## 第二章 基于DOS的多任务系统的实现

# 2.1 设计目的和内容要求

## 1．设计目的

通过对线程（和进程）的创建和撤消、CPU的调度、同步机制、 通信机制的实现，达到以下目的：

1. 加深对线程和进程概念的理解，明确进程和程序的区别。
2. 加深对CPU调度过程（现场保护、CPU的分派和现场恢复）的理解。
3. 进一步认识并发执行的概念，明确顺序执行和并发执行的区别。
4. 加深对临界资源、临界区、信号量以及同步机制的理解。
5. 加深对消息缓冲通信的理解。

2．内容要求

（1） 用C语言完成线程的创建和撤消，并按先来先服务方式对多个线程进行调度。

（2） 将线程调度算法修改为时间片轮转算法，实现时间片轮转调度。（也可以结合优先权，实现优先权加时间片轮转算法的线程调度。）

（3） 改变时间片的大小，观察结果的变化。思考：为什么时间片不能太小或太大。

（4） 假设两个线程共用同一软件资源（如某一变量，或某一数据结构），请用记录型信号量来实现对它的互斥访问。

（5） 假设有两个线程共享一个可存放５个整数的缓冲，其中一个线程不停地计算１至５０的平方，并将结果放入缓冲中，另一个线程不断地从缓冲中取出结果，并将它们打印出来，请用记录型信号量实现这一生产者和消费者的同步问题。

（6） 实现消息缓冲通信，并与4、5中的简单通信进行比较。

（7） 思考：在线程间进行消息缓冲通信时，若对消息队列的访问没有满足互斥要求，情况将会怎样？

3. 学时安排（共21学时）

1. 授课3学时，内容包括线程的创建、撤消、调度等内容。
2. 线程的创建、撤消、先来先服务调度，8学时上机。
3. 时间片轮转调度，3学时上机。
4. 信号量的实现，3学时上机。
5. 线程间的消息缓冲队列通信，4学时上机。

4． 开发平台

TurboC 2.0或3.0。

### 线程描述

2.2.1线程基本概念

在一些多任务的环境下，用户可以同时运行多个完整的程序。例如，在UNIX环境下，你可以用ＣＣ命令编译一个Ｃ程序，并把它作为一个后台进程运行（只需在命令行后加上字符‘＆’）；在前台，你又可以做其他的事情，比如，编辑另一个文件。我们把这种系统称为基于进程的多任务系统。另外有一种多任务系统，在其下，一个程序的多个部分可同时运行，我们把这种环境下的任务，即程序的每个部分叫做线程，称这种系统为基于线程的多任务系统。在这种环境下，处理机的调度单位为线程，它们共享整个进程的资源，还拥有一些自己的私有资源。我们将通过本课程设计实现多个线程的并发执行。

线程，有时也叫做轻进程（lightweight process），是CPU调度的基本单位，每个线程有自己的一个指令计数器、一组寄存器和一个私有堆栈。而代码段、数据段以及操作系统的其它资源（如打开的文件）是由一组线程共享的，这一组线程组成一个Task（传统的进程，即heavyweight process相当于只有一个线程的Task）。

在许多方面，对线程的操作类似于进程：线程可处于就绪、阻塞、执行三种状态之一；线程可共享CPU， 在单机系统中，任何时刻最多只能有一个线程处于执行状态；一个Task中的多个线程可并发执行。但与进程不同，一个Task中的多个线程并不互相独立，因为，所有线程均可访问所属Task的地址空间的任一单元，所以，一个线程读写其它线程的私有堆栈是十分容易的，即系统不提供线程间的保护。

注：上面讲的Task是指一个完整的作业，其中可包括多个线程，与本课程设计中所讲的多任务中的任务（系统中可并发执行的部分，如线程或进程）含义不同，除此之外，本课程设计中所提到的Task或任务均代表后者。

线程的切换只需切换寄存器组的值，而不需做有关内存管理方面的工作，实现起来也就比较简单。

2.2.2 线程控制块

与进程类似，基于线程的多任务系统中的任务，即线程，它不单是指静态的、可并发执行的程序段本身，其实也是一个动态的概念，是指可并发执行的程序段及其执行过程。因此，我们要用一个类似于进程控制块PCB的数据结构——线程控制块TCB，来记录有关描述线程情况和控制线程运行所需的全部信息，具体来说，在一个TCB中主要应包括以下几方面的信息：

1．有关线程私有堆栈的信息

在线程调度的过程中，为了保护线程的现场信息，每个线程都必须有自己的私有堆栈。我们把被切换线程的现场信息，包括目前各寄存器的值和下一条指令的地址都保存在它的堆栈中，再从新线程的私有堆栈中恢复出一组新值来布置系统的寄存器，并从私有堆栈中得到新线程的下一条指令地址。另外，每个线程中用到的局部变量也是存放在它自己的私有堆栈中的。因此，在TCB 中必须有线程的私有堆栈的信息，包括它在内存的起始地址、 堆栈的栈顶指针的段地址和偏移等信息。

DOS中内存的地址是20位的，而且DOS的内存管理采用分段的方式，每个段的基址的低4位必须为0，指令和数据的逻辑地址可用两个16位的整数来描述，即：段地址seg和段内偏移off，其中段地址seg中有段基址的高16位，故逻辑地址seg:off对应的物理地址为seg×24+off。C语言经常用指针来描述一个地址，Turbo C提供了三个宏函数用来实现指针方式到段地址、偏移地址方式的相互转换：若P为一个指针，则可通过FP\_SEG(p)得到该地址的段地址，FP\_OFF(p)得到该地址的段内偏移；若seg为一个地址的段地址，off为其段内偏移，则可通过MK\_FP(seg,off)得到对应的指针。

2．有关线程的状态的信息

在基于线程的多任务系统中，一个线程的状态在它的生命周期中是在不断地变化的，在此，我们把线程的主要状态划分为：就绪、执行、阻塞和终止态。如果，一个线程拥有CPU，我们就说它处于执行态；如果它现在虽不在执行，但一旦获得CPU 就可执行，我们就说它处于就绪态；如果它在等待CPU以外的其他资源，则说它处于阻塞状态；如果线程所对应的程序段已运行完毕，则它处于终止状态。因此，在TCB中要设置一状态字段，用来记录各线程的现行状态。

3．线程的标识符

线程标识符用于惟一地标识一个线程，与进程一样，通常一个线程有两个标识符：

（1）外部标识符：它由创建者提供，通常是一个由字母、数字组成的字符串，记录在线程的TCB中。

（2）内部标识符，它通常是一个整数，由多任务系统在创建线程时设置。在本课程设计中，我们在多任务系统的初始化过程中，设置了一个struct类型的TCB数组来统一为各新建线程提供空白TCB，为了简单起见，我们可以隐含地用各线程所分配到的TCB在整个TCB数组中的下标来表示该线程的内部标识符，所以不需要再专门记录在TCB中了。

4．其它信息

TCB中记录的信息量可随系统的复杂情况而变化，如当采用优先权算法进行调度时,在TCB中还必须设置优先权字段；当TCB要按某种方式排队时，在其中必须设置一链接指针字段；当必须唤醒因某种原因而阻塞的相关线程时，则必须设置阻塞原因字段；在使用消息缓冲队列机制实现线程通信时，则必须设置通信机制需要的字段，如接收线程的消息队列队首指针、消息队列的互斥信号量和资源信号量等。

