操作系统是如何工作的

在上一章计算机的工作模型以及汇编语言的基础上，我们可以进一步理解操作系统的 核心工作机制。本章的目标是在 mykernel 的基础上编写一个简单的内核，在此之前进一步 分析了函数调用堆栈机制，以及 C 代码中内嵌汇编的写法。

## 2.1 函数调用堆栈

前面分析了存储程序计算机的内容，也就是冯·诺依曼体系结构。通过对 x86 汇编的 简要介绍，读者有了在 Linux 下汇编的基础，然后分析了一个实际的小程序，看它的汇编 代码是怎么工作的，以此来深入理解存储程序计算机，让读者看到代码在执行层面遵守着 存储程序计算机的基本逻辑框架。第 1 章的内容和实验实际上已经涉及了计算机里面的 3 个非常重要的基础性概念中的两个：一个是存储程序计算机，它基本上是所有计算机的基 础性的逻辑框架；另一个在分析程序和学习汇编指令时也涉及了，就是堆栈。

堆栈是计算机中一个非常基础性的内容，然而堆栈不是一开始就有的，在最早的时候， 计算机没有高级语言，没有 C 语言，只有机器语言和汇编语言。这时堆栈可能并不是太重 要，用汇编写代码时可以跳转语句到代码的前面形成一个循环。有了函数的概念，也就是 有了高级语言之后，函数调用的实现机制就成为一个关键问题，必须要借助堆栈机制，可 以说堆栈是高级语言可以实现的基础机制。

除了存储程序计算机和函数调用堆栈机制，还有一个非常基础性的概念就是中断，这 3 个关键性的方法机制可以叫作计算机的 3 个法宝。

前面内容讲到了 3 个法宝中的两个，第一个是存储程序计算机，第二个是函数调用堆 栈。接下来需要仔细分析函数调用堆栈，因为函数调用堆栈是比较基础性的概念，对读者理解操作系统的一些关键代码非常重要，然后简单介绍中断的机制。

首先来看堆栈，堆栈是 C 语言程序运行时必须使用的记录函数调用路径和参数存储的 空间，堆栈具体的作用有：记录函数调用框架、传递函数参数、保存返回值的地址、提供 函数内部局部变量的存储空间等。

C 语言编译器对堆栈的使用有一套规则，当然不同的编译器对堆栈的使用规则会有一 些差异，但总体上大同小异。了解堆栈存在的目的和编译器对堆栈使用的规则是读者理解 操作系统一些关键性代码的基础。前面已经涉及堆栈的部分内容，这里再具体研究一下堆 栈相关的内容。

### 堆栈相关的寄存器

1、ESP：堆栈指针（stack pointer）。

2、EBP：基址指针（base pointer），在 C 语言中用作记录当前函数调用基址。

对于 x86 体系结构来讲，堆栈空间是从高地址向低地址增长的，如图 2-1 所示。

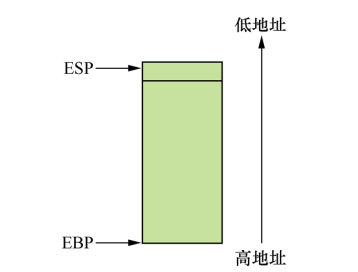


图2-1 x86堆栈空间

### 堆栈操作

1、push：栈顶地址减少 4 个字节（32 位），并将操作数放入栈顶存储单元。

2、pop：栈顶地址增加 4 个字节（32 位），并将栈顶存储单元的内容放入操作数。

之前也介绍过堆栈相关的寄存器，主要就是 ESP 栈顶指针寄存器和 EBP 基址指针寄 存器，堆栈主要的操作是 push 和 pop。对于 x86 体系结构来讲，栈是从高地址向低地址 增加的。

EBP 寄存器在 C 语言中用作记录当前函数调用的基址，如果当前函数调用比较深，每 一个函数的 EBP 是不一样的。函数调用堆栈就是由多个逻辑上的栈堆叠起来的框架，利用

这样的堆栈框架实现函数的调用和返回。

### 其他关键寄存器

CS:EIP 总是指向下一条的指令地址，这里用到了 CS 寄存器，也就是代码段寄存器和EIP 总是指向下一条的指令地址。如果程序比较简单，像我们上一章的实验里编译的一个 小程序，它只有一个代码段，所有的 EIP 前面的 CS 代码段寄存器的值都是相同的。当然 这是一个特例，一般程序都至少会使用到标准库，整个程序会有多个代码段。

1、顺序执行：总是指向地址连续的下一条指令。

2、跳转/分支：执行这样的指令时，CS:EIP 的值会根据程序需要被修改。

3、call：将当前 CS:EIP 的值压入栈顶，CS:EIP 指向被调用函数的入口地址。

4、ret：从栈顶弹出原来保存在这里的 CS:EIP 的值，放入 CS:EIP 中。

堆栈是 C 语言程序运行时必需的一个记录函数调用路径和参数存储的空间，堆栈实际上 已经在 CPU 内部给我们集成好了功能，是 CPU 指令集的一部分。比如 32 位的 x86 指令集 中就有 pushl 和 popl 指令用来做压栈和出栈操作，enter 和 leave 指令更进一步对函数调用 堆栈框架的建立和拆除进行封装，帮我们提供了简洁的指令来做函数调用堆栈框架的操作。 堆栈里面特别关键的就是函数调用堆栈框架，如图 2-2 所示。



