

内核进程管理子系统

进程是什么

老师：“同学们，操作系统里的进程是什么？”
小白：“正在运行的程序”
老师：“那正在运行的程序与程序有哪些区别呢？”

4.1 概述

HguOS 是支持多进程的操作系统内核。这意味着，各个用户进程在运行过程中，彼此不能相互干扰，这样才能保证进程在主机中正常地运行。在某个 CPU 上，某一时刻，处理器只会执行一个任务，如图 4-1 所示。

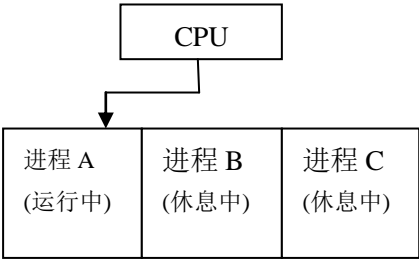


图 4-1 进程调度

程序是静态的、存储在文件系统上、尚未运行的指令代码；进程则是指正在运行的程序，即进行中的程序。一个进程的指令地址所组成的执行轨迹称为控制执行流。进程的执行流是独立的，互不干扰的。为了保证每个进程执行流的独立性，每个进程的运行必须获得运行所需要的各类资源，这些资源包括进程所使用的栈、一套自己的寄存器映像和内存资源等。操作系统为每个进程提供一个 PCB，Process Control Block，即进程控制块，记录、描述和管理程序执行的动态变化过程，它就是进程的身份证，用它来记录与进程相关的信息，比如进程状态、PID 等。另外，在中断到来或发生进程切换时，必须(也只需要)把进程的寄存器组信息完整地保存下来。在 Linux 内核中，通常由 task_struct 结构描述。

每个进程都有自己的 PCB，所有 PCB 放到一张表中维护，这就是进程表，如图 4-2 所示，PCB 就成为进程表中的“项”，因此 PCB 又可称为进程表项。另外进程 PCB 没有具体格式，其实际格式取决于操作系统的功能复杂度。

寄存器映像	寄存器映像	寄存器映像
栈	栈	栈	
PID	PID	PID	
进程状态	进程状态	进程状态	
进程名	进程名	进程名	
Linklist	Linklist	Linklist	
.....	

图 4-2 进程表

进程从执行到结束的整个过程中，并不是所有阶段都一直开足马力在处理器上运行，有时也会由于某些原因不得不停下来。为此，通常把进程“执行过程”中所经历的阶段按状态进行分类，如图 4-3 所示。进程有哪些状态，取决于操作系统对进程的管理办法，咱们可以创造一套自己的进程状态。

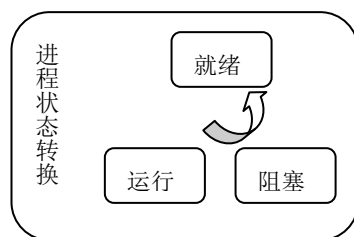


图 4-3 进程的状态变化

当一个进程在执行时，**CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文**。当内核需要切换(switch)至另一个进程时，它就需要保存当前进程的所有状态，即保存当前线程的上下文，以便在再次执行该进程时，能够恢复到切换的状态执行下去。进程调度过程如图 4-4 所示。

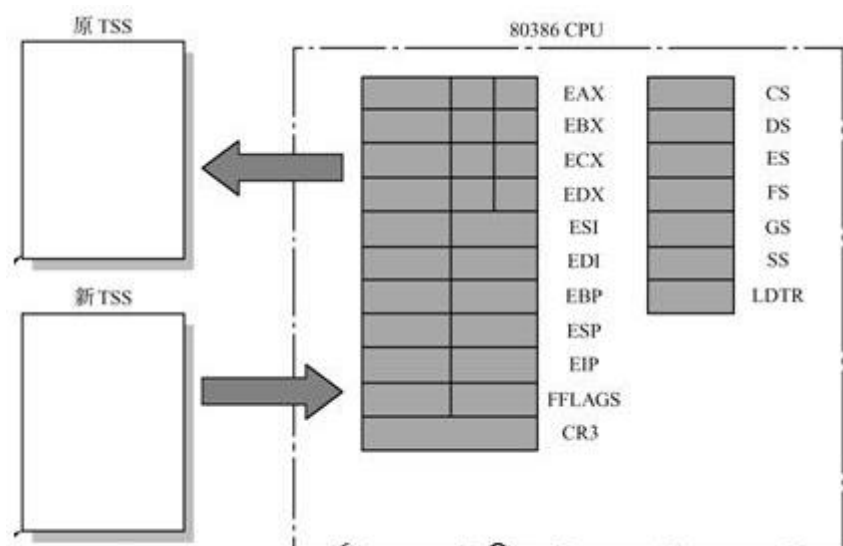


图 0-4 任务切换操作示意图

在本章，将实现进程管理子系统进程中的进程创建、切换及调度机制，共设置三个实验单元：
(1) 进程创建 (2)进程切换 (3) 进程调度机制。文件目录结构如表 4-1 所示。

目录结构

	文件名	状态	功能说明
进程创建	src/kernel/proc/process.c	添加	进程管理子系统的功能源文件
	include/sched.h	添加	进程管理子系统的头文件
进程切换	src/kernel/irq/idt.c	阅读	定义中断描述符表
	src/kernel/irq/do_irq.S	修改	中断机制
	src/kernel/irq/irq_handle.c	修改	中断处理程序
	src/kernel/proc/schedule.c	添加	切换进程函数
调度机制	src/kernel/proc/schedule.c	修改	调度函数
测试程序	src/test/test.c	添加	测试程序
	include/test.h	添加	测试程序的头文件

