

《操作系统设计与实践》第八周实验

多进程与进程调度



姓名：刘竞优 2021302181057

赵伯俣 2021302181156

钱嘉乐 2021312181234

钱一铭 2021302181115

指导教师： 严飞 副教授

钱一铭 2021302181115

2023年12月3日

**目录**

[1 引言 4](#_Toc152524574)

[1.1 实验目的 4](#_Toc152524575)

[1.2 实验内容 4](#_Toc152524576)

[1.3 实验环境 4](#_Toc152524577)

[2 多进程问题 5](#_Toc152524578)

[2.1 相关宏定义 5](#_Toc152524579)

[2.2 初始化LDT 6](#_Toc152524580)

[2.3 时钟中断程序 7](#_Toc152524581)

[2.4 进程切换 8](#_Toc152524582)

[2.5 添加一个进程体 9](#_Toc152524583)

[2.6 拓展中断支持多进程 10](#_Toc152524584)

[3 扩展实现多进程应该考虑的问题 14](#_Toc152524585)

[4 初始化多进程控制块的过程流程图 15](#_Toc152524586)

[5 扩展初始化LDT和TSS流程图 15](#_Toc152524587)

[6 修改时钟中断来支持多进程管理 16](#_Toc152524588)

[7 系统调用的基本框架、基本功能，系统调用的流程图 17](#_Toc152524589)

[7.1 基本框架 17](#_Toc152524590)

[7.2 基本功能 18](#_Toc152524591)

[7.3 流程图 19](#_Toc152524592)

[7.4 运行结果 19](#_Toc152524593)

[8 操控可编程计数器 20](#_Toc152524594)

[8.1 理论知识 20](#_Toc152524595)

[8.2 实际操作 22](#_Toc152524596)

[8.3 运行结果 23](#_Toc152524597)

[9 进程调度的框架，实现优先级调度 24](#_Toc152524598)

[9.1 进程调度 24](#_Toc152524599)

[9.2 优先级调度 25](#_Toc152524600)

[10 实现多级反馈队列调度算法 29](#_Toc152524601)

[11 从用户态进程读和写内核段的数据 37](#_Toc152524602)

[12 贡献与体会 40](#_Toc152524603)

[12.1 赵伯俣 40](#_Toc152524604)

[12.2 刘竞优 41](#_Toc152524605)

[12.3 钱嘉乐 42](#_Toc152524606)

[12.3 钱一铭 43](#_Toc152524607)

[13 教师评语 44](#_Toc152524608)

# 1 引言

## 实验目的

1. 多进程的实现机理与进程调度
2. 对应章节：第六章6.4、6.5、6.6

## 实验内容

1. 多进程问题，如何扩展单进程到多进程，如何扩展中断支持多进程？

2. 如何实现系统调用

3. 进程调度问题，弄清楚实现调度的基本思路

## 1.3 实验环境

1. VMware Workstation 17 Pro

2. Ubuntu-16.04.6-desktop-i386

3. Bochs 2.7

# 2 多进程问题

## 2.1 相关宏定义

上周的操作系统实验中已实现单进程，这次通过tasktab数组实现多进程。在初始化的时候，只要用一个for循环依次读取每一项，然后

填充到相应的进程表项中即可。

Proc.h中声明的一个数组类型定义如下，分别为函数地址、存放进程栈大小及定义名字。

typedef **struct** s\_task {

    task\_f  initial\_eip;

**int** stacksize;

**char**    name[32];

}TASK;

定义函数指针task\_f：

typedef **void** (\*task\_f) ();

NR\_TASKS表示任务的数量；STACK\_SIZE\_TESTA和STACK\_SIZE\_TESTB是分别表示任务A和任务B的堆栈大小的宏。它们被定义为0x8000，这意味着每个任务的堆栈大小为32KB。

STACK\_SIZE\_TOTAL表示任务堆栈的总大小。它使用了STACK\_SIZE\_TESTA和STACK\_SIZE\_TESTB的值进行计算，即将两个任务的堆栈大小相加。

task\_f是一个函数指针类型，定义了一个指向无返回值、无参数的函数的指针。这个函数指针可以用来指向任务函数，以便在任务调度时执行相应的任务代码。

*/\* Number of tasks \*/*

#define NR\_TASKS    2

*/\* stacks of tasks \*/*

#define STACK\_SIZE\_TESTA    0x8000

#define STACK\_SIZE\_TESTB    0x8000

#define STACK\_SIZE\_TOTAL    (STACK\_SIZE\_TESTA + \

                STACK\_SIZE\_TESTB)def **void** (\*task\_f) ();

## 2.2 初始化LDT

每一个进程都会在GDT中对应一个LDT描述符。于是在for循环中，我们将每个进程表项中的成员p\_proc->ldt\_sel赋值。

下列代码循环遍历了进程表（proc\_table），对每个进程都执行以下操作：

1. 获取当前进程的指针，即p\_proc。
2. 计算当前进程的LDT选择子（selector\_ldt），通过将索引值（INDEX\_LDT\_FIRST）左移3位得到选择子的值。选择子是GDT中的一个索引，通过左移3位可以将索引转换为选择子的格式。
3. 使用init\_descriptor函数初始化GDT中与LDT相关的描述符。该函数接受四个参数：要初始化的描述符、LDT的物理地址、LDT的大小（以字节为单位），以及描述符的属性。在这里，属性被设置为DA\_LDT，表示这是一个LDT的描述符。
4. 增加进程指针的值，以便处理下一个进程。
5. 增加LDT选择子的值，以便在GDT中为下一个进程的LDT分配描述符。

*// 填充 GDT 中进程的 LDT 的描述符*

**int** i;

    PROCESS\* p\_proc = proc\_table;

    u16 selector\_ldt = INDEX\_LDT\_FIRST << 3;

    for(i=0;i<NR\_TASKS;i++){

        init\_descriptor(&gdt[selector\_ldt>>3],

                vir2phys(seg2phys(SELECTOR\_KERNEL\_DS),

                    proc\_table[i].ldts),

                LDT\_SIZE \* sizeof(DESCRIPTOR) - 1,

                DA\_LDT);

        p\_proc++;

        selector\_ldt += 1 << 3;

    }

## 2.3 时钟中断程序

该函数实现打印“#”

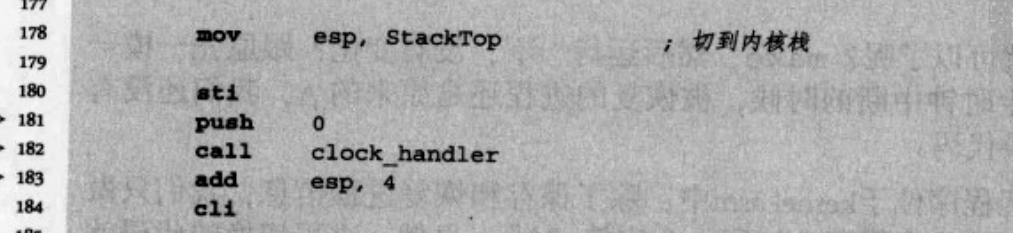
PUBLIC **void** clock\_handler(**int** irq)

{

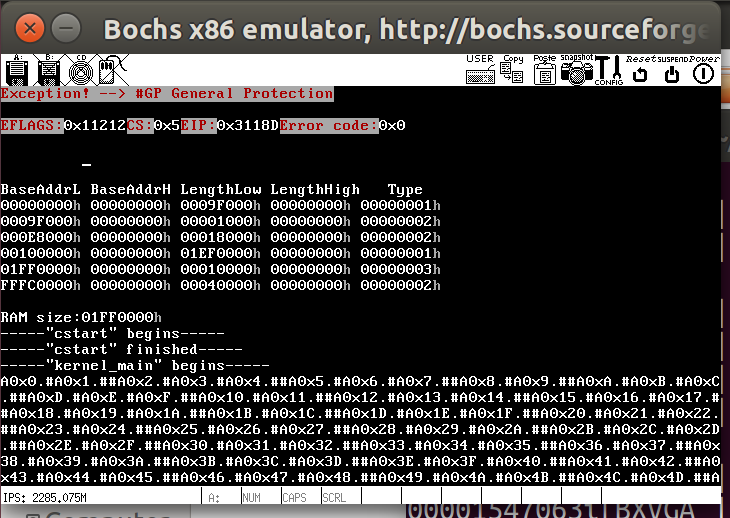
    disp\_str("#");

}

中断处理程序



运行结果如下



## 2.4 进程切换

每一次让p\_proc\_ready指向进程表中的下一个表项，如果切换前已经到达进程表结尾则回到第一个表项。最关键的这几行添加完后就可以看到结果了

PUBLIC **void** clock\_handler(**int** irq)

{

    disp\_str("#");

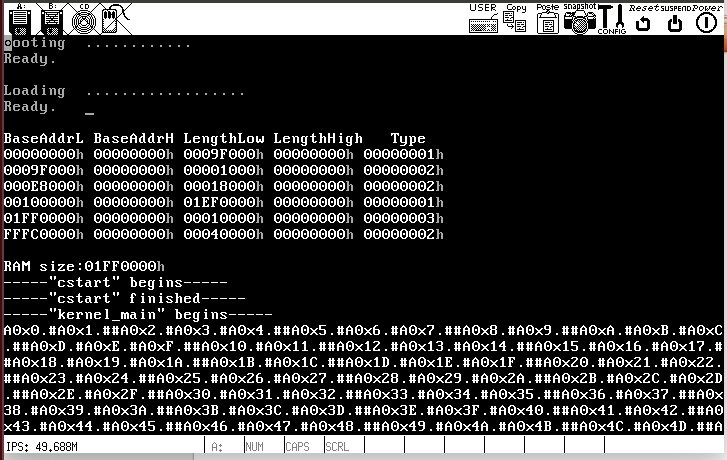
    p\_proc\_ready++;

    if (p\_proc\_ready >= proc\_table + NR\_TASKS)

        p\_proc\_ready = proc\_table;

}

运行结果如下



## 2.5 添加一个进程体

TestC函数会以循环的方式，在屏幕上不断显示字符"C"和递增的整数，并在每次显示后延迟1毫秒。

**void** TestC()

{

**int** i = 0x2000;

    while(1){

        disp\_str("C");

        disp\_int(i++);