图2-2 函数调用堆栈框架

### 用堆栈来传递函数的参数

对 32 位的 x86 CPU 来讲，通过堆栈来传递参数的方法是从右到左依次压栈，64 位机 器在传递参数的方式上可能会稍有不同，这里不仔细研究它们之间的差异，下面以 32 位的 x86 CPU 为例。

### 函数是如何传递返回值的

这里涉及保存返回值和返回地址的方式，保存返回值，就是程序用 EAX 寄存器来保存 返回值。如果有多个返回值，EAX 寄存器返回的是一个内存地址，这个内存地址里面可以 指向很多的返回数据，EAX 寄存器可以保存返回地址。

函数还可以通过参数来传递返回值，如果参数是一个指针且该指针指向的内存空间是 可以写的，那么函数体的代码可以把需要返回的数据写入该内存空间。这样调用函数的代 码在函数执行结束后，就可以通过该指针参数来访问这些函数返回的数据。

### 堆栈还提供局部变量的空间

函数体内的局部变量是通过堆栈来存储的，目前的编译器一般在函数开始执行时预留 出足够的栈空间用来保存函数体内所有的局部变量，但早期的编译器并没有智能地预留空 间，而是要求程序员必须将局部变量的声明全部写在函数体的头部。

### 编译器使用堆栈的规则

C 语言编译器对堆栈有一套使用规则，而且不同版本的编译器规则还有所不同。比如在做 试验时，在读者的平台上，你可能用 GCC 把汇编出来的代码换到另外一台机器上，汇编一个 相同的 C 程序，汇编出来代码可能会有一些不同，这可能是因为编译器的版本不同。如果两台 机器的处理器指令集不同，汇编出来的汇编代码也会有所不同，这个需要读者了解。因为不同 的汇编指令序列可以实现完全相同的功能，有时用这个指令能实现这个功能，用那个指令也能 实现这个功能。总之汇编出来的代码可能会有一些细微的差异，需要读者清楚产生差异的原 因。了解堆栈存在的目的和编译器对堆栈使用的规则，是理解操作系统一些关键性代码的基础。

## 2.2 借助Linux 内核部分源代码模拟存储程序计算机工作模型及时钟中断

在开始动手实践前，我们还需要了解 C 代码中内嵌汇编的写法。

### 2.2.1 内嵌汇编

内嵌汇编语法如下：

\_\_asm\_\_ \_\_volatile\_\_ (

汇编语句模板：

输出部分：

输入部分：

破坏描述部分

)；

下面通过一个简单的例子来熟悉内嵌汇编的语法规则。

#include <stdio.h>

int main()

{

/\* val1+val2=val3 \*/

unsigned int val1 = 1;

unsigned int val2 = 2;

unsigned int val3 = 0;

printf("val1:%d,val2:%d,val3:%d\n",val1,val2,val3);

asm volatile(

"movl $0,%%eax\n\t" /\* clear %eax to 0\*/

"addl %1,%%eax\n\t" /\* %eax += val1 \*/

"addl %2,%%eax\n\t" /\* %eax += val2 \*/

"movl %%eax,%0\n\t" /\* val2 = %eax\*/

: "=m" (val3) /\* output =m mean only write output memory variable\*/

: "c" (val1),"d" (val2) /\* input c or d mean %ecx/%edx\*/

);

printf("val1:%d+val2:%d=val3:%d\n",val1,val2,val3);

return 0;

}

这个例子是用汇编代码实现 val3 = val1 + val2 的功能，我们具体来看其中涉及的语法 规则。

\_\_asm\_\_是 GCC 关键字 asm 的宏定义，是内嵌汇编的关键字，表示这是一条内嵌汇编 语句。\_\_asm\_\_和 asm 可以互相替换使用：

#define \_\_asm\_\_ asm

\_\_volatile\_\_是 GCC 关键字 volatile 的宏定义，告诉编译器不要优化代码，汇编指令保 留原样。\_\_volatile\_\_和 volatile 可以互相替换使用：

#define \_\_volatile\_\_ volatile

内嵌汇编关键词 asm volatile 的括号内部第一部分是汇编代码，这里的汇编代码和之前 学习的汇编代码有一点点差异，体现在%转义符号。寄存器前面会多一个%的转义符号， 有两个%；而%加一个数字则表示第二部分输出、第三部分输入以及第四部分破坏描述（没 有破坏则可省略）的编号。

上述内嵌汇编范例中定义了 3 个变量 val1、val2 和 val3，希望求解 val3 = val1 + val2； 内嵌式汇编代码就是 asm volatile 后面的一段汇编代码，下面来具体分析。

