**武汉大学计算机学院**

**本科生实验报告**

**操作系统总体设计与实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统设计

指 导 教 师 ：曾平 副教授

学 生 学 号 ：2016300030043

学 生 姓 名 ：彭凯飞

二○一八年四月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 日期：

摘 要

本次操作系统设计实验的实验目的是：加深了解处理器调度的工作；理解在不同的存储管理方式下应怎样进行存储空间的分配和回收；掌握磁盘存储空间的分配和回收算法；模拟进程的并发执行，了解进程并发执行时同步机构的作用。

实验设计主要遵循上学期操作系统概论课程中所学知识与实验说明文档所指明的原则。

实验内容主要包括：实现处理器优先级调度算法；利用位示图实现主存空间的分配和回收；利用的空闲块成组链接法实现磁盘存储空间的分配和回收；通过对生产者-消费者问题的模拟实现进程同步机构。

**关键词：**操作系统；处理器调度；存储空间管理；进程同步

**目 录**

**1** 处理器调度：按优先数调度算法实现处理器调度的程序

1.1 实习内容及上机实验所用平台 ……………………………………………………6

1.2 数据结构及代码段分析………………………………………………………………6

1.3 调试过程…………………………………………………………………………………8

1.4 总结………………………………………………………………………………………8

**2** 主存空间的分配与回收：在分页管理方式下采用位示图来表示主存分配情况

2.1 实习内容及上机实验所用平台 ……………………………………………………10

2.2 数据结构及代码段分析………………………………………………………………10

2.3 调试过程…………………………………………………………………………………13

2.4 总结………………………………………………………………………………………16

**3** 磁盘存储空间的分配和回收：模拟UNIX系统的空闲块成组链接法

3.1 实习内容及上机实验所用平台 ……………………………………………………17

3.2 数据结构及代码段分析………………………………………………………………18

3.3 调试过程…………………………………………………………………………………21

3.4 总结………………………………………………………………………………………24

**4** 进程同步：模拟P、V操作实现同步机构、用P、V操作解决生产者-消费者问题（选做题）

4.1 实习内容及上机实验所用平台………………………………………………………25

4.2 数据结构及代码段分析………………………………………………………………29

4.3 调试过程…………………………………………………………………………………31

4.4 总结………………………………………………………………………………………34

**附录……………………………………………………………………35**

**1处理器调度：按优先数调度算法实现处理器调度的程序**

**1.1 实习内容及上机实验所用平台**

本实验模拟在单处理器环境下的处理器调度，加深对处理器调度的工作的了解。

**1．1．1** **实习内容：选择一个调度算法，实现处理器调度**

设计一个按优先数调度算法实现处理器调度的程序。

**1．1．2** **设计思路**

(1)假定系统有5个进程，每个进程用一个PCB来代表。PCB的结构为：

·进程名——如P1~P5。

·指针——按优先数的大小把5个进程连成队列，用指针指出下一个进程PCB的首地址。

·要求运行时间——假设进程需要运行的单位时间数。

·优先数——赋予进程的优先数，调度时总是选取优先数大的进程先执行。

·状态——假设两种状态：就绪和结束，用R表示就绪，用E表示结束。初始状态都为就绪状态。

(2) 开始运行之前，为每个进程确定它的“优先数”和“要求运行时间”。通过键盘输入这些参数。

(3) 处理器总是选择队首进程运行。采用动态改变优先数的办法，进程每运行1次，优先数减1，要求运行时间减1。

(4) 进程运行一次后，若要求运行时间不等于0，则将它加入就绪队列，否则，将状态改为“结束”，退出就绪队列。

(5) 若就绪队列为空，结束，否则转到(3)重复。

**1．1．3** **上机实验所用平台**

模拟算法所用语言：c语言。编辑器为Dev C++。

**1.2数据结构及代码段分析**

**1．2．1** **部分主要数据结构**

typedef struct PCB {

unsigned pid; //进程编号（代替进程名）

struct PCB \*next; //指向该进程所在队列中下一个PCB的指针

unsigned remain\_time; //该进程剩余的执行时间

unsigned priority; //该进程当前的优先级

int status; //该进程当前的状态（running, ready, terminated）

} PCB;

PCB \*ready\_queue\_head; //就绪进程队列

PCB \*terminated\_queue\_head; //终止运行进程队列

**1．2．2** **主要代码段分析**

int insertProc(PCB \*new\_proc, PCB \*queue\_head)

{

//insert to right position according to priority(decreasing)