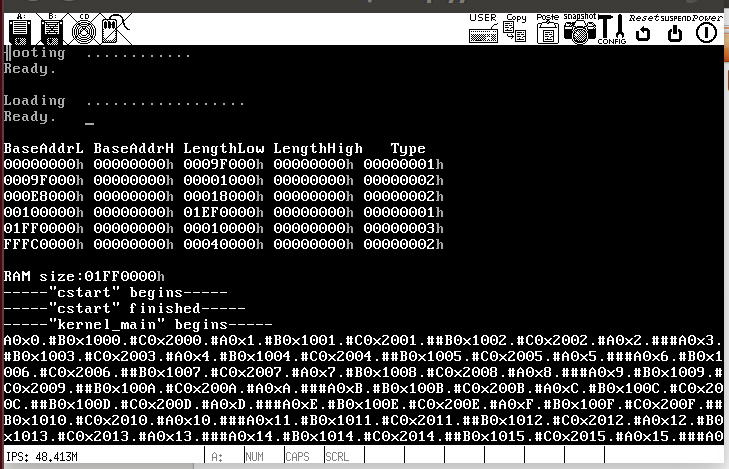
        disp\_str(".");

        delay(1);

    }

}

运行结果如下



## 2.6 拓展中断支持多进程

下列代码为宏定义，用于处理硬件中断的中断向量

* 首先，调用save函数保存寄存器的状态。
* 然后，读取主8259中断控制器的屏蔽寄存器的值，并将当前中断号对应的屏蔽位设置为1，屏蔽当前中断。
* 接着，更新主8259中断控制器的屏蔽寄存器的值，将结束中断的命令码加载到命令端口，以通知中断处理的结束。
* 打开中断，允许响应新的中断。
* 将中断号压入栈中，并调用中断处理程序。
* 恢复栈顶，关闭中断，禁止其他中断的发生。
* 读取主8259中断控制器的屏蔽寄存器的值，并将当前中断号对应的屏蔽位恢复为0，接受当前中断。
* 更新主8259中断控制器的屏蔽寄存器的值。
* 最后，从中断例程返回。

; 中断和异常 -- 硬件中断

; ---------------------------------

%macro  hwint\_master    1

    call    save

    in  al, INT\_M\_CTLMASK   ; `.

    or  al, (1 << %1)       ;  | 屏蔽当前中断

    out INT\_M\_CTLMASK, al   ; /

    mov al, EOI         ; `. 置EOI位

    out INT\_M\_CTL, al       ; /

    sti ; CPU在响应中断的过程中会自动关中断，这句之后就允许响应新的中断

    push    %1          ; `.

    call    [irq\_table + 4 \* %1]    ;  | 中断处理程序

    pop ecx         ; /

    cli

    in  al, INT\_M\_CTLMASK   ; `.

    and al, ~(1 << %1)      ;  | 恢复接受当前中断

    out INT\_M\_CTLMASK, al   ; /

    ret

%endmacro

同时，新引入函数指针数组：irq\_table

PUBLIC irq\_handler irq\_table[NR\_IRQ];

初始化irg table：这项工作分为两部分，首先将所有的元素初始化为spurious\_irq,然后再处理需要单独赋值的元素

PUBLIC **void** init\_8259A()

{

    out\_byte(INT\_M\_CTL, 0x11); *// Master 8259, ICW1.*

    out\_byte(INT\_S\_CTL, 0x11); *// Slave  8259, ICW1.*

    out\_byte(INT\_M\_CTLMASK, INT\_VECTOR\_IRQ0); *// Master 8259, ICW2. 设置 '主8259' 的中断入口地址为 0x20.*

    out\_byte(INT\_S\_CTLMASK, INT\_VECTOR\_IRQ8); *// Slave  8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28*

    out\_byte(INT\_M\_CTLMASK, 0x4); *// Master 8259, ICW3. IR2 对应 '从8259'.*

    out\_byte(INT\_S\_CTLMASK, 0x2); *// Slave  8259, ICW3. 对应 '主8259' 的 IR2.*

    out\_byte(INT\_M\_CTLMASK, 0x1); *// Master 8259, ICW4.*

    out\_byte(INT\_S\_CTLMASK, 0x1); *// Slave  8259, ICW4.*

    out\_byte(INT\_M\_CTLMASK, 0xFF); *// Master 8259, OCW1.*

    out\_byte(INT\_S\_CTLMASK, 0xFF); *// Slave  8259, OCW1.*

**int** i;

    for (i = 0; i < NR\_IRQ; i++)

        irq\_table[i] = spurious\_irq;

}

使用以下函数处理时钟中断程序

PUBLIC **int** kernel\_main()

{

    disp\_str("-----\"kernel\_main\" begins-----\n");

    TASK\*       p\_task      = task\_table;

    PROCESS\*    p\_proc      = proc\_table;

**char**\*       p\_task\_stack    = task\_stack + STACK\_SIZE\_TOTAL;

    u16     selector\_ldt    = SELECTOR\_LDT\_FIRST;

**int** i;

    for (i = 0; i < NR\_TASKS; i++) {

        strcpy(p\_proc->p\_name, p\_task->name); *// name of the process*

        p\_proc->pid = i; *// pid*

        p\_proc->ldt\_sel = selector\_ldt;

        memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS >> 3],

               sizeof(DESCRIPTOR));

        p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK << 5;

        memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3],

               sizeof(DESCRIPTOR));

        p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK << 5;

        p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK)

            | SA\_TIL | RPL\_TASK;

        p\_proc->regs.ds = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

            | SA\_TIL | RPL\_TASK;

        p\_proc->regs.es = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

            | SA\_TIL | RPL\_TASK;

        p\_proc->regs.fs = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

            | SA\_TIL | RPL\_TASK;

        p\_proc->regs.ss = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

            | SA\_TIL | RPL\_TASK;

        p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK)

            | RPL\_TASK;

        p\_proc->regs.eip = (u32)p\_task->initial\_eip;

        p\_proc->regs.esp = (u32)p\_task\_stack;

        p\_proc->regs.eflags = 0x1202; */\* IF=1, IOPL=1 \*/*

        p\_task\_stack -= p\_task->stacksize;

        p\_proc++;

        p\_task++;

        selector\_ldt += 1 << 3;

    }

    k\_reenter = 0;

    p\_proc\_ready    = proc\_table;

        put\_irq\_handler(CLOCK\_IRQ, clock\_handler); */\* 设定时钟中断处理程序 \*/*

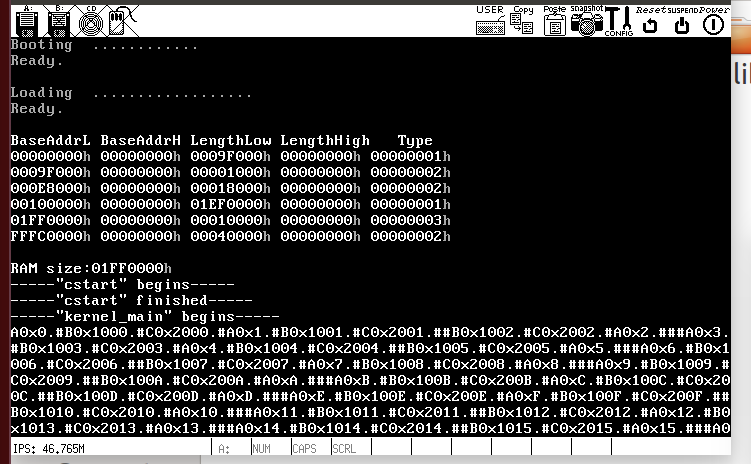
        enable\_irq(CLOCK\_IRQ); */\* 让8259A可以接收时钟中断 \*/*

    restart();

