16 级计科 7 班: 操作系统原理实验 #6

Due on Saturday, April 28, 2018

凌应标 周一 *9-10* 节

颜彬

16337269

Content

1 实验目的

理解多道程序和分时系统的实现原理。理解二进程模型的原理。熟悉进程调度的方法。理解并能自行设计进程控制块，存储必要的进程信息。设计进程表和相关底层函数接口供内核调用。进一了解时钟中断的工作原理，完成用户程序的轮流执行。

2 实验过程

2.1 进程控制块

进程控制块中最核心的部分是寄存器映像 RegisterImage. 其实现如代码1所示。整个寄存器映像可以分为若干个部分。

第一个部分是 flags, cs 和 ip。它在中断发生时自动入栈。

第二部分是 ax, cs, dx, bx, sp, bp, si, di。这部分可以由 pusha 指令推入栈中。结构体中这些寄存器的排列顺序恰好是 pusha 指令入栈的顺序。根据文档，pusha 压入栈中的 sp 是 ‘‘执行 pusha 前的 sp 的值’’。在结构体中，该 sp 称为 “old_sp’’。(与以后提到的 “sp’’ 相区分) 值得一提的是该 C 语言编译时必须关闭所有优化，否则编译器可能会进行重排操作。

第三个部分是段寄存器，从 ds 到 ss。这部分在保护现场时，用 push 指令手动入栈。

最后一个部分只包含一个寄存器,sp. 该 sp 值的是在把所有的其他寄存器都压入栈中后的 sp 的值。该值在2.2节的现场保护和第2.3节的现场恢复中极其重要。

2.2 保存上下文

保存上下文的代码如代码2所示。代码大概分为若干的几个部分（空行隔开)。

pusha 和一系列的 push 把寄存器保存在栈中。必须在时钟中断到达后的第一时间保存寄存器。这是因为这些寄存器应该与 flag, cs, ip 连续存放。一旦这些寄存器和 flag, cs, ip 分隔开后，会对 PCB 块的保存带来很大的麻烦。

只有将所有寄存器都保存到栈后，才能执行 mov es, 0 和 mov ds, 0 (本代码仅作示意) 的操作。由于需要将 sp 压入栈中，但 push sp 是有歧义的汇编代码，为了可读性和可维护性，这里采用 mov [ss:sp-2], sp 的写法（本代码仅作示意)。

寄存器都保存完毕后，采用 rep movsb 指令将栈中保存的所有内容复制到 PCB 块中。

2.3 恢复上下文

恢复现场是保护现场的逆过程。这里采用略为讨巧的方法。恢复现场时，先恢复 sp 和 ss，让栈指针指向 ‘‘被恢复进程’’ 的栈顶。随后，只需要按照 ‘‘保存上下文的逆序” 执行 pop 指令。即可顺利地恢复所有寄存器的值。随后采用 iret，再把 flag, cs, ip 出栈的同时，把 cs 和 ip 恢复到正确的值.

这里有若干细节需要注意。首先是，popa 指令不会修改 sp 的值（但它会把 sp 从栈中弹出)。popa 的这个行为很好理解，因为随意修改 sp 会导致极其严重的错误。(虽然在进程切换时，我会更希望 popa

代码 1: PCB 寄存器映像

```
struct RegisterImage {
    uint16_t sp;

    //segments
    uint16_t ss;
    ...
    uint16_t ds;

    // pusha
    uint16_t di;
    uint16_t si;
    uint16_t bp;
    uint16_t old_sp;
    uint16_t bx;
    uint16_t dx;
    uint16_t cx;
    uint16_t ax;

    // pushed by interupt
    uint16_t ip;
    uint16_t cs;
    uint16_t flags;
}__attribute__((packed));
```

指令能帮我把 sp 也修改了)。第二个细节是，无论如何都要先把 PCB 中的值复制到栈上（哪怕栈上已经有一份一模一样的副本），再做进程恢复。这是因为，整个操作系统的第一个进程（INIT 进程）需要靠“恢复上下文”操作来自动启动。而第一个进程（INIT 进程）的初始化操作是在 PCB 上做的，而不是栈上做的。