用C语言来描述，一个最简单的TCB的数据结构可以表示如下：

/\* 状态码常量定义 \*/

/\* null 0 not assigned \*/

#define FINISHED 0 /\*表示线程处于终止态或TCB是空白状态\*/

#define RUNNING 1 /\*表示线程处于运行态\*/

#define READY 2 /\*表示线程处于就绪态\*/

#define BLOCKED 3 /\*表示线程处于阻塞态\*/

struct TCB{

unsigned char \*stack; /\* 线程堆栈的起始地址 \*/

unsigned ss; /\* 堆栈段址 \*/

unsigned sp; /\* 堆栈指针 \*/

char state; /\* 线程状态 ，取值可以是FINISHED、RUNNING、READY、BLOCKED\*/

char name[10]; /\* 线程的外部标识符 \*/

} tcb[NTCB]; /\*NTCB是系统允许的最多任务数\*/

### 线程的创建和撤消

2.3.1 线程的创建

在创建一个新线程时，线程的创建者必需提供一些信息，如线程的外部标识符、线程所需的私有堆栈空间的大小、与线程所对应的程序段的入口地址的有关信息（这里假设一个线程执行程序里的一个函数，所以创建者只需提供线程要执行的函数的函数名即可）。

1．线程创建函数格式说明

（1）函数申明原型： typedef int (far \*codeptr)(void); /\*定义了一个函数指针类型\*/

Int create(char \*name,codeptr code,int stck) ；

（2）函数功能描述：在main()函数中调用，创建一个新线程，让其执行code开始的代码。

（3）输入：

name：新创建线程的外部标识符；

code：新创建线程要执行的代码的入口地址，此处用函数名作为传入地址；

stck： 新创建线程的私有堆栈的长度。

（4）输出：新创建线程的内部标识符，若创建失败，返回-1

2． 函数实现的算法描述

在创建一个线程时主要应完成以下工作：

1. 为新线程分配一个空闲的线程控制块TCB，该TCB 的数组下标即为新线程的内部标识符。如果没有空闲的TCB，则返回-1，创建失败。
2. 为新线程的私有堆栈分配内存空间（因为同一进程的多个线程共享该进程的程序段和数据段空间，所以创建线程时不必象创建进程那样再为程序段和数据段分配内存空间）。
3. 初始化新线程的私有堆栈，即按CPU 调度时现场信息的保存格式布置堆栈，这一点是非常重要的，因为当CPU首次调度该线程运行时，CPU中的SS寄存器和SP寄存器将指向该线程的私有堆栈，并从该堆栈中获得线程运行的正确的指令地址和其它现场信息。新线程的首次执行是从对应函数的入口开始的；而且，执行时CPU的寄存器ES、DS应置上恰当的值；Flags 寄存器的允许中断位也应置上1，这样，线程执行过程中才允许硬中断（如时钟中断）发生并及时响应中断；其它寄存器（AX、BX、CX、DX、SI、DI、BP）的值只在线程执行过程中才有意义，它们的初值可为任意值。初始化工作完成后堆栈中各信息项的值及其相应位置如图2-1b所示。

为了方便堆栈的初始化工作，我们可以按照堆栈中的内容设计一个以下的数据结构：

struct int\_regs {

unsigned bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags,off,seg;

};

然后用一个指向该数据结构的指针给堆栈赋值。

1. 初始化线程控制块，即填入线程的外部标识符，设置好线程私有堆栈的始址、段址和栈顶指针，将线程的状态置成就绪态READY，如图2-1a所示。

另外，如果线程调度算法是按优先权方式进行CPU调度，则需在TCB中置上新线程的优先权信息（初始优先数可由用户提供）；若TCB的组织方式是按某种方式拉链，系统设置了线程就绪队列，则还需将新线程的TCB插入就绪队列；如果要实现通信，还需要将线程的消息队列队首指针设置为Null、消息队列的互斥信号量和资源信号量分别设置为{1，Null}和{0，Null}

1. 最后，返回新线程的内部标识符。

在Turbo C的small编译模式下，调用create("f1",(codeptr)f1,1024) 创建一个对应于函数f1()的线程后新线程的内存映象如图2-1所示。



图2-1 对应函数f1()的新线程的内存映像

2.3.2 线程的撤消

引起线程撤销的原因主要有两个：一是系统或用户要求撤销某个线程；二是当前线程所对应的函数已经运行完成。对于第一种情况比较简单，只需调用线程撤销原语将指定线程撤销即可；对于第二钟情况，首先必须自动调用线程撤销原语撤销当前已经运行完成的线程，然后还需要自动地重新进行CPU调度。

1． 线程撤销函数设计：

（1）函数申明原型：void destroy（int id）；

（2）功能：撤销内部标识符为id的指定线程。

（3）输入：

id：将要被撤销的线程的内部标识符。

（4）输出：无

（5）函数实现的算法描述：

撤销线程所要完成的工作比较简单，主要是将线程所占据的资源归还给系统。在操作系统原理中已经介绍了线程本身基本不占据资源，它与同进程的其他线程共享该进程的代码段和数据段空间；但是线程作为一个可以独立调度和运行的基本单元也拥有一些必不可少的资源，如线程控制块TCB和私有堆栈。所以撤销线程所要做的事情主要就是两个：

（1）将线程的私有堆栈所占的内存空间归还给系统；

（2）将线程控制块TCB的各成员变量进行初始化操作。

2． 撤销线程并重新进行调度

前面提到如果是因为当前线程运行完成而引起线程撤消，则系统应能自动撤消该线程，并重新进行CPU调度。我们可以设置一个称为over（）的函数来完成这个工作，该函数需要顺序做两件事情：首先调用destroy（）撤销当前线程，然后重新进行CPU调度。所以现在关键的问题是在当前线程运行完成后CPU应能自动转去执行over（），这可通过在创建线程时进行一些相关的处理来实现：在进行堆栈初始化时可预先将over( ) 的入口地址压入线程的私有堆栈中，如前面图2-3b所示；这样，当线程所对应的函数正常结束时，over()函数的入口地址将作为函数的返回地址被弹出至CPU的CS、IP寄存器，从而使CPU的控制权自动转向over()去执行。

### 线程调度设计

2.4.1 CPU调度中的关键问题

CPU调度所要做的事情是保护旧线程的现场、找到新线程、恢复新线程的现场、并把处理机交给新线程让它执行。其中，找一新线程是比较容易实现的，只需按某种线程调度算法从所有处于就绪状态的线程中选择一个即可；剩余的问题——旧线程的现场保护和新线程的现场恢复、CPU 控制权的转移才是CPU调度的关键，它们是通过堆栈的切换来实现的。在介绍堆栈切换的内容之前，我们先来看看函数调用和进行中断处理时控制转移的情况。

1．函数调用时的控制转移情况

在执行函数调用指令时，系统会自动地先将主调函数的下一条指令的地址（在CS:IP中）压入堆栈，然后把被调函数的入口地址装入CS和IP 寄存器（段内函数调用只需压入和装配IP），控制就从主调函数转向被调函数；当执行函数返回指令时，系统将当前堆栈的栈顶的两个字（主调函数下一条指令的地址）弹出并送到IP和CS中（段内函数返回只需弹出一个字送到IP中），控制就从被调函数返回到主调函数。

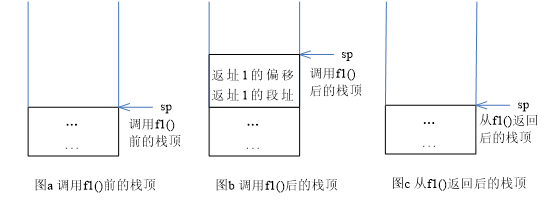
例如，我们编写了一个main()函数和一个f1()函数，在main()中调用f1()。程序的设计及调用返回关系如图2-2所示：