    while(1){}

}

运行结果如下

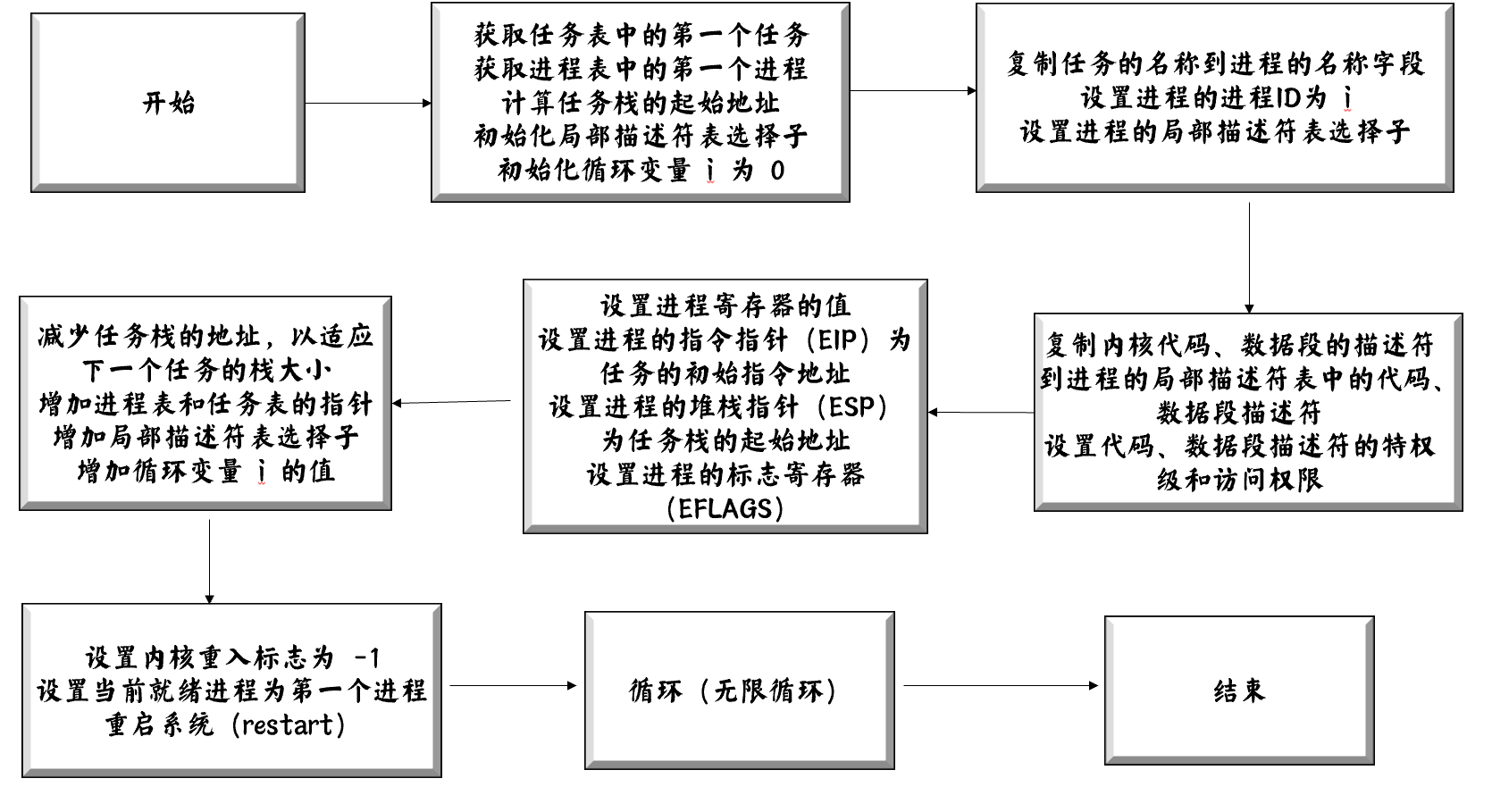


# 3 扩展实现多进程应该考虑的问题

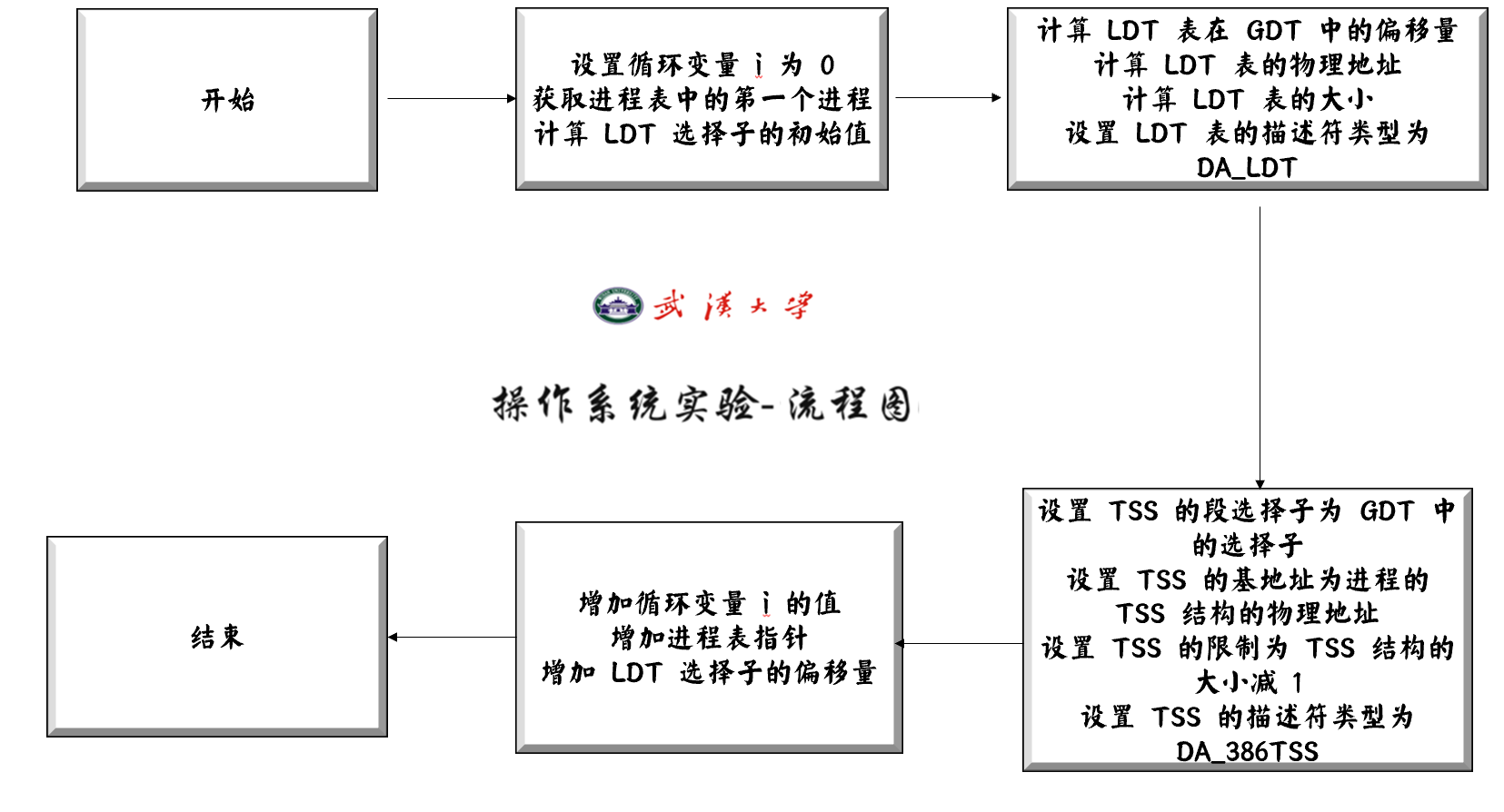
主要应考虑的问题如下：

1. 进程管理：引入进程管理机制来创建、调度、终止和管理多个进程。这包括设计进程控制块（PCB）数据结构来表示进程的状态、标识符、资源分配和进程间通信等信息。进程管理器需要提供接口和算法来管理进程的创建、销毁和切换。
2. 进程调度：设计调度算法来决定哪个进程在何时执行。常见的调度算法包括先来先服务（FCFS）、最短作业优先（SJF）、轮转法（Round Robin）等。调度器需要根据进程的优先级、调度策略和系统负载等因素，选择合适的进程进行执行。
3. 进程同步和互斥：确保多个进程之间的正确协作和资源共享。这包括设计和实现各种同步和互斥机制，如信号量、互斥锁、条件变量等，以防止竞态条件和数据不一致问题的发生。进程间的同步和互斥机制可以通过共享内存、消息传递或管道等方式实现。
4. 内存管理：为每个进程分配独立的内存空间，并进行内存保护和地址转换等操作。需要设计内存管理器来管理进程的地址空间，包括分配和回收内存、页表管理、虚拟内存管理等。如果采用虚拟内存技术，还需要设计页替换算法和页面置换策略。
5. 文件系统：设计文件系统来支持多个进程对文件的读写操作。需要考虑文件共享、文件权限和并发访问等问题。文件系统应提供适当的接口和机制，支持进程间的文件共享和保护，同时处理并发访问请求，如读写锁或文件锁。

# 4 **初始化多进程控制块的过程流程图**



# 5 扩展初始化LDT和TSS流程图

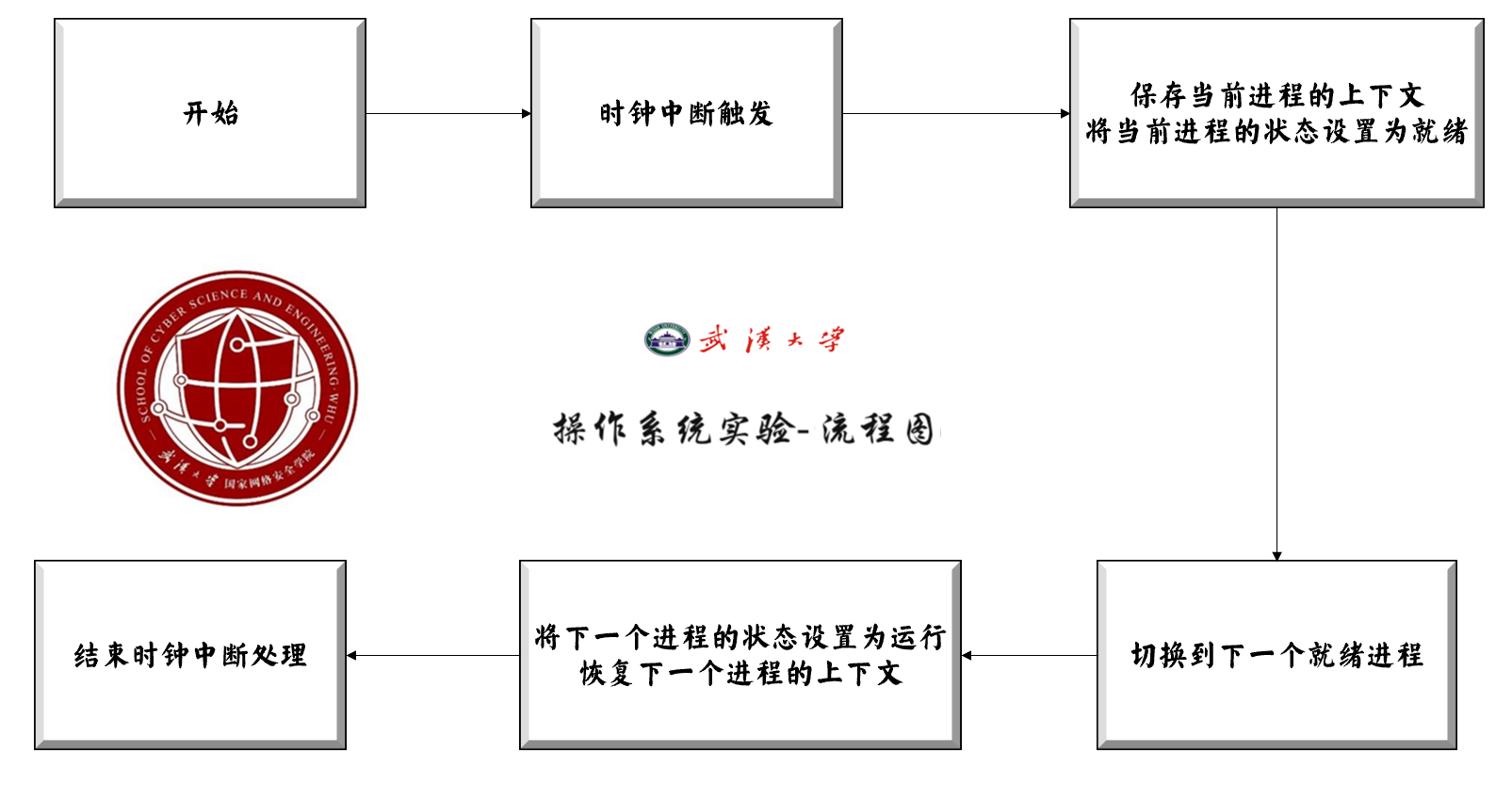


# 6 修改时钟中断来支持多进程管理

每当时钟中断触发时，操作系统会保存当前进程的状态，并选择下一个就绪进程来运行。这样，多个进程可以交替执行，实现多进程管理。

1. 在操作系统初始化时，设置时钟中断处理函数。这可以通过设置中断向量表中时钟中断的入口地址来实现。
2. 在时钟中断处理函数中，保存当前进程的上下文。这包括保存当前进程的寄存器状态、堆栈指针等信息。这些信息将用于在切换回当前进程时恢复其执行状态。
3. 将当前进程的状态设置为就绪状态，以便调度程序可以选择下一个要运行的进程。
4. 调用调度程序来选择下一个就绪进程。调度程序可以使用不同的调度算法，如轮转调度、优先级调度等，根据其策略选择下一个进程。
5. 将下一个进程的状态设置为运行状态。
6. 恢复下一个进程的上下文。这包括恢复其寄存器状态、堆栈指针等信息。
7. 结束时钟中断处理，返回到被切换进程的执行点。

