[**Linux 线程调度与优先级**](https://www.cnblogs.com/xiaotlili/p/3510224.html)

【转】

<http://blog.chinaunix.net/uid-20788636-id-1841334.html>

<http://blog.chinaunix.net/uid-20788636-id-1841335.html>

Linux内核的三种调度策略：  
  
　　1，SCHED\_OTHER 分时调度策略，  
　　2，SCHED\_FIFO实时调度策略，先到先服务。一旦占用cpu则一直运行。一直运行直到有更高优先级任务到达或自己放弃

　　3，SCHED\_RR实时调度策略，时间片轮转。当进程的时间片用完，系统将重新分配时间片，并置于就绪队列尾。放在队列尾保证了所有具有相同优先级的RR任务的调度公平

Linux线程优先级设置  
   首先，可以通过以下两个函数来获得线程可以设置的最高和最低优先级，函数中的策略即上述三种策略的宏定义：  
  
　　int sched\_get\_priority\_max(int policy);  
  
　　int sched\_get\_priority\_min(int policy);  
  
　 SCHED\_OTHER是不支持优先级使用的，而SCHED\_FIFO和SCHED\_RR支持优先级的使用，他们分别为1和99，数值越大优先级越高。  
设置和获取优先级通过以下两个函数

|  |
| --- |
| int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param); 　　int pthread\_attr\_getschedparam(const pthread\_attr\_t \*attr, struct sched\_param \*param);  param.sched\_priority = 51; //设置优先级 |

   系统创建线程时，默认的线程是SCHED\_OTHER。所以如果我们要改变线程的调度策略的话，可以通过下面的这个函数实现。

|  |
| --- |
| int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy); |

上面的param使用了下面的这个数据结构：

|  |
| --- |
| struct sched\_param {     int \_\_sched\_priority; //所要设定的线程优先级 }; |

我们可以通过下面的测试程序来说明，我们自己使用的系统的支持的优先级：

|  |
| --- |
| #include <stdio.h> #include <pthread.h> #include <sched.h> #include <assert.h>  static int get\_thread\_policy(pthread\_attr\_t \*attr) {   int policy;   int rs = pthread\_attr\_getschedpolicy(attr,&policy);   assert(rs==0);   switch(policy)   {   case SCHED\_FIFO:     printf("policy= SCHED\_FIFO\n");     break;   case SCHED\_RR:     printf("policy= SCHED\_RR");     break;   case SCHED\_OTHER:     printf("policy=SCHED\_OTHER\n");     break;   default:     printf("policy=UNKNOWN\n");     break;   }   return policy; }  static void show\_thread\_priority(pthread\_attr\_t \*attr,int policy) {   int priority = sched\_get\_priority\_max(policy);   assert(priority!=-1);   printf("max\_priority=%d\n",priority);   priority= sched\_get\_priority\_min(policy);   assert(priority!=-1);   printf("min\_priority=%d\n",priority); }  static int get\_thread\_priority(pthread\_attr\_t \*attr) {   struct sched\_param param;   int rs = pthread\_attr\_getschedparam(attr,&param);   assert(rs==0);   printf("priority=%d",param.\_\_sched\_priority);   return param.\_\_sched\_priority; }  static void set\_thread\_policy(pthread\_attr\_t \*attr,int policy) {   int rs = pthread\_attr\_setschedpolicy(attr,policy);   assert(rs==0);   get\_thread\_policy(attr); }  int main(void) {   pthread\_attr\_t attr;   struct sched\_param sched;   int rs;   rs = pthread\_attr\_init(&attr);   assert(rs==0);    int policy = get\_thread\_policy(&attr);   printf("Show current configuration of priority\n");     show\_thread\_priority(&attr,policy);   printf("show SCHED\_FIFO of priority\n");  show\_thread\_priority(&attr,SCHED\_FIFO);   printf("show SCHED\_RR of priority\n");   show\_thread\_priority(&attr,SCHED\_RR);   printf("show priority of current thread\n");   int priority = get\_thread\_priority(&attr);    printf("Set thread policy\n");   printf("set SCHED\_FIFO policy\n");   set\_thread\_policy(&attr,SCHED\_FIFO);   printf("set SCHED\_RR policy\n");   set\_thread\_policy(&attr,SCHED\_RR);   printf("Restore current policy\n");   set\_thread\_policy(&attr,policy);    rs = pthread\_attr\_destroy(&attr);   assert(rs==0);   return 0; } |