2.4 内存分配模型

本次项目还做了简单的内存分配，支持在内核上对某个保留内存空间的 malloc 和 free 的操作。这允许了用户程序在运行前才分配和指定内存空间。配合项目 4 完成的文件系统，本次项目能更自由和智能地执行用户程序。

内存分配器采用若干个链表来实现。首先将 0x10000 到 0x20000 的内存空间分成大小为 0x20, 0x800, 0x1000 的块。这些块组成 3 个独立的链表。每次分配内存时，先计算出最合适的块，从将该块从链表中移除，并返回该块的首地址。在回收内存时，只需将该内存块回收入链表中即可。在经典的内存池实现中，内存块的大小应包括 4, 8, 16, ..., 2^n, ..., 1024 字节。如此，维护 9 个独立的链表。本实现简化了这些细节（考虑到项目时间问题），仅提供了 3 种不同大小的块。

内存分配需要做初始化。初始化函数如代码3所示。代码只截取了有代表性的部分。

初始化中，“强行”将内存 p 中的值修改为 p + offset, 然后把 p 修改为 p + offset, 不断重复以上步骤。如此即可在内存空间中“强行”构造出一个链表。

在内存分配时，通过将 phead = *phead 这一操作，将头指针的值改为头指针指向的内存空间的值（下

代码 2: 保存上下文的实现

```
saveRegisterImage:
    ; 高地址
    pusha

    push ds ; sp + 8
    push es ; sp + 6
    push fs ; sp +4
    push gs ; sp + 2
    push ss ; sp
    ; 低地址

    mov ax, 0 ; cs is 0
    mov es, ax
    mov ds, ax

    calll get_current_PCB_address

    mov bx, sp
    mov word [ss:bx-2], sp

    mov di, ax
    mov ax, sp
    sub ax, 2
    mov si, ax

    mov cx, 17 * 2
    cld
    rep movsb
```

一个块的地址)。如此即可实现将块弹出链表的操作。内存回收的代码如4所示。若返回的内存块大小大于最大的块，则说明程序出错，直接返回。否则，找到该块属于哪个链表，通过两行语句将回收的内存放回到链表的头。在下一次内存申请时，回收的内存可以被再次使用。

内存分配的示意图如图1所示。其中白色空心箭头（带有 HEAD 字样）的为内存分配器的头指针。白色空心箭头（带有 RETUREN 字样）的为 malloc 函数的返回值。内存回收的示意图如2所示。其中标号为 x 的 block 代表待回收的代码块。

由于本次项目时间紧迫。所以没有将 malloc 和 free 封装成系统调用。这就导致了分配 0x10000 0x20000 空间时，段还处在 0x0000 处。而访问 0x10000 会因为偏移量过大而导致访问错误。所以此处采用了一个变通方案。在进入所有的内存分配器函数之后，先手动把 ds 置为 0x1000. 在离开后手动恢复 ds。如此即可暂时解决这个跨段修改内存的问题。在以后的版本中，会把内存分配封装成系统调用。

