mqmelon L55 2015-06-26 00:27:40 基于内核栈切换的进程切换

“操作系统原理与实践”实验报告

基于内核栈切换的进程切换

一、tss方式的进程切换##

Linux0.11中默认使用的是硬件支持的tss切换，系统为每个进程分配一个tss结构用来存储进程的运行信息（上下文环境），然后通过CPU的一个长跳转指令ljmp来实现进程的切换，这种方式易于实现，但一者不便于管理多CPU进程，二者效率不佳，故此次实验要将系统中使用的tss切换方式修改为栈切换方式。而由于CPU管理方式的原因，tr寄存器必须指向当前正在运行的进程的tss结构，所以系统在运行时必须要有一个tss结构来存储当前运行的进程（线程）的上下文信息，但只需要保留一个全局的tss结构即可（由于这里不考虑多CPU情况，如果考虑则要为每一个CPU准备一个tss结构来储存每个cpu运行时的进程或线程上下文信息）。

从tss结构（见下面）中可以看出，tss结构基本上就是一个全部运行时寄存器保存结构，就是用来保存进程运行时全部的寄存器信息的。当然，也有其他一些进程相关信息，如进程分组链接back_link，ＩＯ使用信息trace_bitmap，以及协处理器i387的信息。当进程进行切换时这些信息通过ljmp语句进行交换，被调度到的进程信息进入相应寄存器，而被调度出的进程则使用这个结构保存进程被调度时的寄存器状态及数据，以备下次运行时恢复。而这一切均通过ljmp语句进行。见switch_to宏的代码（如下）。从代码中可以看到这种进程的切换方式简洁方便，但其缺乏效率，而且也不能很好地向多cpu结构扩展。所以要修改为基于栈的切换方式。

```
struct tss_struct {
    long    back_link;    /* 16 high bits zero */
    long    esp0;
    long    ss0;        /* 16 high bits zero */
    long    esp1;
    long    ss1;        /* 16 high bits zero */
    long    esp2;
    long    ss2;        /* 16 high bits zero */
    long    cr3;
    long    eip;
    long    eflags;
    long    eax,ecx,edx,ebx;
    long    esp;
    long    ebp;
    long    esi;
    long    edi;
    long    es;         /* 16 high bits zero */
    long    cs;         /* 16 high bits zero */
    long    ss;         /* 16 high bits zero */
    long    ds;         /* 16 high bits zero */
    long    fs;         /* 16 high bits zero */
    long    gs;         /* 16 high bits zero */
    long    ldt;        /* 16 high bits zero */
    long    trace_bitmap;    /* bits: trace 0, bitmap 16-31 */
    struct i387_struct i387;
};
```

宏switch_to(next)：

实验数据

学习时间	244分钟
操作时间	80分钟
按键次数	0次
实验次数	6次
报告字数	16748字
是否完成	完成

评分

未评分

下一篇

篇

相关报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 基于内核栈切换的进程切换 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 信号量的实现和应用 实验报告

```
#define switch_tss(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,current\n\t" \   //比较当前要切换的进程是否是当前运行的进程
    "je 1f\n\t" \                 //如果是则不调度直接退出
    "movw %%dx,%1\n\t" \        //将要调度的tss指针存到tmp.b中
    "xchgl %%ecx,current\n\t" \       //交换pcb值，要调度的存储在ecx中
    "ljmp *%0\n\t" \              //进行长跳转，即切换tss
    "cmpl %%ecx,last_task_used_math\n\t" \         //进行切换后的处理协处理器
    "jne 1f\n\t" \
    "clts\n" \
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

二、基于内核栈的切换##

2.1、关于切换的分析###

不管使用何种方式进行进程切换（此次实验不涉及线程），总之要实现调度进程的寄存器的保存和切换，也就是说只要有办法保存被调度出cpu的进程的寄存器状态及数据，再把调度的进程的寄存器状态及数据放入到cpu的相应寄存器中即可完成进程的切换。由于切换都是在内核态下完成的所以两个进程之间的tss结构中只有几个信息是不同的，其中esp和trace_bitmap是必须切换的，但在0.11的系统中，所有进程的bitmap均一样，所以也可以不用切换。

调度进程的切换方式修改之前，我们考虑一个问题，进程 0 不是通过调度运行的，那进程 0 的上下文是如何建立的？因为在进程 0 运行时系统中并没有其他进程，所以进程 0 的建立模板一定可以为进程栈切换方式有帮助。所以先来分析一下进程 0 的产生。进程 0 是在move_to_user_mode宏之后直接进入的。在这之前一些准备工做主要是task_struct结构的填充。

