第 6 章 进程调度

月出皓兮　苏曙光老师的课堂笔记

2022 年 4 月 14 日

目录

1 进程调度（schedule） 2

1.1 定义 2
1.2 分类 2
1.3 目标 2
1.3.1 周转时间/平均周转时间 2
1.3.2 带权周转时间/平均带权周转时间 3
1.4 进程调度算法 3
1.4.1 先来先服务调度（First Come First Serve） 3
1.4.2 短作业优先调度算法（Short Job First） 3
1.4.3 响应比高者优先调度算法 3
1.4.4 优先数调度算法 4
1.4.5 循环轮转调度法（ROUND-ROBIN） 4
1.5 进程调度方式 5
1.5.1 非抢占方式 5
1.5.2 抢占方式 5
1.6 Linux 调度机制 5
1.6.1 宏观评价 5
1.6.2 关键参量 priority, counter, rt_priority, policy 5
1.6.3 nice 指令 6
1.6.4 调度函数的实现 6
1.6.5 调度时钟 do_timer() 函数 6
1.6.6 调度函数 schedule 7

1 进程调度（schedule）

1.1 定义

在一个队列中，按某种策略选择一个最合适个体。

1.2 分类

1. 长程调度/宏观调度/作业调度
2. 中程调度/交换调度
3. 短程调度/进程调度
4. I/O 调度/设备调度

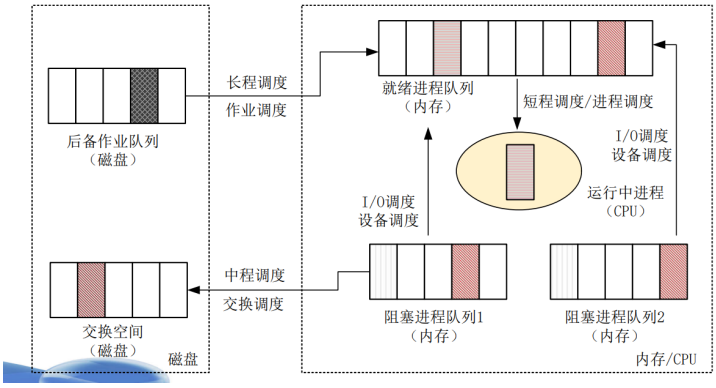

图 1: 进程调度分类

在这里我们主要讨论的是短程调度（一直将其称为进程调度）。
在合适的时候以一定策略选择一个就绪进程运行.

1.3 目标

1.3.1 周转时间/平均周转时间

周转时间，即进程提交给计算机到完成所花费的时间（t），周转时间说明了进程在系统中停留时间的长短。

t = tc - ts

ts——进程的提交时间（Start）

tc——进程的完成时间（Complete ）

平均周转时间，平均周转时间越短，意味着这些进程在系统内停留的时间越短，因而**系统吞吐量**也就越大，**资源利用率**也越高。

t = （t1 + t2 + ···+ tn）/ n

1.3.2 带权周转时间/平均带权周转时间

带权周转时间 w，说明了进程在系统中的相对停留时间。

w = **周转时间 / 进程运行时间** = t / tr

t：进程的周转时间

tr ： 进程的运行时间（run）

平均带权周转时间。

w = （w1 + w2 + ···+ wn）/ n

1.4 进程调度算法

1.4.1 先来先服务调度（First Come First Serve）

按照作业进入系统的时间先后次序来挑选作业。先进入系统的作业优先被运行。

其特点有：

容易实现，效率不高

只考虑作业的等候时间，而没考虑运行时间的长短。因此一个晚来但是很短的作业可能需要等待很长时间才能被运行，因而本算法**不利于短作业**。

1.4.2 短作业优先调度算法（Short Job First）

参考运行时间，选取时间最短的作业投入运行。

其特点有：

易于实现，效率不高

忽视了作业等待时间，**一个早来但是很长的作业**将会在很长时间得不到调度，易出现资源“饥饿”的现象。

1.4.3 响应比高者优先调度算法

调度作业时计算作业列表中每个作业的响应比，选择响应比最高的作业优先投入运行。

响应比定义：作业的响应时间和与运行时间的比值

响应比 = 响应时间/运行时间 =（等待时间 + 运行时间）/运行时间 = 1 + 等待时间 / 运行时间。

特点：

有利于短作业

有利于等候已久的作业。

兼顾长作业

1.4.4 优先数调度算法

根据进程优先数，把 CPU 分配给最高的进程。

进程优先数 = 静态优先数 + 动态优先数

进程创建时确定，在整个进程运行期间不再改变。静态优先数的确定基于

1. 进程所需的资源多少（不一定）

2. 基于程序运行时间的长短（长的进程静态优先数应该越小）

3. 基于进程的类型（IO/CPU 中偏向 IO 的进程交互性强，优先数高；前台/后台中偏向前台的进程对于用户体验比较重要，优先数高，核心/用户中用户态进程优先数更高）