下面是测试程序的运行结果：

|  |
| --- |
| policy=SCHED\_OTHER Show current configuration of priority max\_priority=0 min\_priority=0 show SCHED\_FIFO of priority max\_priority=99 min\_priority=1 show SCHED\_RR of priority max\_priority=99 min\_priority=1 show priority of current thread priority=0Set thread policy set SCHED\_FIFO policy policy= SCHED\_FIFO set SCHED\_RR policy policy= SCHED\_RRRestore current policy policy=SCHED\_OTHER |

这里测试一下其中的两种特性，SCHED\_OTHER和SCHED\_RR，还有就是优先级的问题，是不是能够保证，高优先级的线程，就可以保证先运行。  
    下面的这个测试程序，创建了三个线程，默认创建的线程的调度策略是SCHED\_OTHER，其余的两个线程的调度策略设置成SCHED\_RR。我的Linux的内核版本是2.6.31。SCHED\_RR是根据时间片来确定线程的调度。时间片用完了，不管这个线程的优先级有多高都不会在运行，而是进入就绪队列中，等待下一个时间片的到了，那这个时间片到底要持续多长时间？在《深入理解Linux内核》中的第七章进程调度中，是这样描诉的，Linux采取单凭经验的方法，即选择尽可能长、同时能保持良好相应时间的一个时间片。这里也没有给出一个具体的时间来，可能会根据不同的CPU 来定，还有就是多CPU 的情况。

|  |
| --- |
| #include <stdio.h> #include <unistd.h> #include <stdlib.h> #include <pthread.h>  void Thread1() {   sleep(1);   int i,j;   int policy;   struct sched\_param param;   pthread\_getschedparam(pthread\_self(),&policy,&param);   if(policy == SCHED\_OTHER)     printf("SCHED\_OTHER\n");   if(policy == SCHED\_RR);   printf("SCHED\_RR 1 \n");   if(policy==SCHED\_FIFO)     printf("SCHED\_FIFO\n");    for(i=1;i<10;i++)   {     for(j=1;j<5000000;j++)     {     }     printf("thread 1\n");   }   printf("Pthread 1 exit\n"); }  void Thread2() {   sleep(1);   int i,j,m;   int policy;   struct sched\_param param; pthread\_getschedparam(pthread\_self(),&policy,&param);  if(policy == SCHED\_OTHER)     printf("SCHED\_OTHER\n");   if(policy == SCHED\_RR);   printf("SCHED\_RR\n");   if(policy==SCHED\_FIFO)     printf("SCHED\_FIFO\n");    for(i=1;i<10;i++)   {     for(j=1;j<5000000;j++)     {            }     printf("thread 2\n");   }   printf("Pthread 2 exit\n"); }  void Thread3() {   sleep(1);   int i,j;   int policy;   struct sched\_param param; pthread\_getschedparam(pthread\_self(),&policy,&param);  if(policy == SCHED\_OTHER)     printf("SCHED\_OTHER\n");   if(policy == SCHED\_RR)     printf("SCHED\_RR \n");   if(policy==SCHED\_FIFO)     printf("SCHED\_FIFO\n");    for(i=1;i<10;i++)   {     for(j=1;j<5000000;j++)     {     }     printf("thread 3\n");   }   printf("Pthread 3 exit\n"); }  int main() {   int i;   i = getuid();   if(i==0)     printf("The current user is root\n");   else     printf("The current user is not root\n");    pthread\_t ppid1,ppid2,ppid3;   struct sched\_param param;    pthread\_attr\_t attr,attr1,attr2;      pthread\_attr\_init(&attr1); pthread\_attr\_init(&attr); pthread\_attr\_init(&attr2);   param.sched\_priority = 51;  pthread\_attr\_setschedpolicy(&attr2,SCHED\_RR);  pthread\_attr\_setschedparam(&attr2,&param);  pthread\_attr\_setinheritsched(&attr2,PTHREAD\_EXPLICIT\_SCHED);//要使优先级其作用必须要有这句话   param.sched\_priority = 21;  pthread\_attr\_setschedpolicy(&attr1,SCHED\_RR);  pthread\_attr\_setschedparam(&attr1,&param);  pthread\_attr\_setinheritsched(&attr1,PTHREAD\_EXPLICIT\_SCHED);    pthread\_create(&ppid3,&attr,(void \*)Thread3,NULL);  pthread\_create(&ppid2,&attr1,(void \*)Thread2,NULL);  pthread\_create(&ppid1,&attr2,(void \*)Thread1,NULL);    pthread\_join(ppid3,NULL);  pthread\_join(ppid2,NULL);  pthread\_join(ppid1,NULL);  pthread\_attr\_destroy(&attr2);  pthread\_attr\_destroy(&attr1);  return 0; } |

