**北京邮电大学软件学院、**

**2017-2018学年第1学期实验报告**

**课程名称： 操作系统**

**实验名称：** 实验四 进程调度

**实验完成人：**

**姓名：**\_\_\_\_\_苏可欣\_\_\_**学号：**\_\_\_2016211954\_\_\_\_\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_陈晋鹏\_\_\_\_\_\_\_\_\_\_

**日 期： 2018 年 5 月 29 日**

1 实验目的

1. 理解 Linux 管理进程所用到的数据结构。

2. 理解 Linux 的进程调度算法的处理逻辑及其实现所用到的数据结构。

2 实验内容

1. 通过查阅参考书或者上网找资料，熟悉/usr/src/linux（注意：这里最后一级目录名

可能是个含有具体内核版本号和“linux”字符串的名字）下各子目录的内容，即所含

Linux源代码的情况。

2. 分析 Linux 进程调度有关函数的源代码，主要是 schedule()函数，并且要对它们引

用的头文件等一并分析。

3. 实现 Linux 的进程调度算法及理解其实现所用的主要数据结构。

3 实验准备

3.1.Linux进程状态的描述

Linux将进程状态描述为如下五种：

TA S K \_ R U N N I N G：可运行状态。处于该状态的进程可以被调度执行而成为当

前进程。

TA S K \_ I N T E R R U P T I B L E：可中断的睡眠状态。处于该状态的进程在所需资源

有效时被唤醒，也可以通过信号或定时中断唤醒。

TA S K \_ U N I N T E R R U P T I B L E：不可中断的睡眠状态。处于该状态的进程仅当所

需资源有效时被唤醒。

TASK\_ZOMBIE：僵死状态。表示进程结束且已释放资源，但其 task\_struct 仍

未释放。

TA S K \_ S TO P P E D：暂停状态。处于该状态的进程通过其他进程的信号才能被

唤醒。

3.2. 进程的虚拟地址空间

调度方式Linux中的每个进程都分配有一个相对独立的虚拟地址空间。该虚存

空间分为两部分：用户空间包含了进程本身的代码和数据；内核空间包含了操作系

统的代码和数据。Linux采用“有条件的可剥夺”调度方式。对于普通进程，当其时间

片结束时，调度程序挑选出下一个处于TA S K \_ R U N N I N G状态的进程作为当前进程

（自愿调度）。对于实时进程，若其优先级足够高，则会从当前的运行进程中抢占

CPU 成为新的当前进程（强制调度）。发生强制调度时，若进程在用户空间中运行，

就会直接被剥夺CPU；若进程在内核空间中运行，即使迫切需要其放弃CPU，也仍

