|  |  |  |  |
| --- | --- | --- | --- |
| **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 | 2019/12/13 |
| 实验名称 | **进程（二）：多进程与进程调度** | 实验序号 | 八 |
| 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：**  多进程的实现机理与进程调度  对应章节：第六章6.4、6.5、6.6  **实验内容：**  1. 多进程问题，如何扩展单进程到多进程，如  何扩展中断支持多进程？  2. 如何实现系统调用  3. 进程调度问题，弄清楚实现调度的基本思路  **完成本次实验要思考的问题：**  1. 在单进程的基础上扩展实现多进程要考虑哪些问题？  2. 画出以下关键技术的流程图： – 初始化多进程控制块的过程、扩展初始化LDT和TSS  3. 如何修改时钟中断来支持多进程管理，画出新的流程图。  4. 系统调用的基本框架是如何的，应该包含哪些基本功能，画出  流程图。  5. 如何操控可编程计数器？  6. 进程调度的框架是怎样的？优先级调度如何实现？  7. 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度5-8个任务，输出性能评价信息。  8. 思考题：从用户态进程读和写内核段的数据，看能否成功，是  否会触发保护，并解释原因。 | | | |
| 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：**  多进程扩展，实现系统调用，实现进程调度  **实验步骤：**  1. 多进程问题，如何扩展单进程到多进程，如何扩展中断支持多进程？  **扩展到多进程:**  **添加进程体**（mian.c）。这里添加了进程B。  **初始化进程表，LDT。**  在proc.h声明结构TASK，包含进程名，开始地址，堆栈大小。然后用TASK在global.h声明数组task\_table，并填入A,B。用于存储各个进程的TASK。然后修改进程表proc\_table的NR\_TASKS为2，说明有两个进程。同时初始化进程B的堆栈大小，然后修改堆栈总大小STACK\_SIZE\_TOTAL（proc.h）.  之后在proto.h声明进程B（void TestB()）。  之后在main.c循环初始化各进程对应的进程表项。读取task\_table中各进程的入口地址，堆栈栈顶，进名，修改对应项，初始化寄存器，并设置进程pid。  在循环中设置了各进程的ldt（p\_proc->ldt\_sel = selector\_ldt;）。每次循环selector\_ldt += 1 << 3;。这里初始化selector\_ldt = SELECTOR\_LDT\_FIRST;。SELECTOR\_LDT\_FIRST在protect.h中定义，这样就为每个进程在GDT中设置了选择子。  有了选择子，接下来设置LDT的内容。在proctect.c设置LDT。通过循环读取  proc\_table的ldt选择子然后填充。于是完成LDT初始化。    **扩展中断：**  中断恢复到进程的过程是esp指向进程表中对应进程的开始位置，然后lldt，pop恢复寄存器值。于是进程切换只需要修改esp的值，指向要切换的进程对应的进程表项。中断返回时设置esp语句为（mov esp, [p\_proc\_ready] ）  编写C代码clock.c修改p\_proc\_ready，并在时钟中断（kernel.asm）call clock\_handler即可。    **效果：chapter6/e**  此时每次进入时钟中断，返回时都会切换到进程表中下一进程。到达进程表尾即开始进程表首部的进程，循环。打印多个\*是在A,B的delay中发生的时钟中断引起的。    **总结：在以上修改的基础上，增加一个新进程需要的步骤**  1. 在task\_table中增加一项（global.c）。 （name，eip，stack）  2. 让NR\_TASKS加1（proc.h）。  3. 定义任务堆栈（proc.h）。  4. 修改STACK\_SIZE\_TOTAL（proc.h）。  5. 添加新任务执行体的函数声明（proto.h）。  **继续修改时钟中断，使中断处理更清晰：**  目的是使得所有中断类型（16个）的处理都更加流程化，标准化。  分离中断例程：  保存寄存器，修改k\_reenter，切换内核栈部分分离为save。Save最后jmp [eax + RETADR - P\_STACKBASE]是返回到hwint\_master中call jmp的地址下面的地址。Save中根据是否是中断重入，push了restart或restart\_reenter。hwint\_master中最后的ret会跳转到restart或restart\_reenter恢复进程运行。  返回进程部分分离为restart（restart\_reenter)。  同时添加代码操作8259A使得禁止同一中断重入（打开sti，故仍然允许其他中断重入）。  