|  |  |  |  |
| --- | --- | --- | --- |
| **武汉大学国家网络安全学院教学实验报告** | | | |
| 课程名称 | 操作系统及安全设计 | 实验日期 | 2019/11/29 |
| 实验名称 | **进程（一）：简单的进程** | 实验序号 | 七 |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） | | | |
| **实验目的：**  进程的实现机理与进程管理  对应章节：第六章6.1、6.2、6.3  **实验内容：**  1. 掌握进程相关数据结构的定义方法：  进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、 进程相关的TSS，以及数据结构的关系  2. 掌握构造进程的关键技术：  初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动  3. 进程的现场保护与切换，弄清楚需要哪些关键数据结构与步骤  时钟中断与进程调度关系，现场保护与恢复机理，从ring0-- >ring1的上下文切换方法，中断重入机理  **完成本次实验要思考的问题：**  1. 描述进程数据结构的定义与含义：  进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS，画出数据结构的关系图  2. 画出以下关键技术的流程图：  初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动  3. 怎么实现进程的现场保护与恢复？  4. 为什么需要从ring0-->ring1，怎么实现？  5. 进程为什么要中断重入，具体怎么实现，画出流程图？  6. 动手做：修改例子程序的进程运行于ring3，并增加一个自定义的中断向量，使得进程因为这个中断而被暂停运行5s，并因为 这个中断的再次触发而继续运行。 | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） | | | |
| **实验环境：**  VMwareWorkstationPro 15.5.0  Ubuntu 12.04.5 desktop i386 32位  bochs 2.6.9  **关键技术：**  进程的构造，切换  **实验步骤：**  **概述：进程切换过程**  假设进程运行在ring1：  1. 进程A运行中。  2. 时钟中断发生，ring1→ring0，时钟中断处理程序启动。  3. 进程调度，下一个应运行的进程（假设为进程B）被指定。  4. 进程B被恢复，ring0→ring1。  5. 进程B运行中。  要想实现这些功能，我们必须完成的应该有以下几项： 时钟中断处理程序，进程调度模块，两个进程。     1. **描述进程数据结构的定义与含义：进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS，画出数据结构的关系图**   进程控制块：  进程主要由代码，数据，堆栈组成。进程控制块记录一个进程的状态，在进程要被挂起的时候，进程信息就被写入这个数据结构，等到进程重新启动的时候，这个信息重新被读出来。进程的可能被改变的状态才有保存的必要。进程要运行，不外乎CPU和内存在相互协作，而不同进程的内存互不干涉（我们考虑最简单的情况，假设内存足够大），CPU只有一个，不同进程共用一个CPU的一套寄存器。所以，要把寄存器的值保存，进程被恢复执行时使用。为了保证进程状态完整，不被破坏，在进程刚刚被挂起时保存所有寄存器的值。  “进程表”（有的书中称之为进程控制块，也即PCB）。把进程表定义成一个结构体，对它的操作是非常方便的。  会有很多个进程，所以会有很多个进程表，形成一个进程表数组。进程表是用来描述进程的，所以它必须独立于进程之外。所以，当我们把寄存器值压到进程表内的时候，已经处在进程管理模块中了。    进程结构体：  进程表被C语言定义为结构体，便于操作。  进程相关GDT/LDT：  由于每个进程相对独立，我们把涉及到的描述符放在局部描述符表LDT中，所以，我们还需要为每个进程准备LDT。 LDT作为一个段，段中含有局部描述符，而指向LDT的描述符存在GDT中。LDT中的段为进程自有。  进程相关TSS:  对于有特权级变换的转移，如果由外层向内层转移时，需要从TSS中取得从当前TSS中取出内层ss和esp作为目标代码的ss和esp。所以，我们必须事先准备好TSS。其描述符存在GDT  数据结构关系图：  在保护模式下由GDTR寄存器保存GDT基址。先从GDTR寄存器中获得GDT基址。     