流程图如下：



# 7 系统调用的基本框架、基本功能，系统调用的流程图

## 7.1 基本框架

下面以教材中给出的进程A调用系统调用得到当前时钟中断次数的例子来说明系统调用的基本框架。

（1）首先需要在进程的代码块中执行系统调用函数get\_ticks,该函数的代码保存在专门用来保存系统调用函数的文件文件syscall.asm中，该函数的功能是将系统调用的问题参数\_NR\_get\_ticks保存在eax中，然后执行中断号为INT\_VECTOR\_SYS\_CALL的中断。

（2）然后跳转到中断门中，该中断号的中断门与系统调用处理函数sys\_call相对应，跳转到该系统调用处理函数中。

（3）在sys\_call中首先调用save函数进行现场的保护以及跳转到内核栈中，然后根据传入的系统调用的问题参数\_NR\_get\_ticks调用系统调用表中对应的处理程序sys\_get\_ticks。

（4）最后在处理函数sys\_get\_ticks中执行系统调用返回全局变量ticks的值。完成一次系统调用。

## 7.2 基本功能

(1)系统调用get\_ticks。该函数的功能是在进程中的代码调用该函数之后，将希望操作系统执行的操作编码即\_NR\_get\_ticks保存到eax寄存器中，以便于在系统调用处理时能够知道需要进行怎样的处理，然后再调用中断号为\_INT\_VECTOR\_SYS\_CALL的中断，该中断号只需选用与原来的中断号不同即可。

（2）中断门。中断门的作用是在系统调用进行中断请求后调用产生中断的向量号所对应的中断处理程序sys\_call

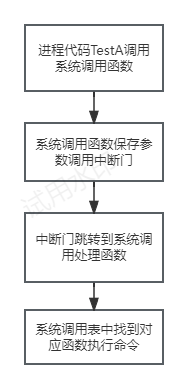
（3）save函数。save函数的作用是保存当前正在运行的进程的寄存器的值和栈帧的值以及切换到内核栈中，便于sys\_call函数进行系统调用处理操作。

（4）sys\_call系统调用处理函数。该函数的功能是根据在调用系统调用时在eax中保存的值在系统调用表中找到对应偏移的位置即sys\_get\_ticks，确定需要执行何种处理程序对系统调用进行处理。

（5）sys\_get\_ticks函数执行系统调用对应的指令将发生的时钟中断数进行输出操作。

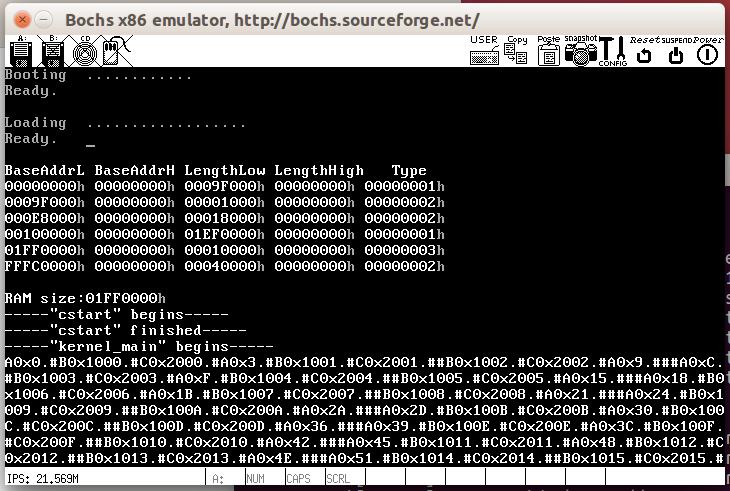
## 7.3 流程图

系统调用过程的流程图如下图所示



## 7.4 运行结果

在将系统调用的所有内容代码编写完毕后运行得到的结果如下图所示



通过观察运行的结果我们可以看出，由于只有A进程中调用了系统调用，所以每一个A进程都会输出一次当前已经产生的时钟中断的次数，又由于每一次时钟中断都会产生一个新的进程，所以每打印一个#都代表产生了一次时钟中断。

第一个A打印了0，说明在这个进程之前并没有时钟中断产生由于；第二个A打印了3，说明在此之前发生了三次时钟中断打印出了三个#；第三个A打印了9说明在此之前产生了9次时钟中断打印出了9个#。并且由于第一个A是进程调用产生而不是时钟中断产生，所以并不会打印出#。

# **8 操控可编程计数器**

## 8.1 理论知识

（1）8253计数器。

在操作系统中，使用芯片8253进行计数操作，在8253中存在有3个16位计数器，三个计数器的功能如下图所示

|  |  |
| --- | --- |
| 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 | 设置为18，大约每15微秒进行一次RAM的刷新 |
| Counter2 | 与PC喇叭进行连接 |

（2）计数器工作原理。

在计数器中会存在有一个初始值和输入频率，每一个输入频率周期则对保存的初始值进行一次减一操作，当初始值减为0时会产生一次时钟中断。因此如果想要改变时钟中断的产生频率需要改变计数器中保存的初始值的大小，因为输入频率由硬件提供不能更改。

