进程调度处理过程：**

进程调度是在时钟中断处理Time\_Interrupt\_Handle(/src/geekos/timer.c)内实现的。进程的Kernel\_Thread结构中有一个numTicks变量，在进程对象初始化时被初始化为零，之后每次时钟中断处理，进程的该变量都会加1，然后系统会检查进程执行的时间是否超过了系统规定的时间片g\_Quantum，如果超过，说明当前进程时间片已用完，系统应调度新的进程运行，于是将变量g\_needReschedule置为true，用以标志需要重新调度新进程运行。

在时钟中断处理函数返回到Handle\_Interrupt后检查g\_needReschedule变量，如果为true，就调用Make\_Runnable函数（kthread.c），将当前运行进程放入准备运行进程队列s\_runQueue；之后再调用Get\_Next\_Runnable函数（kthread.c）找到优先级最高的进程；最后返回Handle\_Interrupt将g\_needReschedule返回为false，并切换到新进程运行。Time\_Interrupt\_Handle(/src/geekos/timer.c)：



**多级调度队列与时间片轮转调度之间的转换：**

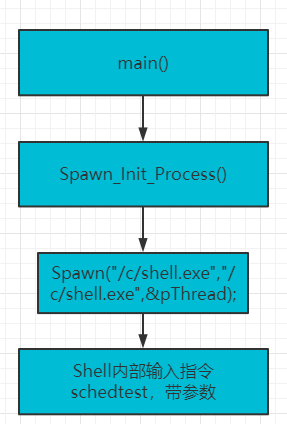
有了系统调用，用户程序就可以通过Set\_Scheduling\_Policy 函数来改变系统的调度策略。下面我们看两种调度策略间的转换。

GeekOS 使用函数Chang\_Scheduling\_Policy(int policy, int quantum)来改变调度策略，其中policy 是设置的调度策略，quantum 是设置的时间片。例如policy 为1 说明设置的是多级反馈队列调度算法，此时若g\_SchedPolicy（为系统添加的标识算法的变量，初始化为0）为 0，说明当前的调度算法为轮转调度，要变成 MLF 就必须把 Idle 线程放入 3 队列，若g\_SchedPolicy 为1，说明当前是多级反馈队列调度算法，则返回。如果policy 为0，则说明设置的是轮转调度，此时若g\_SchedPolicy 为1，则必须把4 个队列变成一个队列，即所有的线程都在队列0 上了。若g\_SchedPolicy 为0，则返回。

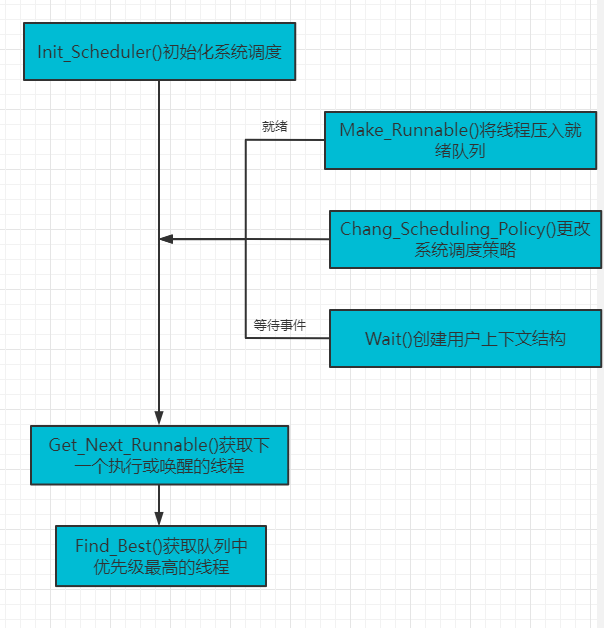
**其他相关函数：**

* 系统调用Sys\_GetTimeOfDay（）用于计算线程剩余的时间片，只需要返回g\_numTicks就可以了。
* 在Init\_Thread（）中都是把线程放在0 队列上的，并且blocked 变量为false。
* 在 Get\_Next\_Runnable（）中，从最高级的队列开始，调用 Find\_Best（）来找线程优先级最大的线程，直到在某级队列中找到符合条件可以运行的线程。其调度算法主要是在该函数中实现。
* 在 Wait（）函数中，线程被阻塞，所以 blocked 变量被设置为 true，并且如果是 MLF算法，则该进程的 currentReadyQueue 加一，下次运行的时候进入高一级的线程队列。同理，在线程被唤醒后，Make\_Runnable的时候，需要根据该进程的currentReadyQueue将其置于相应的队列。