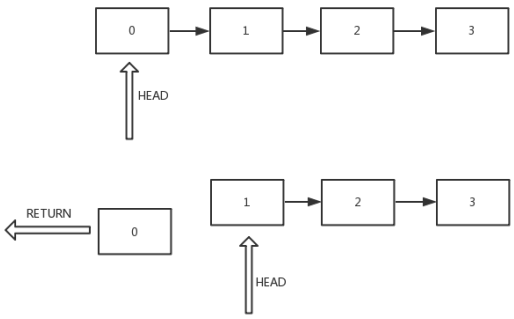

图 1: 分配内存示意图

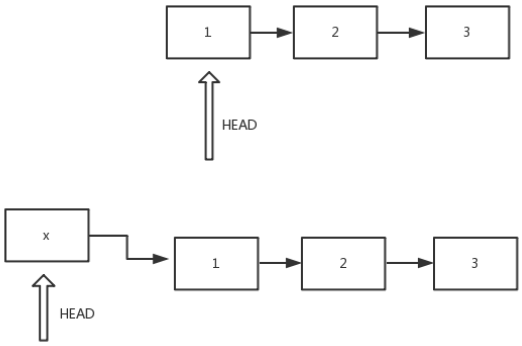

图 2: 内存释放示意图

3 实验结果

在进入程序后，输入 run stoneQ，或 run stoneW，或 run stoneA 或 run stoneS 即可在屏幕的 4 个不同的位置运行 stone 程序。每次新程序运行时，会先调用内存分配器获得分配的内存，初始化进程控制块，并开始进程的执行。
如图3所示，进入程序后连续输入 3 次 run stoneQ。随后输入 pc 指令查看当前线程的状态。可以看到屏幕右下角有 3 个不同的字母'A' 在独立地运动 (**图片效果不好，请自行尝试**)。同时可以看到这 3 个不同的进程有自己独立的进程 id 和内存地址。由于中断占据了 0 号进程，所以这 3 个用户进程的 id 为 1-3.

如图4紧接着上图，继续执行 kill 1, 把 1 号进程杀死，并执行 run stoneW, 产生一个新的 stone 程序。在杀死了 1 进程后，内存管理器会回收掉进程 1 占据的内存。在产生新进程时，可以把回收的内存重新分配出去。所以如图所示，进程 4 的内存起始地址为 0x11000, 恰好为原来进程 1 的内存地址。

图5用于证明，本次实验中系统调用仍能正常进行。

代码 3: 内存分配器的初始化

```
void init_mm() {
    __mm__enter();

    MM_LITTLE_HEAD = (uint32_t*)LITTLE_BEGIN;

    int i = 0;
    void* p = (void* )(LITTLE_BEGIN);
    while (p + BLOCK_LITTLE_SIZE * i <= (void*)LITTLE_END) {
        *(uint32_t*) (p + BLOCK_LITTLE_SIZE * i)
            = (uint32_t)(p + BLOCK_LITTLE_SIZE * (i+1));
        i++;
    }

    ...

    __mm__leave();
}
```

代码 4: 内存分配器的 free 操作实现

```
void mm_free(uint32_t addr, int32_t size) {
    __mm__enter();
    if (size >= BLOCK_LARGE_SIZE) return;
    else if (size > BLOCK_MID_SIZE) {
        *(uint32_t*)addr = (uint32_t)MM_LARGE_HEAD;
        MM_LARGE_HEAD = (uint32_t*)addr;
    }

    ...

    __mm__leave();
}
```