```
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \
    "pushl $0x17\n\t" \
    "pushl %%eax\n\t" \
    "pushfl\n\t" \
    "pushl $0x0f\n\t" \
    "pushl $1f\n\t" \
    "iret\n" \
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

这个宏模拟了一个中断返回，把但却没有设置cs、ss，其原因应该是进程 0 代码和数据均在内核，所以不需要设置，这样就可以知道，在使用栈切换时我们也只要模仿这种方式，手动造一个中断返回即可。但实际上还要麻烦一点，因为进程 0 不通过schedule进行第一次运行，但我们fork出来的新进程却要经过schedule，也就是要经过switch_to来进行调度，所以在新fork的进程栈中不仅要模拟中断返回，还要为schedule的返回准备数据。基于此，新fork的进程的内核栈应该更类似于一个被调度出去的一个进程。经过分析代码我们可以知道一个两个进程在调度时内核的栈的状态如下：

```
            进程A的内核栈              进程B的内核栈
int前
            ss                  ss
            esp                  esp
            eflags                  eflags
            cs                  cs
            eip                  eip
进入Int之后返回之前
            ds                  ds
            es                  es
            fs                  fs
            edx                  edx
            ecs                  ecs
            ebx                  ebx
            eax                  eax
进入schedule的switch后     返回地址retur_from_systemcall     返回地址
retur_from_systemcall
```

所以我们只要把fork的进程的内核栈的状态手工打造成类似的样子就可以了。根据实验指导的内容，此次修改将switch_to宏变成函数，所以需要压栈，故相比上图要做一定的调整，因为c函数的调用参数是通过栈来传递的，其汇编后的代码一般如下：

```
pushl ebp
movl  esp,ebp
sub     $x,esp
…
```

此时栈内会放置函数所用的参数，新进程并没有经过schedule的前一部分之直跳进来的，所以fork中的进程栈中也必须要有这些参数数据。此外由于新进程也没有经过int，所以iret返回时所需要的数据也要在栈内准备好。

2.2、基于栈的切换代码###

本次要修改的文件如下：

1、schedule.h

这个文件中主要修改如下几处。 第一，task_struct结构的相关部分，由于之前使用tss切换，而此次修改要使用栈切换，tss结构弃之不用（只保留一个进程0的），所以esp指针必须要保存到task_struct中，因此要在task_struct结构中添加一个新的成员long kernelstack; //内核栈指针，实际上此次实验的最后结果中我添加了两个，另一个是long eip; //保存切换时使用的EIP指针，具体原因后面会说明。同时进程0的task_struct也要做相应修改。老师在视频中也讲到成员添加的位置要小心，因为有些成员在系统中被硬编码了，所以不能放在结构的最前面，也最好不要放最后面，因为那样，在汇编程序中计算新添加成员偏移位置时就变得困难了。

此外，为了能在函数中使用汇编中的函数，这里还要声明如下几个函数。

```
/*添加用来PCB切换的函数声明*/
extern void switch_to(struct task_struct * pnext,int next); //这里的参数类型是不重要的，记得C语言
                    //的老师说过编译器只并不对此处的类型进行严格检查。
extern void first_return_from_kernel(void);
extern void first_switch_from(void); //这里也是在实验指导之外添加的
```