图2-2 函数调用及返回图

在执行f1()函数的调用指令前的当前堆栈的情况如图2-3a所示。在执行函数调用指令时，系统首先将main（）中函数调用语句的下一条指令即“i=1；”的地址（在CS:IP中，这里用“返址1”表示）压入堆栈，此时堆栈内容如图2-3b所示。然后将f1()函数的入口地址装入CS和IP 寄存器，控制就从main()转向f1()。当执行f1()的最后一条指令“return”（函数返回指令）时，系统将前面保存在堆栈中的返址1的偏移和返址1的段址弹出并送到IP和CS中，则控制就从f1()返回到main()了。

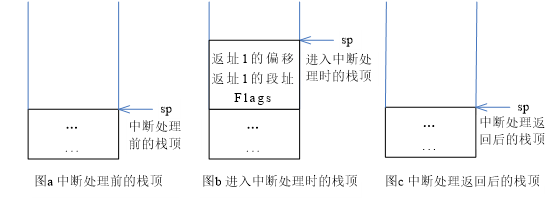
图2-3 函数调用前后堆栈内容的变化



2．中断处理时的控制转移情况

除了函数调用以外，中断也能实现控制的转移。当中断发生时，系统首先将标志寄存器Flags的值压入堆栈，然后将装有被中断程序下一条指令地址的CS和IP寄存器的内容也分别压入堆栈，再从中断向量中获取中断服务程序的入口地址并将它们装入CS和IP寄存器，这样，控制就从被中断的程序转向中断服务程序。中断返回时，系统自动从栈顶弹出返址1的偏移、返址1的段址和Flags并送到IP、CS和Flags寄存器中，CPU就开始继续从断点处执行被中断程序。中断处理前后堆栈内容的变化情况如图2-4所示：

图2-4 中断处理前后堆栈内容的变化



3． Interrupt类型函数的特殊作用

在Turbo C中提供了一个特殊的函数类型说明符interrupt，我们可利用它将一个函数申明为中断处理函数。例如我们写了一个文件名为“aaa.c”的C程序，其内容如下：

void interrupt fun(void);

int i;

main( )

{

i=0;

fun();

i=1;

}

void interrupt fun(void)

{

i=2;

}

用编译命令“tcc -S aaa”得到以上C程序的汇编码：

ifndef ??version

?debug macro

endm

endif

?debug S "aaa.c"

\_TEXT segment byte public 'CODE'

DGROUP group \_DATA,\_BSS

assume cs:\_TEXT,ds:DGROUP,ss:DGROUP

\_TEXT ends

\_DATA segment word public 'DATA'

d@ label byte

d@w label word

\_DATA ends

\_BSS segment word public 'BSS'

b@ label byte

b@w label word

?debug C E93B4F151F056161612E63

\_BSS ends

\_TEXT segment byte public 'CODE'

; ?debug L 3

\_main proc near

; ?debug L 4

mov word ptr DGROUP:\_i,0

; ?debug L 5

**pushf**

**call far ptr \_fun**

; ?debug L 6

mov word ptr DGROUP:\_i,1

@1:

; ?debug L 7

ret

\_main endp

; ?debug L 8

\_fun proc far

**push ax**

**push bx**

**push cx**

**push dx**

**push es**

**push ds**

**push si**

**push di**

**push bp**

mov bp,DGROUP

mov ds,bp

; ?debug L 10

mov word ptr DGROUP:\_i,2

@2:

; ?debug L 11

**pop bp**

**pop di**

**pop si**

**pop ds**

**pop es**

**pop dx**

**pop cx**

**pop bx**

**pop ax**

iret

\_fun endp

\_TEXT ends

\_BSS segment word public 'BSS'

\_i label word

db 2 dup (?)

\_BSS ends

?debug C E9

\_DATA segment word public 'DATA'

s@ label byte

\_DATA ends

\_TEXT segment byte public 'CODE'

\_TEXT ends

public \_main

public \_fun

public \_i

end

从编译后的代码中可以看出，对于fun()函数，由于定义的类型是interrupt类型的中断处理函数，所以在使用tcc命令进行编译时，编译器将自动在fun()的开始加入一组push操作（代码中第二组黑体字部分），以保存被中断程序的CPU现场环境信息；相对应地在fun()的代码最后自动加入一组pop操作（代码中第三组黑体字部分），以便中断返回时恢复被中断程序的现场环境信息。

从编译后的代码段中还可以看出，main()对fun()的调用是通过：

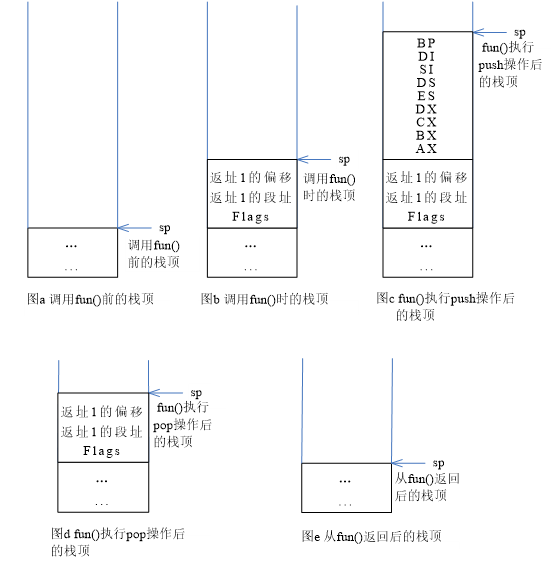
pushf

call far ptr \_fun /\*代码中的第一组黑体字部分 \*/

实现的，即先压入Flags寄存器的内容，再用call指令压入返回地址，并转去执行fun()函数。

而在进入fun()时，系统首先将执行一组push操作，将AX、BX、CX、DX、ES、DS、SI、DI、BP 的值保存到堆栈中，再用fun()的数据段装配DS寄存器，然后才执行fun()中的具体语句“i=2;”。而在返回前，先要执行一组pop操作，从堆栈中恢复BP、DI、SI、DS、ES、DX、CX、BX、AX 的值，然后再用iret指令（而不是ret指令）返回，iret指令将从堆栈中弹出返址的偏移、返址的段址、flags到CPU的IP 、CS和Flags寄存器中，从而使CPU继续从断点处执行被中断程序main()。fun()调用前后的堆栈内容如图2-5所示，图中的“返址1”是指main()函数中fun()函数调用指令的下一条指令“i=1”的地址。

令将从堆栈中弹出返址的偏移、返址的段址、flags到CPU的IP 、CS和Flags寄存器中， 图2-5 fun（）调用前后的堆栈内容的变化



4． 利用堆栈切换实现CPU切换

从上面的描述可知，我们可以用函数或中断服务子程序来处理CPU的切换，但关键仍在于堆栈的切换。例如， 系统中有两个线程：线程１和线程２，它们的私有堆栈分别为 stack1 和stack2；线程１目前拥有CPU，即线程１正在执行，则系统的现行堆栈为线程1的私有堆栈stack1；在线程１中调用一函数F()，函数的返回地址即线程１的下一条指令的地址将压入到现行堆栈stack1中;若在 F() 中，将系统的现行堆栈从stack1切换到stack2：保存现行堆栈的段址SS和栈顶指针SP到变量ss1、sp1中，并将线程２的堆栈stack2的段址和栈顶指针装到CPU的SS与SP寄存器中；如果stack2的栈顶有线程２的下一条指令的地址，则从F()中返回时，将用现行堆栈stack2 的栈顶的字装配IP和CS，CPU就开始执行新线程，即线程２。若再用保存在变量ss1、sp1 中的内容来装配SS和SP寄存器，即将现行堆栈切换回线程１的私有堆栈，则CPU 将被分配给线程１，它将从原来的断点继续往下执行。考虑到CPU 切换时要进行现场保护，用中断服务子程序来实现CPU的切换就显得更方便。