然后定义新的数据结构irq\_table（global.c），大小为16，为指针数组，指向各个中断对应的不同的中断处理函数。  然后初始化irq\_table（i8259a.c）。    先全部初始化为spurious\_irq,然后单独处理需要修改的中断。单独处理某个中断时，又定义了新的函数put\_irq\_handler（i8259a.c），其中包含汇编函数disable irq（kliba.asm）和修改irq\_table[x]指针的部分。另外还有enable.irq，用于配合put\_irq\_handler使用。    这样做是因为单独修改的步骤比较多，定义了put\_irq\_handler后类似于C++的封装，更清晰。之后修改某个中断的方法为在mian.c的kernel\_main()函数中调用put\_irq\_handler，enable\_handler.以时钟中断为例，下面的代码定义时钟中断处理函数为clock\_handler，并开启8259的时钟中断端口。    现在已经可以通过enable\_irq，disable\_irq控制某个中断端口的开启，故在初始化设置8259a时先关闭所有端口（out\_byte(INT\_M\_CTLMASK, 0xFF); ）。  完成上述修改后，已经有了**扩展中断处理的接口**，若想添加某个中断处理模块，只需要将完成中断处理的函数入口地址赋给irq\_table中相应的元素就够了。  所有中断的格式均为宏（macro hwint\_master）的格式。  发生某中断时，如hwint 00，会传中断号0给hwint\_master，然后hwint\_master中先save，save过程保存了寄存器值，并根据是否中断重入通过push，设置好了hwint\_master最后的ret地址，并切换到内核栈。然后save返回，处理8259a禁止同类中断，sti开启其他中断，然后call irq\_table中对应中断号的中断函数，然后开启被禁止的中断，最后ret（restart）。  Restart和restart\_reenter的区别是restart中先离开了内核栈。若是中断重入，restart\_reenter，这样中断重入的hwint\_master最后不会离开内核栈。而非中断重入的最后Restart离开了内核栈。这样可以避免其他类型的中断重入导致当前中断离开了内核栈。    **效果：chapter6/k**    2. 如何实现系统调用  通过中断实现系统调用。设置0x90中断处理系统调用。先将eax中移入要调用的系统调用号，然后调用0x90中断，然后在0x90中断中根据eax中的中断号调用相应的系统调用（使用数组实现，sys\_call\_table（eax）），然后将返回值写入eax，再跳回进程。过程与上面的hwint master实现中断类似。  例子中get\_ticks函数定义在syscall.asm，完成的工作是mov eax 0（在sys\_call\_table中0号为对应函数），然后调用0x90中断。Ticks为全局变量，在global.h定义，在kernel\_main初始化为0，每次时钟中断触发都+1.  而0x90中断INT\_VECTOR\_SYS\_CALL对应的中断门初始化在protect.c的init\_prot函数。  其中断代码在kernel.asm，和hwint master一样利用了之前分离出的save，restart。  注意为了不改变eax，save函数中用到的eax都改为esi。mov [esi+EAXREG-P\_STACKBASE],eax是把sys\_call\_table[eax]的返回值放在进程表中eax的位置。  sys\_call\_table数组存放系统调用函数的指针，定义在global.c。sys\_call\_table[0]被初始化为get\_ticks。于是call [sys\_call\_table+4\*eax]就是call get\_ticks。  各系统调用函数定义在proc.c。  各函数声明在proto.h。    **效果：chapter6/m**  在TestA进程中调用get\_ticks，会int 0x90，eax存get\_ticks的调用号0，然后在0x90中断中call sys\_get\_ticks,然后返回TestA。  由于每次clock\_handler执行会打印一个#并ticks++，两次A见出现#次数应该与打印的getticks相同。由图可见A0X0和A0X3间确实有3个#。     1. 进程调度问题，弄清楚实现调度的基本思路   **简单的方法**：修改进程delay（延迟）时间，延迟时间长，做的事情少，相当于优先级更低。  **过去的处理**：之前的多进程中，clock handler按顺序选择下一个执行的进程（时间片轮转）。  **进一步调度**：给每一进程都添加一个变量（可以放在进程表中），在一段时间的开头，这个变量的值有大有小，进程每获得一个运行周期，这个变量就减1，当减到0时，此进程就不再获得执行的机会，直到所有进程的变量都减到0为止。这样，每个进程获得的执行时间就不一样了。  先修改进程表。