4.2 实验项目

4.2.1 进程创建

1. 实验目的

- (1) 理解进程的本质；
- (2) 理解进程控制块及其管理方法；
- (3) 掌握模拟器的基本使用方法和调试方法。

2. 实验准备

操作系统为每个进程管理一个 PCB，结构如图 4-5 所示。在 Linux 内核中，通常定义 task_struct 结构描述。其实，进程 PCB 没有具体格式，其实际格式取决于操作系统的功能复杂度。



图 4-5 PCB 结构

所谓**创建进程**，其实是初始化进程控制块各项。如**错误!未找到引用源。**所示。

一个进程开始之前，只要指定各段寄存器、eip、esp 和 eflags，就可以正常运行了，至于其他寄存器是用不到的，所以，在创建进程体时，必须初始化的寄存器列表为：cs、ds、es、fs、esp、eip 和 eflags。其中，cs、ds 等段寄存器对应的是 GDT 中的描述符，如图 4-6 所示。

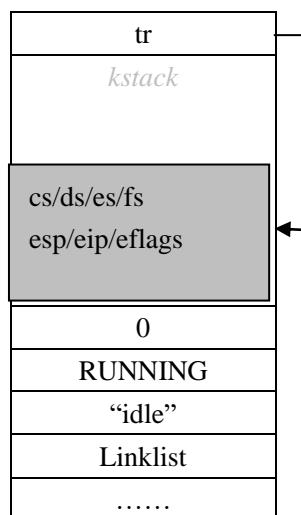


图 4-6 idle 进程初始化信息

每个进程都有自己的 PCB，系统通过进程表来管理所有进程。进程表的组织形式比较灵活。其中在 Linux 系统中，进程表定义成进程队列数组的形式，例如 struct task_struct *task[NR_TASKS]；规定系统可同时运行的最大进程数(见 kernel/sched.c)；每个进程占一个

数组元素。可以通过 `task[]` 数组遍历所有进程的 PCB。但 Linux 也提供一个宏定义 `for_each_task()`(见 `include/linux/sched.h`)，它通过 `next_task` 遍历所有进程的 PCB。

在 HguOS 源代码的 `include/adt/list.h` 文件中，采用了一种(数据结构)类型无关的双循环链表实现方式。其思想是将指针 `prev` 和 `next` 从具体的数据结构中提取处理构成一种通用的“双链表”数据结构 `list_head`，而 `list_head` 被作为一个成员嵌入到要拉链的数据结构(被称为宿主数据结构)中。这样，只需要一套通用的链表操作函数就可以将 `list_head` 成员作为“连接件”，把宿主数据结构链接起来，如图 0-7 所示。

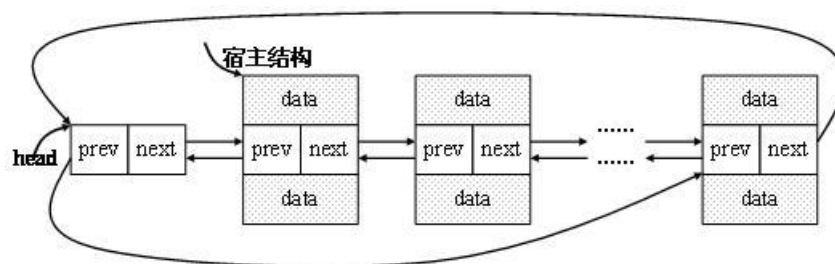


图 0-7 双向循环链表

3. 实验方案

(1) 数据结构

在本实验中，HguOS 进程控制块 PCB 主要包括以下几个部分：

- ①进程标识符，主要 `pid`；
- ②处理机状态信息：通用寄存器、指令计数器、程序状态字、用户栈指针；
- ③进程调度信息：进程状态、优先级等；
- ④进程控制信息：程序和数据地址、同步和通信机制、资源清单、链接指针等；

由于，本节实验是在内核空间，创建内核进程，实现基本的进程管理及切换，因此 PCB 不包含用户进程需要的资源记录，同时关于调度、通信、同步等信息会在后面实验中陆续添加。

代码 4-1 PCB 数据结构

```
#define KSTACKSIZE 1024

enum proc_state{UNUSED=0,RUNNABLE,RUNNING,BLOCKED,STOPED};
struct task_struct{
    TrapFrame *tf;
    char kstack[KSTACKSIZE];
    pid_t pid;
    char pname[20];
    enum proc_state state;
    ListHead *linklist;
};
```

其中，`TrapFrame` 结构用于定义系统寄存器信息，定义在 `include/x86.h` 中。

```
struct TrapFrame {
    uint32_t edi, esi, ebp, esp_;
    uint32_t ebx, edx, ecx, eax;    // Register saved by pushal
    uint32_t gs, fs, es, ds;        // Segment register
    int irq;                        // # of irq
};
```

```

uint32_t err, eip, cs, eflags;    // Execution state before trap
uint32_t esp, ss;                // Used only when returning to DPL=3
};
typedef struct TrapFrame TrapFrame;

