第3章　进 程 管 理

3.1　有人说，一个进程是由伪处理机执行的一个程序，这话对吗？为什么？

答：对。

因为伪处理机的概念只有在执行时才存在，它表示多个进程在单处理机上并发执行的一个调度单位。因此，尽管进程是动态概念，是程序的执行过程，但是，在多个进程并行执行时，仍然只有一个进程占据处理机执行，而其他并发进程则处于就绪或等待状态。这些并发进程就相当于由伪处理机执行的程序。

3.2　试比较进程和程序的区别。

答：(1) 进程是一个动态概念，而程序是一个静态概念，程序是指令的有序集合，无执行含义，进程则强调执行的过程。

(2) 进程具有并行特征(独立性、异步性)，程序则没有。

(3) 不同的进程可以包含同一个程序，同一程序在执行中也可以产生多个进程。

3.3　我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

答：并非对所有的程序均成立。例如：

```
begin
    local x
    x:=10
    print(x)
end
```

上述程序中 x 是内部变量，不可能被外部程序访问，因此这段程序的运行不会受外部环境影响。

3.4　试比较作业和进程的区别。

答：一个进程是一个程序对某个数据集的执行过程，是分配资源的基本单位。作业是用户为了让计算机完成某项任务而要求计算机所做工作的集合。一个作业的完成要经过作业提交、作业收容、作业执行和作业完成 4 个阶段。而进程是已提交完毕的程序的执行过程的描述，是资源分配的基本单位。二者的主要区别如下：

(1) 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业之后，系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体，是向系统申请分配资源的基本单位。任一进程，只要它被创建，总有相应的部分存在于内存中。

(2) 一个作业可由多个进程组成，且必须至少由一个进程组成，但反过来不成立。

(3) 作业的概念主要用在批处理系统中，像 UNIX 这样的分时系统中则没有作业的概

念。而进程的概念则用在几乎所有的多道程序系统中。

3.5 在 UNIX System Ⅴ中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

答：因为系统程序的代码被用户程序所共享，因此如果每个进程在保存进程上下文时，都将系统程序代码放到其进程上下文中，则大大浪费了资源。因此系统程序的代码不放在进程上下文中，而是统一放在核心程序所处的内存中。

3.6 什么是临界区？试举一个临界区的例子。

答：临界区是指不允许多个并发进程交叉执行的一段程序。它是由于不同并发进程的程序段共享公用数据或公用数据变量而引起的，所以它又被称为访问公用数据的那段程序。例如：

```
getspace:
begin
    local g
    g=stack[top]
    top=top-1
end
release(ad):
begin
    top=top+1
    stack[top]=ad
end
```

3.7 并发进程间的制约有哪两种？引起制约的原因是什么？

答：并发进程所受的制约有两种：直接制约和间接制约。

直接制约是由并发进程互相共享对方的私有资源所引起的。间接制约是由竞争共有资源而引起的。

3.8 什么是进程间的互斥？什么是进程间的同步？

答：进程间的互斥是指：一组并发进程中的一个或多个程序段，因共享某一公有资源而导致它们必须以一个不许交叉执行的单位执行，即不允许两个以上的共享该资源的并发进程同时进入临界区。

进程间的同步是指：异步环境下的一组并发进程因直接制约互相发送消息而进行互相合作、互相等待，是各进程按一定的速度执行的过程。

3.9 试比较 P、V 原语法和加锁法实现进程间互斥的区别。

答：互斥的加锁实现是这样的：当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否是上锁的，如果该临界区已被锁住，则该进程要等到该临界区开锁之后才有可能获得临界区。

但是加锁法存在如下弊端：(1)循环测试锁定位将损耗较多的 CPU 计算时间；(2)产生不公平现象。

为此，P、V 原语法采用信号量管理相应临界区的公有资源，信号量的数值仅能由 P、V 原语操作改变，而 P、V 原语执行期间不允许中断发生。其过程是这样的：当某个进程正在临界区内执行时，其他进程如果执行了 P 原语，则该进程并不像 lock 时那样因进不了临界区而返回到 lock 的起点，等以后重新执行测试，而是在等待队列中等待由其他进程做 V 原语操作释放资源后，进入临界区，这时 P 原语才算真正结束。若有多个进程做 P 原语操作而进入等待状态之后，一旦有 V 原语释放资源，则等待进程中的一个进入临界区，其余的继续等待。