在进程结构增加两个变量ticks和priority。ticks是递减的，从某个初值到0。为了记住ticks的初值，另外定义一个变量priority，它是恒定不变的。当所有的进程ticks都变为0之后，再把各自的ticks赋值为priority，然后继续执行。每个进程的两个初始化在main.c kernel\_main()。    同时不再在中断中（如clock handler）选择下一个进程，而定义一个新函数schedule（proc.c）  作用为选择当前进程表中ticks最大的进程，作为返回的进程。这样所有中断返回时调用schedule即可。同时schedule在最后判断，若各个进程的ticks都减为0了，那么就重新赋值priority。    三个进程均修改为delay（200）ticks。也就是20ms。A,B,C的ticks初始化为150,50,30.每次  时钟中断（10 ticks 一次）只有一个进程的ticks-1，由于delay 200，所以每次进程得到执行，其ticks会-20。  为了方便观察先设置当各进程的ticks降到0时不重新赋为priority。同时清屏。修改schedule使得每次时钟中断发生都打印当前A,B,C三个进程中最大的ticks，而A,B,C打印A,B,C和开始milli\_delay时系统的ticks。区别在于有无 <> 包围。    这样运行过程为：A先执行，100/20=5次，然后A,B的ticks均为50，降到30前共有（50-30）\*2ticks的时间，40/20 \* 2=4次，然后最后A,B,C一起运行，30\*3/20\*3=13.5次。  **注意这样算的原因是milli\_delay(200)的原理是获取系统的ticks，其他进程运行系统的ticks也会改变，所以要用40/20 \* 2，而不是40/2。**  **效果：chapter6/q**    **更进一步：为了使得进程执行的时间更符合priority的比例，在中断（clock handler）中增加判断，当当前进程ticks未减到0时一直执行当前进程。**  **同时修改priority为15,5,3，delay时间均为10ms（1个ticks）。**  **效果：chapter/s**  非常接近15:5:3，不完全相同是因为打印字符，中断，进程切换，等等也需要消耗ticks。 | | | |
| 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. **在动手做实现多级队列调度时遇到困难**   解决：利用定义变量的方法代替了链表，简化了实现，实现了同样的调度效果。  **实验结果记录：**   1. 在单进程的基础上扩展实现多进程要考虑哪些问题？   进程函数体，进程函数的声明，初始化进程表（使用task\_table标准化简化），设置各进程的GDT,LDT（选择子的初始化和填充），从中断返回时进程的选择（切换）。   1. 画出以下关键技术的流程图： – 初始化多进程控制块的过程、扩展初始化LDT和TSS      1. 如何修改时钟中断来支持多进程管理，画出新的流程图。     4. 系统调用的基本框架是如何的，应该包含哪些基本功能，画出  流程图。  **系统调用框架：**  进程根据需求传递所需求的系统调用号给系统，进入中断，在中断例程中系统根据调用号调用相关函数，把返回值传递给进程，中断结束返回进程。  通过中断实现系统调用。设置0x90中断处理系统调用。先将eax中移入要调用的系统调用号，然后调用0x90中断，然后在0x90中断中根据eax中的中断号调用相应的系统调用（使用数组实现，sys\_call\_table（eax）），然后将返回值写入eax，再跳回进程。过程与上面的hwint master实现中断类似。  **应包含的功能：**  包含系统调用号的传递，进程状态的保存，系统调用函数表的初始化和调用，对返回值的传递，进程与中断的切换，以及中断重入的处理。  **流程图：**     1. 如何操控可编程计数器？     中断是由一个被称做PIT（Programmable Interval Timer）的芯片来触发的。在IBM XT中，这个芯片用的是Intel 8253，在AT以及以后换成了Intel 8254。8253有3个计数器（Counter），它们都是16位的，各有不同的作用。时钟中断实际上是由8253的Counter 0产生的。 计数器的工作原理是这样的：它有一个输入频率，在PC上是1193180Hz。在每一个时钟周期（CLK cycle），计数器值会减1，当减到0时，就会触发一个输出。由于计数器是16位的，所以最大值是65535，因此，默认的时钟中断的发生频率就是1193180/65536≈18.2Hz。  **编程修改8253计数器值，那么中断产生频率就改变了。**  如果想让系统每10ms产生一次中断，也就是让输出频率为100Hz，那么需要为计数器赋值为1193180/100≈11931。  改变计数器值通过写端口实现。8253端口如下。改变Counter 0计数值需要操作端口40h。我们需要先通过端口43h写 8253模式控制寄存器，其格式如图。其中各位取整不同的作用见课本。    