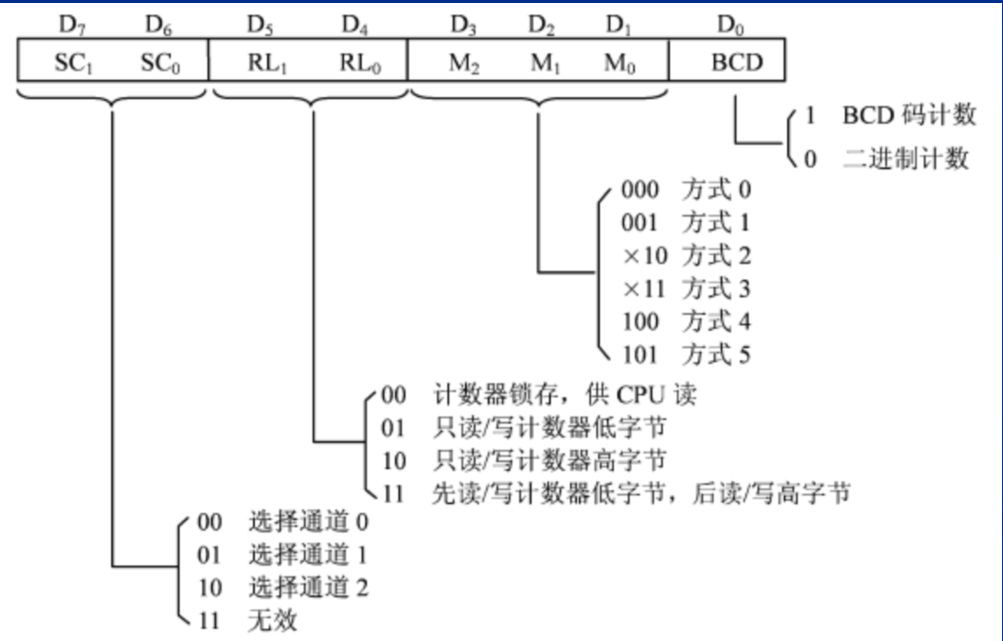
（3）8253端口。

修改计数器的初始值通过8253提供的端口进行操作，8253中的各个端口的情况如下表所示

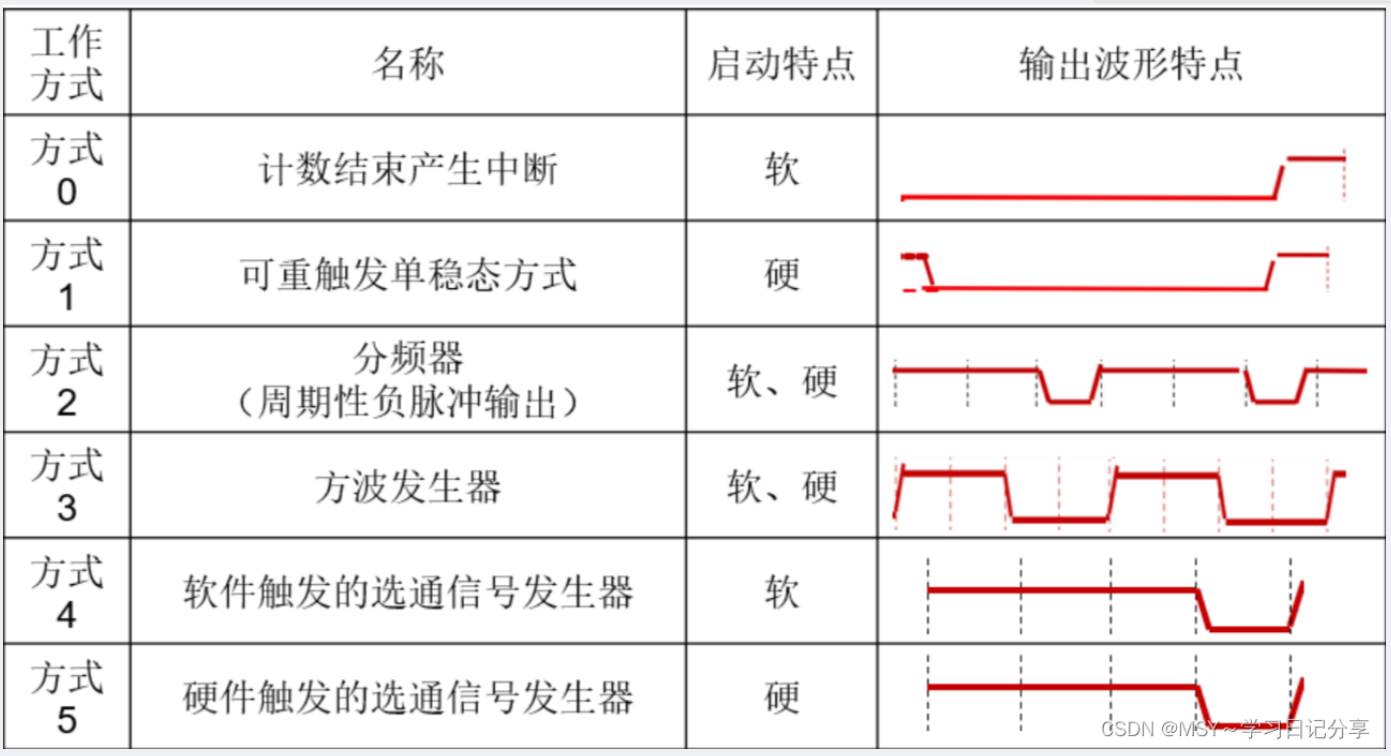
|  |  |
| --- | --- |
| 端口 | 描述 |
| 40h | 8253 Counter0 |
| 41h | 8253 Counter1 |
| 42h | 8253 Counter2 |
| 43h | 8253模式控制寄存器 |

（4）通过端口修改初始值

想要修改计数器的初始值就需要先通过43h端口写入控制字然后再将初始值的大小通过对应的端口进行传送。其中43h端口需要写入的控制字的格式如下图所示。



其中5种不同的工作方式如下图所示



## 8.2 实际操作

（1）根据需求选择控制字。

在本次实验中需要操作的计数器是Counter0,因此需要选用通道0，所以控制字的7、6位应为00；计数值大小为16位，所以低字节的内容和高字节的内容都要写入，所以控制字的第5、4位应为11；在本次实验中需要采用与方式2相同的工作方式和波形，所以控制字的第3、2、1位应为010；在传输初始值时使用2进制表示，所以控制字的第0位应为0。由此便可以得到控制字为00110100B=34H。

（2）计算计数值。

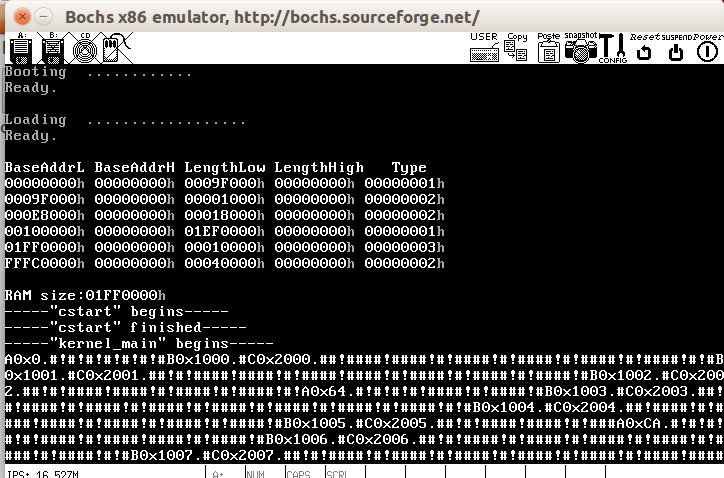
在本次实验中PC上计数器的输入频率为1193180Hz，想要产生的时钟中断为每10ms产生一次时钟中断，由此可得计数器输出的频率应为1s/10ms=100Hz,可以得到计数器的初始计数值应为1193180/100=11931=2E9B。

（3）编程控制计数值。

通过调用out\_byte函数将对应的内容传送到对应的端口中去，首先将控制字直接传送到8253的43号端口中，然后将初始计数值的低八位9B传入40h号端口，然后将初始计数值的高八位传入40号端口，从而完成对可编程计数器的控制。

## 8.3 运行结果

将修改完的代码运行后得到的结果如下图所示



通过观察结果可以发现，相比于之前，打印结果中的!数量明显增多，字母的数量减少，产生的原因是我们修改了计数器的值，使得时钟中断产生的频率更快，从而导致发生了非常多次的中断重入。

# 9 进程调度的框架，实现优先级调度

## 9.1 进程调度

Linux进程调度的基本框架包括进程队列、调度策略、调度器、进程状态、时间片、优先级、上下文切换和调度器的触发时机等功能模块，各个模块的功能如下：

（1）进程队列：Linux内核维护了多个进程队列，其中包括就绪队列、运行队列和等待队列。进程被放置在这些队列中等待执行。

（2）调度策略：Linux支持多种调度策略，包括完全公平调度、实时调度、以及按优先级调度等。不同的策略适用于不同的应用场景。

（3）调度器：调度器是Linux内核的一部分，它负责根据选定的调度策略选择下一个要执行的进程。常见的调度器包括CFS调度器、实时调度器等。

（4）进程状态：每个进程都有一个状态，包括就绪、运行、等待等状态。调度器根据进程的状态来确定下一个要执行的进程。

（5）时间片：CFS调度器使用时间片的概念，它为每个进程分配一个时间片，然后按照时间片大小来轮流执行进程。

（6）优先级：在优先级调度中，每个进程都有一个优先级值，调度器根据优先级来选择下一个执行的进程。

（7）上下文切换：当调度器决定切换到另一个进程时，会进行上下文切换，保存当前进程的状态，并加载下一个进程的状态。

（8）调度器的触发时机：调度器可以在多种情况下触发，包括进程的时间片用尽、进程进入等待状态、新进程创建等情况。

## 9.2 优先级调度

在Linux中，优先级调度是通过动态改变进程的优先级值来实现的。Linux使用一个抽象的优先级值来决定进程在多任务环境下的调度顺序。较高的优先级值表示较高的优先级，较低的优先级值表示较低的优先级。下面是Linux中实现优先级调度的关键步骤：

（1）优先级值：每个进程都有一个优先级值，通常取值范围在-20到19之间，其中-20表示最高优先级，19表示最低优先级，0表示默认优先级。负数表示实时进程，正数表示普通进程。

（2）动态调整优先级：Linux中的进程调度器会根据进程的行为和状态来动态调整其优先级值。这意味着进程在运行过程中的表现会影响其将来的调度优先级。

（3）基于进程行为的调整：Linux根据进程的行为和资源利用情况来调整优先级值。例如，一个长时间占用CPU资源的进程可能会被降低优先级，以便其他等待执行的进程有机会获得CPU时间。

（4）基于进程状态的调整：进程状态的改变也可以影响优先级。例如，当进程进入等待状态时，它的优先级可能会提高，以鼓励它更早地执行。

（5）进程创建时的默认优先级：新创建的进程通常会继承其父进程的优先级，但也可以通过系统调用来指定初始优先级。

（6）实时进程：实时进程通常具有较高的优先级，它们的调度不受动态调整的影响，确保它们在规定的时间内得到执行。

总的来说，Linux中的优先级调度是通过动态调整进程的优先级值来实现的，以便根据进程的行为、状态和类型来确定下一个执行的进程。这种灵活性使得Linux能够在多任务环境中有效地管理进程，以满足不同应用的需求。

书中例程代码实现了基本的非抢占式优先级调度功能，例程代码利用前文实现的系统调用，对进程表中每个进程赋予一个ticks属性，ticks属性用于表示当前进程所剩余的时间片，具体实现则是每次时钟中断时，将当前处于运行态的进程的ticks属性自减1，接着判断运行态的进程的ticks属性是否为0，当该进程的ticks属性等于0时，表示该进程的分配的时间片使用完毕，应该调用进程调度程序对进程表中的其他进程进行调度。

PUBLIC void clock\_handler(int irq)

{

ticks++;

p\_proc\_ready->ticks--;

if (k\_reenter != 0) {

return;

}

if (p\_proc\_ready->ticks > 0) {

return;

}

schedule();

}

进程调度程序则需要在进程表中查找出ticks属性值最高的进程，即优先级最高的进程，并将该进程状态更改为运行态，这里为了方便观察进程的调度情况，简单处理，使用进程需要执行的时间作为优先级，并且在所有进程的ticks属性等于0时，将他们ticks赋值为初始值，继续进行进程调度，即让进程表中的三个进程永远执行下去。

PUBLIC void schedule()

{

PROCESS\* p;

int greatest\_ticks = 0;

while (!greatest\_ticks) {

for (p = proc\_table; p < proc\_table+NR\_TASKS; p++) {

if (p->ticks > greatest\_ticks) {

greatest\_ticks = p->ticks;

p\_proc\_ready = p;

}

}

if (!greatest\_ticks) {

for (p = proc\_table; p < proc\_table+NR\_TASKS; p++) {

p->ticks = p->priority;