1. **画出以下关键技术的流程图： 初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动**   **IMG_20191204_200742**    **初始化进程表：**  结构定义：在/include/proc.h    在global.c声明进程表数组  PUBLIC PROCESS proc\_table[NR\_TASKS];  NR\_TASKS定义了最大允许进程，把它设为1。  由于kernel\_main( )是最后一部分被执行的代码，那么初始化进程表的代码理应添加在这    进程表需要初始化的主要有3个部分：寄存器、LDT Selector和LDT。代码中，LDT Selector被赋值SELECTOR\_LDT\_FIRST，为简化起见，分别被初始化成内核代码段和内核数据段，只是改变了一下DPL以让其运行在低的特权级下。  cs指向LDT中第一个描述符，ds、es、fs、ss都设为指向LDT中的第二个描述符，gs仍然 指向显存，只是其RPL发生改变。  eip指向TestA，表明进程将从TestA的入口地址开始运行。esp指向了单独的栈，栈的大小为 STACK\_SIZE\_TOTAL。最后一行设置eflags，0x1202恰好设置了IF位并把IOPL设为1。这样，进程就可以使用I/O指令，并且中断会在iretd执行时被打开。  LDT跟GDT是联系在一起的，填充GDT中进程的LDT的描述符。在/kernel/protect.c。代码放在init\_prot( )中。seg2phys由段名求绝对地址，也在protect.c。vir2phys由线性地址到物理地址，在/include/protect.h.      **初始化TSS,GDT：**  剩下的没有初始化的只有TSS和它对应的描述符了。init\_prot( )填充TSS以及对应的描述符。TSS结构定义在/include/protect.h。  准备TSS:    在kernel.asm加载tr寄存器。指向TSS。    **Iretd：**  IRETD从栈顶弹出32位内容送入EIP。  简化restart（见4如何实现ring0->ring1）。Kernel.asm。  p\_proc\_ready是指向进程表结构的指针：  EXTERN PROCESS \* p\_proc\_ready;  由于进程的各寄存器值如今已经在进程表里面保存好了，现在我们只需要让esp指向栈顶，然后将各个值弹出就行了。最后一句iretd执行以后，eflags会被改变成pProc->regs.eflags的值。我们事先置了IF位，所以进程开始运行之时，中断其实也已经被打开了，最后在nernel/main.c添加调用。    **进程启动：**  Make image    我们看到了不断出现的字符“A”和不断增加的数字。这意味着我们实现了ring0到ring1的跳转，再进一步，这意味着我们的进程在运行，而这一切意味着我们编写的这个东西已经可以称之为一个“操作系统”了。因为它已经有了“进程”       1. **怎么实现进程的现场保护与恢复？**   使用进程表保存进程的状态（**寄存器值**），以便中断处理程序完成之后需要被恢复的进程能够被顺利地恢复。进程表给每一个寄存器预留了位置，以便把它们所有的值都保存下来。这样就可以在进程调度模块中使用这些寄存器，而不必担心会对进程产生不良影响。  Kernel3.asm    在进程切换到中断前读TSS，获取eps0，esp0指向进程对应的进程表的起始位置，依次把当前寄存器值压入进程表，就完成了现场保护。当从中断返回时，先pop一系列的寄存器，然后iretd再将进程表高地址的5个寄存器（ss，esp，...）赋值，就完成现场恢复。   1. 为什么需要从ring0-->ring1，怎么实现？   IMG_20191204_200742  我们假设的初始状态是“进程A运行中”。到目前为止我们的代码完全运行在 ring0。所以，当我们准备开始第一个进程时，面临一个从ring0到ring1的转移，并启动进程A。这跟我们从进程B 恢复的情形很相似，所以在准备就绪之后跳转到中断处理程序的后半部分，“假装”发生了一次时钟中断来启动进程A，利用iretd来实现ring0到ring1的转移。  Restart在内核启动后就开始运行，工作在ring0，这时写入进程A的寄存器值和设置好ss0，esp0，就相当于A调了中断然后中断返回。当然此时返回时写入寄存器的值并非之前压入进程表的值，而是初始化进程表时设置好的值。  