要等到从它系统空间返回的前夕才被剥夺CPU。

－! －

15

3.3. 调度策略

1）SCHED\_OTHER

SCHED\_OTHER 是面向普通进程的时间片轮转策略。采用该策略时，系统为

处于TA S K \_ R U N N I N G状态的每个进程分配一个时间片。当时间片用完时，进程调

度程序再选择下一个优先级相对较高的进程，并授予CPU 使用权。

2）SCHED\_FIFO

SCHED\_FIFO 策略适用于对响应时间要求比较高，运行所需时间比较短的实时

进程。采用该策略时，各实时进程按其进入可运行队列的顺序依次获得 CPU。除了

因等待某个事件主动放弃CPU，或者出现优先级更高的进程而剥夺其CPU 之外，该

进程将一直占用CPU运行。

3）SCHED\_RR

SCHED\_RR 策略适用于对响应时间要求比较高，运行所需时间比较长的实时

进程。采用该策略时，各实时进程按时间片轮流使用 CPU。当一个运行进程的时间

片用完后，进程调度程序停止其运行并将其置于可运行队列的末尾。

3.4. 相关函数

1）Schedule()函数

Schedule()函数首先对所有任务（进程）进行检测，唤醒任何一个得到信号的任务。

具 体方法是针对任务数组中的每个任务，检查其报警定时值alarm。如果任务的

alarm时间已经 过期（alarm<jiffies），则在它的信号位图中设置SIGALRM信号，然

后清alarm值。jiffies是 系统从开机开始算起的滴答数（10ms/滴答），在sched..h 中

定义。如果进程的信号位图中 除去被阻塞的信号外还有其他信号，并且任务处于可

中断睡眠状态 （TA S K \_ I N T E R R U P T I B L E），则置任务为就绪状态

（TASK\_RUNNING）。

2）sleep\_on()函数

sleep\_on()函数的主要功能是当一个进程（或任务）所请求的资源正忙或不在内存中

时 暂时切换出去，放在等待队列中等待一段时间，当切换回来后再继续运行。放入

等待队列的 方式是利用了函数中的tmp指针作为各个正在等待任务的联系。

3）wake\_up()函数 唤醒操作函数wake\_up()把正在等待可用资源的指定任务置为就绪

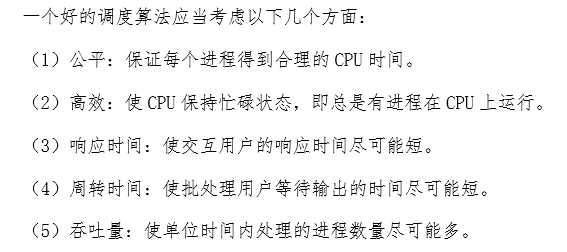
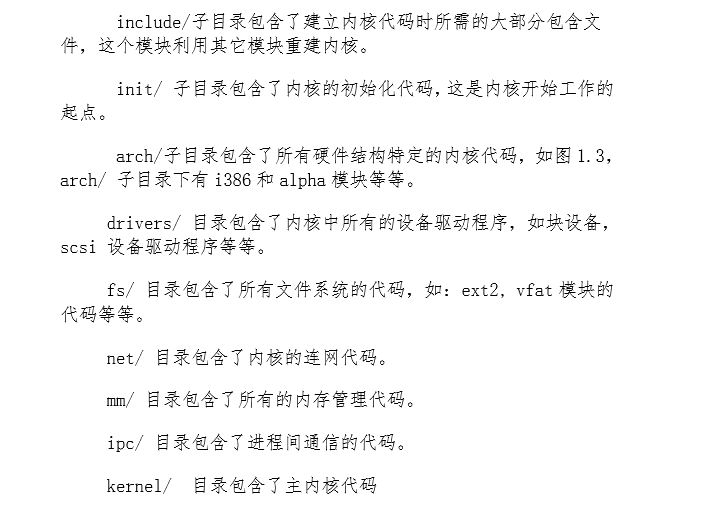
状态。该函数是一个通用唤醒函数。在有些情况下，例如读取磁盘上的数据块，由

于等待队列中的任何一个任务 都可能被先唤醒，因此还需要把被唤醒任务结构的指

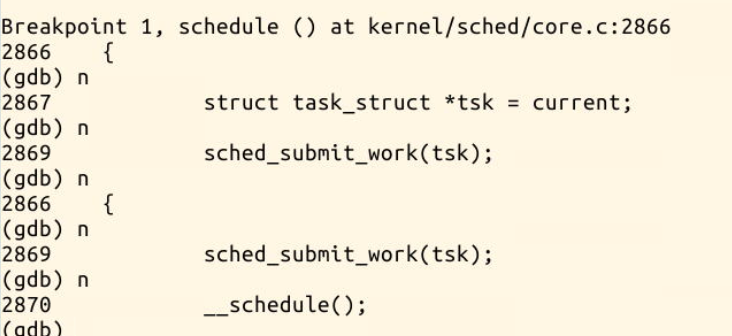
针置空。这样，在其后进入睡眠的进程 被唤醒而又重新执行sleep\_on()时，就无需