总之，加锁法是采用反复测试 lock 而实现互斥的，存在 CPU 浪费和不公平现象，P、V 原语使用了信号量，克服了加锁法的弊端。

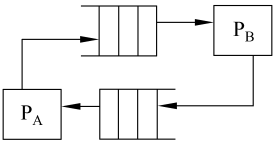
3.10 设在 3.6 节中所描述的生产者-消费者问题中，其缓冲部分由 *m* 个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度，生产者和消费者可对缓冲区同时操作。重新描述发送过程 deposit(data)和接收过程 remove(data)。

答：设第 i 块缓冲区的公用信号量为 mutex[i]，保证生产者进程和消费者进程对同一块缓冲区操作的互斥，初始值为 1。设信号量 avail 为生产者进程的私用信号量，初始值为 m；信号量 full 为消费者进程的私用信号量，初始值为 0。从而有

```
deposit(data)
    begin
        P(avail)
        选择一个空缓冲区 i
        P(mutex[i])
        送数据入缓冲区 i
        V(full)
        V(mutex[i])
    end
remove(data)
    begin
        P(full)
        选择一个满缓冲区 i
        P(mutex[i])
        取缓冲区 i 中的数据
        V(avail)
        V(mutex[i])
    end
```

3.11 两个进程 P_A 和 P_B 通过两个 FIFO 缓冲区队列连接(如下图所示)，每个缓冲区长度等于传送消息长度。进程 P_A 和 P_B 之间的通信满足如下条件：

(a) 至少有一个空缓冲区存在时，相应的发送进程才能发送一个消息。



(b) 当缓冲队列中至少存在一个非空缓冲区时，相应的接收进程才能接收一个消息。

试描述发送过程 send(i,m)和接收过程 receive(i,m)。这里 i 代表缓冲队列。

答：定义数组 buf[0]、buf[1]、bufempty[0]和 buffull[1]是 P_A 的私有信息量，buffull[0]、bufempty[1]是 P_B 的私有信息量。

初始时：

```
bufempty[0]=bufempty[1]=n,(n 为缓冲区队列的缓冲区个数)
buffull[0]=buffull[1]=0
send(i,m)
begin
    local x
    P(bufempty[i])
    按 FIFO 方式选择一个空缓冲区
    buf[i](x)
    buf[i](x)=m
    buf[i](x)置满标记
    V(buffull[i])
end
receive(i,m)
begin
    local x
    P(buffull[i])
    按 FIFO 方式选择一个装满数据的缓冲区 buf[i](x)
    m=buf[i](x)
    buf[i](x)置空标记
    V(bufempty[i])
end
```

P_A 调用 send(0,m)和 receive(1,m)。

P_B 调用 send(1,m)和 receive(0,m)。

3.12 在和控制台通信的例中，设操作员不仅回答用户进程所提出的问题，而且还能独立地向各用户进程发出指示。对于这些指示，操作员不要求用户进程回答，但它们享有比其他消息优先传送的优先度。即如果 inbuf 中有指示存在，系统不能进行下一次通信会话。试按上述要求重新描述 CCP、KCP 和 DCP。

答：KCP 描述如下：

设 T_Ready 和 T_Busy 分别为键盘 KP 和键盘控制进程 KCP 的私有信号量，其初值为 0 和 1。设 inbuffer 为 inbuf 的共有信号量，初值为 1，表示其中没有控制消息。

```
初始化{清除所有 inbuf 和 echobuf}
begin
    local x
    P(T_Ready)
    从键盘数据传输缓冲 x 中取出字符 m,记为 x.m
    if 为控制消息
        P(inbuffer)
    send(x.m)
    将 x.m 送入 echobuf
    V(T_Busy)
end
```

键盘控制进程 KCP：

```
repeat
    P(T_Busy)
    把输入字符放入数据传输缓冲
    V(T_Ready)
until 终端关闭
```

显示器控制进程 DCP：

设 D_Ready 和 D_Busy 分别为 DP 和 DCP 的私有信号量，且初始值为 0 和 1。

```
初始化{清除输出缓冲 outbuf,echo 模式置 false}
begin
    if outbuf 满
    then
        receive(k)
        P(D_Busy)
        把 k 送入显示器数据缓冲区
        V(D_Ready)
    else
        echo 模式置 true
        echo buf 中字符置入显示器数据缓冲区 fi
end
```

