【内核】进程切换 switch_to 与 __switch_to - visayafan - 博客园

笔记本: 内核
创建时间: 2021/4/20 10:59
URL: https://www.cnblogs.com/visayafan/archive/2011/12/10/2283660.html

【内核】进程切换 switch_to 与 __switch_to

哥今生不想再看到这段代码！！！

```
#define switch_to(prev, next, last)     \
do {          \
/*        \
  * Context-switching clobbers(彻底击败) 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   */ \
       "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),  \
       /*m表示把变量放入内存，即把[prev_sp]存储的变量放入内存，最后再写入prev->thread.sp*/\
         [prev_ip] "=m" (prev->thread.ip),  \
         "=a" (last),                                          \
         /*=表示输出,a表示把变量last放入ax,eax = last*/         \
         \
         /* clobbered output registers: */  \
         "=b" (ebx), "=c" (ecx), "=d" (edx),  \
         /*b 变量放入ebx,c表示放入ecx, d放入edx,S放入si,D放入edi*/\
         "=S" (esi), "=D" (edi)    \
                \
         /* input parameters: */    \
       : [next_sp]  "m" (next->thread.sp),  \
       /*next->thread.sp 放入内存中的[next_sp]*/\
         [next_ip]  "m" (next->thread.ip),  \
                \
         /* regparm parameters for __switch_to(): */ \
         [prev]     "a" (prev),    \
```

2011年8月(14)
2011年7月(2)

C/C++/STL/Boost

CppReference

Python

Google's Python Class
Google-Python-Groups
Python官网
itlong-Python论坛
PyGame

博客链接

我的CSDN博客
宇宙吾心
结构之法 算法之道
王垠主页
yinwang0
Joel Spolsky
Jeff Atwood

大师主页链接

Bjarne Stroustrup's homepage
Donald Knuth's homepage
W. Richard Stevens' Home Page

酷站收藏

刘未鹏 MindHacks
酷壳 CoolShell
目光博客
月光博客
在线公式LaTeX编辑--CodeCog
开源中国
xahlee
technoboffalo
techcrunch
slideshare

阅读排行榜

1. 【转】【NVL】oracle中nvl()函数(43316)
2. 【LaTeX】Lyx/LaTeX笔记04---插入伪代码(25507)
3. 【Perl】二维数组(18377)
4. 【Matlab 】【转】元胞数组--cell(17877)
5. 【Java】边框总结(15077)

评论排行榜

1. 【LaTeX】Lyx/LaTeX笔记04---插入伪代码(1)
2. 【Linux】代码统计工具sloccount(1)
3. 【Algorithm】一般约束优化问题——PHR算法及其Matlab实现(1)
4. Wolfe和Armijo准则之Matlab实现(1)
5. 【内核】Linux 2.6 内存反向映射机制 Reverse Mapping(1)

```
        /*eax = prev  edx = next*/\
        [next]     "d" (next)    \
        \
      : /* reloaded segment registers */   \
   "memory");      \
} while (0)
```

首先简单提一下这个宏和函数的被调用关系：

schedule() --> context_switch() --> switch_to --> __switch_to()

这里面，schedule是主调度函数，涉及到一些调度算法，这里不讨论。当schedule()需要暂停A进程的执行而继续B进程的执行时，就发生了进程之间的切换。进程切换主要有两部分：1、切换全局页表项；2、切换内核堆栈和硬件上下文。这个切换工作由context_switch()完成。其中switch_to和__switch_to()主要完成第二部分。更详细的，__switch_to()主要完成硬件上下文切换，switch_to主要完成内核堆栈切换。

阅读**switch_to时请注意：这是一个宏，不是函数，它的参数prev, next, last不是值拷贝，而是它的调用者context_switch()的局部变量。局部变量是通过%ebp寄存器来索引的，也就是通过n(%ebp)**，n是编译时决定的，在不同的进程的同一段代码中，同一局部变量的n是相同的。在switch_to中，发生了堆栈的切换，即ebp发生了改变，所以要格外留意在任一时刻的局部变量属于哪一个进程。**关于__switch_to()这个函数的调用，并不是通过普通的call来实现，而是直接jmp，函数参数也并不是通过堆栈来传递，而是通过寄存器来传递。**