第 1 行语句“movl $0,%%eax”是把 EAX 清 0。

第 2 行语句“addl %1，%%eax”，%1 是指下面的输出和输入的部分，第一个输出编 号为%0，第二个编号为%1，第三个就是%2。%1 是指 val1，前面有一个“c”，是指用 ECX 寄存器存储 val1 的值，这样编译器在编译时就自动把 val1 的值放到 ECX 里面。%1 实际上 就是把 ECX 的值与 EAX 寄存器求和然后放到 EAX 寄存器中，本例中由于 EAX 为 0，所 以结果是 ECX 的值放入了 EAX 寄存器。

第 3 行语句“addl %2，%%eax”，%2 是指 val2 存在 EDX 寄存器中，就是把 val1 的 值加上 val2 的值再放到 EAX 里。

最后一条指令“movl %%eax，%0”是把 val1 加上 val2 的值存储的地方放到%0，%0 就是 val3，我们这里用=m 修饰，它的意思就是写到内存变量里面去，m 就是内存 memory， 不是使用寄存器了，这条指令是直接把变量放到内存 val3 里面。

至此，这段代码就实现了 val3 = val1 + val2 的功能。

简单总结一下，如果把内嵌汇编当作一个函数，则第二部分输出和第三部分输入相当 于函数的参数和返回值，而第一部分的汇编代码则相当于函数内部的具体代码。

### 2.2.2 虚拟一个 x86 的 CPU 硬件平台

上一章内容用了相当多的篇幅来介绍 x86 的汇编，又仔细分析了函数调用堆栈和内嵌 汇编的写法，这是读者理解计算机的基础，接下来要做一个有趣的实验。一个操作系统那 么复杂，它的本质上是怎么工作的？“天下大事必作于细，天下难事必作于易”。下面我们 来还原整个系统，首先搭建一个虚拟的平台，虚拟一个 x86 的 CPU，然后使用 Linux 的源 代码把 CPU 初始化配置好，并配置好整个系统，开始执行编写的程序。

前文讲的计算机的 3 个法宝只有中断没有介绍过了，为了便于理解实验内容，这里简 要介绍中断的概念。有了中断才有了多道程序，在没有中断的机制之前，计算机只能一个 程序一个程序地执行，也就是批处理，而无法多个程序并发工作。有了中断机制的 CPU 帮 我们做了一件事情，就是当一个中断信号发生时，CPU 把当前正在执行的程序的 CS:EIP 寄存器和 ESP 寄存器等都压到了一个叫内核堆栈的地方，然后把 CS:EIP 指向一个中断处 理程序的入口，做保存现场的工作，之后执行其他程序，等重新回来时再恢复现场，恢复 CS:EIP 寄存器和 ESP 寄存器等，继续执行程序。

实验中模拟了时钟中断，每隔一段时间，发生一次时钟中断，这样我们就有基础写一 个时间片轮转调度的操作系统内核，这也是后面的实验目标。下面来具体看看如何虚拟一 个 x86 的 CPU。

先来看如何把这个虚拟的 x86 CPU 实验平台搭建起来。

sudo apt-get install qemu # install QEMU

sudo ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu

wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.4.tar.xz #

下载 linux-3.9.4.tar.xz，

linux-3.9.4源代码压缩包

<https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.4.tar.xz>。

wget https://raw.github.com/mengning/mykernel/master/mykernel\_for\_linux3.9.4sc.pat ch # download mykernel\_for\_linux3.9.4sc.patch

mykernel\_for\_linux3.9.4sc.patch补丁文件

<https://raw.github.com/mengning/mykernel/master/mykernel_for_linux3.9.4sc.patch>

xz -d linux-3.9.4.tar.xz

tar -xvf linux-3.9.4.tar

cd linux-3.9.4

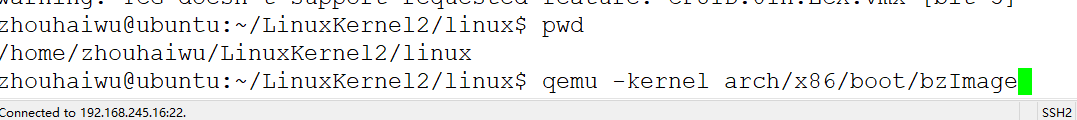
patch -p1 < ../mykernel\_for\_linux3.9.4sc.patch