**多级反馈策略测试运行流程：**



系统调度相关的流程图：



本项目主要实现多级反馈队列(MLF)和轮询(RR)调度算法以及信号量的相关操作，需要我们填写 syscall.c 和 kthread.c 等不同文件中的多个函数。在之前的项目中，GeekOS使用的系统调度算法均为轮询调度算法，因此在此项目中，我们需要实现MLF调度算法的相关操作以及 RR 与 MLF 两种算法之间的队列转换算法。GeekOS中，MLF 算法的规则描述为：进程就绪队列共分为 4 级，按照优先级从 高到低排列分别为 Q0、Q1、Q2 和 Q3 队列；新创建的进程会被置入最高优先级的就绪 队列（此处为 Q0）；每当一个进程运行完一个时间片长度之后，它就会被置入比之前低 一级的就绪队列，直到到达优先级最低的队列（Q3）， 因此，CPU 密集型的进程最终会 被放到最低优先级的队列中；如果一个进程被阻塞（blocked），它的队列优先级就会提 升一个等级，直到被阻塞三次后达到最高优先级队列（Q0）。

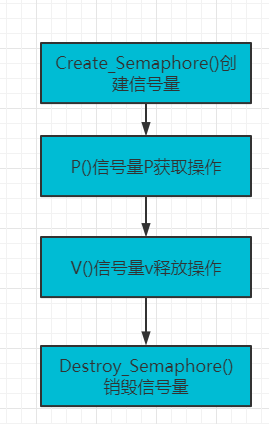
在 MLF 策略与 RR 策略的进程队列转换时，GeekOS 给出的规则如下：MLF->RR 时，将 Q1-Q3 队列中所有进程转移至 Q0 队列，然后按照优先级从高到低的顺序重新排 序；RR->MLF 时，只需将原本位于 Q0 队列中的 Idle（空闲）进程转移至 Q3 队列，其 他进程无需再做修改。在实际编写代码的时候，由于 GeekOS 提前实现了从队列中获取 最高优先级进程的函数 Find\_Best()，因此在 MLF->RR 时只需要将所有队列中进程移至 Q0 队列中即可。

为实现四级队列，设计中首先要修改s\_runQueue（src/geekos/kthread.c）的定义，从原来一个结构体改为一个4元素的结构体数组，每一个结构体元素用于存放一个优先级队列的队首指针。

GeekOS系统将Idle进程始终放在优先级为3的进程队列末尾，且不允许移动到其它队列，以保证进程调度时一定能找到进程投入运行。

当实现四级反馈队列调度策略后，GeekOS系统就拥有2种进程调度策略，即单队列时间片轮转调度策略和四级反馈队列调度策略，然后通过系统调用Sys\_SetSchedulingPolicy（src/geekos/syscall.c）实现的。

信号量相关的流程图：



**五、所遇问题及解决方法**

* 在完成项目 3 代码填写进行测试的时候，可能发现 schedtest.exe 这个文件在 /project3/build/user/文件夹中存在，但是却无法在系统中执行，总是会显示找不到文件。

解决方法：分析 GeekOS 系统编译阶段 make程序的执行结果，会发现 schedtest.exe 文件的名称并没有显示完全，也就是说在GeekOS的挂载磁盘中，这个文件的名称为“schedtest.ex” 而非“schedtest.exe”，这也就是为什么直接输入程序名或者加上路径都无法执行该文件的原因。同时还能发现了 Makefile文件中并没有对 semtest.c和 null.c进行编译。exe文件是通过/tools/builtFat.ex这个小程序生成 img格式磁盘映像文件的，找到这个文件的源代码文件位置/project3/src/tools/buildFat.c，会发现是其中使用的 directoryEntry 这个结构中 fileName 字段 的长度不足。因此通过 grep 指令找到 directoryEntry 结构体所在文件 /project3/include/geekos/pfat.h。最后修改/build/Makefile文件，在“USER\_C\_SRCS”后面添加“semtest.c”和“null.c”。