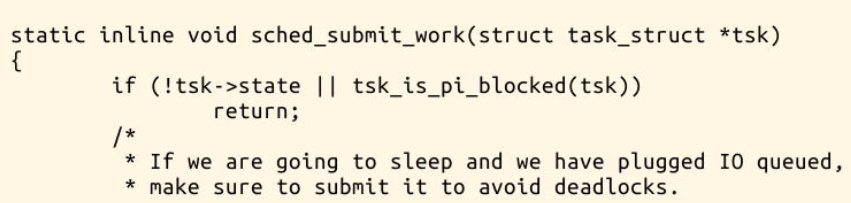
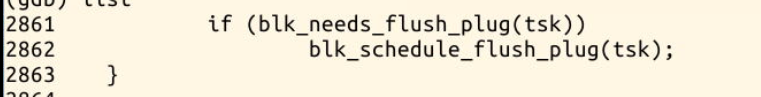
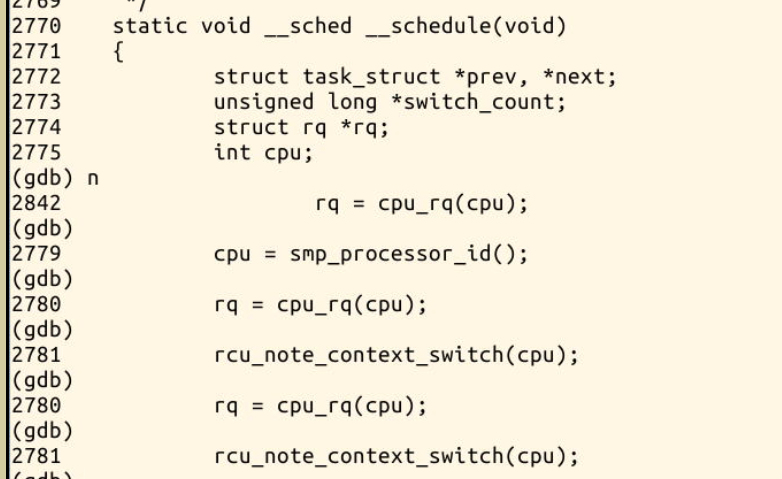
唤醒该进程了。

3.5程序示例



schedule();  
struct task\_struct \*tsk = current; //current是当前进程  
sched\_submit\_work(tsk);  //避免死锁  
\_\_schedule();//这就是调度的主函数了  
  
static void \_\_sched \_\_schedule(void)  
{  
struct task\_struct \*prev, \*next;  
unsigned long \*switch\_count;  
struct rq \*rq;  
int cpu;  
  
need\_resched:  
preempt\_disable();   //关闭内核抢占，关于内核抢占详见注释1  
cpu = smp\_processor\_id();  
rq = cpu\_rq(cpu);   //跟当前进程相关的runqueue的信息被保存在rq中  
rcu\_note\_context\_switch(cpu);  
prev = rq->curr;   //当前进程放入prev  
  
schedule\_debug(prev);  
  
if (sched\_feat(HRTICK))  
hrtick\_clear(rq);  
  
raw\_spin\_lock\_irq(&rq->lock);  
  
switch\_count = &prev->nivcsw;  
//如果内核态没有被抢占，并且内核抢占有效  
if (prev->state && !(preempt\_count() & PREEMPT\_ACTIVE)) {  
//如果当前进程有非阻塞等待信号，并且它的状态是TASK\_INTERRUPTIBLE  
if (unlikely(signal\_pending\_state(prev->state, prev))) {  
prev->state = TASK\_RUNNING; //将当前进程的状态设为：TASK\_RUNNING  
}  
else{  
deactivate\_task(rq,prev, DEQUEUE\_SLEEP);//将当前进程从runqueue(运行队列)中删除   
prev->on\_rq = 0;  //标识当前进程不在runqueue中  
  
  //这里涉及到工作队列的知识，我们在以后的章节里在来说，这里略过  
if (prev->flags & PF\_WQ\_WORKER) {  
struct task\_struct \*to\_wakeup;  
  
to\_wakeup = wq\_worker\_sleeping(prev, cpu);  
if (to\_wakeup)  
try\_to\_wake\_up\_local(to\_wakeup);  
}  
}  
switch\_count = &prev->nvcsw;  
}  
  
pre\_schedule(rq, prev);  
  
if (unlikely(!rq->nr\_running))//如果runqueue中没有正在运行的进程  
idle\_balance(cpu, rq); //就会从其它CPU拉入进程  
  
put\_prev\_task(rq, prev);   //通知调度器，当前进程要被另一个进程取代，做好准备  
next = pick\_next\_task(rq); //从runqueue中选择最适合的进程  
clear\_tsk\_need\_resched(prev); //清除当前进程的重调度标识  
rq->skip\_clock\_update = 0;  
//当前进程与所选进程是否是同一进程，不属于同一进程才需要切换  
if (likely(prev != next)) {  
rq->nr\_switches++;  
rq->curr = next; //所选进程代替当前进程  
++\*switch\_count;  
  
context\_switch(rq, prev, next); //负责底层上下文切换  
  
cpu = smp\_processor\_id();  
rq = cpu\_rq(cpu);  
}  
else  
raw\_spin\_unlock\_irq(&rq->lock);  //如果不需要切换进程，则只需要解锁  
  
post\_schedule(rq);  
  
sched\_preempt\_enable\_no\_resched();  
if (need\_resched())  
goto need\_resched;