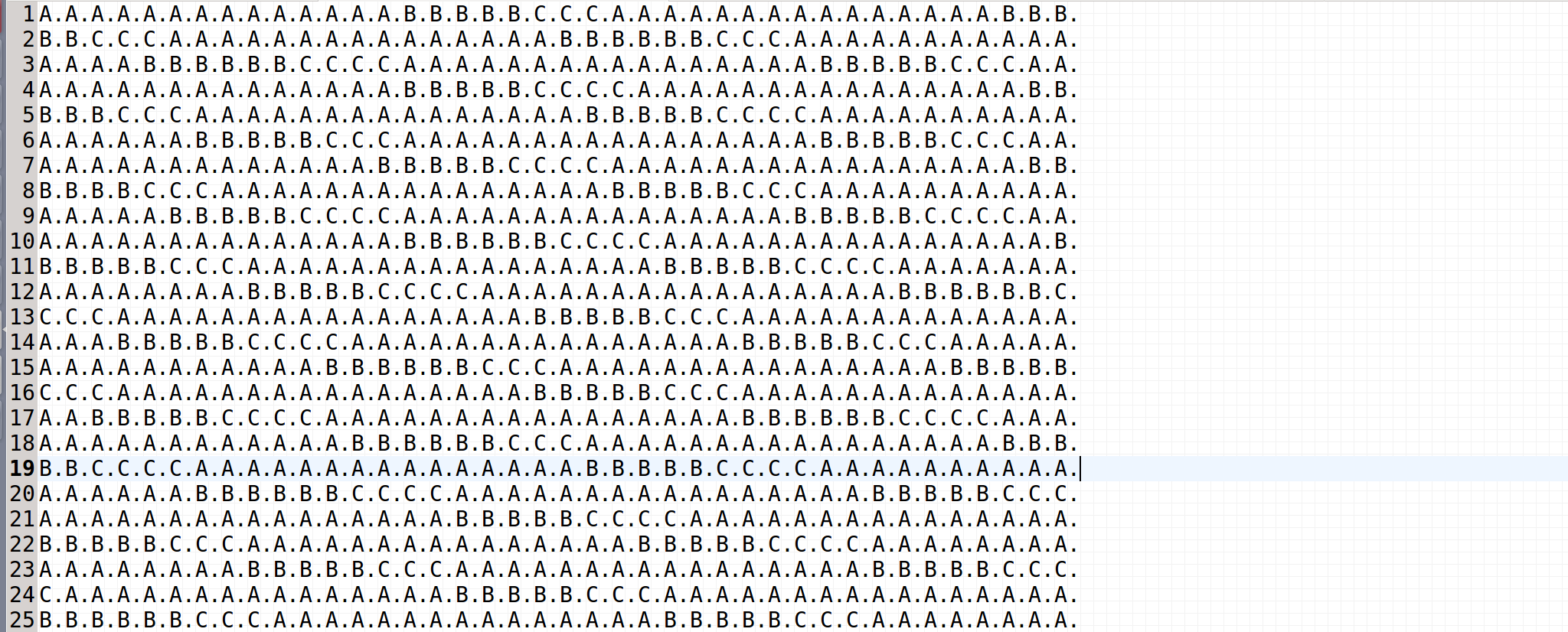
}

}

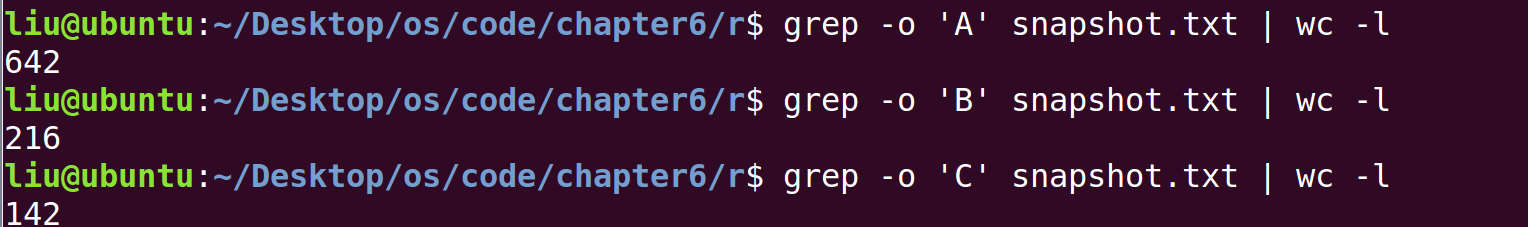
}

}

使用bochs提供的屏幕捕捉工具，将进程输出内容输出到snapshot.txt中，使用工具计算每个进程输出的字母个数，即每个进程执行的时钟周期数。



这里将文本末的8个字符‘A’去掉，则字符A的数量为634，这样便满足三个进程完成数个完整调用，三种字符的比值为634: 216: 142=16.05: 5.4: 3.55，近似于进程表中设置的各个进程的ticks属性值的比值15: 5: 3，说明实现的基本的优先级调度是有效的。



# 10 实现多级反馈队列调度算法

在本部分中，我们将要动手实现一个多级反馈队列的调度算法，其需要满足 符合立即抢占，即如果有一个新的优先级高的进程加入进来，当前正在运行的优先级较低的进程会被立即剥夺执行权限进入休眠状态，被剥夺的进程仍然放到当前队列中；

本次实验也有一定的局限性：实验中所有用到的进程都是我们实现创建好的，我们无法在运行时动态地随机创建一个进程，所以抢占与剥夺的效果在本实验中体现得并不明显。

本实验的整体思路如下：

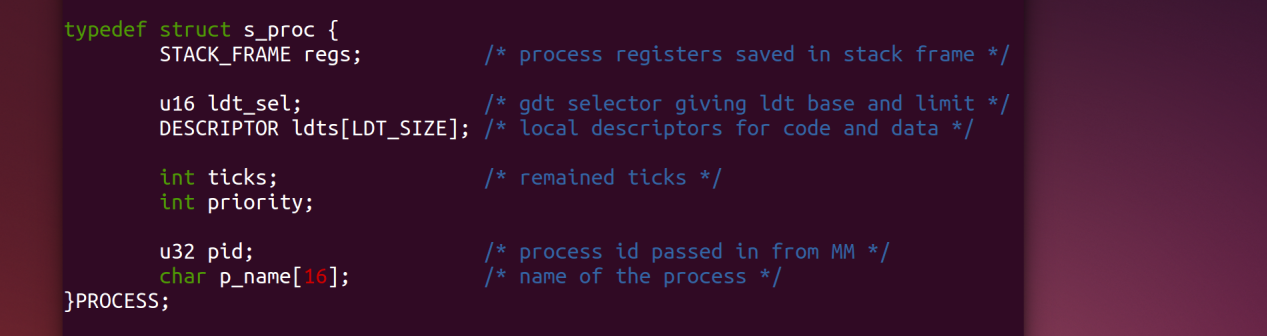
仍然使用上次实验中基于时钟中断的定时器，每次定时器一到时间，和教材中给出的代码一样，遍历proc\_table找出最高优先级的进程并赋予执行权限；

对于多级反馈队列的时间问题，我们规定一个基本的时间单位T = n次时钟中断，然后各级队列的时间片长度分别为T、2T、4T ……

所以我们先实现相关的数据结构：

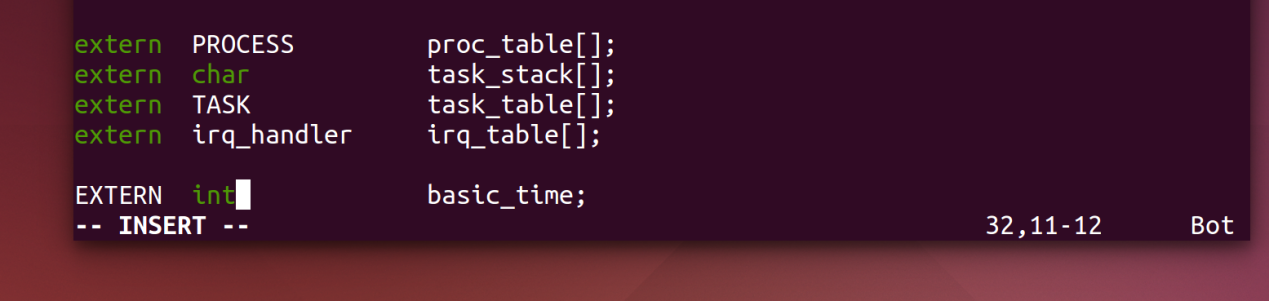
1. 用一个变量记录该进程属于哪一个优先级队列，然后再用一个变量来记录该进程在当前队列中的剩余时间；

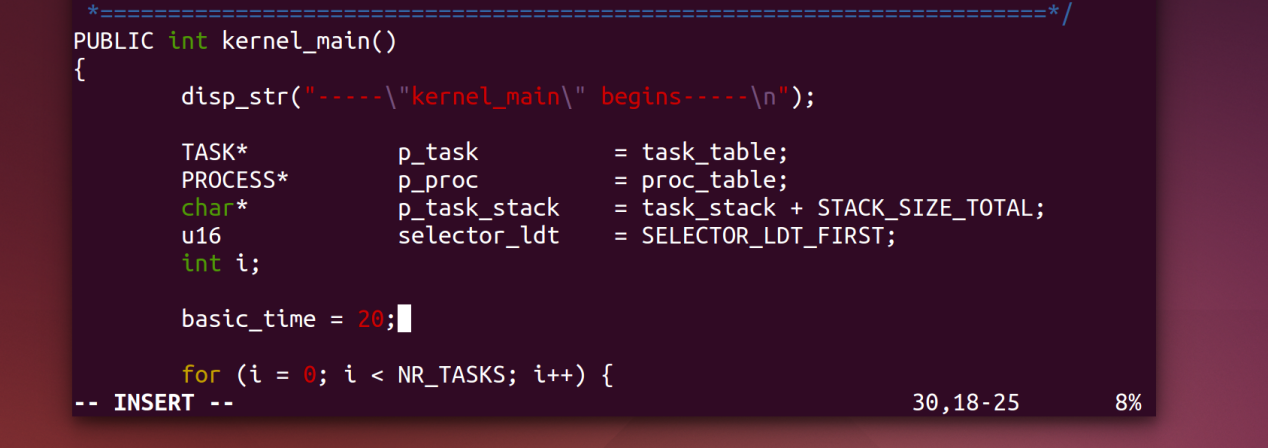
经过分析，我们仍然可以使用/include/proc.h中的PROCESS结构体：



我们使用priority变量来描述进程属于哪一个优先级队列，ticks变量表示该进程在当前队列中剩余的时间；

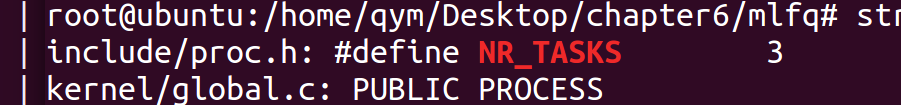
在这里我们定义时间片基本单位T = 20，在global.h中定义一个全局变量basic\_time，并在kernel\_main中进行初始化：