PCB \*p, \*pre;

pre = queue\_head;

p = pre->next;

while (p!=NULL)

{

if (new\_proc->priority > p->priority)

break;

pre = p;

p = p->next;

}

new\_proc->next = pre->next;

pre->next = new\_proc;

return 1;

}

上述函数将一个进程的PCB插入就绪或终止运行进程队列中。p为一临时PCB指针，从队首向队尾移动。当找到第一个比当前要插入队列中的PCB优先级低的队列中PCB时，将前者插入到后者前面的位置。这样一来，就绪队列中PCB按优先级降序排列，保证队首为优先级最高的就绪进程。处理器调度时，只需取出就绪队列首的进程运行即可。

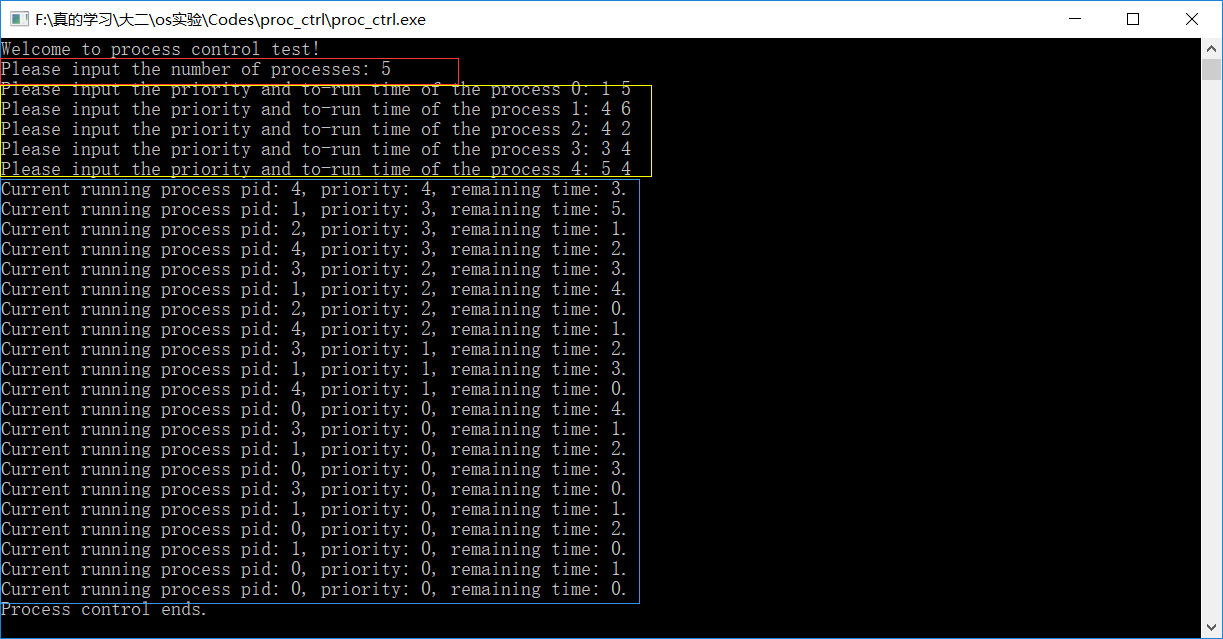
运行main函数主体为：

while (runProc(ready\_queue\_head, terminated\_queue\_head));

runProc(PCB\* , PCB\*, )模拟运行就绪队列首的进程。每调用一次，先检查就绪队列首有没有就绪进程。若无返回0，跳出循环；若有返回1，继续运行进程。被运行的进程的priority和remain\_time均减1，然后判断该进程剩余运行时间是否为0。若为0，将该进程加入终止队列；若不为0，将该进程重新用insertProc函数加入就绪队列。

**1.3 调试过程**

程序运行截图如下：



上图中，红色部分为用户输入要运行的进程数，本例中为5。黄色部分为用户输入各个进程的优先级和运行时间。

蓝色部分为运行结果。由图可知，在最开始，进程4优先级最高，所以首先运行进程4，其优先级由5降为4，剩余运行时间由4降为3。紧随其后，因为进程1，2与4优先级均为最高值4，在此先运行进程1，再运行进程2，再运行进程4，三者优先级均降为3，剩余时间均减1。如此反复，每次运行优先级最高的进程，最后输出所有的运行过程如上。