动态优先数在进程运行期间可以改变。

1. 当使用 CPU 超过一定时长时：适当降低

2. 当进程等待时间超过一定时长时：适当提高

3. 当进行 I/O 操作后：适当提高

1.4.5 循环轮转调度法（ROUND-ROBIN）

把所有就绪进程按先进先出的原则排成队列。新来进程加到队列末尾。进程以时间片 q 为单位轮流使用 CPU。刚刚运行了一个时间片的进程排到队列末尾，等候下一轮调度。队列逻辑上是环形的。

优点：

1. 公平性：每个就绪进程有平等机会获得 CPU

2. 交互性：每个进程等待 (N-1)* q 的时间就可以重新获得 CPU

时间片 q 的大小，如果 q 太大，交互性差。甚至退化为 FCFS 调度算法。如果 q 太小，进程切换频繁，系统开销增加。

改进：时间片的大小可变 (可变时间片轮转调度法)，组织多个就绪队列 (多重时间片循环轮转)

1.5 进程调度方式

当一进程正在 CPU 上运行时，若有更高优先级的进程需要运行，系统如何分配 CPU。

1.5.1 非抢占方式

让正在运行的进程继续执行，直到该进程完成或发生某事件而进入“完成”或“阻塞”状态时，才把 CPU 分配给新来的更高优先级的进程。

1.5.2 抢占方式

当更高优先级的进程来到时，便暂停正在运行的进程，立即把 CPU 分配给新来的优先级更高的进程。

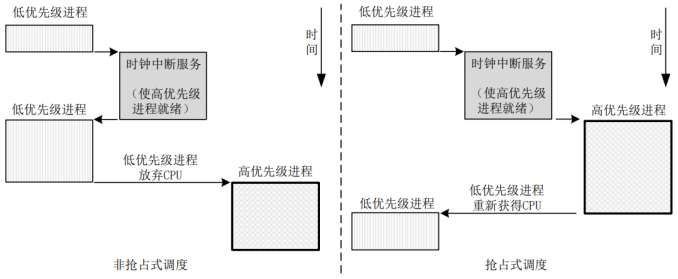

图 2: 非抢占方式与抢占方式的对比

1.6 Linux 调度机制

1.6.1 宏观评价

基于**优先级调度**。支持普通进程，也支持实时进程；实时进程优先于普通进程；普通进程公平使用 CPU 时间。

1.6.2 关键参量 priority, counter, rt_priority, policy

以下变量均定义在 task_struck 结构体中。(以 Linux0.11 为例)
priority：进程（包括实时和普通）的静态优先级

counter：进程能连续运行的时间（动态优先级）其表示的是当前一轮调度中：进程能连续运行的时间片数量。较高优先级的进程有更大的 counter，counter 初值 = priority。counter 的改变：时钟中断服务程序：counter，在新的一轮调度开始时：counter 初值 = priority。

policy 指明进程使用何种调度策略。

```
/*
 * Scheduling policies
 */
#define SCHED_OTHER     0   普通的分时进程
#define SCHED_FIFO      1   先进先出的实时进程
#define SCHED_RR        2   基于优先级轮转的实时进程
```