进行调试跟踪schedule的执行过程。   
  
进程调度时，首先进入schedule()函数，将一个task\_struct结构体的指针tsk赋值为当前进程。   
然后调用sched\_submit\_work(tsk)   
我们进入这个函数，查看一下做了什么工作   
我们在执行到sched\_submit\_work时，输入si进入函数。

  
可以看到这个函数时检测tsk->state是否为0 （runnable）若为运行态时则返回，   
tsk\_is\_pi\_blocked(tsk),检测tsk的死锁检测器是否为空，若非空的话就return。   
  
然后检测是否需要刷新plug队列，用来避免死锁。   
sched\_submit\_work主要是来避免死锁。   
然后我们进入\_\_schedule()函数。   
  
\_\_schedule()是切换进程的真正代码，我们来分析一下具体的关键代码   
1.创建一些局部变量，

struct task\_struct \*prev, \*next;//当前进程和一下个进程的进程结构体

unsigned long \*switch\_count;//进程切换次数

struct rq \*rq;//就绪队列

int cpu;

关闭内核抢占，初始化一部分变量

need\_resched:

preempt\_disable();//关闭内核抢占

cpu = smp\_processor\_id();

rq = cpu\_rq(cpu);//与CPU相关的runqueue保存在rq中

rcu\_note\_context\_switch(cpu);

prev = rq->curr;//将runqueue当前的值赋给p

3.选择next进程

next = pick\_next\_task(rq, prev);//挑选一个优先级最高的任务排进队列

clear\_tsk\_need\_resched(prev);//清除prev的TIF\_NEED\_RESCHED标志。

clear\_preempt\_need\_resched();

4.完成进程的调度

if (likely(prev != next)) {//如果prev和next是不同进程

rq->nr\_switches++;//队列切换次数更新

rq->curr = next;

++\*switch\_count;//进程切换次数更新

context\_switch(rq, prev, next); /\* unlocks the rq \*///进程上下文的切换

/\*

\* The context switch have flipped the stack from under us

\* and restored the local variables which were saved when

\* this task called schedule() in the past. prev == current

\* is still correct, but it can be moved to another cpu/rq.

\*/

cpu = smp\_processor\_id();

rq = cpu\_rq(cpu);

} else//如果是同一个进程不需要切换

raw\_spin\_unlock\_irq(&rq->lock);

这段代码中context\_switch(rq,prev,next)完成了从prev到next的进程上下文的切换。我们进入这个函数查看

static inline void

context\_switch(struct rq \*rq, struct task\_struct \*prev,

struct task\_struct \*next)

{

struct mm\_struct \*mm, \*oldmm;//初始化进程地址管理结构体mm和oldmm

prepare\_task\_switch(rq, prev, next);//完成进程切换的准备工作

mm = next->mm;

oldmm = prev->active\_mm;

/\*完成mm\_struct的切换\*/

if (!mm) {

next->active\_mm = oldmm;

atomic\_inc(&oldmm->mm\_count);

enter\_lazy\_tlb(oldmm, next);

} else

switch\_mm(oldmm, mm, next);

if (!prev->mm) {

prev->active\_mm = NULL;

rq->prev\_mm = oldmm;

}

switch\_to(prev, next, prev);//进程切换的核心代码

barrier();

finish\_task\_switch(this\_rq(), prev);