**1.4 总结**

编写按优先数调度算法实现处理器调度的程序，模拟处理器调度的情况，加深了对处理器调度各作业工作的理解。

该实验的关键是将就绪队列中的PCB按优先级降序排列。在编写程序时，思路不应该是对就绪队列进行sort排序，而是在每次插入PCB时就将该PCB插入正确的位置，其前的PCB优先级大于或等于该PCB优先级，其后的PCB优先级小于该PCB优先级。实现时要注意大于、等于、小于各判断条件的正确使用。另外，在找插入新PCB的位置时，要注意遍历指针当前的位置是否为NULL，避免bug。

另外需要注意的一个细节是进程优先级应为非负数。当一个进程的优先级已经降为0后，若该进程仍需运行，在以后运行时该进程优先级不应再下降，而应保持为0。

**2主存空间的分配与回收：在分页管理方式下采用位示图来表示主存分配情况**

**2.1 实习内容及上机实验所用平台**

通过本实习帮助，理解在不同的存储管理方式下应怎样进行存储空间的分配和回收。

**2．1．1** **实习内容：主存储器空间的分配和回收**

在分页管理方式下采用位示图来表示主存分配情况，实现主存分配和回收。

**2．1．2** **设计思路**

（1）假定系统的主存被分成大小相等的64个块，用0/1对应空闲/占用。

（2）当要装入一个作业时，根据作业对主存的需求量，先查空闲块数是否能满足作业要求，若能满足，则查位示图，修改位示图和空闲块数。位置与块号的对应关系为：

块号=j\*8+i，其中i表示位，j表示字节。

根据分配的块号建立页表。页表包括两项：页号和块号。

（3）回收时，修改位示图和空闲块数。

**2．1．3** **上机实验所用平台**

模拟算法所用语言：c语言。编辑器为Dev C++。

**2.2数据结构及代码段分析**

**2．2．1** **部分主要数据结构**

typedef struct PageEntry

{

int page\_num;

int block\_num;

PageEntry;

}

页表项数据结构定义如上。每个页表项包括两个元素：页号与对应的内存块号。

typedef struct PCB

{

unsigned pid;

unsigned mem\_size;

PageEntry\* pagetable;

struct PCB\* next;

} PCB;

PCB数据结构如上。每个PCB包含如下信息：进程号；进程所需内存大小；进程页表起始地址；PCB所在列表的下一个PCB的地址。pagetable项所指的地址为该进程页表的起始地址，从该地址开始，连续存储了mem\_size个PageEntry结构，构成该进程的页表。当需要查询该进程的页表时，用指针从pagetable处开始，不断增加，遍历所有页表项即可。

typedef struct Bitmap

{

char data[MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1)];

} Bitmap;

位示图结构如上。用char数组表示。当需要1-8位时，位示图只需一个char（1字节）的内存单元；需要9-15位时，需2个char（2字节）的内存单元……依次类推。

**2．2．2** **主要代码段分析**

int setBitmap0(Bitmap\* bitmap, int block\_id)

{

int index = block\_id/8;

int offset = block\_id%8;

char temp = bitmap->data[index];

temp = (temp & ~(1<<offset));

bitmap->data[index] = temp;

return 1;

}

setBitmap0函数：将内存中第block\_id个内存块在位示图中所对应的位置为0（即释放该内存块）。Bitmap的实质是一个char[8]数组。找到第block\_id/8个char，将该char的第block\_id%8位置为0即可。

int\* setBitmap1(Bitmap\* bitmap, int size)

{

/\*set lowest available bit to 1\*/

int i, j, k=0, n=MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1);

int \*rtr = (int\* )malloc(size\*sizeof(int));

char mask;

char temp;

for (i=0; i<n; i++)

{

mask = 1;

temp = bitmap->data[i];

for (j=0; j<8; j++)

{

if ((int)(temp & mask) == 0) // when bit == 0

{

temp = temp | mask; // set the bit to 1

rtr[k++] = 8\*i + j;

if (k>=size) break;

}

mask = mask << 1;

}

bitmap->data[i] = temp;

if (k>=size) break;

}

return rtr; // block\_num array

}

setBitmap1函数：将位示图低位的size个为0的位置为1。对位示图Bitmap（实质为char[8]数组）的每个char，即首先从char[0]开始到char[7]的每个char做如下操作：从第0位开始，检查其是否为0。若为0，置其为1，并将该位所对应的块号记录在rtr数组中。之后对第1，2……，7位做同样操作，直到该char被检查完。每次置某一位为1时，检查已经被置1的总共位数是否已达到size。若是，从该函数返回，返回值为包含该次内存分配所有被分配块号的rtr数组，便于建立页表。

建立页表的代码段如下：

int\* blocks;