下面我们用interrupt类型的函数swtch()来实现CPU在线程1和线程2间的切换，另外，必须注意的是，在堆栈切换过程中一定要做到操作的原子性，这一点我们可以通过关中断、开中断来达到。Swtch()的设计如下：

void **interrupt** swtch(void)

{

disable(); /\*关中断\*/

/\* 保存现行堆栈的段址和栈顶指针供下次切换时用 \*/

ss1=\_SS; /\* ss1保存线程1的堆栈段址 \*/

sp1=\_SP; /\* sp1保存线程1的堆栈栈顶指针 \*/

/\* 切换堆栈 \*/

\_SS=ss2; /\* ss2是线程2的堆栈段址 \*/

\_SP=sp2; /\* ss2是线程2的堆栈的栈顶指针 \*/

enable(); /\*开中断\*/

}

上面代码中用到的\_SS，\_SP是Turbo C提供的伪变量。 所谓的伪变量是一个和给定寄存器相一致的简单的标识符，通过它们，我们可以在C 语言程序中直接访问相应的寄存器。表2-1中给出了Turbo C可用的伪变量的完整列表、它们的类型、相应的寄存器以及那些寄存器通常的用处。

表2-1 Turbo C 伪变量表

|  |  |  |  |
| --- | --- | --- | --- |
| 伪变量 | 类型 | 寄存器 | 通常用处 |
| \_AX | 无符号整型 | AX | 通用／累加器 |
| \_AL | 无符号字符型 | AL | AX的低字节 |
| \_AH | 无符号字符型 | AH | AX的高字节 |
| \_BX | 无符号整型 | BX | 通用／变址器 |
| \_BL | 无符号字符型 | BL | BX的低字节 |
| \_BH | 无符号字符型 | BH | BX的高字节 |
| \_CX | 无符号整型 | CX | 通用／计数和循环 |
| \_CL | 无符号字符型 | CL | CX的低字节 |
| \_CH | 无符号字符型 | CH | CX的高字节 |
| \_DX | 无符号整型 | DX | 通用／存放数据 |
| \_DH | 无符号字符型 | DH | DX的高字节 |
| \_CS | 无符号整型 | CS | 代码段地址 |
| \_DS | 无符号整型 | DS | 数据段地址 |
| \_SS | 无符号整型 | SS | 堆栈段地址 |
| \_ES | 无符号整型 | ES | 附加段地址 |
| \_SP | 无符号整型 | SP | 堆栈栈顶指针（对SS的偏移） |
| \_BP | 无符号整型 | BP | 基址指针 |
| \_DI | 无符号整型 | DI | 用于寄存器变量 |
| \_SI | 无符号整型 | SI | 用于寄存器变量 |

由于swtch()是interrupt类型的函数，因此编译后的伪代码如图2-6b所示：

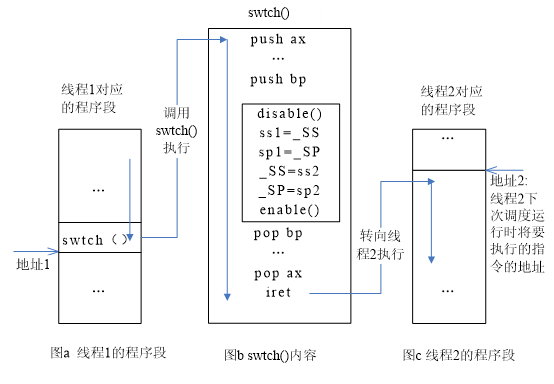


图2-6堆栈切换过程中CPU控制的转移情况

假设线程1对应的程序段如图2-6a所示，线程2对应的程序段如图2-6c所示。又假设现在是线程1正在CPU上运行，则当前堆栈是线程1的堆栈stack1，其内容如图2-7a所示。线程2目前处于就绪状态，其堆栈stack2的内容如图2-8a所示。

图2-7 CPU切换过程中线程1堆栈stack1的变化情况

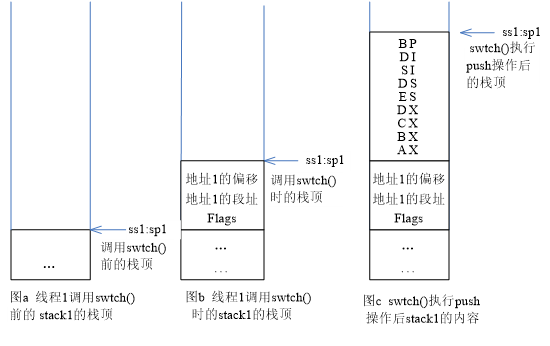
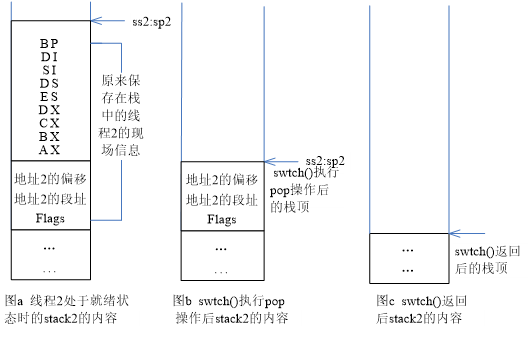


图2-8 CPU切换过程中线程2堆栈stack2的变化情况



下面我们来分析CPU从线程1切换到线程2的过程中控制的转移情况以及堆栈的变化情况。

（1）线程1执行swtch()函数调用指令，系统首先将Flags、地址1的段址、地址1的偏移压栈，此时stack1的内容如图2-7b所示。

（2）然后CPU转去执行swtch()，首先执行一组push操作，由于此时的当前堆栈仍然是线程1的stack1，因此push操作执行完毕后stack1的内容如图2-7c所示。

（3）接下来swtch()进行堆栈切换：首先保存线程1的堆栈stack1的段址和栈顶指针到变量ss1和sp1中；然后将线程2的堆栈stack2的段址和栈顶指针装配到CPU的SS好SP寄存器中，则从现在开始，系统的当前堆栈已经变成了stack2。

（4）接着swtch()执行一组pop操作，将stack2中从BP开始到AX结束的所有内容弹出并装入CPU的相应寄存器中，此时stack2的内容如图2-8b所示。

ret只弹出堆栈里的端地址（如果是远调用）和偏移地址  
iret还要弹出flag，这是因为中断发生的时候压入的是：flag、cs、offset

 ret  是普通的子程序的返回指令  
iret  是中断服务子程序的返回指令

（4）最后swtch()执行中断返回指令“iret”，从stack2中弹出线程2的偏移、线程2的段址和Flags并送到CPU的IP、CS和FLAGS寄存器中，此时stack2的内容如图2-8c所示。