显示器动作 DP：

```
repeat
    if echo 模式
    then
        打印显示器数据缓冲区中的字符
    else
        P(D_Ready)
        打印显示器数据缓冲区中的消息
        V(D_Busy)
until 显示器关机
```

设过程 read(x)把 inbuf 中的所有字符读到用户进程数据区 x 处，过程 write(y)把用户进程 y 处的消息写到 outbuf 中。read(x)和 write(y)的描述略。

3.13 编写一个程序使用系统调用 fork 生成 3 个子进程，并使用系统调用 pipe 创建一个管道，使得这 3 个子进程和父进程公用同一管道进行信息通信。

答：

```
main()
{
    int r,p1,p2,p3,fd[2];                          /* fd[2]为管道文件读写标识 */
    char buf[50],s[5];
    pipe(fd);                                      /* 创建管道 pipe() */
    while((p1=fork())==-1);                        /* 创建子进程 1 */
    if(p1==0)                                      /* 在子进程 1 中执行 */
    {
        lockf(fd[1],1,0);                          /* 锁定写过程 */
        sprintf(buf,"child process P1 is sending message!\n");
        printf("child process P1!\n");
        write(fd[1],buf,50);                       /* 将 buf 中的数据写入 pipe */
        sleep(5);                                  /* 睡眠等待父进程读出 */
        lockf(fd[1],0,0);                          /* 解锁 */
        exit(0);
    }
    else
    {
        while((p2=fork())==-1);                    /* 创建进程 2 */
        if(p2==0)                                  /* 在子进程 2 中执行 */
        {
            lockf(fd[1],1,0);                      /* 锁定写过程 */
            sprintf(buf, "child process P2 is sending message!\n");
                                                   /* 数据写入 buf */
            printf("child process P2!\n");
            write(fd[1],buf,50);                   /* 将 buf 中的数据写入 pipe */
            sleep(5);                              /* 同步等待父进程读 */
            lockf(fd[1],0,0);                      /* 解锁 */
            exit(0);                               /* 释放进程资源 */
        }
        else
        {
            while((p3=fork())==-1);                /* 创建进程 3 */
            if(p3==0)                              /* 在子进程 3 中执行 */
            {
                lock(fd[1],1,0);
                sprintf(buf,"child process P3 is sending message!\n");
                printf("child process P3!\n");
```

```
            write(fd[1],buf,50);
            sleep(5);
            lockf(fd[1],0,0);
            exit(0);
        }
        wait(0);                                    /* 父进程等待子进程先执行 */
        if(r=read(fd[0],s,50)==-1)                  /* 读管道 pipe 内容到 s 中 */
            printf("can't read pipe\n");
        else
            printf("%s\n",s);
        wait(0);                                    /* 等待另一个子进程执行 */
        if(r=read(fd[0],s,50)==-1)
            printf("can't read pipe\n");
        else
            printf("%s\n",s);
        wait(0);                                    /* 等待最后一个子进程执行 */
        if(r=read(fd[0],s,50)==-1)
            printf("can't read pipe\n")
        else
            printf("%s\n",s);
        exit(0);
    }
  }
}
```

3.14 设有 5 个哲学家,共享一张放有 5 把椅子的桌子,每人分得一把椅子。但是,桌子上总共只有 5 支筷子,在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。条件如下:

(1) 只有拿到两支筷子时,哲学家才能吃饭。

(2) 如果筷子已在他人手上,则该哲学家必须等到他人吃完之后才能拿到筷子。

(3) 任一哲学家在自己未拿到两支筷子吃饭之前,决不放下自己手中的筷子。

试解答以下问题:

(1) 描述一个保证不会出现两个邻座同时要求吃饭的通信算法。

(2) 描述一个既没有两邻座同时吃饭,又没有人饿死(永远拿不到筷子)的算法。在什么情况下,5 个哲学家全部吃不上饭?

