Lab1: 实现分时多线程

徐栋

南京大学，计算机科学技术系

学号：**121220312**，邮箱：**dc.swind@gmail.com**

时间：**2014.5.20**

摘要：在 **Lab1** 框架的基础上，了解中断机制，实现线程创建、初始化，从而实现分时多线程。并实现 **printf** 函数，线程消亡、睡眠唤醒、上锁解锁等功能。

关键字：中断；分时多线程；操作系统；**sleep**；**printf**；**lock**；

1 实验介绍

通过过操作系统与计算机体系结构的课程学习以及 **Lab0** 的实验，我们对操作系统是如何运行的有了更加深入的了解。同时在实验以及对手册的阅读中对 **Intel IA32** 体系结构有了更多的认识，同时对中断机制、**I/O** 机制有了一定的了解。本次实验的目的就是要在对操作系统，更确切的说是对中断机制的理解的基础上 (理解中断执行过程的细节)，通过中断机制来实现分时多线程。同时完成一些基本的函数、功能，比如实现 **printf** 函数，线程创建消亡、睡眠唤醒、线程上锁解锁等。

2 实验目标

1. 理解框架代码，清楚框架代码运行过程。
2. 利用提供的 **putchar()** 函数实现 **printf()** 的一些功能。
3. 深入理解中断机制的详细过程，并通过中断机制实现分时多线程。这是本实验的主要部分。
4. 理解线程切换上下文，并可以正确设置初始值，使得创建的线程可以运行。并且可以使进程运行结束后正确消亡。
5. 实现 **lock()**、**unlock()** 提供临界区管理。
6. 实现线程的睡眠 **(sleep)**、唤醒 **(wakeup)** 功能。
7. 培养阅读文献、搜索资料、独自学习的能力。

3 涉及知识

3.1 Git[1]

3.2 int 0x80[2]

int 0x80 is the assembly language instruction that is used to invoke system calls in Linux on x86。我们可以借用 **int 0x80** 通过软件中断来主动申请一个中断。

3.3 中断处理过程

以时钟中断为例，在时钟中断到来时，由硬件将 **EFL**、**CS**、**EIP** 的值 **push** 到当前进程的栈中。然后根据中断向量，在 **IDT** 中找到中断向量对应的中断门，继而找到中断处理函数。这里需要说明的是，由于 **CPU** 进入中断门，所以这条中断线会被禁止使用，从而在中断处理过程中不会发生二次中断。之后中断处理函数将中断号 **push** 进栈，然后调用 **asm** 代码将寄存器现场保存，**call irq_handle()** 函数进行中断处理，之后根据 **current** 更改 **esp** 寄存器的值，**pop** 现场，最后 **iret**。至此中断完成，进程也完成切换。

4 设计思路及具体实现

4.1 printf

printf() 通过调用 **putchar()** 来实现。**printf()** 的第一个参数是一个字符串，其中也隐含了需要输出值的个数以及格式。需要做的就是将其中的% 所规定的格式用参数替换。难点在于 **printf** 的参数是可变的，所以我们无法得到参数的参数名。但是我们可以根据第一个参数 (格式字符串) 的地址推出后面参数的地址。因为后面的参数存储的只是参数的地址，而非参数的值，所以每个参数只占 **4** 个字节。

具体实现: 函数原型为 **int printf(const char *format, ...)**。其中 **format** 为格式字符串的地址，而 **printf** 的第一个参数的地址是 **format** 指针的地址 (存为 **base**)。所以遍历 **format** 字符串，每当遇到% 则进行格式替换，否则直接调用 **putchar** 输出字符。两个连续% 时输出%，而其他格式如%d、%x、%s、%c 时将 **base+4** 则取得对应参数存放的地址。要注意的细节有两个，一是%d、%x 格式需要注意正负，二是函数的返回值是 **putchar** 的个数。

4.2 初始化 Thread(PCB) 现场

Thread 结构主要包括 **tf** 指针及一个栈。其中 **tf** 指向线程的现场 **TrapFrame** 结构。我们需要正确设置 **TrapFrame** 中线程现场的初始值。**Thread** 的其他成员还包括 **PID**(因为是以数组的格式提前申请了 **Thread** 空间，所以我初始化 **PID=-1** 表示未分配，而分配后将其 **PID** 设为数组的下标。)、**sleep**(是否睡眠，初始化为 **0**)、**lock** 计数器 (初始化为 **0**)、指向前后线程的指针、存放线程函数地址的 **entry** 等。而 **TrapFrame** 的初始化，需要初始化 **8** 个通用寄存器，**irq** 中断号，和三个寄存器 **efl**、**eip**、**cs**。**8** 个通用寄存器除 **xxx** 外均可以初始化为 **0**，而 **irq** 也可以初始化为 **0**。**xxx** 初始化为 **tf+32**，这是 **pushal** 时 **esp** 寄存器的值。最重要的是设置 **efl**、**eip**、**cs** 的值。我通过在 **irq_handle()** 函数中打印 **tf** 的这三个值，得到 **cs=8**。所以我初始化 **cs=8**，然后通过 **IA32** 手册 **Volume-3A-61** 页，可以查到 **eflags**