```

为了管理系统中的所有进程控制块，系统还要维护如下全局变量见错误!未找到引用源。。

代码 4-2 与进程 PCB 相关的几个数据结构

```

#define NR_PROC 64
struct task_struct task[NR_PROC]; //进程表
extern struct task_struct *current; //当前进程
ListHead RunnableList; //就绪队列

```

说明 task_list是进程管理队列，可以根据系统需求创建多种队列，例如就绪队列、阻塞队列等等；

在这里，task[]是进程表，NR_TASKS 定义了最大允许进程；current 表示当前占用 CPU 且处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证原子操作；RunnableList 是用于管理就绪进程的双向循环列表结构，task_struct 中的成员变量 linklist 将链接入这个链表中(链表结构 LishHead 的定义在 include/adts/list.h 中)。

(2)算法设计

操作系统通过进程表来管理所有进程 PCB，task[]数组大小固定，因此系统中的进程是受限资源，即系统的最大进程数量是有限的。进程创建的本质就是为进程分配 PCB，并设置为就绪状态。

在 HguOS 操作系统中，为了实现进程管理需要进行如下操作：

- A. task_init:task 进程表初始化
- B. idle_init: 创建 0 号进程
- C. kthread_create:进程创建

下面根据系统的具体实现，给出参考算法。

算法 4-1 进程表初始化

```

void task_init(){
    for(i=0; i<NR_PROC; i++){
        task[i].state = UNUSED; //当前系统中没有进程
    }
}

```

算法 4-2 创建 0 号进程

```

idle_init(){
    //使用 task[0]，用于表示内核代码执行流代表的进程。
    task[0].state=RUNNING;
    p=&task[0];
    p->pid=nextpid++; //nextpid 为全局变量，初始化为 0，用于记录进程号
    memcpy(p->pname,"idle",5);
    current=p; //系统启动时，0 号进程为当前进程
}

```

算法 4-3 进程创建

A.1 p=alloc_proc(); //申请空白 PCB

A.2 init_pcb(p); //初始化 PCB 信息

A.2.1 初始化标识信息，将系统分配的标识符填入 PCB 中

A.2.2 初始化处理机状态信息，使程序计数器指向程序的入口地址，使栈指针指向栈顶等；

A.2.3 初始化处理机控制信息，将进程状态设置为就绪态；

A.3 将新进程插入就绪队列

A.4 算法结束，返回 PCB

算法 4-4 初始化 PCB

```
struct task_struct init_pcb(struct task_struct *p, void(*proc)(void),const char *name){
    p->pid =nextpid++; //定义进程号
    p->state=RUNNABLE; //设置进程状态
    tf=(TrapFrame*)(memset(p->kstack,0,KSTACKSIZE)+KSTACKSIZE)-1; //初始寄存器组
    p->tf=tf;
    tf->eflags=(uint32_t)(0x01<<9);
    tf->cs=(uint32_t)KSEL(SEG_KCODE);
    tf->ds=(uint32_t)KSEL(SEG_KDATA);
    tf->gs=(uint32_t)KSEL(SEG_KDATA);
    tf->eip=(uint32_t)func;
    tf->esp=(uint32_t)tf;
    tf->irq=0x03e8;
    memcpy(p->pname,name,7); //设置进程名字
    list_init(&p->linklist);
    return p;
}
```

【实验 2-1】完成进程表管理及进程创建工作。

4. 实验验证及分析

(1) 实验测试

目前，本节实验只完成了“进程的创建”，因此可以编写测试函数，遍历进程队列，查看进程号和进程名、进程状态等信息。

测试程序参考如下。

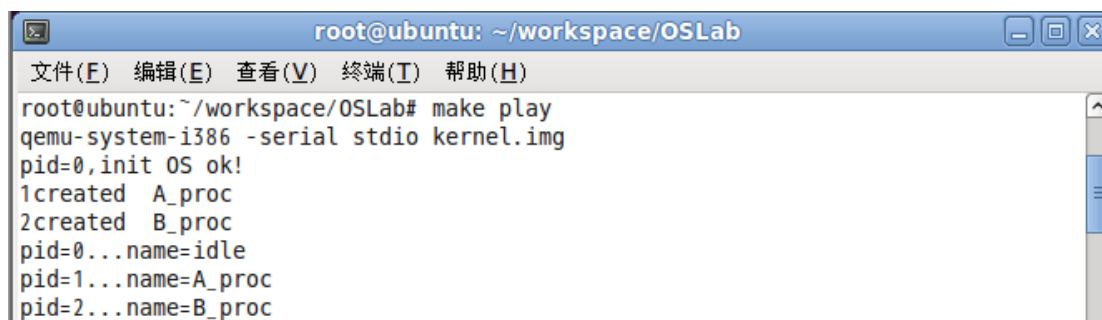
```
#include "sched.h"
#include "debug.h"
void
a(){
    while(1){
        printk("A");
        wait_intr();
    }
}
void
b(){
    while(1){
        printk("B");
        wait_intr();
    }
}
```

```

    }
}
void
c(){
    for(i=1; i<=10; i++){
        printk("C");
        wait_intr();
    }
}
void
test_proc(){
    create_kthread(a, "A_proc");
    create_kthread(c, "B_proc");
    create_kthread(b, "C_proc");
}