下面是该程序的其中之一的运行结果：

|  |
| --- |
| sudo ./prio\_test The current user is root SCHED\_OTHER SCHED\_RR SCHED\_RR 1  thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 Pthread 1 exit thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 Pthread 2 exit thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 Pthread 3 exit |

   这里我们可以看到，由于线程3的调度策略是SCHED\_OTHER，而线程2的调度策略是SCHED\_RR，所以，在Thread3中，线程3被线程1，线程2给抢占了。由于线程1的优先级大于线程2的优先级，所以，在线程1以先于线程2运行，不过，这里线程2有一部分代码还是先于线程1运行了。  
    我原以为，只要线程的优先级高，就会一定先运行，其实，这样的理解是片面的，特别是在SMP的PC机上更会增加其不确定性。  
    其实，普通进程的调度，是CPU根据进程优先级算出时间片，这样并不能一定保证高优先级的进程一定先运行，只不过和优先级低的进程相比，通常优先级较高的进程获得的CPU时间片会更长而已。其实，如果要想保证一个线程运行完在运行另一个线程的话，就要使用多线程的同步技术，信号量，条件变量等方法。而不是绝对依靠优先级的高低，来保证。  
    不过，从运行的结果上，我们可以看到，调度策略为SCHED\_RR的线程1，线程2确实抢占了调度策略为SCHED\_OTHER的线程3。这个是可以理解的，由于SCHER\_RR是实时调度策略。  
   只有在下述事件之一发生时，实时进程才会被另外一个进程取代。  
  （1） 进程被另外一个具有更高实时优先级的实时进程抢占。  
  （2） 进程执行了阻塞操作并进入睡眠  
  （3）进程停止（处于TASK\_STOPPED 或TASK\_TRACED状态）或被杀死。  
  （4）进程通过调用系统调用sched\_yield()，自愿放弃CPU 。  
  （5）进程基于时间片轮转的实时进程（SCHED\_RR），而且用完了它的时间片。  
   基于时间片轮转的实时进程是，不是真正的改变进程的优先级，而是改变进程的基本时间片的长度。所以基于时间片轮转的进程调度，并不能保证高优先级的进程先运行。  
   下面是另一种运行结果：

|  |
| --- |
| sudo ./prio\_test The current user is root SCHED\_OTHER SCHED\_RR 1  thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 thread 1 Pthread 1 exit SCHED\_RR thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 thread 2 Pthread 2 exit thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 thread 3 Pthread 3 exit |

  可以看出并没有每一次都保证高优先级的线程先运行。