make allnoconfig

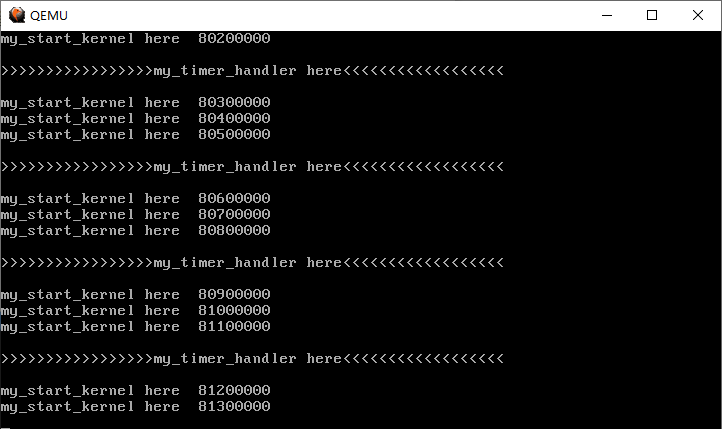
make

qemu -kernel arch/x86/boot/bzImage

命令如2-3图所示



搭建起来后的内核启动效果如图 2-4 所示。



在 Linux-3.9.4 内核源代码根目录下进入 mykernel 目录，可以看到 QEMU 窗口输出的 内容的代码 mymain.c 和 myinterrupt.c，当前有一个虚拟的 CPU 执行 C 代码的上下文环境， 可以看到 mymain.c 中的代码在不停地执行。同时有一个中断处理程序的上下文环境，周期性地产生的时钟中断信号，能够触发 myinterrupt.c 中的代码。这样就模拟一个带有时钟中 断的 x86 CPU，并初始化好了系统环境。读者只要在 mymain.c 的基础上继续写进程描述 PCB 和进程链表管理等代码，在 myinterrupt.c 的基础上完成进程切换代码，就可以完成一 个可运行的小 OS kernel。

## 2.3 在 mykernel 基础上构造一个简单的操作系统内核

庖丁解牛，一开始“所见无非牛者”，是因为对于牛体的结构还不了解，因此无非看见 的只是整头的牛。到“三年之后，未尝见全牛也”，因为脑海里浮现的已经是牛的内部肌理 筋骨了。之所以有了这种质的变化，一定是因为先见全牛，然后进一步深入其中，详细了 解牛的内部结构。我们需要一个“全牛”，才能进一步细致地解析它，那就让我们把整个系 统还原一下，看看全牛是什么样的。

在 mykernel 虚拟的 x86 CPU 基础上实现一个简单的操作系统内核只需要写两三百行代 码，尽管代码量看起来并不大，但是对很多人来说还是很有挑战的，这里给出一份代码范 例供读者参考。

### 2.3.1 代码范例

为了方便查看，特在文件中进行注释，以加深读者的理解。

增加一个 mypcb.h 头文件，用来定义进程控制块（Process Control Block），也就是进 程结构体的定义，在 Linux 内核中是 struct tast\_struct 结构体。

/\*

\* linux/mykernel/mypcb.h

\*

\* Kernel internal PCB types

\*

\* Copyright (C) 2013 Mengning

\*

\*/

#define MAX\_TASK\_NUM 4

#define KERNEL\_STACK\_SIZE 1024\*8

/\* CPU-specific state of this task \*/

struct Thread {

unsigned long ip;

unsigned long sp;

};

typedef struct PCB{

int pid;

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

char stack[KERNEL\_STACK\_SIZE];

/\* CPU-specific state of this task \*/

struct Thread thread;

unsigned long task\_entry;

struct PCB \*next;

}tPCB;

void my\_schedule(void);

对 mymain.c 进行修改，这里是 mykernel 内核代码的入口，负责初始化内核的各个组 成部分。在 Linux 内核源代码中，实际的内核入口是 init/main.c 中的 start\_kernel(void)函数。

/\*

\* linux/mykernel/mymain.c

\*

\* Kernel internal my\_start\_kernel

\*

\* Copyright (C) 2013 Mengning

\*

\*/

#include <linux/types.h>

#include <linux/string.h>

#include <linux/ctype.h>

#include <linux/tty.h>

#include <linux/vmalloc.h>

#include "mypcb.h"

tPCB task[MAX\_TASK\_NUM];

tPCB \* my\_current\_task = NULL;

volatile int my\_need\_sched = 0;

void my\_process(void);

void \_\_init my\_start\_kernel(void)

{

int pid = 0;

int i;

/\* Initialize process 0\*/

task[pid].pid = pid;

task[pid].state = 0;/\* -1 unrunnable, 0 runnable, >0 stopped \*/

task[pid].task\_entry = task[pid].thread.ip = (unsigned long)my\_process;

task[pid].thread.sp = (unsigned long)&task[pid].stack[KERNEL\_STACK\_SIZE-1];

task[pid].next = &task[pid];

/\*fork more process \*/

for(i=1;i<MAX\_TASK\_NUM;i++)

{

memcpy(&task[i],&task[0],sizeof(tPCB));

task[i].pid = i;

task[i].state = -1;

task[i].thread.sp = (unsigned long)&task[i].stack[KERNEL\_STACK\_SIZE-1];

task[i].next = task[i-1].next;

task[i-1].next = &task[i];

}

/\* start process 0 by task[0] \*/

pid = 0;

my\_current\_task = &task[pid];

asm volatile(

"movl %1,%%esp\n\t" /\* set task[pid].thread.sp to esp \*/

"pushl %1\n\t" /\* push ebp \*/

"pushl %0\n\t" /\* push task[pid].thread.ip \*/

"ret\n\t" /\* pop task[pid].thread.ip to eip \*/

"popl %%ebp\n\t"

:

: "c" (task[pid].thread.ip),"d" (task[pid].thread.sp) /\* input c or d me

an %ecx/%edx\*/

);

}

void my\_process(void)

{

int i = 0;

while(1)

{

i++;

if(i%10000000 == 0)