```

实验要求：完成进程创建操作，并进行基本测试，参考输出结果如**错误!未找到引用源。**所示。



```

root@ubuntu: ~/workspace/OSLab
文件(E) 编辑(E) 查看(V) 终端(T) 帮助(H)
root@ubuntu:~/workspace/OSLab# make play
qemu-system-i386 -serial stdio kernel.img
pid=0,init OS ok!
1created A_proc
2created B_proc
pid=0...name=idle
pid=1...name=A_proc
pid=2...name=B_proc

```

图 4-8

(2) 实验思考

执行测试程序观察，此时系统中哪个执行流占用处理器，新创建的进程能否被调度，为什么？

4.2.2 进程切换

1. 实验目的

- (1)了解系统中断机制；
- (2)理解进程切换本质；
- (3)掌握模拟器的基本使用方法和调试方法。

2. 实验准备

操作系统是由“中断驱动”的。中断是现代操作系统实现并行性的基础之一。引入中断机制，操作系统在让应用程序放弃控制权或从应用程序获得控制权将具有更大的灵活性。因此，操作系统中进程的切换也是以中断为基础的。当中断发生时，当前进程放弃处理器，新进程获得处理器开始执行。可以这样讲，进程上下文的切换都是由中断事件引起的。

为了完成本小节关于进程切换的操作，首先了解当前系统中的中断机制实现情况。

本节系统中涉及到的中断为“硬中断”，包括中断(又称外中断或异常中断)和异常(又称内中断或同步中断)。中断机制设置“中断描述符表”，以向量号为索引查找中断向量，然后

转入中断处理程序或异常处理程序。在保护模式下，系统采用中断描述符表(Interrupt Descriptor Table, IDT)，此表可以包含 256 个中断描述符，与中断或异常一一对应。描述符的作用是把程序控制权转给中断异常服务程序，每个描述符结构如图 所示，均占 8 个字节，通过它就能找到服务程序的起始地址、属性以及程序特权级别等。IDT 的位置由硬件中断描述符寄存器 IDTR 指定，它是一个 48 位的寄存器，高 32 位的 IDT 基址，低 16 位限定 IDT 的长度。如图 所示。中断调用过程如图所示。

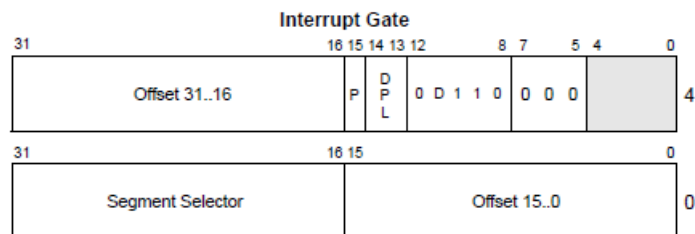


图 4-9 中断描述符格式

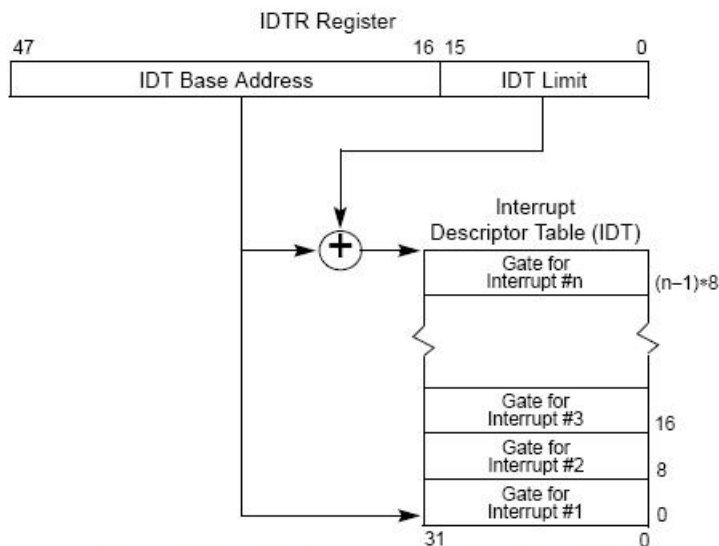


图 4-10 中断描述符表 IDT 和寄存器 IDTR

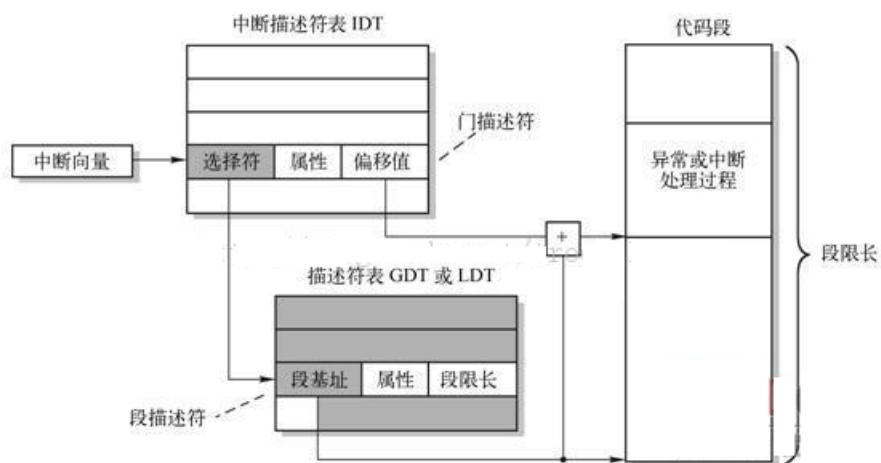


图 4-11 中断过程调用

思考题 3 时钟中断向量为 32, 请根据图描述当时钟中断发生时, 如何进行中断处理? 由此可见, “中断和异常是激活操作系统的仅有方法, 它暂停当前运行进程的执行, 把