PageEntry\* pagetable = (PageEntry\* ) malloc (mem\_size \* sizeof(PageEntry));

new\_proc->pagetable = pagetable;

blocks = setBitmap1(bitmap, mem\_size); // set bitmap

for (i=0; i<mem\_size; i++) //set pagetable

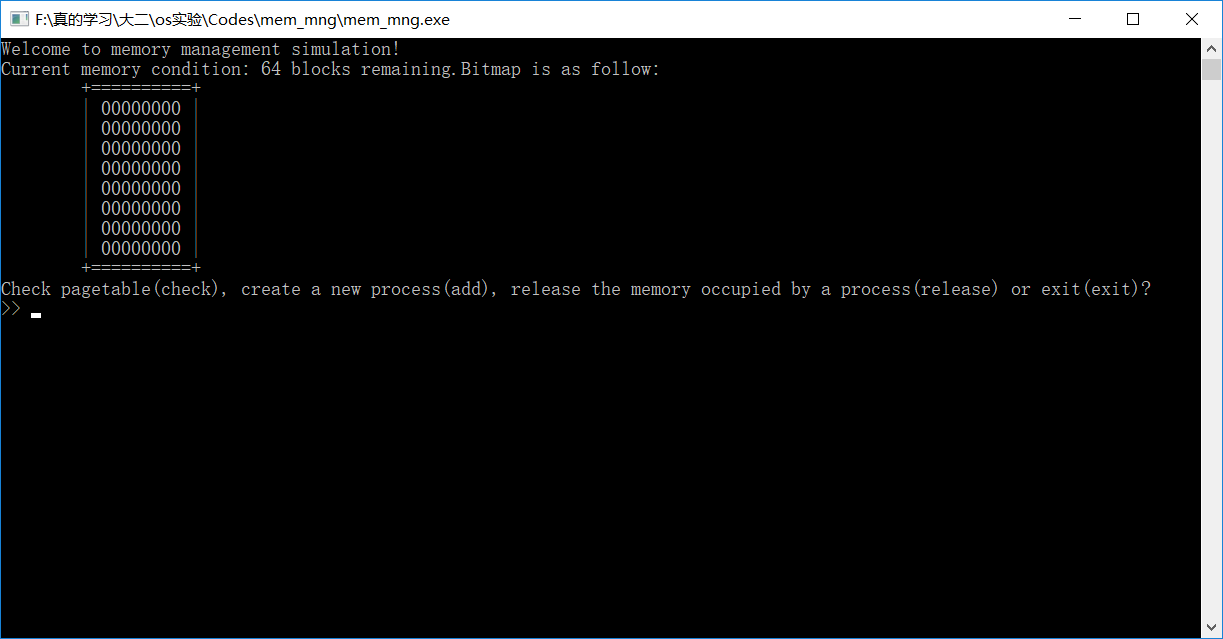
setPageEntry(pagetable++, i, blocks[i]);

free(blocks);