2、schedule.c

要修改的位置就是schedule函数，因为要将switch_to宏修改为一个汇编函数，所以无法在里面使用宏在查找pcb指针以及ldt指针，所以这两个数据均要使用参数来传递，故在schedule中也添加了一个新成员用来保存当前要调度进cpu的进程的pcb指针。如下：

```
struct task_struct * pnext=NULL; //保存PCB指针，
…
while (1) {
        c = -1;
        next = 0;
        /*为pnext赋初值，让其总有值可用。*/
        pnext=task[next]; //最初我并没有加这句，导致如果系统没有进程可以调度时传递进去的是一个空值，系统宕机，所以加上这句，这样就可以在next=0时不会有空指针传递。
        /**/
i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i,pnext=*p; //保存要调度到的pcb指针
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    /*调度进程到运行态*/
    if(task[next]->pid != current->pid)
    {
        //判断当前正在运行的进程状态是否为TASK_RUNNING，
        //如果是，则表明当前的进程是时间片到期被抢走的，这时当前进程的状态还应是TASK_RUNNING，
        //如果不是，则说明当前进程是主动让出CPU，则状态应为其他Wait状态。
        if(current->state == TASK_RUNNING)
        {
            //记录当前进程的状态为J，在此处，当前进程由运行态转变为就绪态。
            fprintk(3,"%ld\t%c\t%ld\n",current->pid,'J',jiffies);
        }
        fprintk(3,"%ld\t%c\t%ld\n",pnext->pid,'R',jiffies);
    }
    /**/

    //switch_tss(next); //由于此次实验难度还是挺高的，所以一般不会一次成功，所以我没有将switch_to宏删除，而只是将其改了一个名字，这样，如果下面的切换出问题，就切换回来测试是否是其他代码出问题了。如果换回来正常，则说明问题就出现在下面的切换上。这样可以减少盲目地修改。
    switch_to(pnext,_LDT(next));
```

3、fork.c

主要修改的是copy_process函数，见下面：

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    /*melon - 添加用来取得内核栈指针*/
    long * krnstack;
    /*melon added End*/
    struct task_struct *p;
    int i;
    struct file *f;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;

    /*melon  -取得当前子进程的内核栈指针*/
    krnstack=(long)(PAGE_SIZE+(long)p); //实际上进程每次进入内核，栈顶都指向这里。
    /*melon added End*/
    task[nr] = p;
    *p = *current;    /* NOTE! this doesn't copy the supervisor stack */
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->father = current->pid;
    p->counter = p->priority;
    //初始化内核栈内容，由于系统不再使用tss进行切换，所以内核栈内容要自己安排好
    //下面部分就是进入内核后int之前入栈内容，即用户态下的cpu现场
    *(--krnstack) = ss & 0xffff; //保存用户栈段寄存器,这些参数均来自于此次的函数调用，
                    //即父进程压栈内容，看下面关于tss的设置此处和那里一样。
    *(--krnstack) = esp; //保存用户栈顶指针
    *(--krnstack) = eflags; //保存标识寄存器
    *(--krnstack) = cs & 0xffff; //保存用户代码段寄存器
    *(--krnstack) = eip; //保存eip指针数据,iret时会出栈使用 ，这里也是子进程运行时的语句
地址。即if(!fork()==0) 那里的地址，由父进程传递
    //下面是iret时要使用的栈内容，由于调度发生前被中断的进程总是在内核的int中，
    //所以这里也要模拟中断返回现场，这里为什么不能直接将中断返回时使用的
    //return_from_systemcall地址加进来呢？如果完全模仿可不可以呢？
    //有空我会测试一下。
    //根据老师的视频讲义和实验指导，这里保存了段寄存器数据。
    //由switch_to返回后first_return_fromkernel时运行，模拟system_call的返回
    *(--krnstack) = ds & 0xffff;
    *(--krnstack) = es & 0xffff;
    *(--krnstack) = fs & 0xffff;
    *(--krnstack) = gs & 0xffff;
    *(--krnstack) = esi;
    *(--krnstack) = edi;
    *(--krnstack) = edx;
    //*(--krnstack) = ecx; //这三句是我根据int返回栈内容加上去的，后来发现不加也可以
                //但如果完全模拟return_from_systemcall的话，这里应该要加上。
    //*(--krnstack) = ebx;
    //*(--krnstack) = 0; //此处应是返回的子进程pid//eax;
               //其意义等同于p->tss.eax=0;因为tss不再被使用，
               //所以返回值在这里被写入栈内，在switch_to返回前被弹出给eax;