{

printk(KERN\_NOTICE "this is process %d -\n",my\_current\_task->pid);

if(my\_need\_sched == 1)

{

my\_need\_sched = 0;

my\_schedule();

}

printk(KERN\_NOTICE "this is process %d +\n",my\_current\_task->pid);

}

}

}

对 myinterrupt.c 进行修改，主要是增加了进程切换的代码 my\_schedule(void)函数，在 Linux 内核源代码中对应的是 schedule(void)函数。

/\*

\* linux/mykernel/myinterrupt.c

\*

\* Kernel internal my\_timer\_handler

\*

\* Copyright (C) 2013 Mengning

\*

\*/

#include <linux/types.h>

#include <linux/string.h>

#include <linux/ctype.h>

#include <linux/tty.h>

#include <linux/vmalloc.h>

#include "mypcb.h"

extern tPCB task[MAX\_TASK\_NUM];

extern tPCB \* my\_current\_task;

extern volatile int my\_need\_sched;

volatile int time\_count = 0;

/\*

\* Called by timer interrupt.

\* it runs in the name of current running process,

\* so it use kernel stack of current running process

\*/

void my\_timer\_handler(void)

{

#if 1

if(time\_count%1000 == 0 && my\_need\_sched != 1)

{

printk(KERN\_NOTICE ">>>my\_timer\_handler here<<<\n");

my\_need\_sched = 1;

}

time\_count ++ ;

#endif

return;

}

void my\_schedule(void)

{

tPCB \* next;

tPCB \* prev;

if(my\_current\_task == NULL

|| my\_current\_task->next == NULL)

{

return;

}

printk(KERN\_NOTICE ">>>my\_schedule<<<\n");

/\* schedule \*/

next = my\_current\_task->next;

prev = my\_current\_task;

if(next->state == 0)/\* -1 unrunnable, 0 runnable, >0 stopped \*/

{

/\* switch to next process \*/

asm volatile(

"pushl %%ebp\n\t" /\* save ebp \*/

"movl %%esp,%0\n\t" /\* save esp \*/

"movl %2,%%esp\n\t" /\* restore esp \*/

"movl $1f,%1\n\t" /\* save eip \*/

"pushl %3\n\t"

"ret\n\t" /\* restore eip \*/

"1:\t" /\* next process start here \*/

"popl %%ebp\n\t"

: "=m" (prev->thread.sp),"=m" (prev->thread.ip)

: "m" (next->thread.sp),"m" (next->thread.ip)

);

my\_current\_task = next;

printk(KERN\_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);

}

else

{

next->state = 0;

my\_current\_task = next;

printk(KERN\_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);

/\* switch to new process \*/

asm volatile(

"pushl %%ebp\n\t" /\* save ebp \*/

"movl %%esp,%0\n\t" /\* save esp \*/

"movl %2,%%esp\n\t" /\* restore esp \*/

"movl %2,%%ebp\n\t" /\* restore ebp \*/

"movl $1f,%1\n\t" /\* save eip \*/

"pushl %3\n\t"

"ret\n\t" /\* restore eip \*/

: "=m" (prev->thread.sp),"=m" (prev->thread.ip)

: "m" (next->thread.sp),"m" (next->thread.ip)

);

}

return;

}

需要说明的是，上述 my\_schedule(void)函数的代码写得并不好，if 和 else 两块代码大 同小异，重复率很高。在新版本的代码中，我们彻底将两块代码统一起 来了，见如下代码。当然初始化的代码也做了一点修改，完整的新版代 码见 GitHub 版本库

mykernel项目github主页

<https://github.com/mengning/mykernel>。

void my\_schedule(void)

{

tPCB \* next;

tPCB \* prev;

if(my\_current\_task == NULL

|| my\_current\_task->next == NULL)

{

return;

}

printk(KERN\_NOTICE ">>>my\_schedule<<<\n");

/\* schedule \*/

next = my\_current\_task->next;

prev = my\_current\_task;

if(next->state == 0)/\* -1 unrunnable, 0 runnable, >0 stopped \*/

{

my\_current\_task = next;

printk(KERN\_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);

/\* switch to next process \*/

asm volatile(

"pushl %%ebp\n\t" /\* save ebp \*/

"movl %%esp,%0\n\t" /\* save esp \*/

"movl %2,%%esp\n\t" /\* restore esp \*/

"movl $1f,%1\n\t" /\* save eip \*/

"pushl %3\n\t"

"ret\n\t" /\* restore eip \*/

"1:\t" /\* next process start here \*/

"popl %%ebp\n\t"

: "=m" (prev->thread.sp),"=m" (prev->thread.ip)

: "m" (next->thread.sp),"m" (next->thread.ip)

);

}

return;