第6章最终实现的代码（chapter6/r）中kernel.asm的一小部分    main.c里函数kernel\_main( )有一行restart( ); 调用的便是代码这一段，它是进程调度的一部分，同时也是操作系统启动第一个进程时的入口。  第358行设置了esp的值，而在下方是若干个pop以及一个popad指令。，p\_proc\_ready是一个向进程表的指针，存放的是下一个要启动进程的进程表的地址。其中的内容必然是以图6.7所示的顺序进行存放。这样，才会使pop和popad指令执行后各寄存器的内容更新一遍。我们找到s\_stackframe这个结构体的声明，它的内容安排与我们的推断完全一致。  进程的状态统统被存放在s\_proc这个结构体中。当要恢复一个进程时，便将esp指向这个结构体的开始处，然后运行一系列的pop命令将寄存器值弹出。。    第359行，lldt指令设置ldtr的。第360行、第361行的作用是将s\_proc这个结构中第一个结构体成员regs的末地址赋给TSS中ring0堆栈指针域（**esp0**）。在下一次中断发生时，**esp0**将变成regs的末地址，然后进程ss和esp两个寄存器值，以及eflags，还有cs、eip这几个寄存器值将依次被压栈，放到regs这个结构的最后面（堆栈是从高地址向低地址生长的）。s\_stackframe这个结构最末端的成员果然便是这5个。 （**ring1->ring0，读TSS.esp0，得到进程表中regs的地址，然后把当前这5个寄存器保存到regs。其他的寄存器在时钟中断时保存到进程表更低的地址。见上图**）结合s\_stackframe的结构定义，esp加4跳过了retaddr这个成员，**这样esp所在位置（栈顶）为eip。**以便执行iretd这个指令，之前堆栈内恰好是eip、cs、eflags、esp和ss的值。Iretd执行时会弹出栈顶4字节（eip），然后依次继续弹出剩下的4个寄存器值。   1. **进程为什么要中断重入，具体怎么实现，画出流程图？**   IMG_20191204_200742  5.1 保存esp0（赋值TSS.esp0，用于ring1->ring0）  为了实现ring1->ring0的堆栈切换，需要保存ring0堆栈信息ss0和esp0。其中ss0在2中已经设置。由于要为下一次ring1→ring0做准备，所以用iretd返回之前要保证tss.esp0 是正确的。 （esp0指向该进程的进程表，这样从ring1到ring0时，先读TSS的esp0，然后把ring1的寄存器压栈到esp0指向的位置）当进程被中断切到内核态，当前的各个寄存器应该被立即保存（压栈）。也就是说，每个进程在运行时， tss.esp0应该是当前进程的进程表中保存寄存器值的地方，即struct s\_proc中struct s\_stackframe的最高地址处。这样，进程被挂起后才恰好保存寄存器到正确的位置。我们假设进程A在运行，那么tss.esp0的值应该是进程表A中regs的最高处，因为我们是不可能在进程A运行时来设置tss.esp0的值的，所以必须在A被恢复运行之前，即iretd执行之前做这件事。换句话说，我们应该在时钟中断处理结束之前做这件事。    5.2 内核栈  现在esp指向的是进程表，如果此时我们要执行复杂的进程调度程序，最简单的例子，如果我们想调用一个函数，这时一定会用到堆栈操作，那么，我们的进程表立刻会被破坏掉。所以我们需要切换堆栈，将esp指向另外的位置。现在就该用内核栈。  修改时钟中断：kernel5.asm  162和169两个mov语句将esp切换到内核栈又切换回进程表。    使用内核栈后，就可以进行更复杂的操作，比如disp\_str.  修改kernel.fianl.asm    运行：我们看到不断出现的字符“^”，说明函数disp\_str运行正常，而且没有影响到中断处理的其他部分以及进程A。之所以在两次 字符A的打印中间有多个“^”，是因为我们的进程执行体中加入了delay( )函数，在此函数的执行过程中发生了多次中断。    5.3 中断重入  我们应该允许中断嵌套，也就是说，在中断处理过程中，应该允许下一个中断发生。不允许肯定是不行的，因为不希望在进程调度时按键就不再响应。于是，我们必须用合适的机制来应付嵌套的情况。  修改一下代码，以便让系统可以在时钟中断的处理过程中接受下一个时钟中断。  首先，因为CPU在响应中断的过程中会自动关闭中断，我们需要人为地打开中断，加入sti指令；然后，为保证中断处理过程足够长，以至于在它完成之前就会有下一个中断产生，我们在中断处理例程中调用一个延迟函数。C/kernel/kernel1.asm    **运行：**  在打印了一个A0x0之后就不停打印“^”，再也进不到进程里面  之所以会产生这种情况，是因为在一次中断还未处理完时，又一次中断发生了。