    //switch_to的ret语句将会用以下地址做为弹出进址进行运行
    *(--krnstack) = (long)first_return_from_kernel;
    //*(--krnstack) = &first_return_from_kernel; //讨论区中有同学说应该这样写，结果同
上
    //这是在switch_to一起定义的一段用来返回用户态的汇编标号,也就是
    //以下是switch_to函数返回时要使用的出栈数据
    //也就是说如果子进程得到机会运行，一定也是先
    //到switch_to的结束部分去运行，因为PCB是在那里被切换的，栈也是在那里被切换的，
    //所以下面的数据一定要事先压到一个要运行的进程中才可以平衡。
    *(--krnstack) = ebp;
    *(--krnstack) = eflags; //新添加
    *(--krnstack) = ecx;
    *(--krnstack) = ebx;
    *(--krnstack) = 0; //这里的eax=0是switch_to返回时弹出的，而且在后面没有被修改过。
             //此处之所以是0，是因为子进程要返回0。而返回数据要放在eax中，
             //由于switch_to之后eax并没有被修改，所以这个值一直被保留。
             //所以在上面的栈中可以不用再压入eax等数据。
             //将内核栈的栈顶保存到内核指针处
    p->kernelstack=krnstack; //保存当前栈顶
    //p->eip=(long)first_switch_from;
        //上面这句是第一次被调度时使用的地址 ，这里是后期经过测试后发现系统修改
        //后会发生不定期死机，经分析后认为是ip不正确导致的，但分析是否正确不得
        //而知，只是经过这样修改后问题解决，不知其他同学是否遇到这个问题。
    /*melon added End*/
```

4、system_call.s

这个文件中主要是添加新的switch_to函数。见下面：

```
.align 2
switch_to:
    pushl %ebp
    movl %esp,%ebp          #上面两条用来调整C函数栈态
    pushfl              #将当前的内核eflags入栈！！！！
    pushl %ecx
    pushl %ebx
    pushl %eax
    movl 8(%ebp),%ebx     #此时ebx中保存的是第一个参数switch_to(pnext,LDT(next))
    cmpl %ebx,current     #此处判断传进来的PCB是否为当前运行的PCB
    je 1f             #如果相等，则直接退出
    #切换PCB
    movl %ebx,%eax          #ebx中保存的是传递进来的要切换的pcb
    xchgl %eax,current     #交换eax和current，交换完毕后eax中保存的是被切出去的PCB
    #TSS中内核栈指针重写
    movl tss,%ecx          #将全局的tss指针保存在ecx中
    addl $4096,%ebx          #取得tss保存的内核栈指针保存到ebx中
    movl %ebx,ESP0(%ecx)     #将内核栈指针保存到全局的tss的内核栈指针处esp0=4
    #切换内核栈
    movl %esp,KERNEL_STACK(%eax)     #将切出去的PCB中的内核栈指针存回去
    movl $1f,KERNEL_EIP(%eax)     #将1处地址保存在切出去的PCB的EIP中！！！！
    movl 8(%ebp),%ebx     #重取ebx值，
    movl KERNEL_STACK(%ebx),%esp     #将切进来的内核栈指针保存到寄存器中
#下面两句是后来添加的，实验指导上并没有这样做。
    pushl KERNEL_EIP(%ebx)          #将保存在切换的PCB中的EIP放入栈中！！！！
    jmp  switch_csip         #跳到switch_csip处执行！！！！
#     原切换LDT代码换到下面
#     原切换LDT的代码在下面
1:    popl %eax
    popl %ebx
    popl %ecx
    popfl             #将切换时保存的eflags出栈！！！！
    popl %ebp
    ret             #该语句用来出栈ip

switch_csip:             #用来进行内核段切换和cs:ip跳转
    #切换LDT
    movl 12(%ebp),%ecx     #取出第二个参数，_LDT(next)
    lldt %cx          #切换LDT
    #切换cs后要重新切换fs,所以要将fs切换到用户态内存
    movl $0x17,%ecx          #此时ECX中存放的是LDT
    mov  %cx,%fs
    cmpl %eax,last_task_used_math
    jne  1f
    clts
1:
    ret             #此处用来弹出pushl next->eip！！！！！！！！


#第一次被调度时运行的切换代码
first_switch_from:
    popl %eax
    popl %ebx
    popl %ecx
    popfl             #将切换时保存的eflags出栈！！！！
    popl %ebp
    ret             #该语句用来出栈ip