}

### 2.3.2 代码分析

对于以上文件中的数据类型定义等代码在此就不赘述了，唯一重要的进程初始化、切 换的几段汇编代码比较难理解，因此这里进行详细分析。

启动执行第一个进程的关键汇编代码。

这里需要注意的是%1 是值后面的“"d"(task[pid].thread.sp)”，%0 是指后面的“"c"(task [pid]. thread.ip)”，在内嵌汇编的部分有介绍过。

asm volatile(

"movl %1,%%esp\n\t" /\* 将进程原堆栈栈顶的地址（这里是初始化的值）存入ESP寄存器 \*/

"pushl %1\n\t" /\* 将当前EBP寄存器值入栈 \*/

"pushl %0\n\t" /\* 将当前进程的EIP（这里是初始化的值）入栈 \*/

"ret\n\t" /\* ret命令正好可以让入栈的进程EIP保存到EIP寄存器中 \*/

"popl %%ebp\n\t" /\* 这里永远不会被执行，只是与前面push指令结对出现，是一种编码习惯 \*/

:

: "c" (task[pid].thread.ip),"d" (task[pid].thread.sp)

);

第一个进程也就是进程 0 被初始化时，进程 0 的堆栈和相关寄存器的变化过程。

如图 2-5 所示，将 ESP 寄存器指向进程 0 的堆栈栈底，task[pid].thread.sp 初始值即为 进程 0 的堆栈栈底。



图2-5 将ESP寄存器指向进程0的堆栈栈底

如图 2-6 所示，将当前 EBP 寄存器的值入栈，因为是空栈，所以 ESP 与 EBP 相同。 这里简化起见，直接使用进程的堆栈栈顶的值 task[pid].thread.sp，相应的 ESP 寄存器指向 的位置也发生了变化。

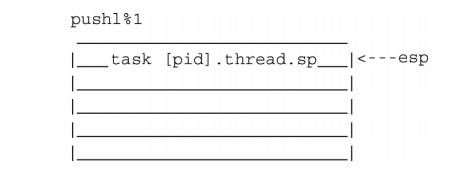


图2-6 将当前EBP寄存器的值入栈

如图 2-7 所示，将当前进程的 EIP（这里是初始化的值 my\_process(void)函数的位置） 入栈，相应的 ESP 寄存器指向的位置也发生了变化。

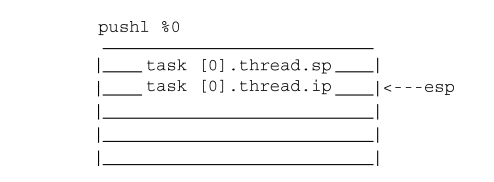


图2-7 将当前进程的EIP入栈

ret 指令将栈顶位置的 task[0].thread.ip，也就是 my\_process(void)函数的位置放入 EIP 寄存器中，相应的 ESP 寄存器指向的位置也发生了变化，如图 2-8 所示。

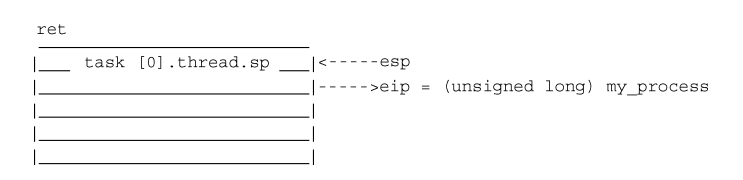


图2-8 将0号进程的起点地址放入EIP寄存器

接下来进程 0 启动，开始执行 my\_process(void)函数的代码。

进程调度代码如下：

if(next->state == 0) /\* next->state == 0对应进程next对应进程曾经执行过 \*/

{

/\* 进行进程调度关键代码 \*/

asm volatile(

"pushl %%ebp\n\t" /\* 保存当前EBP到堆栈中，如图2-9所示 \*/

"movl %%esp,%0\n\t" /\* 保存当前ESP到当前进程PCB中，如图2-10所示\*/

"movl %2,%%esp\n\t" /\* 将next进程的堆栈栈顶的值存到ESP寄存器，如图2-11和图2-12所示

\*/

"movl $1f,%1\n\t" /\* 保存当前进程的EIP值，下次恢复进程后将在标号1开始执行 \*/

"pushl %3\n\t" /\* 将next进程继续执行的代码位置（标号1）压栈，如图2-13所示\*/

"ret\n\t" /\* 出栈标号1到EIP寄存器，如图2-14所示\*/

"1:\t" /\* 标号1，即next进程开始执行的位置 \*/

"popl %%ebp\n\t" /\* 恢复EBP寄存器的值 \*/

: "=m" (prev->thread.sp),"=m" (prev->thread.ip)

: "m" (next->thread.sp),"m" (next->thread.ip)

);

my\_current\_task = next;

printk(KERN\_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);

}

else /\* next该进程第一次被执行 \*/

{

next->state = 0;

my\_current\_task = next;

printk(KERN\_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);

/\* switch to new process \*/

asm volatile(

"pushl %%ebp\n\t" /\* 保存当前进程EBP到堆栈 \*/

"movl %%esp,%0\n\t" /\* 保存当前进程ESP到PCB \*/

"movl %2,%%esp\n\t" /\* 载入next进程的栈顶地址到ESP寄存器 \*/

"movl %2,%%ebp\n\t" /\* 载入next进程的堆栈基地址到EBP寄存器 \*/

"movl $1f,%1\n\t" /\* 保存当前EIP寄存器值到PCB，这里$1f是指上面的标号1 \*/

"pushl %3\n\t" /\* 把即将执行的进程的代码入口地址入栈 \*/

"ret\n\t" /\* 出栈进程的代码入口地址到EIP寄存器 \*/

: "=m" (prev->thread.sp),"=m" (prev->thread.ip)