答:

(1) 设信号量 c[0]～c[4],初始值均为 1,分别表示 i 号筷子被拿(i=0,1,2,3,4)。

```
send(i): 第 i 个哲学家要吃饭
begin
    P(c[i]);
    P(c[i+1 mod 5]);
    eat;
    V(c[i+1 mod 5]);
```

```
        V(c[i]);
    end
```

该过程能保证两邻座不同时吃饭，但会出现 5 个哲学家一人拿一支筷子，谁也吃不上饭的死锁情况。

(2) 解决的思路如下：让奇数号的哲学家先取右手边的筷子，让偶数号的哲学家先取左手边的筷子。

这样，任何一个哲学家拿到一支筷子以后，就已经阻止了他邻座的一个哲学家吃饭的企图，除非某个哲学家一直吃下去，否则不会有人会饿死。

```
send(i):
begin
    if i mod 2==0
    then
    {
        P(c[i]);
        P(c[i+1] mod 5);
        eat;
        V(c[i]);
        V(c[i+1 mod 5])
    }
    else
    {
        P(c[i+1 mod 5]);
        P(c[i]);
        eat;
        V(c[i+1 mod 5]);
        V(c[i]);
    }
end
```

3.15　什么是线程？试述线程与进程的区别。

答：线程是在进程内用于调度和占有处理机的基本单位，它由线程控制表、存储线程上下文的用户栈以及核心栈组成。线程可分为用户级线程、核心级线程以及用户/核心混合型线程等类型。其中用户级线程在用户态下执行，CPU 调度算法和各线程优先级都由用户设置，与操作系统内核无关。核心级线程的调度算法及线程优先级的控制权在操作系统内核。混合型线程的控制权则在用户和操作系统内核二者。

线程与进程的主要区别如下：

(1) 进程是资源管理的基本单位，它拥有自己的地址空间和各种资源，例如内存空间、外部设备等；线程只是处理机调度的基本单位，它只和其他线程一起共享进程资源，但自己没有任何资源。

(2) 以进程为单位进行处理机切换和调度时，由于涉及资源转移以及现场保护等问题，将导致处理机切换时间变长，资源利用率降低。以线程为单位进行处理机切换和调度时，由

于不发生资源变化，特别是地址空间的变化，处理机切换的时间较短，从而处理机效率较高。

(3) 对用户来说，多线程可减少用户的等待时间，提高系统的响应速度。例如，当一个进程需要对两个不同的服务器进行远程过程调用时，对于无线程系统的操作系统来说，需要顺序等待两个不同调用返回结果后才能继续执行，且在等待中容易发生进程调度。对于多线程系统而言，则可以在同一进程中使用不同的线程同时进行远程过程调用，从而缩短进程的等待时间。

(4) 线程和进程一样，都有自己的状态，也有相应的同步机制。不过，由于线程没有单独的数据和程序空间，因此，线程不能像进程的数据与程序那样交换到外存存储空间，从而线程没有挂起状态。

(5) 进程的调度、同步等控制大多由操作系统内核完成，而线程的控制既可以由操作系统内核进行，也可以由用户控制进行。

3.16 使用库函数 clone()与 pthread_create()，在 Linux 环境下创建两种不同执行模式的线程程序。

答：Linux 系统支持用户级线程和核心级线程两种执行模式，其库函数分别为 pthread_create() 和 clone()。创建用户级线程和核心级线程的程序示例如下。

(1) 用户级线程编程示例：

```
#include <pthread.h>
void *ptest(void *arg)
{
    printf("This is the new thread!\n");
    return(NULL);
}

main()
{
    pthread_t tid;
    printf("This is the parent process!\n");
    pthread_create(&tid,NULL,ptest,NULL);     /*创建线程*/
    sleep(1);
    return;
}
```

该程序通过调用 pthread_create()创建一个用户级线程，其指针为 tid，过程名为 ptest。

(2) 核心级线程编程示例：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/unistd.h>

#define STACKSIZE 16384
```

```
#define CSIGNAL 0x000000ff            /* signal mask to be sent at exit */
#define CLONE_VM 0x00000100           /* set if VM shared between processes */
#define CLONE_FS 0x00000200           /* set if info shared between processes */
#define CLONE_FILES 0x00000400        /* set if files shared between processes */
#define CLONE_SIGHAND 0x00000800      /* set if signal handlers shared between
                                         processes */

int show_same_vm;

void cloned_process_start_here(void * data)
{
    printf("child:\t got argument %d as fd\n", (int) data);
    show_same_vm=5;
    printf ("child: \t vm=%d \n", show_same_vm);
    close((int)data);
}

int main()
{
    int fd, pid;

    fd=open (" /dev/null", O_RDWR);
    if (fd<0)
    {
        perror("/dev/null");
        exit(1);
    }
    printf ("mother: \t vm=%d\n",fd);

    show_same_vm=10;
    printf ("mother: \t vm=%d\n", show_same_vm);

    pid=clone (cloned_process_start_here, (void * ) fd);
    if (pid<0)
    {
        perror ("start_thread");
        exit(1);
    }
    sleep(1);
    printf ("mother: \t vm=%d\n", show_same_vm);
    if (write (fd, "c", 1) <0)
        printf ("mother: \t child closed our file descriptor\n");
}