图 3: 进程使用何种调度策略

rt_priority 实时进程特有的优先级：rt_priority+1000

1.6.3 nice 指令

Linux nice 命令以更改过的优先序来执行程序，如果未指定程序，则会印出目前的排程优先序。

nice -n 数字进程

nice 范围：-20（最高）~ 19（最低）

默认 nice = 0

普通用户：可以调整自己进程，而 nice 范围 [0, 19]。

Root 用户：可以调整任何进程，而 nice 范围 [-20, 19]。

1.6.4 调度函数的实现

调度函数 schedule 在可运行队列中找到一个进程给它分配 CPU。

调用时机：

直接调度:时钟中断,即 do_timer(),当资源无法满足被阻塞时,sleep_on( )。

间接调度/松散调度：进程从内核态返回到用户态前。

1.6.5 调度时钟 do_timer() 函数

时钟中断处理函数

```
// 对于一个进程由于执行时间片用完时，则进城任务切换。并执行一个计时更新工作。
void do_timer(long cpl)
{
    extern int beepcount;               // 扬声器发声滴答数
    extern void sysbeepstop(void);      // 关闭扬声器。

    // 如果发声计数次数到，则关闭发声。(向0x61口发送命令，复位位0和1，位0
    // 控制8253计数器2的工作，位1控制扬声器)
    if (beepcount)
        if (!--beepcount)
            sysbeepstop();

    // 如果当前特权级(cpl)为0，则将内核代码运行时间stime递增；
    if (cpl)
        current->utime++;
    else
        current->stime++;

    // 如果有定时器存在，则将链表第1个定时器的值减1.如果已等于0，则调用相应的
    // 处理程序，并将该处理程序指针置空。然后去掉该项定时器。next_timer是定时器
    // 链表的头指针。
    if (next_timer) {
        next_timer->jiffies--;
        while (next_timer && next_timer->jiffies <= 0) {
            void (*fn)(void);       // 这里插入了一个函数指针定义!!!! o(￣ヘ￣)o

            fn = next_timer->fn;
            next_timer->fn = NULL;
            next_timer = next_timer->next;
            (fn)();                 // 调用处理函数
        }
    }
    // 如果当前软盘控制器FDC的数字输出寄存器中马达启动位有置位的，则执行软盘定时程序
    if (current_DOR & 0xf0)
        do_floppy_timer();
    // 如果进程运行时间还没完，则退出。否则置当前任务计数值为0.并且若发生时钟中断
    // 正在内核代码中运行则返回，否则调用执行调度函数。
    if ((--current->counter)>0) return;
    current->counter=0;
    if (!cpl) return;                       // 内核态程序不依赖counter值进行调度
    schedule();
} « end do_timer »

// 系统调用功能 - 设置报警定时时间值(秒)
```

1.6.6 调度函数 schedule

第一步：选择进程，扫描可运行队列，选择一个合适进程

第二步：切换进程，把当前进程放到适当的等待队列里。调用 schedule( )，让新的进程运行。

```c
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

    // 处理信号
    // 从任务数组中最后一个任务开始循环检测alarm。在循环时跳过空指针项。
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            // 如果设置过任务的定时值alarm，并且已经过期(alarm<jiffies)，则在
            // 信号位图中置SIGALRM信号，即向任务发送SIGALARM信号。然后清alarm。
            // 该信号的默认操作是终止进程。jiffies是系统从开机开始算起的滴答数(10ms/滴答)。
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                    (*p)->signal |= (1<<(SIGALRM-1));
                    (*p)->alarm = 0;
                }
            // 如果信号位图中除被阻塞的信号外还有其他信号，并且任务处于可中断状态，则
            // 置任务为就绪状态。其中'~(_BLOCKABLE & (*p)->blocked)'用于忽略被阻塞的信号，但
            // SIGKILL 和SIGSTOP不能叫阻塞。
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */

    //选择优先级最高的进程
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较
        // 每个就绪状态任务的counter(任务运行时间的递减滴答计数)值，哪一个值大，运行时间还
        // 不长，next就值向哪个的任务号。
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        // 如果比较得出有counter值不等于0的结果，或者系统中没有一个可运行的任务存在(此时c
        // 仍然为-1，next=0),则退出while(1)_的循环，执行switch任务切换操作。否则就根据每个
        // 任务的优先权值，更新每一个任务的counter值，然后回到while(1)循环。counter值的计算
        // 方式counter=counter/2 + priority.注意：这里计算过程不考虑进程的状态。
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + //注意此处的修正项
                        (*p)->priority;
    } « end while 1 »
    // 用下面的宏把当前任务指针current指向任务号Next的任务5
    // 并切换到该任务中运行。上面Next
    // 被初始化为0。此时任务0仅执行pause()系统调用，并又会调用本函数。
    switch_to(next);     // 切换到Next任务并运行。
} « end schedule »
```

```
/*
 *  switch_to(n) should switch tasks to task nr n, first
 * checking that n isn't the current task, in which case it does nothing.
 * This also clears the TS-flag if the task we switched to has used
 * tha math co-processor latest.
 */
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,current\n\t" \
    "je 1f\n\t" \
    "movw %%dx,%1\n\t" \
    "xchgl %%ecx,current\n\t" \
    "ljmp *%0\n\t" \                    //跳到其tss上去
    "cmpl %%ecx,last_task_used_math\n\t" \
    "jne 1f\n\t" \
    "clts\n" \                          //clears the TS-flag
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```