: "m" (next->thread.sp),"m" (next->thread.ip)

);

}

为了简便，假设系统只有两个进程，分别是进程 0 和进程 1。进程 0 由内核启动时初 始化执行，然后需要进程调度，开始执行进程 1。下面那从进程 1 被调度开始分析堆栈变 化，因为进程 1 从来没有被执行过，是第一次被调度执行，此时执行 else 中的代码。

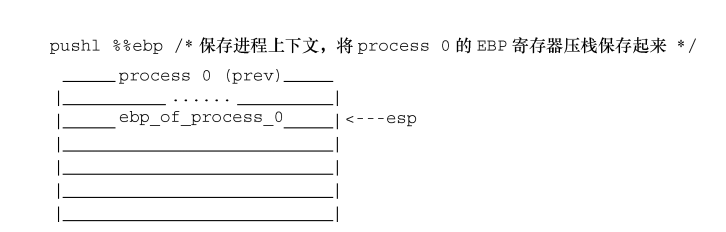


图2-9 保存当前进程EBP到堆栈

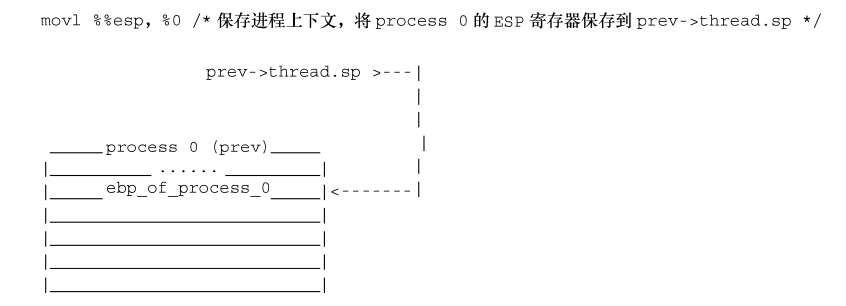


图2-10 保存当前进程ESP到PCB

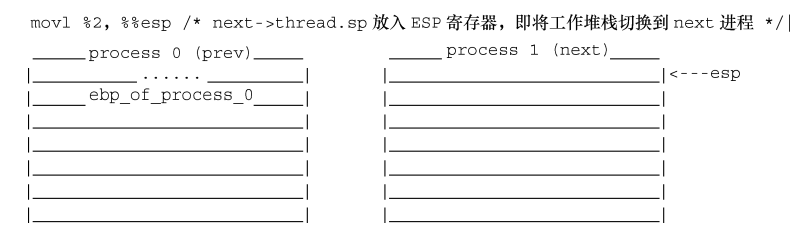


图2-11 载入next进程的栈顶地址到ESP寄存器

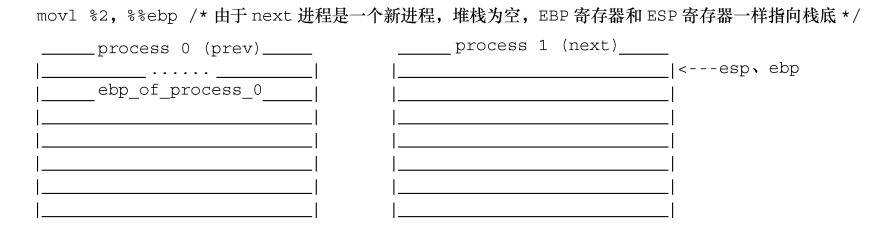


图2-12 载入next进程的堆栈基地址到EBP寄存器

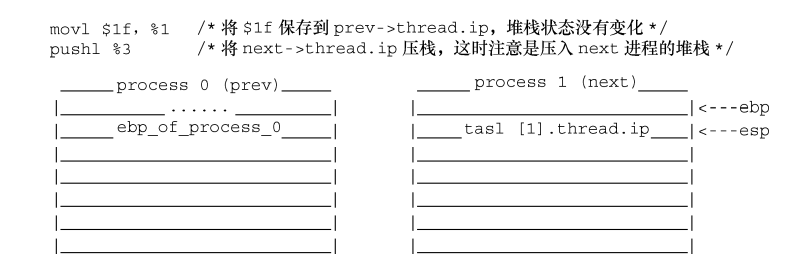


图2-13 把即将执行的进程的代码入口地址入栈

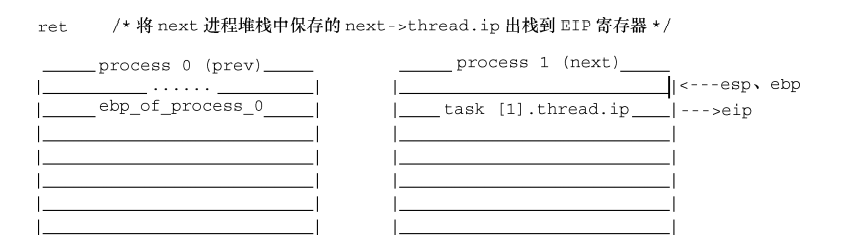


图2-14 出栈进程的代码入口地址到EIP寄存器

到这里开始执行进程 1 了，如果进程 1 执行的过程中发生了进程调度，进程 0 重新被调度 执行了，应该执行前述 if 中的代码，if 中内嵌汇编代码执行过程中堆栈的变化分析如下。

