西北工业大学 操作系统实验 实验报告

班号：10012006　　姓名：夏卓　　学号：2020303245

实验日期：2022/11/6　　实验名称：线程调度的优化与控制

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.

三、实验过程及结果

首先添加设置线程调度策略的系统调用 Sys_SetSchedulingPolicy()函数，其主要功能为调用函数 Chang_Scheduling_Policy()更改线程调度算法：

```
static int Sys_SetSchedulingPolicy(struct Interrupt_State *state)
{
    /* ebx为policy,ecx为quantum */
    if (state->ebx != ROUND_ROBIN && state->ebx != MULTILEVEL_FEEDBACK)
    {
        Chang_Scheduling_Policy(state->ebx, state->ecx);
        return 0;
    }
    return -1;
}
```

在 Chang_Scheduling_Policy()函数中，要根据是否需要变化策略做出不同的更改。

具体来说，当从 MLF —> RR 时，需要把四个队列变成一个队列，即所有线程都移动到队列 0 上：

```
/* 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]);
}
```

当从 RR —> MLF 时，需要把 Idle 线程放在队列 3 中：

```
/* 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);
    }
}
```

接下来需要更改 Get_Next_Runnable()函数，使其支持四级反馈队列调度算法，并能根据当前不同的调度策略选择出下一个可运行的线程：

具体来说，当为 RR 策略时，只需要从 Q0 队列中找出优先级最高的线程即可：

```
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);
    }
}
```

当为 MLF 策略时，需要从高优先级队列依次往低优先级队列查找：

```
else
{
    for (int 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;
        }
    }
}
```

为了观察线程切换的次序，可以在该函数中输出当前进程的 PID，且为了避免内核线程的干扰，需要屏蔽掉 pid < 7 的进程：

```
if (best->pid > 7)
    Print("%d@%d----", best->pid, best->currentReadyQueue);
return best;
```

另外还需要更改 Wait()函数，如果当前调度算法为 MLF，则阻塞后该进程的优先级应该加 1，即将其升到前一个队列中：

```
/* 如果为 MLF 调度策略则下次运行时线程应进入高一优先级的队列(即队列数减一)
   RR 调度策略时不受影响，因为已经运行在最高优先级的线程队列 */
if (current->pid != IdleThread->pid && current->currentReadyQueue > 0)
    --current->currentReadyQueue;
```

同理，在 Make_Runnable()函数中，当线程被唤醒后，需要根据该进程的 currentReadyQueue 将其置于相应的队列中，需要注意的是，当调度算法为 RR 时，需要将其放入 Q0 中，若是 Idle 线程，需要将其放入 Q3 中：

```
int currentQ = kthread->currentReadyQueue;
/* 当调度算法为 RR 时，将线程放入Q0中 */
if (g_curSchedulingPolicy == ROUND_ROBIN)
    currentQ = 0;
/* 当调度算法为 MLF 时，需要将 Idle 线程放入Q3中 */
else if (kthread == IdleThread)
    currentQ = MAX_QUEUE_LEVEL - 1;
kthread->blocked = false;
Enqueue_Thread(&s_runQueue[currentQ], kthread);
```

最后，需要修改 Timer_Interrupt_Handler()函数，当进程时间片消耗完时，若是 MLF 策略，则当前线程的优先级应该减 1：

```
/* 当进程时间片消耗完且为 MLF 策略时，当前线程的优先级减 1 */
if (g_curSchedulingPolicy == MULTILEVEL_FEEDBACK
        && current->currentReadyQueue < (MAX_QUEUE_LEVEL - 1))
    current->currentReadyQueue++;
```

实验结果如下：

Workload rr 1：

```
$ workload rr 1
8@0----9@0----10@0----11@0----
************* long and short process start *********
8@0----9@0----8@0----9@0----8@0----9@0----
Short done at time: 4
8@0----10@0----8@0----10@0----8@0----10@0----
Short done at time: 4
8@0----11@0----8@0----11@0----8@0----11@0----
Short done at time: 4
8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0
----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0---
-8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@
0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0----8@0--
--8@0----8@0----8@0----8@0----8@0----8@0----8@0----
Long done at time: 67
************* Tests Completed at 93 ***********
```