处理器切换至和心态，内核获得处理器的控制权之后，如果需要就可以实现进程切换。“内核在处理中断事件或系统调用或者在处理时钟中断事件期间发现运行进程的时间片耗尽等，都可能引发内核实施进程上下文切换”。

假设，当中断发生时，进程让出处理器时，寄存器上下文将被保存到系统级上下文的相应的现场信息位置，这是内核就把这些信息压入核心栈；当内核处理完中断返回时，内核进行上下文切换，并从核心栈中弹出上下文。

思考 4：进程切换发生时机是什么时候？

3. 实验方案

(1) 中断机制的实现

在系统初始化阶段，创建 IDT。每个中断/异常都有其相应的处理程序，在使用中断之前，必须在 IDT 中注册以保证发生中断时能找到相应的中断处理程序。

以 32 号时钟中断为例。

首先，在 IDT 中定义时钟中断描述符，代码如下所示，构成如图 所示结构。

代码 4-3 节自 src/kernel/irq/idt.c

```
idt[32] = GATE(STS_IG32, KSEL(SEG_KCODE), irq0, DPL_KERN);
```

irq0 高 16 位	属性值
0x08	irq0 低 16 位

图 4-12 时钟中断描述符

定义 irq0 即中断处理程序，如所示。

代码 4-4 节自 src/kernel/do_irq.S

```
.globl irq0;
irq0:
    pushl $0; //错误码
    pushl $1000; //时钟中断号=1000
    jmp trap //

.extern irq_handle

trap:
    cli
    pushl %ds //入栈，段寄存器 ds、es、fs、gs
    pushl %es
    pushl %fs
    pushl %gs
    pushal //入栈，4 个通用寄存器(eax、ebx、ecx、edx)
           //2 个指针寄存器(esp、ebp)
           //2 个变址寄存器(esi、edi)

    movw $KSEL(SEG_KDATA), %ax //设置 ds、es
    movw %ax, %ds
    movw %ax, %es

    pushl %esp //栈顶指针寄存器入栈
```

```

call irq_handle    //中断处理程序
addl $4, %esp
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $8, %esp

iret    //恢复 cs、eip、eflags

```

中断发生前、转移到中断处理过程，中断处理完成后堆栈的情况如图 4-12 所示。在进行中断处理之前，栈指针如①esp；当发生中断(如时钟中断)，则依照图 4-11，找到中断处理程序入口(如 irq0)，此时系统自动压栈 eflags、cs、eip(这三者统称为 PSW，程序状态字)，栈顶指针如②；接下来，继续执行压栈操作，如③esp 指针，此时已经将程序当前的运行现场保存在栈中；转去执行中断处理程序(call irq_handle)；当中断处理程序执行完毕，再执行一系列的出栈操作，恢复程序运行的寄存器线程，而此时如④esp 指针所指位置。

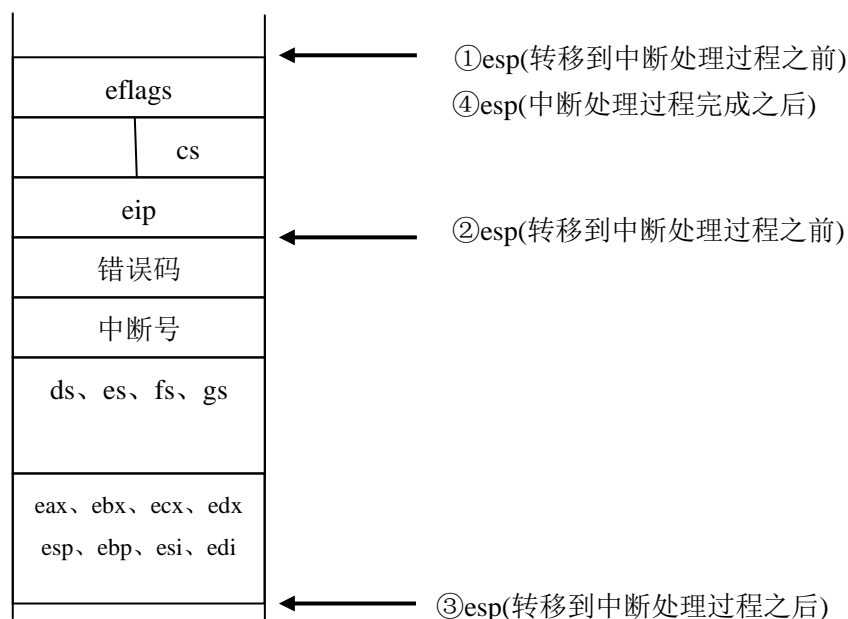


图 1 中断过程中栈的变化

思考题 4 中断发生可能会引起进程切换，那么进程切换的操作应该放在哪里？

【实验 2-2(1)】检测系统中是否开启中断？提示，修改 irq_handle 函数，使得中断发生时，打印信息(Interrupt...)。

(2) 进程切换

在实现进程切换之前，我们不妨来模拟一下进程切换的行为。当前进程正在运行时，堆栈上的内容是与进程相关的，而 current 指针指向了当前运行进程的 PCB。这个时候，中断发生了。中断处理程序会在当前的堆栈上保存 CS、EIP 和 EFLAGS 三个寄存器，并跳转到汇编代码执行。汇编代码如同预期的，把寄存器现场(TrapFrame)保存到堆栈上，并执行 pushl %esp 将栈顶指针保存下来，这下，我们进程的状态就和之前图中的状态一样了。