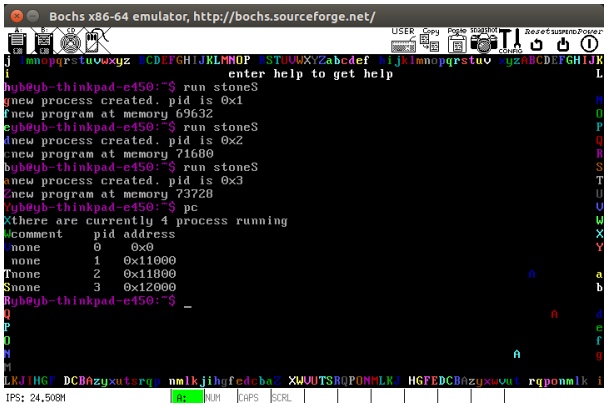

图 3: 同时运行 3 个 stoneQ 程序的画面

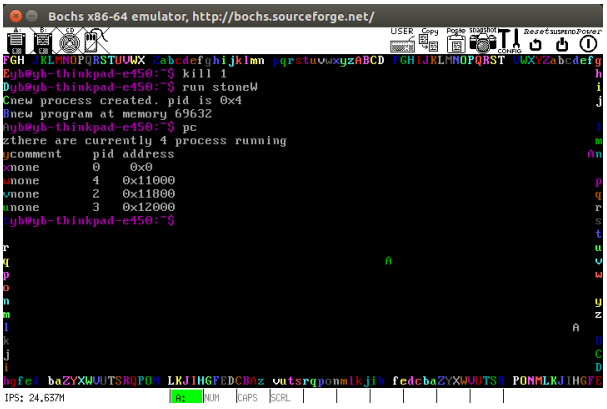

图 4: 杀死一个进程，又创建一个新进程后的画面

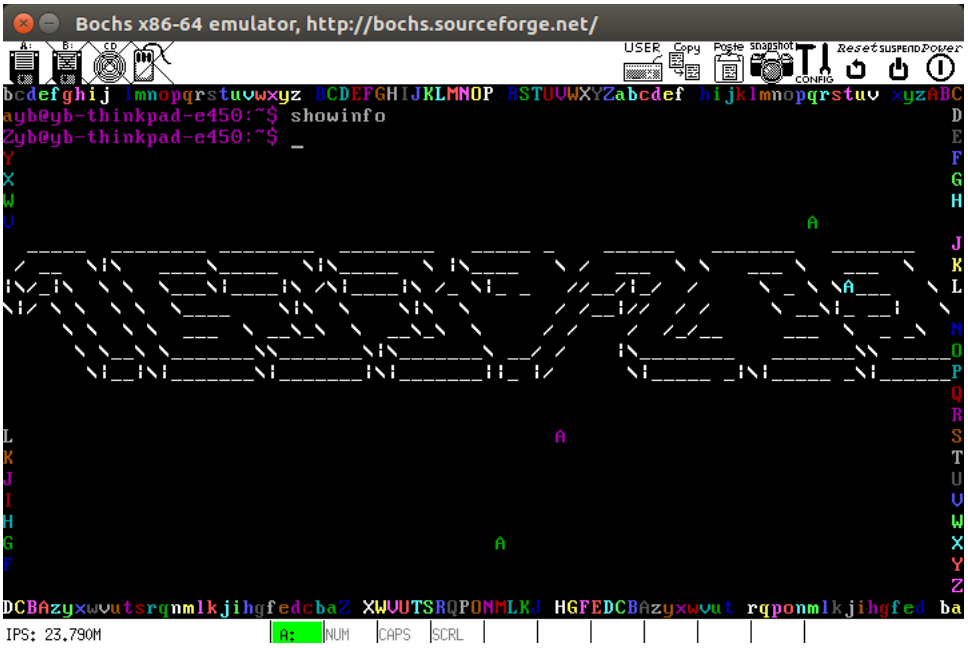

图 5: 用于证明系统调用仍能正常执行的例子

4 实验总结

4.1 亮点介绍

4.1.1 方便的终端操控

本实验提供了新的终端指令。run stoneQ (或把 stoneQ 替换为 stoneW,stoneA, stoneS) 可以产生 4 种不同的用户程序进程。pc 指令可以列出当前存在的所有进程、进程 id 以及他们所在的物理内存地址。kill <pid> 命令可以杀死进程。clear 命令可以做清屏。

本实验创新地采用了如下的实现：在用户程序运行时，不退出终端。于是可以一边观察用户程序的执行，一边通过终端的操作增添或删除进程。这种交互性设计可以良好地检验出程序的正确性。用户只要乐意，可以创建任意多个进程，进程的代码段可以来自于四个用户程序中的任何一个。

4.1.2 内存分配器

本项目实现了内存分配。在运行一个新的用户程序时，不是通过硬编码的方式决定其内存位置，而是由内存分配器分配一段未只用的内存。

回收的内存可以重复使用。图4展示了相关情况。在杀死一个进程后，产生一个新的进程时，被回收的内存又被重新分配出去了。

内存分配器的优点还在于，它允许了同一个用户程序被运行多次，例如图3显示了相关情况。图中，同时运行了 3 个 stoneS 程序，这 3 个程序不分享内存地址，互不干扰地在独立的内存中执行。

4.1.3 与文件系统的配合

由于以前的项目实现了文件系统，本项目又实现了内存分配器，两者可以配合工作，发挥最大的作用。

进程产生时，先通过内存分配器分配内存，再通过文件系统将文件载入到指定的内存空间中，再通过进程调度让进程自动执行。全程自动化程度高，程序灵活。

4.2 实验心得

这个实验表面上看很简单，但我还是花费了比较多的时间。而且上两周的学习任务比较重，这次实验几乎是在舍友都睡着了的时间段写完的。

首先是进程的保存和恢复。由于思路没有完全理清，这部分的代码我重构了两三次。这之间我反复纠结的细节有：sp 该如何存储？存储何时的 sp?sp 和 ss 如何恢复？恢复后栈顶的位置变了怎么办？ds 该合适恢复？恢复后所有访存操作的行为都会改变，怎么办？我甚至还一度认为，“将寄存器从栈上拷贝到 PCB” 和 “从 PCB 拷贝到栈上” 这两个操作是没有必要的。