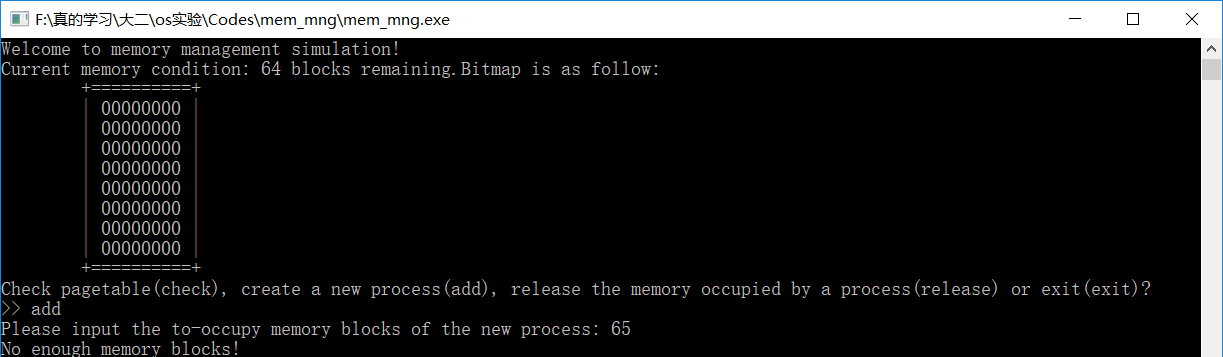
其中new\_proc为该次请求内存分配的进程的PCB。

**2.3 调试过程**

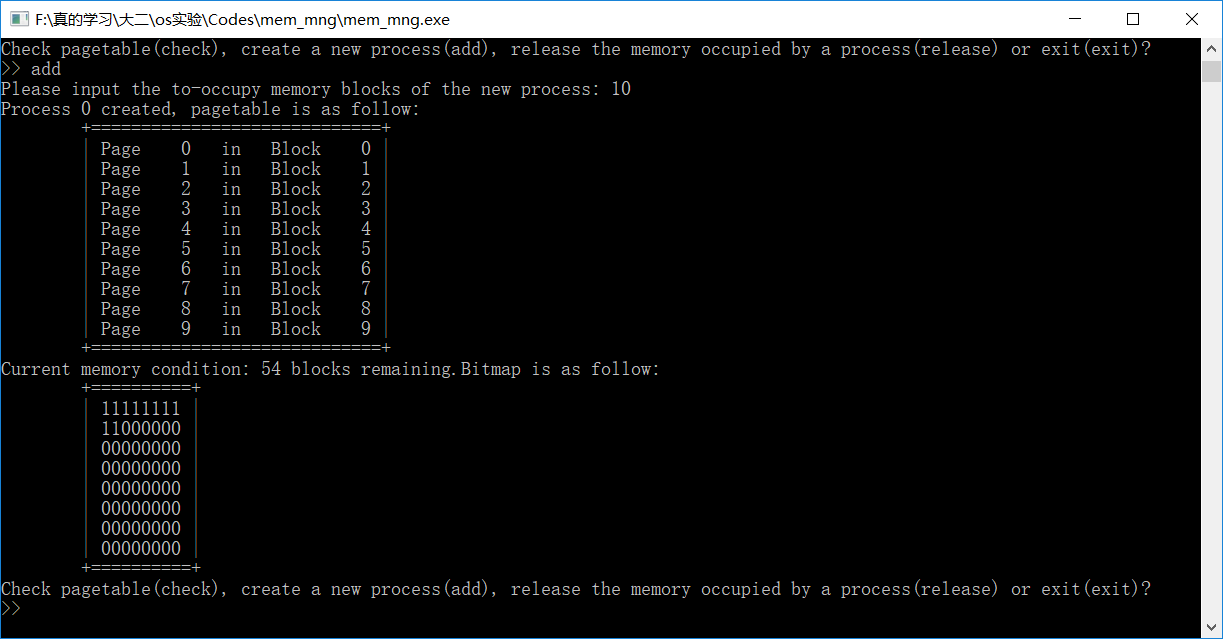
进入程序，初始界面截图如下：



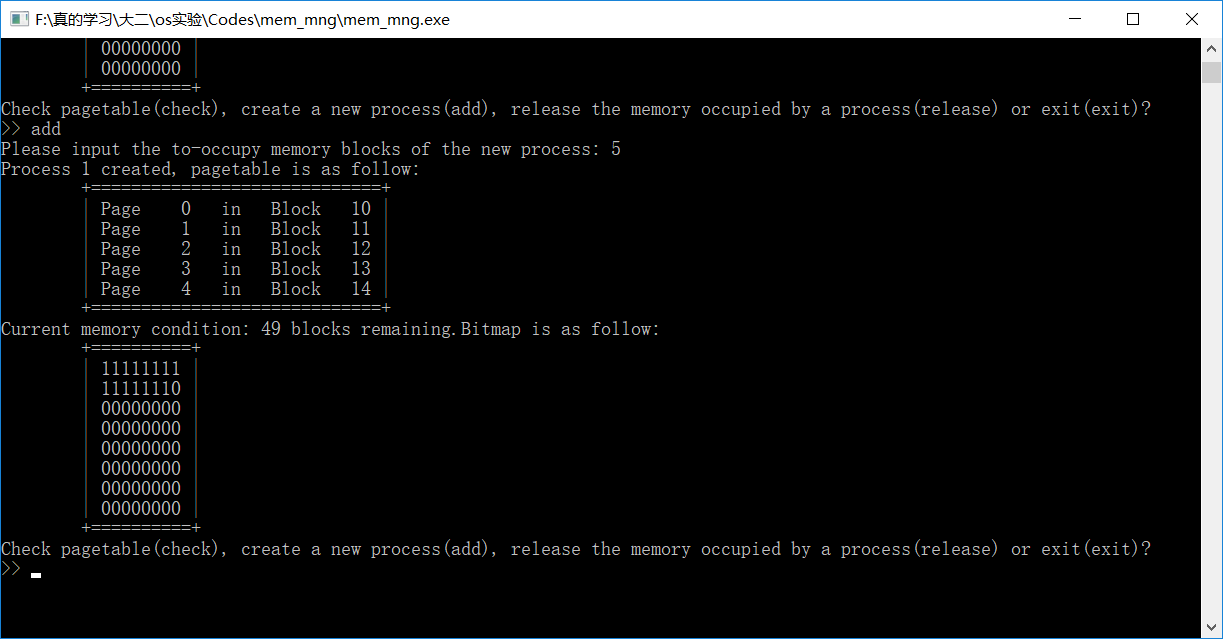
64个内存块均未被分配，位示图为64个0。>>后等待用户输入命令。输入add命令，模拟进程号为0的进程请求内存分配的情况：