C 语言代码继续在堆栈上运行，可能会在堆栈中插入新的内容，但当前进程 PCB 的结构却不会发生任何变化。C 语言代码可能会执行一些如中断处理的工作，然后判断是否需要

执行进程切换。如果需要执行进程切换，C 语言代码只是把 `current` 指针切换：

```
current = next_process();
```

然后就从 C 语言代码返回了。返回后，汇编语言执行堆栈切换，然后照例恢复寄存器现场：

```
popal; popl %gs; popl %fs; popl %es; popl %ds
addl $8, %esp
iret
```

注意到此时的堆栈已经是新进程的堆栈了，在寄存器现场恢复完毕后，另一个线程就恢复执行了！

本实验理论指导实践，在对进程切换理论学习基础上，设计基本的进程切换实现方法，主要涉及到修改 `do_irq.S` 中的 `trap` 子过程，添加 `schedule.c` 中实现 `schedule()` 函数。其中，`trap` 子过程负责在需要进程切换时，处理器现场信息的保存(被中断进程)和恢复(切换后的进程)；`schedule()` 负责选择进程。

算法 4-5 进程切换(Switching of Process)

- A.1 保存被中断进程的处理器现场信息；
- A.2 修改被中断进程 PCB 的有关状态信息，如进程状态等；
- A.3 选择占用处理器的另一个进程；
- A.4 修改被选中进程 PCB 有关信息，如进程状态改为运行态；
- A.5 根据被选中进程的上下文信息来恢复处理器现场。

对于算法 4-5，在实验中可以设计 `trap` 和 `schedule()` 两个函数共同完成，其中 `schedule()` 负责 A.2-A.4 步骤，`trap` 负责 A.1 和 A.5。

算法 4-6 trap

- A.1 保存处理器现场信息，包括段寄存器、通用寄存器、指针寄存器和变址寄存器等等；
- A.2 `call irq_handle`，在中断处理程序 `irq_handle` 中，执行 `schedule()`，使得 `current` 指针指向选中另一个进程。

- A.3 切换内核栈，为恢复处理器现场做准备

```
movl (current), %esi
movl (%esi), %esp
```

- A.4 恢复处理器现场信息

算法 4-6 schedule()

- A.1 `Old_Proc=current;`
`Old_Proc->pstate=RUNNABLE;`
`add(&Runnable_list,&Old_Proc);`
- A.2 从就绪队列选择一个进程为 `New_Proc;`
`New_Proc->pstate=RUNNING;`
`del(New_proc);` //将 `New_Proc` 从就绪队列移除
- A.3 `current=New_Proc;`

本实验进程切换，由中断触发，执行函数调用关系：`trap`→`irq_handle()`→`trap()`

说明 1：`idle` 进程(即 0 号进程)是内核进程，只有在系统空闲时运行。进程切换时总是从 `Runnable_list` 中选择一个就绪进程作为 `current` 进程调度执行，当 `Runnable_list` 为空时才会再次调度 `idle` 进程。

说明 2：算法 4-6 给出的算法是针对进程可以无穷无尽运行的情况。若进程 P 执行一段时间就结束了，需要在结束时，做以下工作：`P->pstate=STOPPED;``P->pid=-1;`并且不在任何队列中存在。

【实验 2-2(2)】完成进程基于时钟中断的进程切换，注意“说明 1 和说明 2”中提到的问题

的解决。

4. 实验验证及分析

(1) 实现测试

根据本小节实验介绍，依次完成以下实验任务：

A. 测试项目 1

检测系统中是否开启中断？提示，修改 `irq_handle` 函数，使得中断发生时，打印信息 (Interrupt...)。参考输出结果如**错误!未找到引用源。**所示。

[illegible]

图 4-13

B. 测试项目 2

完成进程间切换。执行效果参考图。

[illegible]

图 4-14

(2) 实验思考

- 进程执行时使用哪些寄存器？`struct task_struct` 中设置了 `TrapFrame` 结构用于记录寄存器信息，为什么设置此结构？`TrapFrame` 结构的信息存放在内存的什么位置，为什么？
- 进程切换时指令 `pushl` 和 `iret` 指令各起什么作用？
- 如何理解“操作系统是由中断驱动的”这句话，请举例说明。
- 进程的 `struct task_struct` 结构与内核栈的作用，二者有什么联系？
- 处理机有哪些寄存器，在 `PCB` 中如何定义的？`src/kernel/irq/do_irq.S` 中的 `trap` 函数

执行时, 内核栈的变化, 为什么?