#include <signal.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/unistd.h>

#define STACKSIZE 16384
#define CSIGNAL 0x000000ff              /* signal mask to be sent at exit */
#define CLONE_VM 0x00000100             /* set if VM shared between processes */
#define CLONE_FS 0x00000200             /* set if info shared between processes */
#define CLONE_FILES 0x00000400          /* set if files shared between processes */
#define CLONE_SIGHAND 0x00000800
                                /* set if signal handlers shared between processes */

int clone (void (* fn) (void *), void * data)      /* 创建核心线程 */
{
    long retval,errno;
    void **newstack;
    /*
     * allocate new stack for subthread
     */
newstack=(void **) malloc (STACKSIZE);
if (!newstack)
    return -1;
/*
 * Set up the stack for child function, put the (void *)
 * argument on the stack
 */
newstack=(void **) (STACKSIZE+(char *) newstack);
*newstack=data;

/*
 * Do clone () system call. We need to do the low-level stuff
 * entirely in assembly as we're returning with a different
 * stack in the child process and we couldn't otherwise guarantee
 * that the program doesn't use the old stack incorrectly.
 *
 * Parameters to clone() system call:
 *%eax-_NR_clone, clone system call number
 *%ebx-clone_flags, bitmap of cloned data
 *%ecx-new stack pointer for cloned child
 *
 * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
 * CLONE_SIGHAND which shares as much as possible between parent and
 * child. (the signal to be sent on child termination into clone_flags:
 * SIGCHLD makes the cloned process work like a "normal" UNIX child
```

```
 * process)
 *
 * The clone () system call returns (in %eax) the pid of the newly
 * cloned process to the parent, and 0 to the cloned process. If
 * an error occurs, the return value will be the negative errno.
 * In the child process, we will do a "jsr" to the requested function
 * and then do a "exit() " system call which will terminate the child.
 */
_asm_volatile_(
    "int $0x80\n\t"             /* Linux/i386 system call */
    "testl %0, %0\n\t"          /* check return value */
    "jne lf\n\t"                /* jump if parent */
    "call *%3\n\t"              /* start subthread function */
    "movl %2, %0\n\t"
    "int $0x80\n"               /* exit system call: exit subthread */
    "l: \t"
    : "=a" (retval)
    : "0" (_NR_clone), "i" (_NR_exit),
    "r" (fn),
    "b" (CLONE_VM | CLONE_FS | CLONE_FILES |
    CLONE_SIGHAND | CLONE_SIGCHLD),
    "c" (newstack));

