|  |
| --- |
| 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：**  在已有实验代码基础上，将1-6章节进行功能综合，采用汇编和C  语言，实现一个mini-os  **实验内容：**  – 功能要求：  • 用软盘启动该mini-OS，如果能够硬盘启动更好。  • 能够实现内存的分配与释放  • 能够实现多进程管理与调度  • 能够提供一个定时触发的系统中断调用，完成对指定进程内存结构的完整性度量，并打印结果  • 所有代码需用目录树结构管理，并添加完整的makefile编译，以及文档 |
| 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  所有代码在chapter6/me（实验8的代码）的基础上修改  关键技术：  内存管理，进程调度，完整性度量  实验步骤：   1. **实现内存的分配与释放**   **1.1 思路：**  先参考Linux的allocpages和freepages。  <https://blog.csdn.net/tiantao2012/article/details/79029965>  得知allocpages和freepages的管理单位是页面，故在实现中也应该按照页面分配。  而oranges的一页大小为4KB。  由第三章学习可知并非所有的内存都可以用（可供分配）：  图3.33    图5.10 ，5.8给出了更具体的地址（在loader.asm，kernel.asm中也有描述）    可见可供使用的free空间地址为7E00h-30000h。  大小为28200h，/1024=160KB,一页4KB，共40页。  上学期理论课学习过内存管理的各方法，  <https://blog.csdn.net/liuxuezong/article/details/8702382>  其中位图法以二进制的一位表示一块内存，0表示未占用，1表示以占用。所以allocpages就可以实现为将某些位置1，freepages实现为将1置0.  刚刚分析过可用40页，40块，40位。汇编中一个双字32位，因为只是为了简单的实现allocpages，freepages，故取32页用一个双字表示。**即采用一个dw，用位图法表示内存分配情况。**  32\*4\*1024\*8=10 0000 H bits=2 0000 H Bytes。  可用空间为7E00H - 30000h  取7E00H 27E00H分配。  **1.2 代码实现：**  **1.2.1初始化位图kernel.asm**  Page\_map\_len为总长度(32位)    **1.2.2 alloc\_pages kernel.asm**  实现见注释  返回alloc的页在内存中的起始地址    **1.2.3 free\_pages kernel.asm**  freepages应该free指定页，故应该有参数。  而上学期理论课学习过参数和堆栈的关系，当调用函数时，从高地址到低地址分别是参数，ret地址，ebp(基址)。  所以代码实现为：    **1.2.4 声明函数**  **kernel.asm**    **Proto.h**    **1.3 效果**  在main.c添加测试。为了效果先删除了清屏函数。  main()    运行：说明alloc，free成功     1. **实现多进程管理，调度**     **2.1添加进程：**  /main.c /global.c    /proc.h /proto.h    **2.2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。  **2.3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()    **2.4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. **提供一个定时触发的系统中断调用，完成对指定进程内存结构的完整性度量，并打印结果**     1. **思路 进行crc校验**   内存完整性校验的方法有很多，采用crc校验的方法，即计算相关数据得到的余数附加到原数据之后。  而题目另一个要求是定时触发，故参考getticks，因为getticks函数就是一个定时触发的函数，所以模仿他的过程。即设置初值，在进程和handler中定时调用。   * 1. **计算CRC**   计算方法：  **（1）将CRC寄存器（8-bits，比生成多项式少1bit）赋初值0**  **（2）在待传输信息流后面加入8个0**  **（3）While (数据未处理完)**  **（4）Begin**  **（5）   If (CRC寄存器首位是1)**  **（6）        reg = reg XOR 0x31**  **（7）   CRC寄存器左移一位，读入一个新的数据于CRC寄存器的0 bit的位置。**  **（8）End**  **（9）CRC寄存器就是我们所要求的余数。**  参考    实现上用进程表作为输入的数据流，然后得到crc值后打印出来。用u16数据类型作为crc“寄存器”，最后返回即可。  **计算函数 syc\_get\_crc()** :proc.c (和syc\_get\_ticks在一起) proc.c     * 1. **添加函数到系统调用**   **添加到系统调用表 global.c**    **修改系统调用数量 const.h**    **在proto.h声明**     * 1. **实现定时触发**   类似于get-ticks,编写get\_crc函数，将syc\_get\_crc的中断号赋值给eax，然后调用0x90中断（系统调用中断），操作系统根据eax，运行syc\_get\_crc  **Syscall.asm**    在proto.h声明     * 1. **调用**   查看代码可知get-ticks在milli\_delay中被调用。因为校验CRC是校验当前进程，所以不应该放到milli\_delay，应该放到单个进程中  以B为例     * 1. **效果**   **可见每隔0.5s打印当前进程的CRC校验值。成功** |
| 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.**    解决：可能是缓存问题，多试了几次重新make file就解决了。  **实验结果记录：** |