（5）CPU继续执行CS：IP寄存器所指向的指令，即线程2的地址2这个位置的指令。于是CPU的控制权从线程1切换到线程2。

2.4.2 DOS的不可重入性

在本课程设计题目中，要求采用基于优先权的时间片轮转调度算法，即当前线程时间片到时应重新进行CPU调度。那么是不是线程的时间片到了就立即要重新进行调度呢？要回答这个问题，必须先理解“DOS的不可重入性”的概念。

在执行DOS的系统功能调用INT 21H时，DOS内核先将AX、BX、CX、DX、SI、DI、BP、DS、ES九个寄存器的内容依次压入到用户堆栈中，并将用户堆栈当前双字指针（SS：SP）保存到DOS内核的一个双字单元里，就不再使用用户堆栈， 此时使用的是新建立的系统内部堆栈，系统功能调用完毕后再舍弃系统内部堆栈而恢复用户堆栈的使用。

DOS的系统内部堆栈有三个，它们位于DOS操作系统空间的特定位置。当系统功能调用号为01-0CH、50H、51H之一，而且目前已有严重错误（即INT 24H的执行标志为1）时，使用第一个系统内部堆栈（也叫辅助堆栈）；当功能调用号不属01-0CH、50H、51H之一，则使用第二个系统内部堆栈（也叫I/O堆栈）；当功能调用号为01-0CH、50H、51H之一，但目前无严重错误发生时， 使用第三个系统内部堆栈（也叫磁盘堆栈）。

如果在程序１执行DOS系统功能调用的过程中，CPU被调度去执行程序２，而后者又要执行一系统功能调用，而且这两个系统功能调用使用的是同一个系统内部堆栈（如磁盘堆栈），则程序２的入栈数据将会把保存在该系统内部堆栈中的程序１的入栈数据覆盖掉，以后就无法再接着执行程序１。由此可见，以上的这种情况原则上仅允许对DOS的一次性重入，而且要求先后两次功能调用使用不同的系统内部堆栈，其它的对DOS 的重入都将导致错误。这就是DOS的不可重入性。

从另一方面考虑，不仅DOS在上述情况下会出现问题，当硬盘磁头响应INT 13H的调用，正处在读写磁盘的过程中时，进行CPU的调度， 而新的进程把磁头移到其它什么地方，这种重入问题虽对堆栈和可重复代码没什么影响，但将导致硬盘数据的混乱。

若按时间片进行CPU调度，时间片到时时，老线程很有可能是处于DOS的系统功能调用中，而调度后的新线程（或调度程序本身）又不可能完全不做系统功能调用。那么，如何来解决这一问题呢？有两种可考虑的办法：

1. 如果发现INT 21H正在执行中，就推迟对处理机的切换。
2. 在进行CPU调度时，设法为老线程保存DOS的所有内部信息（包括三个内部堆栈），并为新线程恢复DOS的所有内部信息（请参考有关“DOS的可对换数据区SDA”的书）。

相比较而言，第一种办法更加简单。该方法的关键问题是如何确定DOS是否处于忙状态（所谓DOS处于忙状态是指此时INT 21H正在执行中）。

在DOS的内部有一字节，当系统没进入INT 21H时，该字节的值为0； 当系统进入INT 21H时，DOS将该字节的值置成非0，而在退出INT 21H时，DOS又将它的值清0。这一字节被叫做DOS的安全标志，我们称它为INDOS标志，即当该标志的值为0 时，DOS不处于忙状态，可进行处理机调度；否则，DOS处于忙状态，需将处理机调度工作推迟。

使用未公开的34h号系统功能调用，可得到INDOS标志的地址（由ES：BX返回）。由于此地址是在特定操作环境下的常量，所以，可将得到的INDOS 标志的地址保存起来供判断DOS是否处于忙状态时使用。

另外，当DOS出现关键性错误时（如用21H号系统功能调用随机读一个记录时发生的读盘错误），DOS的严重错误处理程序将置上严重错误标志（DOS空间特定位置的一字节），并清除INDOS标志，再调用严重错误的中断处理程序INT 24H；而在INT 24H执行完后，又将严重错误标志清0，再将INDOS标志置上， 然后根据用户输入的Retry、Ignore和Abort决定是重新读盘、还是忽略错误结束系统功能调用、还是结束整个应用程序。即在严重错误标志被置上时，虽然INDOS标志为0，但INT 21H可能仍没结束，所以，此时也应推迟对处理机的调度。在MS-DOS 2.x 版本中，严重错误标志的地址在INDOS标志的后一个字节处，在MS-DOS 3.x中， 此标志的地址位于INDOS标志的前一个字节处，而在MS-DOS 3.10及更高版本中，可通过5D06H 号系统功能调用获得此标志的地址。

下面的C程序说明如何取得INDOS标志和严重错误标志的地址，并根据这两个标志判断DOS是否处于忙状态。

/\* INDOS.C -- Functions to manage DOS flags \*/

#include <stdlib.h>

#include <dos.h>

#define GET\_INDOS 0x34

#define GET\_CRIT\_ERR 0x5d06

char far \*indos\_ptr=0; /\*该指针变量存放INDOS标志的地址\*/

char far \*crit\_err\_ptr=0; /\*该指针变量存放严重错误标志的地址\*/

void InitDos(void);

int DosBusy(void);

/\* InitDos()函数：功能是获得INDOS标志的地址和严重错误标志的地址 \*/

void InitDos(void)

{

union REGS regs;

struct SREGS segregs;

/\* 获得 INDOS 标志的地址 \*/

regs.h.ah=GET\_INDOS;

/\* intdosx() ：Turbo C的库函数，其功能是调用DOS的INT21H中断\*/

intdosx(&regs,&regs,&segregs);

/\* MK\_FP()：不是一个函数，只是一个宏。\*/

/\*其功能是做段基址加上偏移地址的运算，也就是取实际地址。 \*/

indos\_ptr=MK\_FP(segregs.es,regs.x.bx);

/\*获得严重错误标志的地址 \*/

/\*代码中用到的\_osmajor、\_osminor是Turbo C的全程变量，其中前者为\*/

/\*DOS版本号的主要部分，后者为版本号的次要部分。\*/

if (\_osmajor<3)

crit\_err\_ptr=indos\_ptr+1;

else if (\_osmajor==3 && \_osminor==0)

crit\_err\_ptr=indos\_ptr-1;

else {

regs.x.ax=GET\_CRIT\_ERR;

intdosx(&regs,&regs,&segregs);

crit\_err\_ptr=MK\_FP(segregs.ds,regs.x.si);

}

}

/\* DosBusy()：函数功能是获得Indos标志及严重错误标志的值，判断是否dos忙：\*/

/\* 如果返回值是1，表示dos忙；\*/

/\* 如果返回值是0，表示dos不忙；\*/

/\* 如果返回值是-1，表示还没有调用InitDos() \*/

int DosBusy(void)

{

if (indos\_ptr && crit\_err\_ptr)

return(\*indos\_ptr || \*crit\_err\_ptr);

else

return(-1); /\* InitDos() hasn't been called \*/

}

2.4.2 调度程序的实现

1．调度算法设计

本次课程设计题目要求实现基于优先权的时间片轮转调度算法，即当前线程时间片到时，应暂停当前线程的运行，重新选择一个优先级最高的就绪线程参加运行。为了简单，在我们给出的实例中没有使用优先权，也没有建立就绪队列，而只是用TCB 的数组下标隐含地把所有的线程排成一个循环队列，并使用简单的时间片轮转调度算法，即当前正在运行的线程时间片到时，就暂停它的执行，把它的状态从执行态变为就绪态，把它的现场信息压入它的私有堆栈，并从隐含队列中找出下一个就绪线程，恢复它的现场，让它执行。