**六、思考与联系**

**1.在MLFQ算法中，如果为不同队列的线程设置不同的时间片，如何实现？代码要做哪些修改？**

解：设置时间片数组，根据队列级别进行时间选择，代码修改如下:

int Change\_Scheduling\_Policy(int policy,int quantum)  
{   
 struct Kernel\_Thread \*thread,\*ithread;  
 int i;  
 if(policy==0)  
 {  
 if(g\_SchedPolicy==0)return 0;  
 if(g\_SchedPolicy==1)  
 {   
 for(i=1;i<MAX\_QUEUE\_LEVEL;++i)  
 {  
 thread=Get\_Front\_Of\_Thread\_Queue(&s\_runQueue[i]);  
 while(thread!=0){  
 ithread=Get\_Next\_In\_Thread\_Queue(thread);  
 thread->currentReadyQueue=0;  
 Remove\_From\_Thread\_Queue(&s\_runQueue[i],thread);  
 Add\_To\_Back\_Of\_Thread\_Queue(&s\_runQueue[0],thread);  
 thread=ithread;  
 }  
 }  
 g\_Quantum=quantum[i];  
 g\_SchedPolicy=0;  
 }  
 }  
 if(policy==1){   
 if(g\_SchedPolicy==0){   
 thread=Get\_Front\_Of\_Thread\_Queue(&s\_runQueue[0]);  
 while(thread!=0){  
 if(thread->priority==0)break;  
 thread=Get\_Next\_In\_Thread\_Queue(thread);  
 }  
 thread->currentReadyQueue=3;  
 Remove\_From\_Thread\_Queue(&s\_runQueue[0],thread);  
 Add\_To\_Front\_Of\_Thread\_Queue(&s\_runQueue[3],thread);  
 g\_Quantum=quantum;  
 g\_SchedPolicy=1;  
 }  
 if(g\_SchedPolicy==1)return 0;  
 }  
 return 0;  
}

1. **系统调用的作用是什么？简要描述他的执行过程。**

系统调用也叫程序接口，是应用程序请求 OS 内核完成某功能时的一种过程调用，是用户程序对 OS 内核功能进行调用的一种手段。

在设置了系统调用功能号和参数后，执行系统调用指令便开始进入系统调用，其一般处理过程如下：

* 系统产生软中断（或陷入），由中断硬件完成部分现场信息保护： PSW 的值和 PC 寄存器的值；并通过中断向量转向系统总控中断处理程序完成其他 CPU 现场信息的保存：如陷入类型、参数表指针、其他 CPU 寄存器的值等。
* 然后转向执行系统调用处理程序：使用系统调用功能号查找系统调用入口表，找到相应系统调用的处理子程序的入口地址。
* 执行系统调用处理子程序，完毕后返回执行成功与否以及成功时的执行结果给调用者；恢复被中断进程或新调度进程的 CPU 现场，返回被中断进程或新调度进程执行。

1. **如何为操作系统选择进程调度策略？**

**解：常见的进程调度策略如下：**

* 先来先服务调度算法：先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。
* 短作业(进程)优先调度算法：短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。
* 高优先权优先调度算法：为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程。
* 高响应比优先调度算法：在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。
* 时间片轮转法：在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。
* 多级反馈队列调度算法：前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

**选择策略的方式：**

如果响应时间是我们唯一的标准，那么带有合理时间片的RR，就会是非常好的调度程序。但是其周转时间是相当糟糕的；如果周转时间是我们唯一的标准，那么显然SJF，STCF，会是比较好的调度程序；

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。

先来先服务优先调度算法对长作业有利，不利于短作业；短作业优先调度算法能够降低作业的平均等待时间，提高系统的吞吐量，但是对长作业不利，未考虑作业的紧迫程度，执行时间难以预测；非抢占式优先算法适用于批处理系统以及实时性不高的系统，而抢占式优先算法适用于实时性与性能要求高的系统；时间片轮转法依次调度各进程，把一个时间片分出去，多级反馈队列调度算法可兼顾多方面的系统目标，是目前较好的进程调度算法。

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。