在下文中提到一些局部变量和寄存器值，为了不引起混淆，在名字后面加上_X，表示是X进程的成员。如esp_A表示进程A的esp的值，prev_B，表示进程B中的prev变量，等等。

switch_to切换主要有以下三部分：

进程切换	即esp的切换	由于从esp可以找到进程的描述符
硬件上下文切换	_switch_to()	以前通过x86硬件支持，现在使用软件切换
堆栈的切换	即ebp的切换	ebp是栈底指针，它确定了当前变量空间属于哪个进程

上面的四个步骤中，有三个是在switch_to宏中完成，硬件上下文切换由__switch_to()函数完成。

下面来具体看switch_to从A进程切换到B进程的步骤。

step1:复制两个变量到寄存器：

[prev] "a" (prev)

[next] "d" (next)

即:

eax <== prev_A 或 eax <==%p(%ebp_A)

edx <== next_A 或 edx <==%n(%ebp_A)

这里prev和next都是A进程的局部变量。

现在eax中保存prev，ebx中保存next。其中eax中始终都保持prev，最后把该值交给 last

step2:保存进程A的ebp和eflags

pushfl

pushl %ebp

注意，因为现在esp还在A的堆栈中，所以这两个东西被保存到A进程的内核堆栈中。

step3:保存当前esp到A进程内核描述符中：

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

它可以表示成： prev_A->thread.sp <== esp_A

在调用switch_to时，prev是指向A进程自己的进程描述符的。

step4:从next（进程B）的描述符中取出之前从B切换出去时保存的esp_B。

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

它可以表示成：esp_B <== next_A->thread.sp

注意，在A进程中的next是指向B的进程描述符的。

从这个时候开始，**CPU当前执行的进程已经是B进程了，因为esp已经指向B的内核堆栈。**但是，现在的ebp仍然指向A进程的内核堆栈，所以所有局部变量仍然是A中的局部变量，比如next实质上是%n(%ebp_A)，也就是next_A，即指向B的进程描述符。

step5:把标号为1的指令地址保存到A进程描述符的ip域：

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

它可以表示成：prev_A->thread.ip <== %1f，当A进程下次被switch_to回来时，会从这条指令开始执行。具体方法看后面被切换回来的B的下一条指令。

step6:将返回地址保存到堆栈，然后调用__switch_to()函数，__switch_to()函数完成硬件上下文切换。

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

"jmp __switch_to\n" /* regparm call */

这里，如果之前B也被switch_to出去过，那么[next_ip]里存的就是下面这个1f的标号，但如果进程B刚刚被创建，之前没有被switch_to出去过，那么[next_ip]里存的将是ret_ftom_fork（参看copy_thread()函数）。这就是这里为什么不用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。

另外请注意，这里__switch_to()返回时，将返回值prev_A又写入了%eax，这就使得在switch_to宏里面eax寄存器始终保存的是prev_A的内容，或者，更准确的说，是指向A进程描述符的“指针”。这是有用的，下面step8中将会看到。

step7:从__switch_to()返回后继续从1:标号后面开始执行，修改ebp到B的内核堆栈，恢复B的eflags：

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

"popfl\n" /* restore flags */

如果从__switch_to()返回后从这里继续运行，那么说明在此之前B肯定被switch_to调出过，因此此前肯定备份了ebp_B和flags_B，这里执行恢复操作。

注意，**这时候ebp已经指向了B的内核堆栈，所以上面的prev,next等局部变量已经不是A进程堆栈中的了，而是B进程堆栈中的(B上次被切换出去之前也有这两个变量，所以代表着B堆栈中prev、next的值了)，因为prev == %p(%ebp_B)，**而在B上次被切换出去之前，该位置保存的是B进程的描述符地址。如果这个时候就结束switch_to的话，在后面的代码中（即 context_switch()函数中switch_to之后的代码）的prev变量是指向B进程的，因此，进程B就不知道是从哪个进程切换回来。而从context_switch()中switch_to之后的代码中，我们看到finish_task_switch(this_rq(), prev)