只要稍作改动，我们就可使用优先权高者优先的时间片轮转调度算法。为此，我们需在TCB中增加一优先数字段，并将优先数与优先权联系起来（如：优先数大者优先权较低，优先数小者优先权较高，也可反之），我们也可使用某种原则修改优先数（如：一个线程在就绪队列中等待一个时间片，则将它的优先数减去某个值a， 以提高它的优先权而避免长期等待；一个线程执行完一个时间片，则将它的优先数加上某个值b， 以降低它的优先权）。

另外，我们也可在TCB中加一链接字段， 把所有就绪的线程按某种方式排成一显式就绪队列，如优先权从高到低的队列，这样寻找新线程参加运行时只要取出就绪队列的队首线程即可。

2．调度程序的实现思路

引起CPU调度的主要原因有：时间片到时、 线程执行完毕或正在执行的线程因等待某事件发生而不能继续执行。

根据上述调度原因，在实例中的调度功能是通过两个函数来完成的：

（1）void interrupt my\_swtch(void)：该函数主要解决两种原因引起的调度：线程执行完毕或正在执行的线程因等待某事件发生而不能继续执行。

（2）void interrupt new\_int8(void)：该函数主要解决因时间片到时引起的调度，这可通过截取时钟中断（int 08）来完成。

在有的系统中，调度程序只有一个。如在UNIX系统中，时钟中断服务程序内并不直接进行CPU调度， 而只是设置需重新调度的标志； 每当中断（包括时钟中断）或捕俘（类似于PC 机的软中断）结束时，中断总控程序将检查重新调度标志，若重新调度标志已置上且被中断的是用户态程序，则系统的中断总控程序将控制转向调度程序来切换CPU。

3．调度程序的具体设计

（1）void interrupt my\_swtch(void)

该函数要完成的主要工作包括：首先保存当前线程的运行环境，包括私有堆栈的段址和偏移量；然后在系统的TCB数组中查找状态为就绪态的线程，恢复其运行现场，使其在cpu上参加运行。具体的程序设计流程图如图2-9所示。图中的current和timecount是实例中定义的两个全局变量，其中current记录当前正在运行的现场的内部标识符，timecount记录当前线程从上次调度至今已经运行了多少时间，注意timecount的时间单位是一个时钟中断间隔1/18.2 秒，约等于55ms，比如timecount的值为1，则表示当前线程已经运行了约55ms。

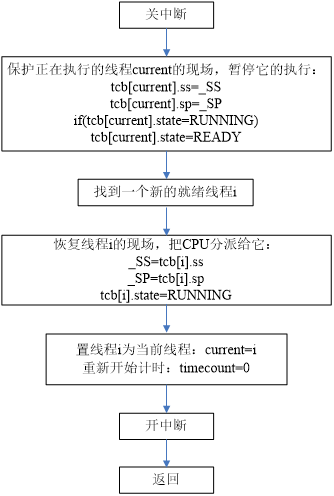


图2-9 my\_swtch()的程序设计流程图

（2）void interrupt new\_int8(void)

该函数要完成的主要工作包括：首先执行老的时钟中断处理程序的功能；然后判断当前线程的时间片是否到了，如果到了，且DOS不忙，则保存当前线程的运行环境，重新选择一个新的就绪线程，恢复其运行现场，使其在CPU上参加运行。

由于该函数必须是自动地定期地运行，所以我们通过截取时钟中断（int 08）来完成。

在PC机上，有一个定时器，每当它发一个时钟信号时（一般情况下，它每秒发出18.2个信号，除非是对产生该中断的定时器芯片重新编程），系统就进入时钟中断处理程序（INT 8)来完成系统计时、调用INT 1CH、关闭磁盘马达等工作。

我们可设计新的时钟中断服务程序，在其中来完成因时间片到时引起的CPU 调度。设计中要注意：新的时钟中断服务程序不能太长，否则系统效率将大大下降甚至使系统无法正常工作；在新的时钟中断服务程序里必须调用系统原来的INT 08H的功能，否则将影响磁盘马达的关闭和系统的计时，另外，我们还要依赖原来的INT 08H 向中断控制器发中断结束指令（EOI）。

在为INT 08H设置新的时钟中断服务程序前，必须提前用Turbo C提供的getvect()函数获取系统原来的INT 08H 的中断服务程序的入口地址并将它保存起来：

void interrupt (\*old\_int8)(void); /\*定义一个函数指针old\_int8\*/

old\_int8=getvect(8);

new\_int8()具体的程序设计流程图如图2-10所示：

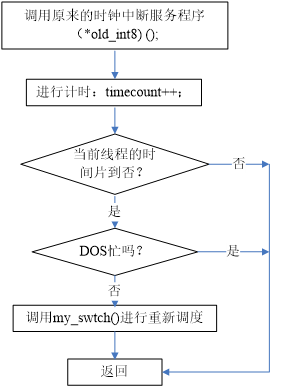


图2-10 new\_int8()的程序设计流程图

为了观察不同的时间片对调度所起的影响，实例中用一符号常量TL来表示时间片的大小，单位与timecount变量一样为时钟中断的间隔1/18.2 秒；对于当前线程来说，在每次时钟中断发生时，timecount加１。将timecount的值与TL 比较可判断当前线程的时间片是否到时，从而决定是否要进行CPU调度。

current 是实例中的另一全程变量，始终等于正在执行线程的内部标识符。

最后有几个问题留待同学们思考：

（1）在用my\_swtch()进行CPU调度时，要不要判断DOS的状态？为什么？

（2）在时间片调度时，时间片TL为什么不能太大或太小？

### 基本实例程序的实现

根据前面对设计思路的分析，我们可以编写一个较完整的实例：首先由main()函数进行一些初始化工作，然后直接创建0#线程对应于main()函数，再由0#线程调用create()创建1#、2#线程分别对应于函数f1()、f2()，最后，将系统的中断服务程序设置成new\_int8()，并把控制交给1#线程，启动多个线程的并发执行。不过在这个基本的实例程序中，我们没有实现线程的阻塞与唤醒、线程的同步与互斥、线程间的通信等内容，请同学们在学习了后面2.6-2.8节的内容后自行补充完成。

0#线程是一个比较特别的线程，它在创建时没使用create()，而是在系统初始化后直接创建的，因它所对应的程序段为main函数中的一段，所以也直接使用整个系统的堆栈，而不在创建时为私有堆栈分配额外的空间；同样，撤消时也不需释放私有堆栈的空间，所以也没使用destroy()，而是直接撤消的，从这方面看，我们可将它看成是一个系统线程。另一方面，在启动多个线程的并发执行后，0#线程就将系统控制权转交出去，直至系统中其它线程都不具备执行条件时，它才可能重新得到CPU，从这一点看，它相当于是一个空转线程。最后，0# 线程还担负一个特别的使命——等待系统中所有其它线程的完成，此时，它将直接撤消自己并恢复原来的时钟中断服务程序，从而终止整个多任务系统。

实例中f1()、f2()、main()函数的具体实现代码如后面所示，其中tcb\_state() 的功能为输出所有线程的状态信息；finished()函数的功能为检查系统中除0#线程以外的所有其它线程是否都已运行完成，是则返回１，否则返回０。

void f1(void)

{ int i,j,k;

for(i=0;i<40;i++){

putchar('a');

/\*延时\*/

for(j=0;j<10000;j++)

for(k=0;k<10000;k++);