这些问题单独出现的时候都不难解决。但当他们在进程保存和进程恢复时同时出现时，就会成为一个比较大的困难。因为我必须同时解决好全部的这些问题。最终我采取的方案是，sp 存储的是 “保存完所有寄存器后” 的 sp，用 mov 的方式把 sp 压入栈中。恢复进程时先恢复 sp 和 ss 到被恢复进程的栈顶。ds 直接用 pop 恢复，并保证保存和恢复现场的全程不访问内存。我经过仔细的考虑（和代码的实践后）得出简论，栈和 PCB 之间的拷贝是必要的，不做拷贝总会出现各种各样的困难，导致实现变得很复杂。

得出上述的思考只用几句话的时间，但这是我一个凌晨的试错换来的。在调试的期间还遇到上述提到的忘关中断的 BUG。这些错误堆积起来，让这次实验的完成变得更加困难了。但经过这次实验我对进程切换的理解也更深刻了。

这次实验还有很多不足。个人认为最大的不足在于，太多的过程没有被封装成系统调用。由于 C 程序是对段透明的，这导致了当一个进程的段不为 0 时，调用许多 C 程序时，一旦发生寻址等操作，就会产生错误。正确的解决办法是利用中断修改 cs, 把 ds 同步成 cs，此时 ds 就被同步到正确的段中。当执行完相关的操作后，再恢复 ds，离开中断。这样就可以让许多内核级的操作在段不为 0 的程序上运行。更进一步地，终端进程就可以脱离内核代码而独立存在。在本项目中，终端依旧只能作为一个程序和内核链接在一起。

4.3 细节上的更新

4.3.1 sscanf 的实现

本项目实现了 sscanf 的大部分操作。实现了%s, %d, %x, %c, 等匹配操作。为终端指令的提取提供基石。

4.3.2 内核拓展

内核从预留 24 个扇区拓展到预留 40 个扇区。修改了文件系统的相关配置。

4.3.3 文件系统的跨段处理

由于本次实验出现了跨段问题，而这在前几次项目中没有出现，故本项目略微修改了文件系统的实现，使其支持跨段的载入。

4.4 BUG 汇总

4.4.1 文件系统的历史遗留问题

我的文件系统调用在传参时，某处地址的类型声明为了 int16_t。在之前这不会发生错误，因为以 0x00 为段寄存器时，偏移量最多为 0xFFFF, 在 int16_t 的范围内。但是本项目发生了跨段，当我传入更高的地址时，会被 int16_t 类型截断。导致程序无法载入到内存中。

这个错误花费了比较多的时间。因为我的文件系统的结构化做得比较好，这也意味着函数调用会比较深。错误的追踪也比较困难。

4.4.2 中断未关闭的错误

在项目实现到一半时，我发现程序会卡死，而且时钟中断没有触发。这极其令人费解，我刚开始考虑了很多的情况，都没有找到这个问题发生的原因。我很疑惑为何时钟中断没有被触发。我试过手动往中断端口中写入数据允许下一次中断，确保硬件得到允许中断的信号。我使用手动调用 int 08H，观察程序执行状态，发现程序正常执行，且栈能回到正确的位置。由于此时我已经写好了进程切换（正在测试中), 我试过反复注释掉不同地方的代码观察情况。结果发现无论怎么改变代码, 总有各种不同的错误。

在折腾了很久后，突然有一个瞬间想到中断的问题。在内核初始化过程中关中断。在所有初始化结束后开中断。于是 bug 解决。

我现在并没有完全弄懂为何这个 BUG 会导致时钟中断不再发生。毕竟时钟中断可能发生在程序执行的任何时刻，未关中断这个错误可能导致程序执行的任何一个时刻出错。分析出具体的原因太难了。但可以得知的是，不正确地开关中断是很严重的错误。

附录 A 参考文献

1. https://blog.csdn.net/longintchar/article/details/50602851
   16 位和 32 位汇编指令的不同（尤其是 push 指令）
2. https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1
   GCC 内嵌汇编的书写。