Workload rr 100：

```
$ workload rr 100
8@0----9@0----10@0----11@0----
************* long and short process start *********
8@0----
Long done at time: 60
9@0----
Short done at time: 2
10@0----
Short done at time: 2
11@0----
Short done at time: 2
************* Tests Completed at 308 ***********
```

Workload mlf 1：

```
$ workload mlf 1
8@0----9@0----10@0----11@0----
************* long and short process start *********
8@0----9@0----8@1----9@1----8@2----9@2----
Short done at time: 4
10@0----10@1----10@1----10@2----
Short done at time: 2
11@0----11@1----11@2----
Short done at time: 2
8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3
----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3---
-8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@
3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3--
--8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8@3----8
@3----
Long don8@3----e at time: 66
************* Tests Completed at 75 ***********
```

Workload mlf 100：

```
$ workload mlf 100
8@0----9@0----10@0----11@0----
************* long and short process start *********
8@0----
Long done at time: 60
9@0----
Short done at time: 2
10@0----
Short done at time: 2
11@0----
Short done at time: 2
************* Tests Completed at 93 ***********
```

四、实验分析

为了对线程切换调度策略有一个直观的感受，我对 workload 做了如下更改：

首先为了避免在 long 进程运行之前 workload 无法加载所有 short 进程，而导致时间上的差异，我使用了信号量作为同步策略，使得只有当 workload 加载完全部进程后方可开始运行，具体操作是在 workload 中创建信号量 test，初始值为 0，当加载完所有进程后 V(test)，而 long 进程和 short 进程想要开始运行，必须 P(test)，因此需要等待 workload 进程结束：

```
test = Create_Semaphore("test", 0);

id1 = Spawn_Program("/c/long.exe", "/c/long.exe");
id2 = Spawn_Program("/c/short.exe", "/c/short.exe");
id3 = Spawn_Program("/c/short.exe", "/c/short.exe");
id4 = Spawn_Program("/c/short.exe", "/c/short.exe");

V(test);
```

可以看到，workload 进程加载了一个 long 进程，三个 short 进程。为了能反映出多级优先队列调度算法的优点，我们使 short 进程串行执行，即当一个 short 进程结束后，才能开启下一个 short 进程。

为了达到这个目的，对于 long 进程，我们使其进入后便唤醒第一个 short 进程开始执行：

```
test = Create_Semaphore("test", 0);
P(test);
V(test);
```

对于 short 进程，当其结束后再唤醒下一个 short 进程：

```
P(scr_sem);
Print("\nShort done at time: %d\n", elapsed);
V(scr_sem);

V(test);
```

另外，实验结果中，采用 pid@queue 的方法输出当前线程的线程号及所处队列，8 号进程为 long 进程，9、10、11 号进程为 short 进程，输出的时间为该进程从创建到结束所消耗的时间，能较好得反映出系统的响应速度。

从实验结果中可以看出，当时间片较短（为 1）时，RR 调度算法下，short 进程的响应时间大致相同，这是因为 RR 算法不区分进程优先级，short 进程和 long 进程交替运行，故响应时间较长；而在 MLF 调度算法下，可以看到第二次和第三次运行的 short 进程的响应时间明显变少，几乎是 RR 算法下的一半，这是由于在首次运行 short 和 long 两个线程时，二者交替运行，均消耗了三次时间片，被调度到了最低优先级队列中，当第一个 short 进程（pid = 9）运行结束后，它唤醒了第二个 short 进程（pid = 10），且第二个 short 进程处于最高优先级 Q0 队列中，于是第二个 short 进程可以连续获得调度，因此响应时间更快，第三个 short 进程（pid = 11）同理，也可获得更快的响应速度。