#此处是从内核站中返回时使用的代码，用来做中断返回
first_return_from_kernel:
    #popl %eax
    #popl %ebx
    #popl %ecx
    popl %edx
    popl %edi
    popl %esi
    popl %gs
    popl %fs
    popl %es
    popl %ds
    iret
```

此外还要添加几个常量值：

```
/*melon 添加用来取内核栈指针的位置宏*/
KERNEL_STACK    =16 //内核栈指针在task_struct结构中的位移，指导上这里是12
ESP0        =4          //tss中内核栈指针位移
KERNEL_EIP    =20      //新添加的task_sturct结构中的eip指针位移
/*melon added End*/
```

这样就修改完毕。编译运行测试即可。

 图片描述

 图片描述

实验指导提出了几个思考问题如下：##

问题1、###

针对下面的代码片段：

```
movl tss,%ecx
addl $4096,%ebx
movl %ebx,ESP0(%ecx)
```

回答问题：

- （1）为什么要加4096；
- （2）为什么没有设置tss中的ss0。
- 1）该代码片段是用来对全局的tss结构进行进程内核栈指针的保存，虽然不再使用tss进行切换，但由于cpu的设计tr寄存器中必须有一个tss结构的指针指向当前cpu中运行的进程的tss结构，在这个修改版本中仅保留一个全局的tss结构用来存储该指针。而在切换进程时要同时将该指针也切换为相应要运行的进程的内核栈栈项指针，由于进程进入内核态运行时其栈是空的，即由用户态经由int转入内核态时其栈为空，由int等代码对其进行操作，如保存用户栈信息等。所以对于tss结构来说，其内核栈指针应总指向进程内核栈空栈时栈顶，而内核栈栈顶位置就位于分配的内存块的4096位置，即PAGE_SIZE，所以这里要将pcb指针加上该值即是空栈时栈顶指针位置。
- 2）对于linux0.11来说，内核栈段的选择子永远都是0x10，所以这里不用修改该值。相同意义的代码见fork.c中的p->tss.ss0 = 0x10;，这行代码明确指出对于所有进程ss0都是0x10。

问题2、###

针对代码片段：

```
*(--krnstack) = ebp;
*(--krnstack) = ecx;
*(--krnstack) = ebx;
*(--krnstack) = 0;
```

回答问题：

- （1）子进程第一次执行时，eax=？为什么要等于这个数？哪里的工作让eax等于这样一个数？
- 回答：子进程第一次执行时要返回 0 ，即eax=0，这样才能和父进程进行分支运行。所以这里要设置eax=0。在使用tss切换时是由p->tss.eax = 0;代码行来设置这个值的。这里只要在switch_to返回时保证eax中的数据为0即可。其实也可不必要在此设置，也可在模仿iret返回时再进行设置。
- （2）这段代码中的ebx和ecx来自哪里，是什么含义，为什么要通过这些代码将其写到子进程的内核栈中？
- 回答：代码段中的ebx、ecx均来自于copry_process函数的调用参数，是在sys_fork之前和之中分段进行入栈的。具体到ebx、ecx两个参数是由system_call函数入栈的。是进入内核态前程序在用户态运行时两个寄存器的状态数据。见下面：

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds          #保存用户态时的寄存器
    push %es
    push %fs
    pushl %edx
    pushl %ecx        # 是下面真正的int调用时的参数
    pushl %ebx        # 调用参数
    movl $0x10,%edx         # 设置ds、es寄存器到内核空间
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx         # fs指向用户空间
    mov %dx,%fs
    call sys_call_table(,%eax,4)      #调用系统调用
    pushl %eax              #返回值
    movl current,%eax
    cmpl $0,state(%eax)        # state
    jne reschedule
    cmpl $0,counter(%eax)        # counter
    je reschedule
```

就个人理解，这里如果不这么设置似乎也是可以的，但这些寄存器的值必须保存在内核栈中，只要可以保证经由iret再返回到用户态时程序可以正常使用就可以。

- （3）这段代码中的ebp来自哪里，是什么含义，为什么要做这样的设置？可以不设置吗？为什么？
- 回答：ebp参数来自于下面程序段中的入栈动作。

```
sys_fork:
    call find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp          #入栈，成为copy_process的参数
    pushl %eax
    call copy_process
    addl $20,%esp
1:    ret
```