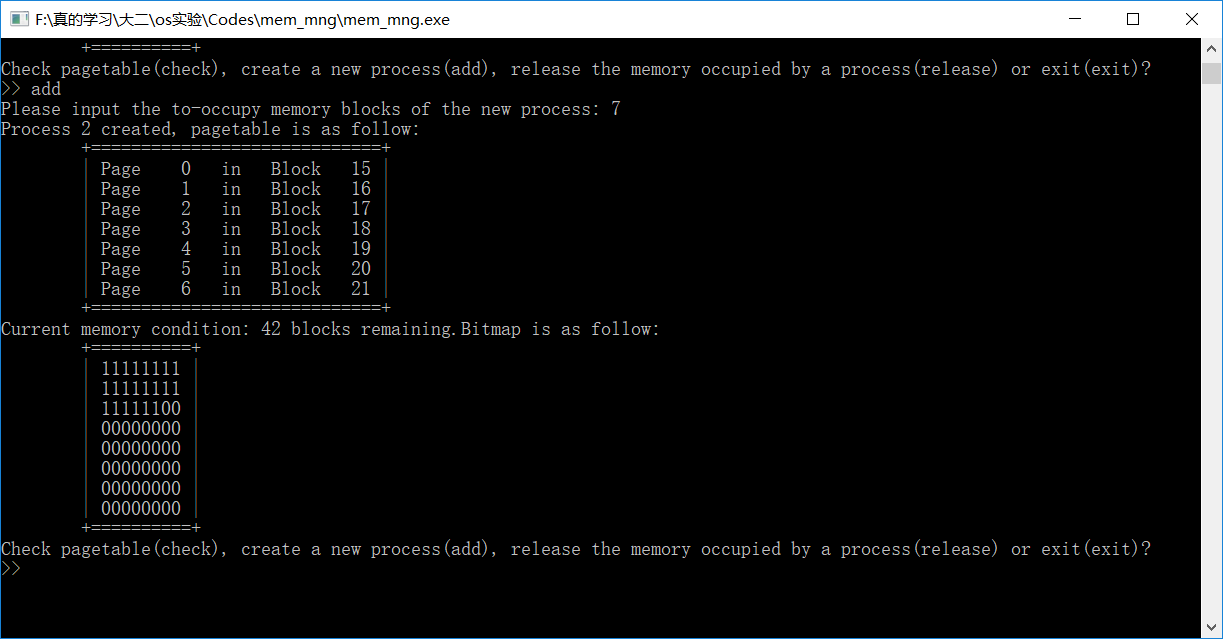


首先尝试分配65个内存块。由于超出剩余内存大小，输出报错信息：No enough memory blocks!