我们要操作的是Counter0，所以第7、6位应该是00；计数值是16位的，所以低字节和高字节都要写入，于是第5、4位应该是“11”；使用模式2，所以第3、2、1位应该是“010”；第0位设 为“0”。这样，整个字节就变成“00110100”，也就是十六进制的0x34。  其修改代码在main.c。其中宏定义见const.h。把时钟中断修改为10ms一次。    **效果**：进一步修改，只有一个进程A并只打印A和ticksnumber，可见每次1秒钟ticks增加为0x64，即100个ticks。（1s=100\*10ms）   1. 进程调度的框架是怎样的？优先级调度如何实现？   **框架：**    **优先级调度的实现：**  给每一进程都添加一个变量（可以放在进程表中），在一段时间的开头，这个变量的值有大有小，进程每获得一个运行周期，这个变量就减1，当减到0时，此进程就不再获得执行的机会，直到所有进程的变量都减到0为止。这样，每个进程获得的执行时间就不一样了。  先修改进程表。在进程结构增加两个变量ticks和priority。ticks是递减的，从某个初值到0。为了记住ticks的初值，另外定义一个变量priority，它是恒定不变的。当所有的进程ticks都变为0之后，再把各自的ticks赋值为priority，然后继续执行。  同时不再在中断中（如clock handler）选择下一个进程，而定义一个新函数schedule（proc.c）  作用为选择当前进程表中ticks最大的进程，作为返回的进程。这样所有中断返回时调用schedule即可。同时schedule在最后判断，若各个进程的ticks都减为0了，那么就重新赋值priority。  这样schedule会选择当前ticks最大的进程进行返回。  为了使得进程执行的时间更符合priority的比例，在中断（clock handler）中增加判断，当当前进程ticks未减到0时一直执行当前进程。     1. 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度5-8个任务，输出性能评价信息。     **添加进程：**  /main.c /global.c    /proc.h /proto.h    **优先级实现：**  实现三级队列。Queue1，queue2，queue3，各自时间片长度5,10,15ticks。进程一开始都在queue1.  思路为在进程表为每个进程添加两个变量int tflag,int cnt。Flag表示当前进程所在队列号flag在main中初始化时为1，表示在第一级就绪队列。Cnt初始化为0，表示该进程在当前队列中运行了几个ticks。  选择返回的进程部分在schedule函数。  先选择flag为1的进程运行，再选择flag为2的进程，最后选择flag为3的进程。Schedule中遍历proc\_table。认为table中flag相同，越靠前的进程到来的越早，先服务。若所有process的tflag均为3，则不再选择最小的process(这样会始终运行A)，而改成轮流执行还没有执行完的进程。这里会判断是否所有进程都已经结束，若都结束则根据priority重新赋值。  实现时间片的部分在clock\_handler()。  clockhandler中每次进程进入，cnt++。在cnt达到边界（5,10,15）前不会进入schedule函数，当cnt达到边界说明用完了时间片，进入schedule，同时cnt清0，tflag++。当tflag为3时不再修改tflag，在schedule中进行时间片轮转。这里会先判断当前进程的ticks是否为0（说明进程已经结束），若结束则将其tflag=3，cnt=15，便于后续统一处理。  这里有一个细节，判断程序结束用ticks<1而不能用ticks=0，因为可能有中断重入导致0--。另外若不是所有队列都进入queue3时，遍历proc\_table时从proc\_ready下一个proc开始判断，这样是为了在每个队列中先来先服务，不让刚刚tflag++的proc抢占之前的proc。  **代码：**  添加flag , cnt: /proc.h    初始化tflag，cnt，ticks : /main.c  为了观察效果，将A,B,C,D,E的ticks分别设为90,90,60,60,30    修改clock\_handler(): /clock.c    修改schedule(): /proc.c      为了观察清屏:/main.c /kernel()    运行效果 :  可见A,B,C,D,E五个进程先后在queue1运行了5个ticks，然后分别在queue2运行10个ticks  ，然后在queue3进行时间片轮转，运行15个ticks。当5个进程全部结束时，打印了finished并在赋值各自ticks后重新开始，5个进程重新进入queue1.     1. 思考题：从用户态进程读和写内核段的数据，看能否成功，是否会触发保护，并解释原因。   不能成功，会触发特权级保护。  原因是用户态进程运行在低特权级和内核段数据在高特权级，只能由高特权级向低特权级访问，不能有低到高。 | | | |