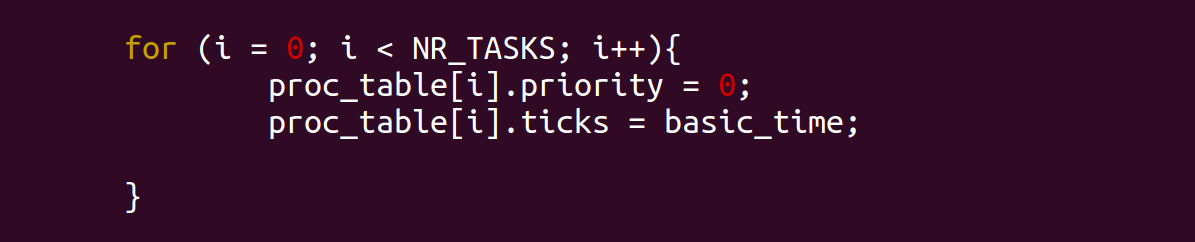




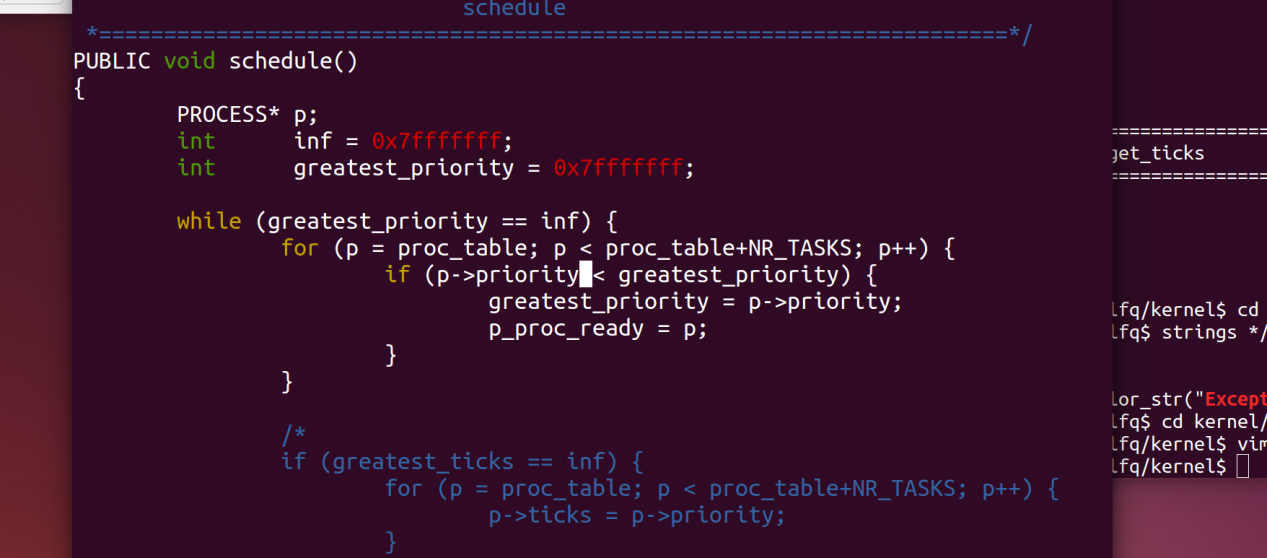
多级调度中我们使用三级队列，且规定级别越高priority的值越小（最高级位0），在初始化时，我们将所有进程的priority赋值为0，均处于最高级队列；

对于ticks的值，则都是以T为单位的，在main函数里初始化每一个进程的ticks：





由于我们采用priority越小优先级越高，所以我们改变一下schedule中的逻辑：



最终我们的时钟中断处理程序如下：

PUBLIC **void** clock\_handler(**int** irq)

{

    ticks++;

    p\_proc\_ready->ticks--;

    PROCESS\* p;

**int** have\_high;

    if (k\_reenter != 0) {

        return;

    }

    if(p\_proc\_ready->priority == 0){

*/\*如果当前进程在最高优先级队列且时间片未用完，则一定不会被抢占\*/*

            if(p\_proc\_ready->ticks != 0) return ;

*/\*时间片用完\*/*

        p\_proc\_ready->ticks = basic\_time \* 2;

        p\_proc\_ready->priority = 1;

        schedule();

    }

    else if(p\_proc\_ready->priority == 1){

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

            p\_proc\_ready->ticks = basic\_time \* 3;

            p\_proc\_ready->priority = 2;

            schedule();

            return ;

        }

        have\_high = 0;

*/\*检查是否有优先级更高的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(p->priority == 0){

                have\_high = 1;

                break;

            }

        }

        if(have\_high){

*/\*如果有优先级更高的进程\*/*

*/\*则发生抢占\*/*

            p\_proc\_ready = p;

        }

        }

    else if(p\_proc\_ready->priority == 2){

**int** have0 = 0;

**int** have1 = 0;

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

*/\*由于进程在最低优先级队列，不会降级，仅恢复时间片\*/*

                        p\_proc\_ready->ticks = basic\_time \* 3;

                        schedule();

                        return ;

                }

*/\*检查是否有优先级为0的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

                        if(p->priority == 0){

                                have0 = 1;

                                break;

                        }

                }

*/\*检查是否有优先级为1的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(have0) break;

            if(p->priority == 1){

                have1 = 1;

                break;

            }

        }

        if(have0 || have1){

*/\*如果有更高优先级的进程就发生抢占\*/*

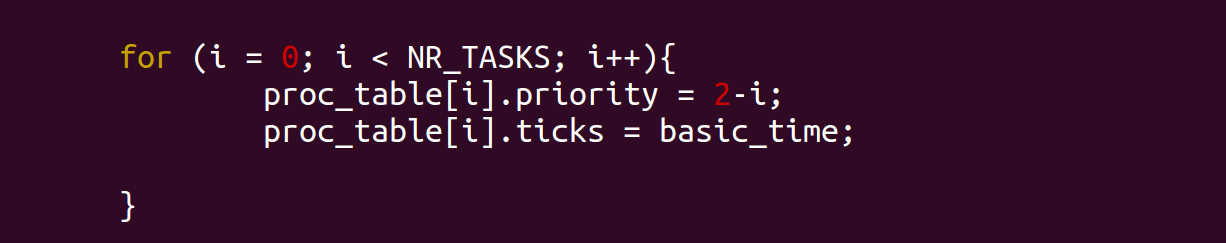
            p\_proc\_ready = p;

        }

        }

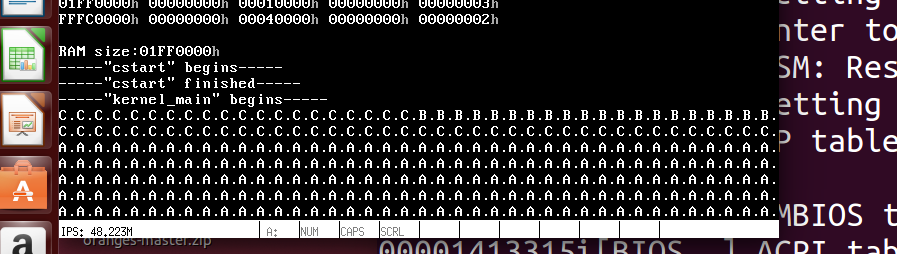
}

下面我们将对我们的调度算法进行测试，为了方便展示效果，我们将三个进程的优先级实现倒序，即C > B > A:



这样，预期效果应该是C优先级最高先打印若干C，之后C时间片用完，CB同时处于优先级1，则先打印B再打印C，之后ABC同时处于优先级2，则A会一直在前面，则会一直打印A；

执行之后看效果：

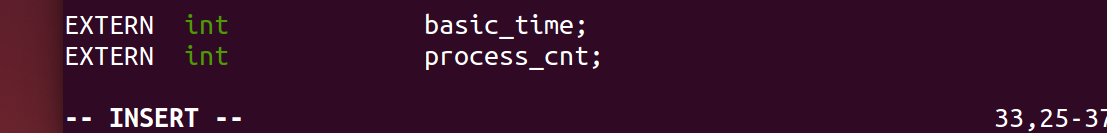


可以看到成功达到了我们的预期效果；

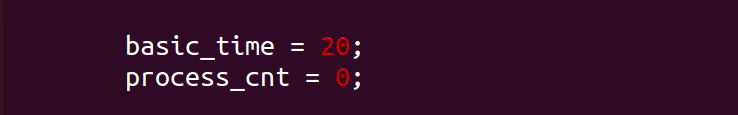
但是，最后BC进程出现了饥饿的现象，这是我们所不希望的，所以下面我们通过改进代码来解决这个问题；

首先解释以下为什么会出现饥饿，这是由于我们每次在遍历p\_proc\_table数组的时候，都是从头开始，如果在最低级别的队列中，则排号在前面的进程将永远抢先于排号在后面的进程，所以我们采取的解决方案是，添加一个全局变量process\_cnt，用它来记录当前遍历的位置，下次接着从这个位置开始，而实现循环迭代则只需要使用一个模加操作即可完成；

首先在/include/global.h中定义这个变量：



然后在kernel/main.c中初始化：



重点是要修改时钟中断处理程序，由于前两级队列都会沉淀到更低优先级队列，所以不会产生饥饿现象，我们可以暂时不做修改，故仅对最低级队列进行修改；

PUBLIC **void** clock\_handler(**int** irq)

{

    ticks++;

    p\_proc\_ready->ticks--;

    PROCESS\* p;

**int** have\_high;

    if (k\_reenter != 0) {

        return;

    }

    if(p\_proc\_ready->priority == 0){

*/\*如果当前进程在最高优先级队列且时间片未用完，则一定不会被抢占\*/*

            if(p\_proc\_ready->ticks != 0) return ;

*/\*时间片用完\*/*

        p\_proc\_ready->ticks = basic\_time \* 2;

        p\_proc\_ready->priority = 1;

        schedule();

    }

    else if(p\_proc\_ready->priority == 1){

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

            p\_proc\_ready->ticks = basic\_time \* 3;

            p\_proc\_ready->priority = 2;

            schedule();

            return ;

        }

        have\_high = 0;

*/\*检查是否有优先级更高的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(p->priority == 0){

                have\_high = 1;

                break;

            }

        }

        if(have\_high){

*/\*如果有优先级更高的进程\*/*

*/\*则发生抢占\*/*

            p\_proc\_ready = p;