    if (retval < 0)
    {
        errno=-retval;
        retval=-1;
    }
    return retval;
}
```

第 4 章　处理机调度

4.1　什么是分级调度？分时系统中有作业调度的概念吗？如果没有，为什么？

答：处理机调度问题实际上也是处理机的分配问题。显然只有那些参与竞争处理及所必需的资源都已得到满足的进程才能享有竞争处理机的资格。这时它们处于内存就绪状态。这些必需的资源包括内存、外设及有关数据结构等。从而，在进程有资格竞争处理机之前，作业调度程序必须先调用存储管理和外设管理程序，并按一定的选择顺序和策略从输入井中选择出几个处于后备状态的作业，为它们分配资源和创建进程，使它们获得竞争处理机的资格。另外，由于处于执行状态下的作业一般包括多个进程，而在单机系统中，每一时刻只能有一个进程占有处理机，这样，在外存中，除了处于后备状态的作业外，还存在处于就绪状态而等待得到内存的作业。需要有一定的方法和策略为这部分作业分配空间。因此处理机调度需要分级。

一般来说，处理机调度可分为 4 级：

(1) 作业调度。又称宏观调度或高级调度。

(2) 交换调度。又称中级调度。其主要任务是按照给定的原则和策略，将处于外存交换区中的就绪状态或等待状态或内存等待状态的进程交换到外存交换区。交换调度主要涉及内存管理与扩充，因此在有些书本中也把它归入内存管理部分。

(3) 进程调度。又称微观调度或低级调度。其主要任务是按照某种策略和方法选取一个处于就绪状态的进程占用处理机。在确立了占用处理机的进程之后，系统必须进行进程上下文切换以建立与占用处理机进程相适应的执行环境。

(4) 线程调度。进程中相关堆栈和控制表等的调度。

在分时系统中，一般不存在作业调度，而只有线程调度、进程调度和交换调度。这是因为在分时系统中，为了缩短响应时间，作业不是建立在外存中，而是直接建立在内存中。在分时系统中，一旦用户和系统的交互开始，用户马上要进行控制。因此，分时系统中没有作业提交状态和后备状态。分时系统的输入信息经过终端缓冲区为系统直接接收，或立即处理，或经交换调度暂存外存中。

4.2　试述作业调度的主要功能。

答：作业调度的主要功能是：按一定的原则对外存输入井中的大量后备作业进行选择，给选出的作业分配内存、输入输出设备等必要的资源，并建立相应进程，使该作业的相关进程获得竞争处理机的权利。另外，当作业执行完毕时，还负责回收系统资源。

4.3　作业调度的性能评价标准有哪些？这些性能评价标准在任何情况下都能反映调度策略的优劣吗？

答：对于批处理系统，由于主要用于计算，因而对于作业的周转时间要求较高，从而作业的平均周转时间或平均带权周转时间被用来衡量调度程序的优劣。但对于分时系统来

说，平均响应时间又被用来衡量调度策略的优劣。

对于分时系统，除了要保证系统吞吐量大、资源利用率高之外，还应保证用户能够容忍的响应时间。因此，在分时系统中，仅仅用周转时间或带权周转时间来衡量调度性能是不够的。

对于实时系统来说，衡量调度算法优劣的主要标志则是满足用户要求的时限时间。

4.4　进程调度的功能有哪些？

答：进程调度的功能如下：

(1) 记录和保存系统中所有进程的执行情况。

(2) 选择占有处理机的进程。

(3) 进行进程上下文切换。

4.5　进程调度的时机有哪几种？

答：进程调度的时机有：

(1) 正在执行的进程执行完毕。这时如果不选择新的就绪进程执行，将浪费处理机资源。

(2) 执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等待状态。

(3) 执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 V 原语操作激活了等待资源的进程队列。

(4) 执行中进程提出 I/O 请求后被阻塞。

(5) 在分时系统中时间片已经用完。

(6) 在执行完系统调用等系统程序后返回用户程序时，可看做系统进程执行完毕，从而调度选择一个新的用户进程执行。

在 CPU 执行方式是可剥夺时，还有：

(7) 就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

4.6　假设有 4 道作业，它们的提交时间及执行时间由下表给出。

作业号	提交时刻/hh:mm	执行时间/hr
1	10:00	2
2	10:20	1
3	10:40	0.5
4	10:50	0.3

计算在单道程序环境下，采用先来先服务调度算法和最短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

答：(1) 先来先服务调度顺序如下：

① $Ts_1=10:00$　$Te_1=12:00$　$T_1=2.00$　$Tw_1=0$

② $Ts_2=10:20$　$Te_2=13:00$　$T_2=1.00$　$Tw_2\approx1.70$

③ $Ts_3=10:40$　$Te_3=13:30$　$T_3=0.50$　$Tw_3\approx2.30$

④ $Ts_4=10:50$　$Te_4=13:48$　$T_4=0.30$　$Tw_4\approx2.70$

$T=0.25*(2+2.7+2.8+3)=2.625$

$W=0.25*(4+0+1.7/1+2.3/0.5+2.7/0.3)=4.825$

(2) 最短作业优先调度顺序如下：

① $Ts_4=10:50$　$Te_4=11:08$　$T_4=0.3$　$Tw_4=0$

② $Ts_3=10:40$　$Te_3=11:28$　$T_3=0.5$　$Tw_3=0.3$

③ $Ts_2=10:20$　$Te_2=12:08$　$T_2=1$　$Tw_2=0.8$