- f) 并发程序与顺序程序在调试技术上的不同?
- g) 例如, 顺序程序的顺序性、封闭性、确定性和可再现性标明程序及执行(计算)是一一对应的, 为程序的编址和调试带来很大的方便其缺点是计算机系统效率不高。
- h) task[0]是系统中哪个程序的进程? 请说明原因。

4.2.3 调度机制

1. 实验目的

- (1) 理解操作系统的调度管理机制。
- (2) 熟悉 mcore 的系统调度框架, 并实现基于优先级的调度算法。

2. 实验准备

进程调度是操作系统最为核心到部分, 执行十分频繁, 其调度策略的优劣降直接影响整个系统的性能, 因而, 这部分代码要求精心设计, 并常驻内存。

思考题 5 处理器调度算法有哪些?

内核中的调度程序用于选择系统中下一个要运行的进程。这种选择机制是多任务操作系统的基础。可以将调度程序看做在所有处于运行状态的进程之间分配 CPU 运行时间的管理代码。

例如在 Linux 系统中通过 `schedule()` 函数来完成调度操作, 为了能让进程有效地使用系统资源, 又能使进程有较快的响应时间, 就需要对进程的切换调度采用一定的调度策略。在 Linux 0.12 中采用了基于优先级排队的调度策略。`schedule()` 函数首先扫描 `task[]` 任务数组。通过比较每个就绪态任务的运行时间递减滴答计数 `counter` 的值来确定当前哪个进程运行的时间最少。哪一个的值大, 就表示运行时间还不长, 于是就选中该进程, 并使用任务切换宏函数切换到该进程运行。如果此时所有处于就绪态进程的时间片都已经用完, 系统就会根据每个进程的优先权值 `priority`, 对系统中所有进程(包括正在睡眠的进程)重新计算每个任务需要运行的时间片值 `counter`。计算的公式是:

$$counter = \frac{counter}{2} + priority \quad (\text{公式 1})$$

这样, 正在睡眠的进程被唤醒时就具有较高的时间片 `counter` 值。然后 `schedule()` 函数重新扫描任务数组中所有处于就绪态的进程, 并重复上述过程, 直到选择一个进程为止。

本节实验实现 2 部分, 其中“计时器”和“进程调度”。

- (1) 完成“计时器”部分实验, 为处理器调度提供基于时间的支持;
- (2) 完成“进程调度”部分内容, 参考 Linux 内核 0.12 版源码。

3. 实验方案

(1) 计时器的原理与实现

在操作系统中, 计时器是其中一个基础而重要的功能, 它提供了基于时间事件的各种机制。

我们一直在讲时钟中断, 那它到底是怎么触发的、何时触发呢?

中断当然不会凭空产生的。实际上, 它是由一个被称作 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, 下文统称为 8253, 如图 所示。

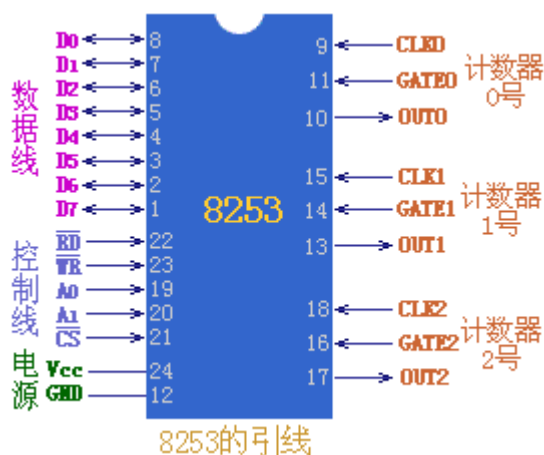


图 4-15 8253 计数器

8253 有 3 个计数器 (Counter)，它们都是 16 位的，各有不同的作用，如表所示。时钟中断实际上是由 8253 的 Counter0 产生的。

表 4-1 8253 计数器

计数器	作用
Counter0	输出到 IRQ0，以便每隔一段时间让系统产生一次时钟中断
Counter1	通常被设为 18，以便大约每 15 μ s 做一次 RAM 刷新
Counter2	连接 PC 喇叭

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，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是 $1193180/65536=18.2\text{Hz}$ ，即 1 秒钟发生 18.2 次中断。

为了为处理器调度机制提供时钟支持，借助系统的系统时钟中断来进行计数或计时。

(1) 定义数据结构

定义全局变量 ticks 来计数；

(2) 基本操作

本实验模拟系统时钟，每次时钟到来时，使得 ticks 加 1，来统计系统时钟触发次数，借此，为处理器调度提供时间上的辅助。

因此，对于计时器的基本操作包括：(1) 初始化系统计时器；(2) 定时器累加；(3) 获得系统时钟值；

算法 4-7 初始化系统计时器

```
init_timer()
[
    //ticks 初始化为 0;
]
```

算法 4-8 定时器累加

```
acc_timer()
[
    Ticks++;
]
```

算法 4-9 获得定时器值

```
get_timer()
[
    return ticks;
]
```

本实验中对操作系统时钟的实现非常简单,但是已经模拟了操作系统内核对于时钟操作的本质。接下来需要对系统计时器进行测试,可以模拟实现 `sleep()` 的功能,在这里称为 `timer_delay()` 来模拟实现让进程延迟睡眠功能。

算法 4-10 延迟函数

```
timer_delay(int value)
[
    //算法自行设计;
]
```

【实验 2-3(1)】测试系统时钟函数,自行设计测试函数。

(2) 调度机制

进程调度是操作系统的核心之一。进程调度的时机多种多样,就目前我们实现的系统功能,进程调度一般发生在进程被动放弃 CPU,当前进程的时间片用完,或一个进程被唤醒且其优先级高于当前进程的优先级等情况。

思考题 6: 目前 mcore 的调度时机是什么时候?