        }

        }

    else if(p\_proc\_ready->priority == 2){

**int** have0 = 0;

**int** have1 = 0;

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

*/\*由于进程在最低优先级队列，不会降级，仅恢复时间片\*/*

                        p\_proc\_ready->ticks = basic\_time \* 3;

                        schedule();

            if(p\_proc\_ready->priority == 2){

*/\*如果新进程的优先级还是2，大概率会出问题\*/*

                while(1){

                    process\_cnt = (process\_cnt + 1) % NR\_TASKS;

                    p\_proc\_ready = proc\_table + process\_cnt;

                    if(p\_proc\_ready->priority == 2) break;

                }

            }

                        return ;

                }

*/\*检查是否有优先级为0的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

                        if(p->priority == 0){

                                have0 = 1;

                                break;

                        }

                }

*/\*检查是否有优先级为1的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(have0) break;

            if(p->priority == 1){

                have1 = 1;

                break;

            }

        }

        if(have0 || have1){

*/\*如果有更高优先级的进程就发生抢占\*/*

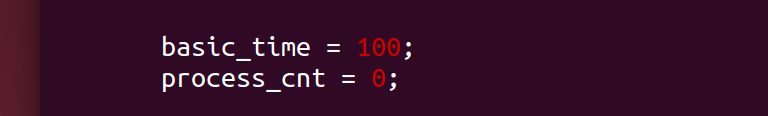
            p\_proc\_ready = p;

        }

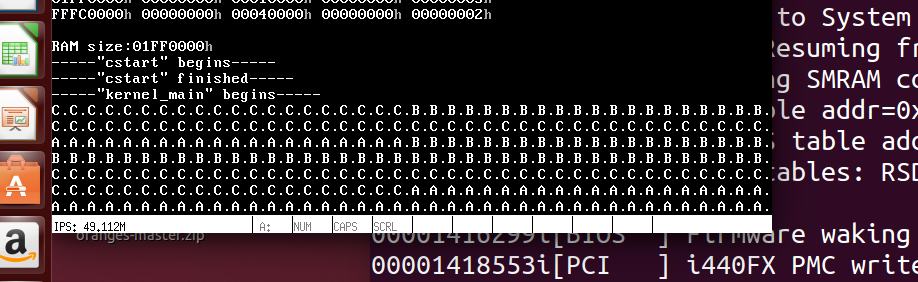
        }

}

此外就是basic\_time如果设置的过小，则会严重影响实验效果，所以我们做如下调整：



运行一下查看效果：



可以看到开始的时候仍然和之前一样，但是后边则是ABC交替出现，并没有出现某个进程饥饿的情况。

# 11 从用户态进程读和写内核段的数据

首先在设置进程表中进程属性时将所有的进程都设置为用户态进程。

p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_USER << 5;

memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3],

sizeof(DESCRIPTOR));

p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_USER << 5;

p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK)

| SA\_TIL | RPL\_USER;

p\_proc->regs.ds = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

| SA\_TIL | RPL\_USER;

p\_proc->regs.es = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

| SA\_TIL | RPL\_USER;

p\_proc->regs.fs = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

| SA\_TIL | RPL\_USER;

p\_proc->regs.ss = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK)

| SA\_TIL | RPL\_USER;

p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK)

| RPL\_USER;

接着在进程A中尝试读取并写内核段的数据，这里尝试读取地址为0处的数据，并将其更改为0。

void TestA()

{

int value,tmp;

memcpy( &value, 0,sizeof(int));

disp\_int(value);

disp\_str("\n");

value=0x0000;

memcpy(0,&value,sizeof(int));

memcpy( &tmp, 0,sizeof(int));

disp\_int(tmp);

disp\_str("\n");

while (1) {

disp\_str("A.");

milli\_delay(10);

}

}

运行结果如下：



发现成功从地址为0处读取出了数据，向该地址写数据时也成功写入数据0，但是并没有触发段错误异常，可能因为c语言程序编译链接后地址会进行重定位。那么尝试读取和修改gdt表中第2项描述符指向的基地址中的数据。

void TestA()

{

int value,tmp;

u32 address = 0;

address |= (u32)gdt[1].base\_low;

address |= ((u32)gdt[1].base\_mid << 16);

address |= ((u32)gdt[1].base\_high << 24);

u32\* addressPointer = (u32\*)address;

value = \*(addressPointer);

disp\_int(value);

disp\_str("\n");

\*addressPointer = 0xAAAA;

tmp=\*addressPointer;

disp\_int(tmp);

disp\_str("\n");

while (1) {

disp\_str("A.");

milli\_delay(10);

}

}

运行后发现读取的结果与上面程序的结果一致，修改效果也一样，都没有触发段错误异常，理论上是会触发异常，但最终仍未找到问题在哪里。



在常规情况下，用户态进程是不能直接读写内核态的数据或内核段的数据的。这是因为现代操作系统使用了内存保护机制，将用户态和内核态的内存空间隔离开来，以确保系统的稳定性和安全性。试图从用户态进程中直接读写内核态的数据通常会触发保护机制，导致操作系统抛出异常，如"Segmentation Fault"（段错误）。

以下是为什么会触发保护机制的原因：

（1）权限不足：内核态的内存空间通常被标记为不可访问或只读，用户态进程没有足够的权限来修改它们。尝试写入内核态内存会导致保护异常。

（2）地址空间隔离：操作系统通过分离用户态和内核态的地址空间来提高系统的安全性和稳定性。用户进程的虚拟地址空间和内核态的虚拟地址空间是不同的，用户进程不能直接访问内核态的地址空间。

（3）系统调用：如果用户态进程需要与内核交互，例如读写硬件设备、请求系统服务等，它们必须通过系统调用接口来请求内核执行这些操作。系统调用提供了一种受控的方式来在用户态和内核态之间传递数据。

（4）安全性和稳定性：允许用户态进程直接访问内核态的数据会增加系统的不稳定性和安全性风险。内核态的数据和代码通常包含操作系统的核心逻辑，不应受到用户进程的随意访问。

因此，为了保护操作系统的稳定性和安全性，用户态进程不能直接读写内核段的数据。必须使用适当的系统调用来进行与内核的交互，以确保在受控的情况下进行数据传递和访问。如果用户态进程尝试越过这些保护机制直接访问内核态数据，操作系统将触发保护异常，并终止进程的执行。这种异常的处理通常会导致进程终止或出现核心转储以便进行故障分析。

# 12 贡献与体会

## 12.1 赵伯俣

贡献：

撰写实验报告中系统调用基本框架以及在该框架之中各个函数实现的功能，并绘制出系统调用过程的流程图并分析出操作系统进行系统调用后的实验结果。撰写对于可编程计数器部分的理论知识和实际操作，并分析操控可编程计数器后得到的实验结果。

体会：

在本次实验中学习到了如何在单进程的操作系统中借助时钟中断拓展实现多进程的运行；学习到多进程控制过程中对于多进程控制块以及LDT表和TSS的初始化操作；学习到系统调用的实现过程以及在系统调用过程中每一个函数模块实现的具体功能；学习到进程之间进行调度的框架以及在进程中如何进行优先级调度操作；尝试了在用户态进程读写内核段数据所触发的保护以及产生该保护的原因

## 12.2 刘竞优

贡献：

完成实验报告中进程调度的基本框架，实现优先级调度以及思考题尝试读取和写内核段的数据相关内容，整合实验报告。

体会：

学习了简单的优先级调度的实现和linux进程调度的基本框架，基本明白多进程在内核中是如何被调度和运行的，在优先级调度的实现时调度程序是否判断当前运行态的程序是否使用完时间片可以分别实现抢断式优先级调度和非抢占式优先级调度。学习了常规情况中在用户态进程读取内核段数据是不被允许的，会引发异常并中止程序，但在本次实验的尝试中并没有触发异常，还没有找到问题所在，期待之后学习过程中能够解决。

## 12.3 钱嘉乐

贡献：

阅读课本及查阅资料，完成多进程问题：如何扩展单进程到多进程，如何扩展中断支持多进程。思考课后习题部分：在单进程的基础上扩展实现多进程要考虑哪些问题；如何修改时钟中断来支持多进程管理。同时，画出以下流程图：初始化多进程控制块的过程、扩展初始化LDT和TSS，修改时钟中断来支持多进程管理。撰写实验报告对应部分内容，提供相关实验结果图。

体会：

在完成这个实验的过程中，我深入了解了多进程的概念和原理，并学习了如何将单进程扩展为多进程系统。多进程系统可以同时运行多个进程，每个进程有自己独立的执行空间和资源。这样可以提高系统的并发性和效率。通过这个实验，我理解了多进程系统的设计和实现原理。我意识到在扩展单进程到多进程时需要考虑很多因素，如进程控制块的管理、调度算法的选择、进程间通信和同步等。同时，我也了解到了如何修改时钟中断来支持多进程管理，以实现进程之间的切换和调度。

## 12.3 钱一铭

贡献：

负责本次实验中的动手做全部工作，前期主要是对整个算法项目的规划以及具体实现问题的提出，然后利用时钟中断实现了一个多级反馈队列的调度算法，完成了相关代码的实现，并通过对实验效果的分析、调试，对其中不理想结果的原因进行定位，并判断为饥饿现象，最后则是通过对代码的改进解决前面出现的饥饿现象，最终实现了一个具有实际意义的多级反馈队列调度算法。

实验体会：

通过本次实验，我首先复习了上学期理论课中所学习的多级反馈队列调度算法，然后将这些理论知识与上节课中所实现的基于时钟中断的定时器相结合，实现了一个自己的多级反馈队列；而在具体实现的过程中，通过对一些不理想情况的分析、调试，我 我更加深入理解了饥饿现象的成因，以及相关的解决办法；总体来说，实现一个多级反馈队列算法涉及多方面的技术，对自己的代码能力是一个很好的提升。

# 13 教师评语

|  |  |  |
| --- | --- | --- |
|  | | |
| **教师评分（请填写好姓名、学号）** | | |
| 姓名 | 学号 | 分数 |
| 刘竞优 | 2021302181057 |  |
| 赵伯俣 | 2021302181156 |  |
| 钱嘉乐 | 2021312181234 |  |
| 钱一铭 | 2021302181115 |  |
| 教师签名：  年 月 日 | | |