观察代码流程，没有发现在这之前对ebp进行过修正，虽然调用过c函数find_empty_process但c函数在返回时会保证ebp值 回到调用之前，所以猜测该值应该是int调用之前进程中的ebp，即进程的用户态的ebp，所以为保证进程在返回用户时能正确处理栈帧，这里要保证该值被正确传递。所以要在这里入栈。在程序切换后时会使用到。

问题3、为什么要在切换完LDT之后要重新设置fs=0x17？而且为什么重设操作要出现在切换完LDT之后，出现在LDT之前又会怎么样？###

回答：cpu的段寄存器都存在两类值，一类是显式设置段描述符，另一类是隐式设置的段属性及段限长等值，这些值必须经由movl、lldt、lgdt等操作进行设置，而在设置了ldt后，要将fs显示设置一次才能保证段属性等值正确。

3、对此次实验的思考##

1、更换切换方式后cpu运行效率是否改变呢？如何测量呢？在此次实验中对比了两种切换的进程状态改变记录（实验三），数据如下：

tss切换时进程记录分析

```
(Unit: tick)
Process   Turnaround   Waiting   CPU  I/O
      0         1929        65           8         406
      1         1660         0           9        1651
      2           25         4          21           0
      3            4         0           4           0
      4          995       140          40         815
      5            3         0           3           0
      6            6         1           5           0
      7          267        81           1         185
      8          269       223          46           0
      9          267       222          45           0
     10          265       220          45           0
     11          283       222          61           0
     12          281       221          60           0
     13            2         0           2           0
     14          614       125          40         449
     15            3         1           2           0
     16            6         1           5           0
     17          237        81           1         155
     18          254       209          45           0
     19          253       207          46           0
     20          251       206          45           0
     21          251       205          46           0
     22          249       204          45           0
     23            3         0           3           0
     24          731         4          39         688
     25            3         1           2           0
     26            0         0           0           0
Average:      337.44     97.89
Throughout: 1.13/s
```

栈切换时进程记录分析

```
(Unit: tick)
Process   Turnaround   Waiting   CPU   I/O
      0         2213        65           8         368
      1         1953         1           8        1944
      2           24         4          20           0
      3            4         0           4           0
      4         1207       127          41        1039
      5            3         1           2           0
      6            6         1           5           0
      7          355        79           1         275
      8          359       298          61           0
      9          357       297          60           0
     10          373       298          75           0
     11          373       297          76           0
     12          371       296          75           0
     13            2         0           2           0
     14          694       136          43         515
     15            3         0           2           0
     16            6         1           5           0
     17          268        82           1         185
     18          269       223          46           0
     19          267       222          45           0
     20          266       221          45           0
     21          283       222          61           0
     22          281       221          60           0
     23            2         0           2           0
     24          327         7          37         283
     25            2         0           2           0
     26            1         1           0           0
Average:      380.33    114.81
Throughout: 1.19/s
```

tss切换时进程运行记录（部分）

```
1    N    48
1    J    48
0    J    48
1    R    48
2    N    49
2    J    49
1    W    49
2    R    49
3    N    64
3    J    64
2    J    64
3    R    64
3    W    68
2    R    68
2    E    74
1    J    74
1    R    74
4    N    74
4    J    74
1    W    74
4    R    74
5    N    105
5    J    105
4    W    105
5    R    105
4    J    107
5    E    108
4    R    108
4    W    113
0    R    113
0    W    113
```

栈切换时进程运行记录（部分）

```
1    N    48
1    J    48
0    J    48
1    R    48
2    N    49
2    J    49
1    W    49
2    R    49
3    N    64
3    J    64
2    J    64
3    R    64
3    W    68
2    R    68
2    E    73
1    J    73
1    R    73
4    N    74
4    J    74
1    W    74
4    R    74
5    N    104
5    J    104
4    W    105
5    R    105
4    J    107
5    E    107
4    R    107
4    W    112
0    R    113
0    W    113
```

从数据上看进程切换时间在缩短。但不是很量化，我尝试去记录切换时间，但未成功，因为无论是哪种切换方式，在switch_to前后记录的时间都无变化，即都是同一个tick。这个有待进一步分析。