④ $Ts_1=10:00$　$Te_1=13:48$　$T_1=2$　$Tw_1=1.8$

$T=0.25*(0.3+0.8+1.8+3.8)=1.675$

$W=0.25*(4+0+0.30/0.50+0.8/1+1.8/2)=1.575$

4.7　设某进程所需要的服务时间 $t=k*q$，其中，k 为时间片的个数，q 为时间片长度且为常数。当 t 为一定值时，令 q 趋于 0，则有 k 趋于无穷，从而服务时间为 t 的进程响应时间 T 为 t 的连续函数。对应于时间片的轮转调度方式(RR)、先来先服务方式(FCFS)和线性优先级调度方式(SRR)，其响应时间函数分别为：

$$T_{rr}(t)=t*\mu/(\mu-\lambda)$$

$$T_{fc}(t)=1/(\mu-\lambda)$$

$$T_{sr}(t)=1/(\mu-\lambda)-(1-t*\mu)/(\mu-\lambda')$$

其中 $\lambda'=(1-b/a)*\lambda=r*\lambda$。

取$(\lambda,\mu)=(50,100)$和$(\lambda,\mu)=(80,100)$，分别改变 r 的值，画出 $T_{rr}(t)$、$T_{fc}(t)$和 $T_{sr}(t)$的时间变化图。

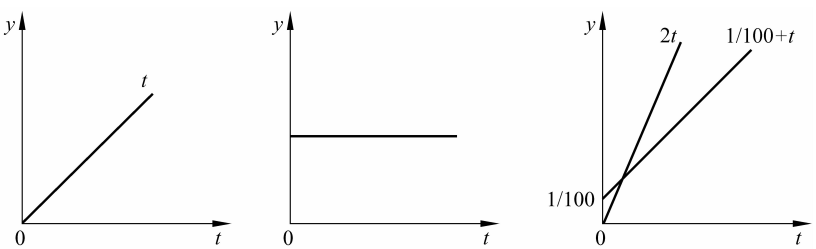
答：(1) 对$(\lambda,\mu)=(50,100)$，有

$$T_{rr}(t)=t,T_{fc}(t)=t/50,T_{sr}(t)=1/50-(1-100t)/(100-50r)$$

$r\to0$ 时，$T_{sr}(t)=1/100+t$；

$r\to1$ 时，$T_{sr}(t)=2t$。

时间变化图如下图所示。



(a) $T_{rr}(t)$　(b) $T_{fc}(t)$　(c) $T_{sr}(t)$

只有 $T_{sr}(t)$受 r 值影响。且 r 增大时，$T_{sr}(t)$斜率增大，服务时间也增加。

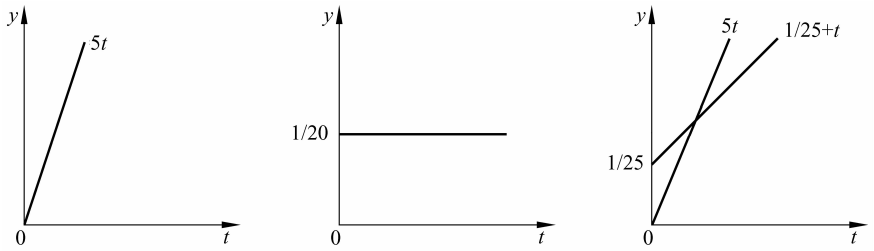
(2) 对$(\lambda,\mu)=(80,100)$，有

$$T_{rr}(t)=5t,T_{fc}(t)=1/20,\ T_{sr}(t)=1/20-(1-100t)/(100-80r)$$

$r\to0$ 时，$T_{sr}(t)\to1/25+t$；

$r \to 1$ 时，$T_{sr}(t) \to 5t$。

时间变化图如下图所示。



(a) $T_{rr}(t)$　　(b) $T_{fc}(t)$　　(c) $T_{sr}(t)$

$T_{sr}(t)$的斜率随 r 增大而增大，y 截距由 1/25 到 0 逐渐移动。

4.8　什么是多处理机系统？并行处理系统、计算机网络、分布式系统和多处理机系统的操作系统之间有何区别？

答：从广义上说，使用多台处理机协调工作来完成用户所要求任务的计算机系统都是多处理机系统。狭义的多处理机系统是利用系统内的多个 CPU 来并行执行用户的几个程序，以提高系统的吞吐量；或用来进行冗余操作，以提高系统的可靠性。

并行处理机系统是利用多个功能单元(CPU)执行同一程序，多个处理机在物理位置上处于同一块电路板上。计算机网络系统则是通过物理通信媒介，包括有线和无线的，把现有的分散的计算机系统互相连接起来，以达到信息传递和资源共享的目的。分布式系统是以计算机网络为基础的，对用户来说是透明的。多处理机系统是指在同一计算机系统内共享内存的计算机系统。

4.9　什么是实时调度？它与非实时调度有何区别？

答：实时调度是为了完成实时处理任务而分配计算机处理器的调度方法。

实时处理任务要求计算机在用户允许的时限范围内给出计算机响应信号。实时处理任务可分为硬实时任务(hard real-time task)和软实时任务(soft real-time task)。硬实时任务要求计算机系统必须在用户给定的时限内处理完毕，软实时任务允许计算机系统在用户给定的时限左右处理完毕。

针对硬实时任务和软实时任务，计算机系统可以有不同的实时调度算法。这些算法采用基于优先级的抢先式调度策略，具体地说，大致有如下几类：

(1) 静态表驱动模式。该模式用于周期性实时调度，它在任务到达之前对各任务抢占处理机的时间进行分析，并根据分析结果进行调度。

(2) 静态优先级驱动的抢先式调度模式。该模式也进行静态分析。分析结果不是用于调度，只是用于给各任务指定优先级。系统根据各任务的优先级进行抢先式调度。

(3) 基于计划的动态模式。该模式在新任务到达后，将以前调度过的任务与新到达的任务一起统一计划，分配 CPU 时间。

(4) 动态尽力而为模式。该模式不进行任何关于资源利用率的分析，只检查各任务的时限是否能得到满足。

代表性的实时调度算法有两种，即时限式调度法(deadline scheduling)和频率单调调度法(rate monotonic scheduling)。

实时调度与非实时调度的主要区别如下：

(1) 实时调度所调度的任务有完成时限，而非实时调度没有。因此，实时调度算法的正确与否不仅与算法的逻辑有关，也与调度算法调度的时限有关。

(2) 实时调度要求较短的进程或线程切换时间，而非实时调度的进程或线程的切换时间较长。

(3) 非实时调度强调资源利用率(批处理系统)或用户共享处理机(分时系统)，实时调度则主要强调在规定时限范围内完成对相应设备的控制。

(4) 实时调度为抢先式调度，而非实时调度则很少采用抢先式调度。

4.10　写出图 4.11 所示周期性任务调度用的时限调度算法。

答：首先设置周期性任务进程的数据结构：