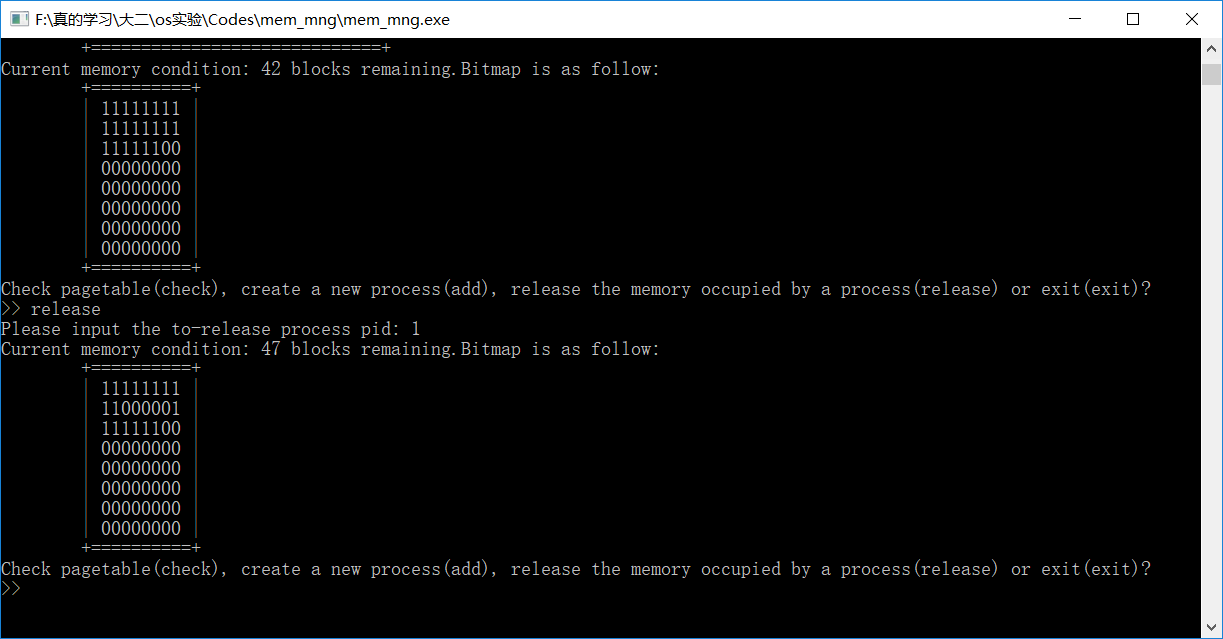


重新add，分配10个内存块。分配成功，第0号进程页表与位示图如上。再次给第1，2号进程分别分配5，7个内存块，运行情况如下：

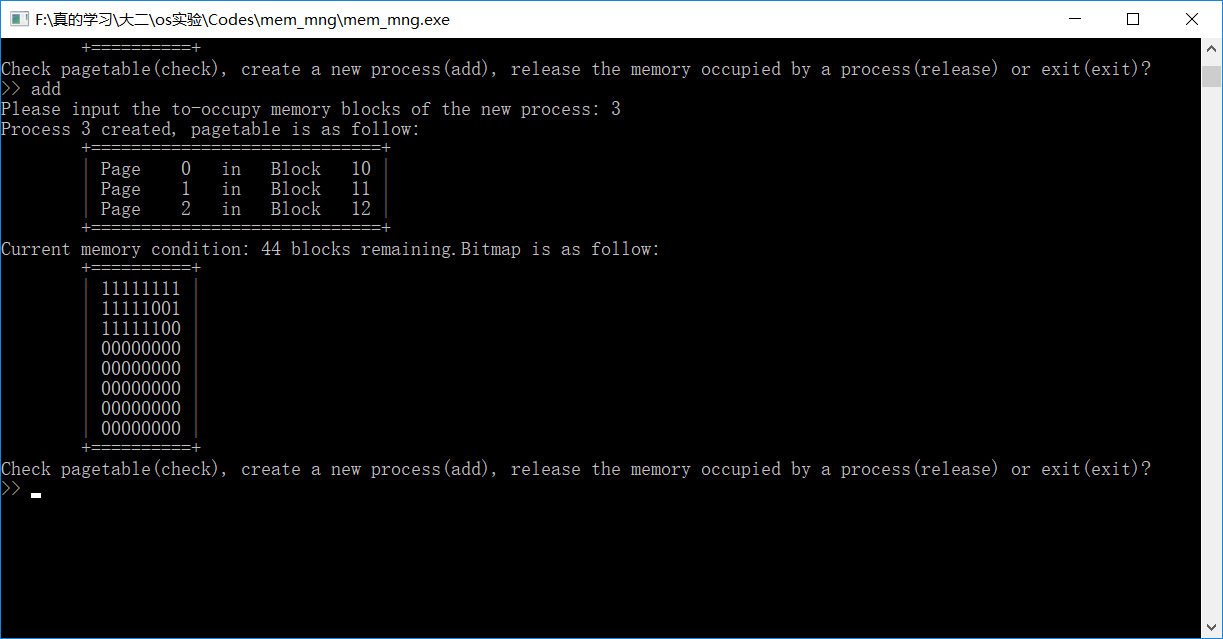




输入release命令，释放第1号进程的5个内存块：



可以看到，位示图中对应于原第1号进程占用的内存块的位均被置为0。再重新分配给第三号进程3个内存块：



填补了中间空着的5个0位的三位。

**2.4 总结**

编写在分页管理方式下采用位示图来表示主存分配情况，实现主存分配和回收的程序，模拟内存管理，加深了对操作系统内存管理的理解。

本实习的关键是setBitmap0()操作和setBitmap1()操作与建立也标操作。setBitmap0()操作较为简单，只需给出块号，在位示图中找到对应的位，将其置0即可。setBitmap1()操作较为复杂，因为需要置1的位并没有直接给出，而是需要去找可用的值为0的位，将它们置为1。找到这些位需要遍历Bitmap中各个char字节，并对每个字节遍历8位，需要用到移位与位运算操作，较为复杂。