这时是从进程 1 再切换到进程 0，当前 prev 进程变成了进程 1，而 next 进程变成进程 0。第一句“pushl %%ebp\n\t”保存当前 EBP 到堆栈中，然后第二句保存 ESP 到进程 PCB 中，如图 2-15 和图 2-16 所示。

图 2-17 所示的 next->thread.ip 即为进程 0 上次被调度出去时保存的$1f，同理这里“movl $1f,%1\n\t”即保存$1f 到进程 1 的 thread.ip。下一句 ret 即出栈$1f 到 EIP 寄存器，$1f 的含 义为前方的标号 1（forwarding label 1），这时即开始从“1:\t”执行。

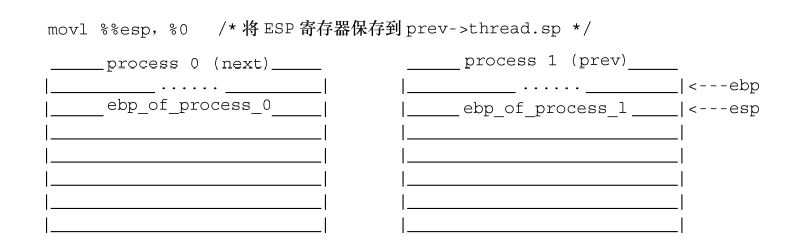


图2-15 保存当前EBP到堆栈中保存ESP到进程PCB中

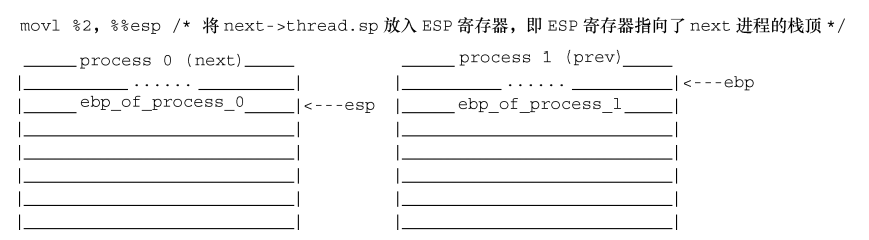


图2-16 恢复next进程堆栈栈顶地址到ESP寄存器中

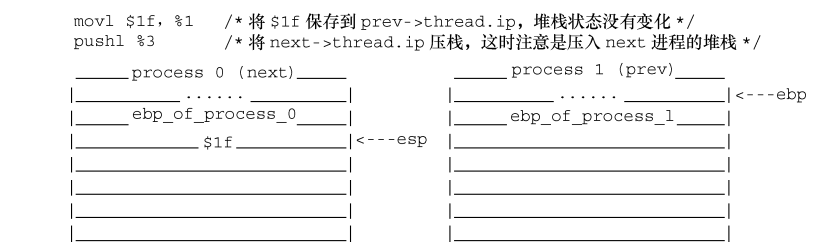


图2-17 把next->thread.ip地址入栈

到这里就恢复到了进程 0 的上下文环境继续执行进程 0，如图 2-18 所示。

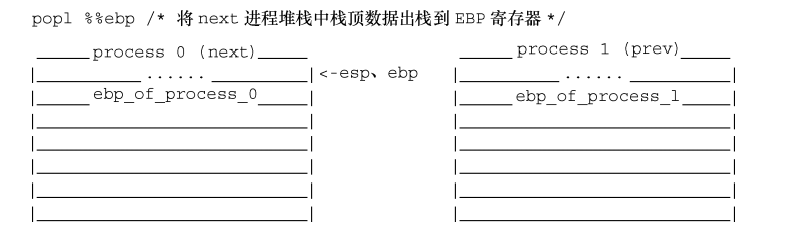


图2-18 恢复EBP寄存器的值

$1f 为前方的标号 1，if 中有标号 1，else 中没有标号 1。很多人会有疑问，else 中的$1f只是将其存入 prev->thread.ip，并没有使用$1f，但当进程被重新调度执行时，prev-> thread.ip 变成了 next->thread.ip，此时进入了 if 代码块中会 next->thread.ip 压栈，并由 ret 出栈到 EIP 寄存器中，这时才实际使用了$1f，因此将执行 if 代码块中的标号 1 处的代码，所以 else 中 没有标号 1 也就不奇怪了。

本章内容最重要的是进程的切换，进程在执行过程中，当时间片用完需要进行进程切 换时，需要先保存当前的进程执行的上下文环境，下次进程被调度时，需要恢复进程的上 下文环境。这样实现多道程序的并发执行。