中需要知道之前是从哪个进程切换过来的，因此，我们必须想办法保存A进程的描述符到B的堆栈中，这就是last的作用。

step8:将eax写入last，以在B的堆栈中保存正确的prev信息。

"=a" (last) 即 last_B <== %eax

而从context_switch()中看到的调用switch_to的方法是：

switch_to(prev, next, prev);

所以，**这里面的last实质上就是prev，因此在switch_to宏执行完之后，prev_B就是正确的A的进程描述符了。**

（既然一样，为什么不直接写入prev中，last不就多余了么？？？？）

这里，last的作用相当于把进程A堆栈中的A进程描述符地址复制到了进程B的堆栈中。

至此，switch_to已经执行完成，A停止运行，而开始了B。在以后，可能在某一次调度中，进程A得到调度，就会出现switch_to(C, A)这样的调用，这时，A再次得到调度，得到调度后，A进程从context_switch()中switch_to后面的代码开始执行，这时候，它看到的prev_A将指向C的进程描述符。

如果读者不是十分清楚这个过程，最好自己画一下堆栈的变化，注意，这里有两个堆栈，在这个过程中，有一个时期esp和ebp并不在同一个堆栈上，要格外注意这个时期里所有涉及堆栈的操作分别是在哪个堆栈上进行的。记住一个简单的原则即可，**pop/push这样的操作，都是对esp所指向的堆栈进行的，这些操作同时也会改变esp本身，除此之外，其它关于变量的引用，都是对ebp所指向的堆栈进行的。**

下面我们从switch_to被调用的情况来看一下这个执行过程。

这里，为了便于理解，我们首先忽略switch_to中的具体细节，仅仅把它当作一个普通的指令。对A进程来说，它始终没有感觉到自己被打断过，它认为自己一直是不间断执行的。switch_to这条“指令”，除了改变了A进程中的prev变量外，对A没有其它任何影响。在系统中任何进程看到的都是这个样子，所有进程都认为自己在不间断的独立运行。然而，实际上switch_to的执行并不是一瞬间完成的，switch_to执行花了很长很长的时间，但是，在执行完switch_to之后，这段时间被从A的记忆中抹除，所以A并没有觉察到自己被打断过。

接着，我们再来看这个“神奇”的switch_to。switch_to是从A进程到B进程的过渡，我们可以认为在switch_to这个点上，A进程被切出，B进程被切入。但是，如果把粒度放小到switch_to里面的单个汇编语句，这个界限就不明显了。进入switch_to的宏里面之后，首先 pushfl和pushl ebp肯定仍然属于进程A，之后把esp指向了B的堆栈，严格的说，从此时开始的指令流都属于B进程了。但是，这个时候B进程还没有完全准备好继续运行，因为ebp、硬件上下文等内容还没有切换成B的，剩下的部分宏代码就是完成这些事情。

另外需要格外强调的是，这部分代码是内核代码，它们跟用户代码不在同一个代码段，所有进程在内核态共用这一段内核代码。这里涉及到的所有堆栈都是内核堆栈，而不涉及用户堆栈。进程切换时需要的页表项的切换不是在这里面做的。

我们现在再向“上“看，从一个高级语言程序员的角度看，内核态的东西就好比这里的switch_to一样，对高级语言程序员是透明的。高级语言程序员始终认为自己的进程在不间断连续执行，而调度点的语句以及调度点之后的整个过程对该程序是完全没有影响的。

关于内核进程切换就讲这么多吧。switch_to只是个普通的宏，但是却能实现进程的切换，很多人对此比较费解。为了正确的理解，大家需要注意：

这些代码是所有进程共用的，代码本身不属于某一个特定的进程，所以判定当前在哪一个进程不是通过看执行的代码是哪个进程的，而是通过esp指向哪个进程的堆栈来判定的。所以，对于上面图中的切换点也可以这样理解，在这一点处，esp指向了其它进程的堆栈，当前进程即被挂起，等待若干时间，当esp指针再次指回这个进程的堆栈时，这个进程又重新开始运行。

分类: 操作系统Win/Linux

« 上一篇： 【Linux】time
» 下一篇： 【内核】address_space与基树

posted @ 2011-12-10 23:50 visayafan 阅读(6089) 评论(0) 编辑 收藏