各个 bit 的含义。其中 if 位为第 9 位 (从 0 开始), 即应该设为 512，同时第 1 位是常数 1，所以我设置 efl=514. 最后设置 eip 为函数入口地址，这里为一个外壳函数 g 函数的地址。同时将线程函数地址 (entry) 存入 Thread 结构中。使得可以在 g 中通过 current->entry 进入对应的线程函数执行，从而可以在函数返回后正确消亡线程。

4.3 汇编代码实现 esp 切换

current 指向的是 Thread 结构，而 Thread 结构的前 4 个字节是 tf，tf 指向 TrapFrame。所以需要两次寻址来使得 esp 指向 TrapFrame。实验中汇编语法为 AT&T 所以通过 movl (current),%eax 和 movl(%eax),%esp 来实现。

4.4 irq_handle() 的细节

首先要做的是线程切换及现场保存，这是无关于中断号的部分，即在每种中断类型中都要执行。而中断类型这里我们只用到了 3 种 1000,1001，0x80，其中 0x80 和 1000 牵扯到线程切换，在当前情况下可以合二为一。1001 处理键盘中断。这里只说明一下线程切换部分。分为三种情况，一是当前 current 为 NULL，这时，需要将当前的寄存器现场保存到 idle 线程的现场中，即保存到 Threads_arr[0] 中，如果当前非 sleep 线程大于 0，则线程切换，否则 current 赋值为 idle。第二种情况是非 sleep 线程 =0，此时保存 current 现场，切换到 idle 线程。第三种情况即当前非 sleep 线程数量大于 0，此时正常进行切换，current->next 存着下一个线程的地址。当然也需要判断一下 current->next 的线程是否 sleep，用 while 循环实现。（报告、实验中提到的线程数量均不计入 idle 线程）

4.5 进程创建、消亡及 idle 线程的实现

首先以数组 Threads_arr[MAX_THREADS_N] 的形式申请 MAX_THREADS_N 个 Thread 结构体的空间。当创建线程时，即 create_kthread()，分配一个空的 Thread 结构体（即 PCB=-1），并初始化（初始化细节在上面介绍过），PCB 置为数组下标。同时加入 Threadlist 链表中、线程计数器 Thread_N+1。消亡时，即 stop_thread()，归还空间，从链表中删除当前进程且 Thread_N-1。最后申请一个中断，因为在线程消亡后，应该立即申请一个中断进行切换，不应该继续执行此线程的代码。这里我使用过两种方式来申请一个中断。考虑到此时可能是关中断状态，第一种方式为开中断，然后 wait 一个时钟中断；第二种方式为立即申请一个 int 0x80 中断。显然第二种方式更为合理，且在 sleep 函数的实现中有更多的细节需要考虑，具体解释将在 sleep 中阐述。

需要注意的是我将 Threads_arr[0] 设置为 idle 线程。且 idle 线程不放入 Threadlist 中，不参与调度。当链表为空，即其他线程个数为 0 或者是其他非 sleep 线程 (Thread_N - Thread_Sleep) 为 0 的时候, 在线程切换时，切换到 idle 线程。实验中我设置的 idle 线程即 bootload 后的线程 (entry)。其中线程创建消亡均在 lock() 下进行。同时需要注意的是线程消亡时，若 Thread_Sleep=1，则需要将 ThreadSleep 计数器也 -1。

这里还需要说明的是，为了使线程可以正确的消亡。在线程函数外包了一个函数 g，将其地址作为每个线程 eip 的初始值，同时将线程函数的地址作为 Thread 结构的一个成员 entry 保存，在 g 函数中执行语句 ((void ()())current->entry)(); 跳转到线程函数执行。并在函数返回后，执行 stop_thread() 使得线程正确消亡。

4.6 lock、unlock

在 Thread 中加入 lock 计数器，当 lock 时，将计数器 +1，关中断。unlock 时则是计数器 -1，如果计数器等于 0 时，则开中断。

4.7 sleep、wakeup

在 Thread 中加入 sleep 标志位，当 sleep 时，将标志位置数。需要特别注意的是，即 sleep 后需要立即等待一个中断，进行切换，且 sleep 可能在 lock 中调度，正如线程消亡中提到过的，我用两种方式实现过，首先第一种先开中断，然后 wait 一个时钟中断，这里不同于线程消亡的是，当 wakeup 后，其中断状态应该是 sleep 前的状态，而这种方式在切换线程前将中断打开了，所以我在 irq_handle() 中判断其 lock 计数器，如果 lock 计数器大于 0，那么 efl 的 if 位应该清零再恢复现场。而第二种方式则简单的多，只要在 sleep 位更改后执行 int 0x80 申请一个中断即可。在 irq_handle() 中也不需要在多做判断。所以我在最终的版本中使用了第二种方式来主动申请中断。wakeup 的实现相对要简单的多，只需要简单的将传入的 Thread 的 sleep 清零即可。这里我的线程调度策略是唤醒线程不立即执行，而是继续执行当前线程，以免出现饥饿。这两个函数需要在 lock() 上锁状态下执行。执行完再 unlock() 解锁。