当时间片较长（为 100）时，可以看到 RR 调度算法与 MLF 调度算法的结果大致相同，这是由于时间片划分过大，导致 long 进程也可以在一个时间片内运行结束，在这个条件下，每个进程只需一次调度即可运行完毕，故二者结果相同。

五、所遇问题及解决方法

本次实验主要考察对多级反馈队列调度算法与系统调用过程的理解，书上已经详细得讲解了这两个部分，并给出了部分参考代码，但是书上对阻塞时优先级升高的部分讲解较少，更是**没有提及需要更改 Timer_Interrupt_Handler()函数**，导致我在按照书上的要求修改完所有代码后，发现当运行 RR 调度算法时，线程会跑到后面的队列中，排查了许久才发现该函数没有对 RR 调度算法进行特判，导致当运行 RR 调度算法时，若时间片耗尽，线程仍然会进入低优先级队列中，因此该实验我耗时较长，希望老师之后能对教材进行完善并提醒同学该处需要注意的地方。

六、思考与练习

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？

首先修改全局时间片的个数等于线程队列的个数，并初始化时间为默认值 4：

```
/*
 * Settable quantum.
 */
int g_Quantum[4] = {4, 4, 4, 4};
```

然后需要在 Chang_Scheduling_Policy()函数中对这个变量进行赋值：

```
for (int i = 0; i < 4; i++)
{
    g_Quantum[i] = quantum[i];
}
```

接着在 Timer_Interrupt_Handler()函数中，根据当前线程所在队列确定时间片大小：

```
int quantum = g_Quantum[current->currentReadyQueue];

if (current->numTicks >= quantum)
{
    g_needReschedule = true;
```

最后修改 workload 用户程序，使得当调度策略为 mlf 时，从命令行中接收 4 个不同的时间片的值，通过系统调用传递给函数 Chang_Scheduling_Policy()：

```
if (argc >= 3)
{
    if (!strcmp(argv[1], "rr"))
    {
        policy = 0;
        quantum[0] = atoi(argv[0]);
    }
    else if (!strcmp(argv[1], "mlf"))
    {
        policy = 1;
        for (int i = 2; i < argc; i++)
            quantum[i] = atoi(argv[i]);
    }
```

这样一来就可以通过运行 workload mlf 1 2 3 4 来分别设置 0、1、2、3 号线程队列的时间片为 1，2，3，4。

2. 系统调用的作用是什么？简要描述它的执行过程。

通过系统调用，操作系可以使得用户线程调用内核线程内的函数，而不会破坏用户态与内核态的隔离机制，保护内核的安全。这是由于内核线程使用 GDT 保存段基址，而用户线程使用 LDT 保存段基址，二者处于不同的内存空间，无法相互直接调用，同时为了保护内核的安全性，从而引进了系统调用的概念，作为用户线程和内核线程之间的一座“桥梁”。

系统调用首先需要操作系统为用户提供相应的编程库函数，并在这些库中提供功能接口，用户通过调用这些功能接口，就可以实现对内核代码和数据的访问。具体来说，在 GeekOS 中，若用户程序想要调用内核中的某个库函数，首先需要启动一个系统中断，中断号为 0x90，相应的中断处理函数会使得操作系统进入内核态，并根据系统调用号 SysNum 参数访问 g_syscallTable，从而找到对应的库函数入口地址并跳转执行。

3. 如何为操作系统选择进程调度策略？

时间片轮转调度和四级反馈队列调度各有各的优缺点，需要根据实际任务情况加以选择。时间片轮转法相对公平，响应速度较快，适用于分时操作系统，但其不能区分任务的紧急程度，且需要频繁地进行进程切换，消耗较大；多级反馈队列调度是对其他调度算法的折中权衡，可以使用优先级区分紧急程度，适用于实时 OS，但其可能会导致某低优先级进程饥饿。