```
process struct
{
    int p_num;                  /* 进程号 */
    int arr_time;               /* 进程到达时间 */
    int exe_time;               /* 进程所需执行时间 */
    int end_deadline;           /* 时限 */
    int period;                 /* 周期 */
    int passed_time;            /* 该进程已占有处理机时间 */
} pro[N],pro1[n];
local int N,n,T1,T2;            /* N,n 为正整数,T1,T2 为进程周期 */
```

算法描述如下：

```
begin
    if {n 个进程 pro1[n]到达,且要求调度}
    then {
        初始化 pro1[n]
        和 T1,T2 分别比较 pro1[n].period
        if {pro1[n].period ≠ T1 且 pro1[n].period ≠ T2}
        then {返回错误信息：该进程周期错误}
        else {    比较 pro1[n]与 pro[N]中的各进程时限
                  选择 pro1[n].end_deadline 或 pro[N].end_deadline 中时间最近者占据
                  处理机
                  保护当前进程现场,并将其放入 pro[N]队列
                  将未选中的 pro1[n]的其他进程置入 pro[N]
             }
    }
    else
        if 当前进程执行完毕要求调度
        {
```

```
            比较 pro[N]中每个进程的 end_deadline
            选择 end_deadline 最小的进程占有处理机
        }
        else
            等待占有处理机的进程执行完毕
end
```

4.11　设周期性任务 P_1、P_2、P_3的周期 T_1、T_2、T_3分别为 100、150、350；执行时间分别为 20、40、100。问：是否可用频率单调调度算法进行调度？

答：根据频率单调调度公式，能进行周期性调度的进程应满足下式：

$$c_1/T_1+c_2/T_2+\cdots+c_n/T_n\leqslant n(2^{1/n}-1)$$

在上式中，$n=3$，$T_1=100$，$T_2=150$，$T_3=350$，而 c_1、c_2、c_3 分别等于 20、40、100，从而有

$$20/100+40/150+100/350=0.2+0.267+0.286=0.753 \quad (1)$$

而

$$3\times(2^{1/3}-1)=0.779 \quad (2)$$

比较式(1)与式(2)，有 (1)≤(2)。因此，本题给出的周期性任务可以用频率单调调度算法进行调度。