}

我们看到在context\_switch中使用switch\_to(prev,next,prev)来切换进程。我们查看一下switch\_to的代码。   
switch\_to是一个宏定义，完成进程从prev到next的切换，首先保存flags，然后保存当前进程的ebp，然后把当前进程的esp保存到prev->thread.sp中，然后把标号1:的地址保存到prev->thread.ip中。   
然后把next->thread.ip压入堆栈。这里，如果之前B也被switch\_to出去过，那么next->thread.ip里存的就是下面这个1f的标号，但如果

next进程刚刚被创建，之前没有被switch\_to出去过，那么next->thread.ip里存的将是ret\_ftom\_fork   
\_\_switch\_canqry应该是现代操作系统防止栈溢出攻击的金丝雀技术。   
jmp \_\_switch\_to使用regparm call, 参数不是压入堆栈，而是使用寄存器传值，来调用\_\_switch\_to   
eax存放prev,edx存放next。这里为什么不用call \_\_switch\_to而用jmp，因为call会导致自动把下面这句话的地址(也就是1:)压栈，然后\_\_switch\_to()就必然只能ret到这里，而无法根据需要ret到ret\_from\_fork   
当一个进程再次被调度时，会从1:开始执行，把ebp弹出，然后把flags弹出。、

#define switch\_to(prev, next, last)

do {

/\*

\* Context-switching clobbers all registers, so we clobber

\* them explicitly, via unused output variables.

\* (EAX and EBP is not listed because EBP is saved/restored

\* explicitly for wchan access and EAX is the return value of

\* \_\_switch\_to())

\*/

unsigned long ebx, ecx, edx, esi, edi;

asm volatile("pushfl\n\t" /\* save flags \*/

"pushl %%ebp\n\t" /\* save EBP \*/

"movl %%esp,%[prev\_sp]\n\t" /\* save ESP \*/

"movl %[next\_sp],%%esp\n\t" /\* restore ESP \*/

"movl *$1f,%[prev\_ip]\n\t" /\* save EIP \*/*

*"pushl %[next\_ip]\n\t" /\* restore EIP \*/*

*\_\_switch\_canary*

*"jmp \_\_switch\_to\n" /\* regparm call \*/*

*"1:\t"*

*"popl %%ebp\n\t" /\* restore EBP \*/*

*"popfl\n" /\* restore flags \*/*

*/\* output parameters \*/*

*: [prev\_sp] "=m" (prev->thread.sp),*

*[prev\_ip] "=m" (prev->thread.ip),*

*"=a" (last),*

*/\* clobbered output registers: \*/*

*"=b" (ebx), "=c" (ecx), "=d" (edx),*

*"=S" (esi), "=D" (edi)*

*\_\_switch\_canary\_oparam*

*/\* input parameters: \*/*

*: [next\_sp] "m" (next->thread.sp),*

*[next\_ip] "m" (next->thread.ip),*

*/\* regparm parameters for \_\_switch\_to(): \*/*

*[prev] "a" (prev),*

*[next] "d" (next)*

*\_\_switch\_canary\_iparam*

*: /\* reloaded segment registers \*/*

*"memory");*

5.开启抢占

sched\_preempt\_enable\_no\_resched();

if (need\_resched())

goto need\_resched;