这时程序又跳到中断处理程序的开头，如此反复，永远也执行不到中断处理程序的结尾——跳回进程继续执行。而且，由于压栈操作多而出栈操作少，随着时间的继续，当堆栈溢出的时候，意料不到的事情就可能发生了。    中断处理程序是被动的，它只知道当忠实的中断发生时执行那段代码，完全不理会中断在何时发生。为了避免这种嵌套现象的发生，我们必须让中断处理程序知道自己是不是在嵌套执行。 设置一个全局变量就可以了。这个全局变量有一个初值-1，当中断处理程序开始执行时它自加， 结束时自减。在处理程序开头处这个变量需要被检查一下，如果值不是0(0=－1+1)，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。武断地结束新的中断并不是一个好的办法，我们姑且这样来做。我们按照这个思路把程序修改一下。  **Kernel/main.c 全局变量**  **Kernel/kernel2.asm 加入k\_reenter自加以及判断是否为0的代码**    **运行：**  字符A和相应的数字又在不停出现了，说明修改生效了。而且，屏幕左上角的字母跳动速度 快而字符“^”打印速度慢，说明有很多时候程序在执行了inc byte [gs:0]之后并没有执行disp\_str，说明中断重入的确发生了。  **注释掉刚才的打印字符以及Delay等语句。kernel3.asm**     1. **动手做：修改例子程序的进程运行于ring3，并增加一个自定义的中断向量，使得进程因为这个中断而被暂停运行5s，并因为 这个中断的再次触发而继续运行。**   6.1修改使程序运行在ring3  修改RPL和DPL，特权级的宏定义在const.h，其中权限为DPL    初始化进程表时定义了进程的RPL,DPL，在main.c    修改DPL,RPL为user（ring 3）即可将进程运行在ring3      运行效果图：    6.2 增加自定义中断，使程序暂停5s  思路为：修改键盘中断，使得中断内计时5s，然后中断停止，返回进程。  因为已经进入了保护模式，实模式的中断向量表已经被替代为IDT，不能用int 21中断读系统时间，所以采用直接读cmos ram中的时间。  使用in，out读取时间，其中cmos ram中时间分布如图，一字节表示秒，分，时。高四位为十位，第四位为个位,BCD码保存。  IMG_256  思路为读取采用键盘中断，触发中断开始计时cmos ram 中的60s（cmos ram中秒速较快，本地约为 次/s）  读取当前秒数-1，记录到BL寄存器。之后不断读取当前秒数，当当前秒数==BL时，中断结束，返回进程。  **代码：**  先打开键盘中断，关闭时钟中断。为演示效果更好，修改进程，delay(10)。      修改键盘中断（kernel.asm）  当计时够60s时，打印 5seconds。在屏幕0行70列72列打印当前cmosram中秒。        **运行效果：**  右上角秒是动态显示的。下方可见在进程打印两个数后键盘中断，暂停5s后打印5seconds并继续。然后进程打印几个数后又一次键盘中断，暂停5s，打印5seconds，继续。 | | | |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | |
| **遇到的问题及解决：**   1. **对 chapter6/b make时遇到问题**     因为makefile中使用的是kernel/kernel.asm，而文件中为kernel1...5.final等asm，所以失败。修改目标kernel为kernel.asm即可。  **实验结果记录：**   1. **进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS数据结构的关系图**      1. **关键技术的流程图： 初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动**   **IMG_20191204_200742**     1. **实现进程的现场保护与恢复**      1. **实现从ring0-->ring1**   **IMG_20191204_200742**     1. **实现中断重入**   **IMG_20191204_200742**       1. **修改程序运行在ring3，实现中断使进程暂停5s**   **6.1修改使程序运行在ring3**  修改DPL,RPL为user（ring 3）即可将进程运行在ring3      运行效果图：    6.2 增加自定义中断，使程序暂停5s | | | |