}

}

void f2(void)

{ long i,j,k;

for(i=0;i<30;i++){

putchar('b');

for(j=0;j<10000;j++)

for(k=0;k<10000;k++);

}

}

main()

{

InitInDos();

InitTcb();

old\_int8=getvect(8); /\* 获取系统原来的时钟中断服务程序的入口地址 \*/

/\* 创建0#线程 \*/

strcpy(tcb[0].name, "main");

tcb[0].state=RUNNING;

current=0;

create("f1",(codeptr)f1,1024);

create("f2",(codeptr)f2,1024);

tcb\_state();

/\* 启动多个线程的并发执行 \*/

setvect(8,new\_int8); /\*为时钟中断设置新的中断服务程序\*/

my\_swtch();

while(!finished());

/\* 终止多任务系统 \*/

tcb[0].name[0]='\0';

tcb[0].state=FINISHED;

setvect(8,old\_int8);

tcb\_state();

printf("\n Multi\_task system terminated. \n");

}

实例的总流程如图2-11所示。

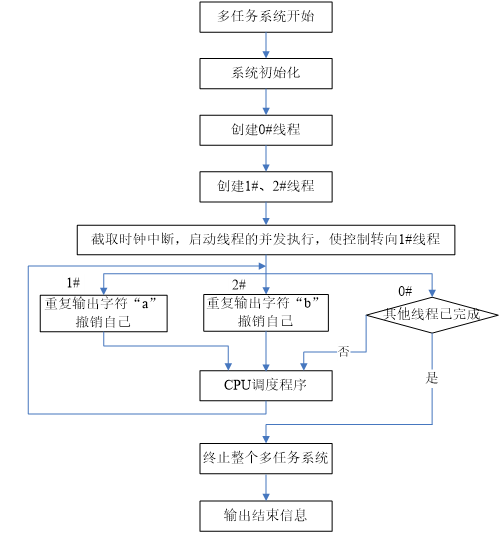


图2-11 实例总流程图

最后，补充说明一点，在程序运行的过程中，若DOS发现输入流中有Ctrl-C，DOS将执行INT 23H的中断服务程序。通常，该中断服务程序将终止当前程序的运行，并将控制转给当前程序的父进程（在这里是DOS的COMMAND程序），但它不会将时钟中断程序恢复成原先的(\*old\_int8)()， 这种不正常的结束将导致整个系统死机。要避免这个问题有两种方法：一种是不使用Ctrl-C；另一种更好的办法是修改INT 23H的中断服务程序，使它或是不起作用，或是撤消当前正在执行的线程， 或是正常地终止整个多任务系统。我们把这个问题留给同学们自己去解决。

### 线程的阻塞和唤醒

跟进程一样，正在执行的线程因为某种事件的发生而无法继续执行时，系统应允许它调用阻塞原语阻塞自己。

1．引起线程阻塞的事件

从操作系统原理课程中我们已经知道，引起进程阻塞的典型事件之一为等待I/O操作的完成：相对高速的CPU而言，I/O 设备的速度显得较慢，所以在一般情况下，一个进程在调用某类设备的设备驱动程序启动相应的I/O操作后，设备驱动程序将调用阻塞原语阻塞现行进程（即启动该I/O操作的进程），直至指定的I/O操作完成后，再由相应的I/O中断服务程序将阻塞的进程唤醒。虽然，引起线程阻塞的事件类似于进程，但在我们的实例中，由于直接使用DOS 操作系统提供的I/O功能，所以不考虑因启动I/O操作引起线程阻塞这一情况。

这里引起线程阻塞的典型事件有两个：一是并发执行的线程间竞争临界资源，如线程１与线程２共享某种软件资源（如某个变量、或缓冲区、或某个队列），为了互斥，线程１正在使用该资源时，若线程２也请求使用该资源，则线程２必须阻塞，等线程１使用完后，再唤醒线程２；二是相互合作的线程间的同步，如线程２等待接收线程１发来的消息，在线程１将消息发送来之前，线程２阻塞，直至线程１发完消息后再将线程２唤醒。

2. 阻塞与唤醒的过程

我们将所有处于阻塞状态的线程的TCB按阻塞的不同原因排成多个队列。为此，需在TCB中增加一链接字段next，以方便TCB的排队。

一个正在执行的线程因某种原因而阻塞时，必须给出因该原因而阻塞的阻塞队列的队首信息，阻塞原语应完成以下主要工作：将线程的状态改成阻塞态；将线程插入指定的阻塞队列末尾；重新进行CPU调度。

当阻塞线程所等待的事件完成后，必须唤醒相应线程。唤醒原语中同样需指出跟指定事件有关的阻塞队列队首信息。另外，有可能同时有多个线程在等待该事件完成，实例中采取的方法是唤醒阻塞队列头上的第一个线程。唤醒原语应完成以下主要工作：把阻塞队列头上的第一个线程的TCB取下来，并将其状态改为就绪态，如果系统设置了就绪队列，还应插入就像队列。

### 2.7 线程的同步与互斥

当多个线程并发执行时，与多个进程并发执行一样，它们可能会共享临界资源，也可能相互协作以完成总的任务，因此线程之间也存在着同步与互斥的问题，所以必须有相应的同步机制来实现线程间的同步与互斥。这里我们选择操作系统原理中介绍的记录型信号量机制来作为线程的同步机制。

记录型信号量的数据结构定义如下：

typedef struct{

int value;

struct TCB \*wq;

} semaphore;

对信号量的P操作和V操作的描述如下：

void p(semaphore \*sem)

{

struct TCB \*\*qp;

disable();

sem->value=sem->value-1;

if(sem->value<0){

qp=&(sem->wq);

block(qp);

}

enable();

}

void v(semaphore \*sem)

{

struct TCB \*\*qp;

disable();

qp=&(sem->wq);

sem->value=sem->value+1;

if(sem->value<=0)

wakeup\_first(qp);

enable();

}

为了实现互斥，可设置一互斥信号量mutex：

semaphore mutex; mutex=1;

然后，在相应的需要互斥的线程中，将临界区CS放在P(&mutex)和V(&mutex)之间。线程同步问题也同样可使用该信号量机制来解决，在这里就不再举例说明了。

在本课程设计中要求使用上面介绍的记录型信号量来实现生产者消费者问题，请同学们自行完成。

### 2.8 利用消息缓冲队列通信机制实现线程间通信

2.8.1 消息缓冲队列通信机制介绍

消息缓冲队列通信机制首先是由美国的Hansan提出来的，后来被广泛地应用于本地进程之间的通信中。其基本思想是根据“生产者——消费者”原理，利用内存中公用消息缓冲区实现进程间的信息交换。

在这种通信机制中，首先需要在内存中开辟若干空闲消息缓冲区，用以存放要通信的消息。每当一个进程需要向另一个进程发送消息时，便向系统申请一个空闲消息缓冲区，并把已准备好的消息复制到该缓冲区，然后把该消息缓冲区插入到接收进程的消息队列中，最后通知接收进程。接收进程接收到发送进程发来的通知后，从本进程的消息队列中摘下一消息缓冲区，取出其中的信息，然后把消息缓冲区作为空闲消息缓冲区归还给系统。系统负责管理公用消息缓冲区以及消息的传递。

2.8.2主要数据结构设计

1．消息缓冲区

在消息缓冲队列通信机制中，主要用到的数据结构是消息缓冲区，它可描述如下：

struct buffer

{

int sender; /\* 消息发送者的内部标识 \*/

int size; /\* 消息长度 <=NTEXT 个字节 \*/

char text[NTEXT]; /\* 消息正文 \*/

struct buffer \*next; /\* 指向下一个消息缓冲区的指针 \*/

};