到此，进程的切换过程就完成了。   
总结：

自己补充。

3.4实验结果

结果展示以及分析原因。

－!－

/home/sue/mycode/OSLab/Lab4/cmake-build-debug/Lab4

create main thread: ok

Process pid: 1 priority: 18 run time: 14

Process pid: 2 priority: 18 run time: 10

Process pid: 3 priority: 4 run time: 11

Process pid: 4 priority: 12 run time: 3

Process pid: 5 priority: 15 run time: 10

Process pid: 6 priority: 9 run time: 7

Process pid: 7 priority: 1 run time: 1

Process pid: 8 priority: 6 run time: 7

Process pid: 9 priority: 19 run time: 12

Process pid: 10 priority: 11 run time: 14

Process pid: 11 priority: 10 run time: 10

Process pid: 12 priority: 2 run time: 10

Process pid: 13 priority: 10 run time: 1

Process pid: 14 priority: 18 run time: 1

Process pid: 15 priority: 10 run time: 2

Process pid: 16 priority: 17 run time: 8

Process pid: 17 priority: 12 run time: 13

Process pid: 18 priority: 15 run time: 15

Process pid: 19 priority: 4 run time: 4

Process pid: 20 priority: 18 run time: 19

--------------FCFS----------------------

Thread1: start 0; run time 14

Thread2: start 14; run time 10

Thread3: start 24; run time 11

Thread4: start 35; run time 3

Thread5: start 38; run time 10

Thread6: start 48; run time 7

Thread7: start 55; run time 1

Thread8: start 56; run time 7

Thread9: start 63; run time 12

Thread10: start 75; run time 14

Thread11: start 89; run time 10

Thread12: start 99; run time 10

Thread13: start 109; run time 1

Thread14: start 110; run time 1

Thread15: start 111; run time 2

Thread16: start 113; run time 8

Thread17: start 121; run time 13

Thread18: start 134; run time 15

Thread19: start 149; run time 4

Thread20: start 153; run time 19

total waiting time: 1596

average waiting time: 79.8

--------------SJF----------------------

Thread7: start 0; run time 1

Thread13: start 1; run time 1

Thread14: start 2; run time 1

Thread15: start 3; run time 2

Thread4: start 5; run time 3

Thread19: start 8; run time 4

Thread6: start 12; run time 7

Thread8: start 19; run time 7

Thread16: start 26; run time 8

Thread2: start 34; run time 10

Thread5: start 44; run time 10

Thread11: start 54; run time 10

Thread12: start 64; run time 10

Thread3: start 74; run time 11

Thread9: start 85; run time 12

Thread17: start 97; run time 13

Thread1: start 110; run time 14

Thread10: start 124; run time 14

Thread18: start 138; run time 15

Thread20: start 153; run time 19

total waiting time: 1053

average waiting time: 52.65

enter RR time:

10

--------------RR----------------------

14

Thread1: start 0; run time 14

10

Thread2: start 10; run time 10

11

Thread3: start 20; run time 11

3

Thread4: start 30; run time 3

10

Thread5: start 33; run time 10

7

Thread6: start 43; run time 7

1

Thread7: start 50; run time 1

7

Thread8: start 51; run time 7

12

Thread9: start 58; run time 12

14

Thread10: start 68; run time 14

10

Thread11: start 78; run time 10

10

Thread12: start 88; run time 10

1

Thread13: start 98; run time 1

1

Thread14: start 99; run time 1

2

Thread15: start 100; run time 2

8

Thread16: start 102; run time 8

13

Thread17: start 110; run time 13

15

Thread18: start 120; run time 15

4

Thread19: start 130; run time 4

19

Thread20: start 134; run time 19

10

Thread1: start 144; run time 14

10

Thread2: start 154; run time 10

10

Thread3: start 164; run time 11

10

Thread4: start 174; run time 3

10

Thread5: start 184; run time 10

10

Thread6: start 194; run time 7

10

Thread7: start 204; run time 1

10

Thread8: start 214; run time 7

10

Thread9: start 224; run time 12

10

Thread10: start 234; run time 14

10

Thread11: start 244; run time 10

10

Thread12: start 254; run time 10

10

Thread13: start 264; run time 1

10

Thread14: start 274; run time 1

10

Thread15: start 284; run time 2

10

Thread16: start 294; run time 8

10

Thread17: start 304; run time 13

10

Thread18: start 314; run time 15

10

Thread19: start 324; run time 4

10

Thread20: start 334; run time 19

total waiting time: 4808

average waiting time: 240.4

--------------priority-Scheduling----------------------

Thread7: start 0; run time 1

Thread12: start 1; run time 10

Thread3: start 11; run time 11

Thread19: start 22; run time 4

Thread8: start 26; run time 7

Thread6: start 33; run time 7

Thread11: start 40; run time 10

Thread13: start 50; run time 1

Thread15: start 51; run time 2

Thread10: start 53; run time 14

Thread4: start 67; run time 3

Thread17: start 70; run time 13

Thread5: start 83; run time 10

Thread18: start 93; run time 15

Thread16: start 108; run time 8

Thread1: start 116; run time 14

Thread2: start 130; run time 10

Thread14: start 140; run time 1

Thread20: start 141; run time 19

Thread9: start 160; run time 12

total waiting time: 1395

average waiting time: 69.75

Process finished with exit code 0

由结果可见SJF的平均等待时间最短，最短作业优先调度（SJF）   
原因：采用SJF调度会选择具有最短CPU运行时间的进程分配CPU使用权。如果两个进程的CPU区间相同，则按照FCFS来进行选择。  
SJF调度可以证明是最佳的，它降低了**平均等待时间**。