不论哪一种进程调度算法,都按照以下三个步骤完成进程的调度管理。

①**处理当前进程:** 根据调度切换原因,将当前进程加入到就绪或阻塞队列;

②**选择进程运行:** 扫描就绪队列中的所有就绪进程,从中选择合适的进程运行,将其设为 `current`,并从就绪队列中移除该进程。

③**进程切换:** 进程上下文切换。

在本实验中,可以选取实现“时间片轮转”调度算法或“动态优先级”调度算法。二者选其一即可。

这里重点分析“动态优先级”调度算法,至于“时间片轮转”调度算法,请同学们自行补课,设计实现。

参考 Linux0.12 设计动态优先级调度算法: 优先级高的就绪进程先运行,优先级低的首就绪进程后运行,优先级相同的进程按轮转方式运行。

下面的结构用来描述基于优先级的进程轮转调度算法在 `mcore` 中的实现方案。

在 `task_struct` 结构中,存放于进程调度相关的成员供调度模块使用。

```
int nice; //Lab 2.3 +进程可控优先因子
```

```
int counter; //Lab 2.3 +进程目前时间片配额,也称进程动态优先级
```

注释

①就绪队列中的普通进程都有 `counter` 值,每次时钟中断时,其值由 `update_process_times()` 函数减一。`Counter` 值等于 0 时,表示进程的时间片耗尽,此时要释放处理器。

②当就绪队列中所有进程的 `counter` 值变为 0 后,表明一轮调度已经结束。此时,系统要重新计算进程的 `counter` 值,这既包括就绪进程,也包括等待态进程。计算公式可参考:

$$p \rightarrow counter = (p \rightarrow counter \gg 1) + NICE_TO_TICKS(p \rightarrow nice); \text{(公式 4-2)}$$

即进程的 `counter` 值右移一位,加上此进程的 `nice` 值,得到新的 `counter` 值。对于就绪进程而言,因其 `counter` 值都为 0,计算结果就是由 `nice` 值转换而来的时钟滴答数。等待进程则未然,其 `counter` 值都不为 0,计算结束后,等待态进程的动态优先级会大于 `nice` 值。因此,等待态进程会有较高的优先级,处于等待态越久的进程,其动态优先级越高。

算法 4-11 调度算法

```

schedule()
[
    A.1 处理当前进程;
    A.2 选择新进程
    A.2.1 初始化 c=0, next(c 记录权值, next 记录进程)
        遍历就绪队列
        [
            若 c<(next 的权值)
            [
                更新 c; 更新 next;
            ]
        ]
    A.2.2 若 c==0 则
        按照 (公式 1) 更新就绪队列和等待队列中进程的 counter 值;
        转入 A.2.1;
    A.2.3 若 c!=0
        [将 next 所指进程设置为 current;
        current->state=RUNNING;
        list_del(&current);
        ]
]

```

【实验 2-3(1)】基于优先级的调度算法实现。具体实施方案如下面描述。

4. 实验验证及分析

(1) 实验实现

实现【实验 2-3(1)】，自行设计测试算法，比较分析运行结果。

实现【实验 2-3(2)】，采用 4.2.2 的测试算法。运行结果如图所示。

测试程序采用第 4.2 节的测试案例。执行结果参考如图 4-16。其中，a()、b()、c() 进程的 counter 和 nice 值都初始设置为 15。

```

root@ubuntu: ~/workspace/OSLab
文件(E) 编辑(E) 查看(V) 终端(T) 帮助(H)
root@ubuntu:~/workspace/OSLab# make play
qemu-system-i386 -serial stdio kernel.img
Athread:1
Bthread:2
Cthread:3
CCCCBBBBBBBBBBBBAAAAAAAAAAAAACCCBBBBBBBBBBBBBBBBAAAAAAAAAAAAACCCBBBBBBB
BBBBBBAAAAAAAAAAAAACCCBBBBBBBBBBBBBBBBAAAAAAAAAAAAACCCBBBBBBBBBBBBBBB
AAAAAAAAACCCBBBBBBBBBBBBBBBBAAAAAAAAAAAAACCCBBBBBBBBBBBBBBBBAAAAAAAAAAC
BBBBBBBBBBBBBBBBAAAAAAAAAAAAACCCBBBBBBBBBBBBBBBBAAAAAAAAAAAAAC

```

图 4-16 进程调度运行结果参考

(2) 实验思考

如何灵活应用链表等数据结构管理进程调度?

4.3 实验思考与拓展

4.3.1 实验思考

思考题 1: 进程创建的本质是什么?

思考题 2: idle 进程在操作系统中的作用是什么?

思考题 3: 链表结构在操作系统中有哪些应用?

思考题 4: 进程切换或调度的时机是什么, 包括哪几个步骤?

4.3.2. 实验拓展

请根据你在做实验过程中遇到的问题及你对实验的认识, 提出对该实验的疑惑或改进措施。可以从以下几方面提出问题:

(1) 实验中哪些设计使你感觉不合理?

(2) 实验中哪些地方使你理解不透彻?

4.3.3 开篇故事版本更新

请给出新版本的“老师和小白”关于引导程序的故事。