假设，系统在对消息缓冲区进行初始化时，一共设置了NBUF个空闲的消息缓冲区，为了便于管理，通常将所有空闲缓冲区插入到空闲缓冲队列freebuf中，该队列是一临界资源， 无论是发送线程发送消息时从该队列中摘下一空闲缓冲区的操作（即申请空闲缓冲区的操作），还是接收线程在消息接收完毕时归还消息缓冲区的操作，所有对该队列的操作都必须互斥地进行。为此，系统为空闲消息缓冲队列设置了一互斥信号量mutexfb，其初值为{1,NULL}；另外，当空闲缓冲队列为空时，发送线程不能再获得空闲消息缓冲区发送消息，必须阻塞等待——直到接收线程归还一个空闲消息缓冲区为止，为解决这个问题，又为空闲缓冲队列设置了一个计数信号量sfb，其初值为{NBUF,NULL}。

2．TCB中与通信有关的数据项

在利用消息缓冲队列进行线程间的通信时，除了需要设置消息缓冲区外，还需对线程控制块TCB进行扩充，在原来的TCB的基础上再增加以下字段：

struct buffer \*mq; /\* 接收线程的消息队列队首指针 \*/

semaphore mutex; /\* 接收线程的消息队列的互斥信号量 \*/

semaphore sm; /\* 接收线程的消息队列的计数信号量，用于实现同步 \*/

在对TCB进行初始化时，mq的初值为NULL，mutex的初值为{1,NULL}，sm的初值为{0,NULL}。

2.8.3主要函数设计

在实现消息缓冲队列通信机制时，主要设计了以下函数：

1．获取空闲缓冲区函数getbuf()

（1）函数申明原型：struct buffer \*getbuf(void)

（2）功能：从空闲消息缓冲队列头上取下一空闲消息缓冲区。

（3）输入：无

（4）输出：指向所获得的空闲消息缓冲区的指针

（5）函数实现的程序描述：

struct buffer \*getbuf(void)

{

struct buffer \*buff;

buff=freebuf;

freebuf=freebuf->next;

return(buff);

}

2．插入缓冲区到缓冲队列函数insert()

（1）函数申明原型：void insert(struct buffer \*\*mq,struct buffer \*buff)

（2）功能：将buff所指的缓冲区插到\*mq所指的缓冲队列末尾。

（3）输入：

mq：将要插入的缓冲队列的队首指针的指针；

buff：消息缓冲区指针。

（4）输出：无

（5）函数实现的程序描述：

void insert(struct buffer \*\*mq,struct buffer \*buff)

{

struct buffer \*temp;

if(buff==NULL) return;

buff->next=NULL;

if(\*mq==NULL)

\*mq=buff;

else

{

temp=\*mq;

while(temp->next!=NULL)

temp=temp->next;

temp->next=buff;

}

}

3．发送原语send()

在发送消息时，消息的发送者必须提供接收者的标识符、消息长度及消息正文的起始地址等信息。然后在发送原语里申请一空闲的消息缓冲区，用相应的信息来装配该消息缓冲区，并将它插入到接受者的消息队列中去。

（1）函数申明原型：void send(char \*receiver,char \*a,int size)

（2）功能：将地址a开始的size个字节发送给外部标识符为receiver的线程。

（3）输入：

receiver：接收线程的外部标识符的指针；

a：要发送的消息正文的指针；

size：要发送的消息正文的长度。

（4）输出：无

（5）函数实现的程序描述：

void send(char \*receiver,char \*a,int size)

{

struct buffer \*buff;

int i,id=-1;

disable();

for(i=0;i<NTCB;i++){

/\*如果接收者线程不存在，则不发送，立即返回\*/

if(strcmp(receiver,tcb[i].name)==0){

id=i;

break;

}

}

if(id==-1) {

printf("Error: Receiver not exist!\n");

enable();

return;

}

/\*获取空闲消息缓冲区\*/

p(&sfb);// 空闲缓冲队列设置了一个计数信号量sfb

p(&mutexfb);// 系统为空闲消息缓冲队列设置了一互斥信号量mutexfb

buff=getbuf();

v(&mutexfb);

/\*填写消息缓冲区各项信息\*/

buff->id=current;

buff->size=size;

buff->next=NULL;

for(i=0;i<buff->size;i++,a++)

buff->text[i]=\*a;

/\*将消息缓冲区插入到接收者线程的消息队列末尾\*/

p(&tcb[id].mutex);

insert(&(tcb[id].mq),buff);

v(&tcb[id].mutex);

v(&tcb[id].sm);

enable();

}

4．获取消息缓冲区函数remov()

（1）函数申明原型：struct buffer \*remov(struct buffer \*\*mq,int sender)

（2）功能：接收线程从它自己的消息队列中取下sender发送给它的消息缓冲区。

（3）输入：

mq：接收线程的消息队列的队首指针的指针；

sender：发送线程的内部标识符；

（4）输出：所获得的消息缓冲区的指针

（5）函数实现的算法描述：

该函数需要完成的工作比较简单，首先在mq消息队列中查找发送线程标识符为sender的消息缓冲区，若找到则从mq所指示的消息缓冲队列中移除该消息缓冲区并返回该缓冲区，否则返回空。程序实现请同学们自己完成。

5．接收原语receive()

（1）函数申明原型：int receive(char \*sender,char \*b)

（2）功能：从接收线程的消息缓冲队列中取一个从sender发送过来消息并复制到b中。

（3）输入：

sender：发送线程的外部标识符；

b：接收线程内部的接收区，用来存放接收到的消息正文。

（4）输出：接收到的消息的长度

（5）函数实现的算法描述：

该函数要完成的主要工作是：首先检测指定的sender是否存在，如果不存在则直接返回；接着判断当前线程消息队列是否空闲，如果空闲，则调用remov()从消息队列取得sender发给它的消息；读取消息到b中；最后调用insert()将取完消息后的消息缓冲区插入到系统空闲消息缓冲队列中。具体代码请同学们自己完成。

另外，在设计receive()函数时，也可假设接收者能接收任一其他线程发送给它的消息，因此这时候就只需要指出接收区的起始地址信息，然后从自己的消息队列的队首取得一个消息缓冲区，将消息正文复制到接收区中，并释放相应的消息缓冲区即可。

在线程通信时，发送者和接收者都有可能在消息处理以前被撤消。其中，当一接收者等待一个来自某个已终止的线程的消息时，若不采取相应的动作，接收者将无限期阻塞下去。系统一旦发现这种情况，应结束接收动作，从而避免接收者被无限期阻塞。

注意，在示例代码中，在信号量的P、V操作中使用指向信号量的指针（而不是直接使用信号量本身）作为函数参数；另外，所有涉及到队列（包括线程控制块队列和消息缓冲队列）的插入和摘取操作的函数参数中，都没有直接使用队首指针来描述一个队列，而是用了指向队首指针本身的一个双重指针来描述一个队列。这都是因为C语言函数参数的传递是值传递，而不是地址传递的缘故。对队列操作，我们可稍加改进，在每个队列的队首加一个空成员，使队首指针始终指向该空成员，这样不管是插入还是摘取都不会改变队首指针本身，我们就可直接将队首指针作为函数的参数，以增加程序的可读性。同学们可根据这个思路去修改示例代码。