4.8 int 0x80 的实现

int 0x80 的实现并不麻烦，因为中断处理的过程是首先在 IDT 中根据中断向量寻找对应的中断门，所以首先需要在 IDT 中注册一个中断，然后在 do_irq.S 中实现中断处理函数即可。在中断处理函数中，同其他中断一样，push 中断号后 call asm_do_irq 进行进一步处理。

5 问题及解决方法

Q1：根据 current 始终无法得到正确的 esp 值。

A1：研究排查了好久，最终才发现是汇编代码写错。esp 是 32bit 的值，而在 move 的过程中，使用了 16bit 的 movw 指令。改成 movl 后解决。（汇编还是要熟悉一下，而且不能随手拿来指令不仔细了解就使用。）

Q2：在中断处理过程中，不会产生时钟中断，但是找不到显式的关中断和开中断指令。

A2：在 irq_handle() 中我写了一个死循环进行测试，系统即死在循环中，但是没有发现显式的关中断指令。经过查阅资料得知，是通过中断门进行中断处理，此时中断线会禁用，所以无法产生时钟中断。而在切换线程后，中断线恢复有效。

Q3：Git log 发生错误。

A3：git 中有一个 master 中的 current 指向当前版本，而当出现一些意外的情况时，current 往往会指向一个空文件。我开始将这个空文件删除。再 git log 的时候就会 fatal。因为它无法找到当前版本。最后我找到 git log 的记录，找到最后的一个版本，然后将 current 修改过去。Done

Q4：sleep、lock 等逻辑错误。

A4：仔细思考后实现。

6 总结

首先很高兴将实验目标达成，同时更加高兴的是培养了自己查阅资料、文档的能力并且使用自己实现的功能会有一种满足感。同时也正如 **JYY** 学长说的那样，不知不觉中学到了很多东西，在阅读 **IA32** 手册、搜索资料的过程中，对中断机制的整个过程有了详细的理解和把握，对操作系统一些以前并不熟悉的功能的实现有了清晰的认识。就像之前只知道线程可以切换，但却对具体如何切换不甚理解。同时对之前不想用心的去学习的一些知识也进行了一定程度上的了解，比如说 **git**，之前觉得只是 **JYY** 学长用来" 收集有用数据" 的东西，不了解也就算了。但是当真正自己的 **git** 出现问题后，就会千方百计的去了解它的原理，了解如何去修复它。知识也就这样一点一点的累积起来。总的来说，这是一个非常棒的实验。

致谢

特别感谢 **JYY** 学长在实验课中给予的正确引导！可以说没有 **JYY** 学长的引导，我不会去读那么多文献，了解这么多知识，更不可能完成这个实验。

References

[1] wikipedia-Git 维基百科 -Git

[2] int 0x80

[3] IA32 手册

附：问题回答

1. 使用 gdb，统计以下内容时钟和键盘中断处理程序 (从执行 asm_do_irq 算起，到执行 iret 为止) 时，分别执行了多少条 x86 指令。在一个真实的计算机上，仅仅执行这些指令大约要耗费多少时间？如果刚才的问题你是手工完成的，现在交给你一个更艰巨的任务：统计从 entry() 函数开始执行，到第一次执行到 entry() 函数中的 wait_for_interrupt()，总共执行了多少条 x86 指令。

第一问：进入 **gdb** 后，我设置断点 **hb asm_do_irq** 然后输入 **stepi** 进行单步调试，**stepi** 跟 **step** 以及 **next** 和 **step** 是有区别的，**step** 会进入函数调用，**next** 不会，而且 **stepi** 是单步调试机器指令，一条代码可能是多条机器指令实现的。所以我用 **stepi** 进行单步调试，最后统计出执行了 **36** 条 **x86** 指令 (而 **step** 是 **19** 条，这里可以看出不同)。在真实的计算机上，执行这些指令需要花费的时间大约是 **15ns**（假设主频为 **2.4GHz**，**CPI** 为 **1**）

第二问：这里指令条数大大增加，一次次输入 **stepi** 调试是不现实的，好在 **stepi** 可以加参数表示执行当前指令下面的第 **x** 条指令，所以我使用 **100** 的偏移量逐次尝试，统计出 **x86** 指令条数为 **2262** 条，其中执行到 **initserial** 为 **2234** 条。