建立页表的关键是给出每一页在内存中的块号。由于setBitmap1()与建立页表是两个连续的操作，且setBitmap1()时已经遍历过Bitmap，我们可以让setBitmap1()记住分配的块号（对应Bitmap中被置1的位号），将这些号码作为建立页表操作的输入即可。

**3磁盘存储空间的分配和回收：模拟UNIX系统的空闲块成组链接法**

**3.1 实习内容及上机实验所用平台**

怎样有效地管理磁盘存储空间是操作系统应解决的一个重要问题，通过本实习使学生掌握磁盘存储空间的分配和回收算法。

**3．1．1** **实习内容：磁盘存储空间的分配和回收**

模拟UNIX系统的空闲块成组链接法，实现磁盘空间的分配和回收。

**3．1．2** **设计思路**

(1) 假定磁盘存储空间已被划分成长度为n的等长块，共有M块可供使用。UNIX系统中采用空闲块成组链接的方法来管理磁盘存储空间，将磁盘中的每N个空闲块（N<M）分成一组，最后一组可以不足N块，每组的第一块中登记了下一组空闲块的块数和块号，第一组的块数和块号登记在专用块中，登记的格式如下：

|  |  |
| --- | --- |
| 0 | 空闲块数k |
| 1 | 空闲块号1 |
| 2 | 空闲块号2 |
| … | … |
| K | 空闲块号k |
|  |  |

当第一项内容为“0”时，则第二项起指出的空闲块是最后一组。

（2）可用二维数组A [0…M-1] [0…n-1]来模拟管理磁盘空间，用A[i]表示第I块，第0块A[0]作为专用块。

(3) 成组链接的分组情况记录在磁盘物理块中，为了查找链接情况，必须把它们读入主存，故当磁盘初始化后，系统先将专用块内容复制到主存中。定义一个数组MA存放专用块内容，即MA: =A[0]。申请一块磁盘空间时，查MA，从中找出空闲块号，当一组的空闲块只剩第一块时，则应把该块中指出的下一组的空闲块数和块号复制到专用块中，然后把该块分配给申请者。当一组的空闲块分配完后则把专用块内容（下一组链接情况）复制到主存，再为申请者分配。

(4) 归还一块时给出归还的块号，若当前组不满规定块数时，将归还块登记入该组；若当前组已满，则另建一新组，这时归还块作为新一组的第一块，应把主存中登记的一组链接情况MA复制到归还块中，然后在MA重新登记一个新组。

(5) 设计分配和归还磁盘空间的程序。

**3．1．3** **上机实验所用平台**

模拟算法所用语言：c语言。编辑器为Dev C++。

**3.2数据结构及代码段分析**

**3．2．1** **部分主要数据结构**

每一个内存块的数据结构如下：

typedef struct Disk\_block

{

int occupied;

int data[BLK\_SIZE];

} Disk\_block;

BLK\_SIZE为一个内存块的大小（本次实验为64个字节），occupied指示该块是否被占用。

磁盘中共M个Disk\_block：

Disk\_block\* disk = (Disk\_block\* )malloc(M\*sizeof(Disk\_block));

MA为内存中存储专用块的内存空间，定义如下：

typedef Disk\_block MA;

**3．2．2** **主要代码段分析**

int disk\_init(Disk\_block\* disk)

{

int i, j , group\_num=(M-1)/N+((M-1)/N==0?0:1);

Disk\_block \*p = disk+1;

disk->data[0] = (M<N)? M : N; //super\_block

for (i=1; i<disk->data[0]+1; i++)

disk->data[i] = i;

for (i=1; i<group\_num+1; i++)

{

if (i<group\_num-1)

{

p->data[0] = N;

for (j=1; j<N+1; j++)

p->data[j] = N\*i + j;

}

else if (i==group\_num-1)

{

p->data[0] = N - M%N; // descripting the last group

for (j=1; j<(M-1)%N+2; j++) //1-16 ! not {1-15, 0}

p->data[j] = N\*i + j;

}

else if (i==group\_num)

p->data[0] = 0;

p += N;

}

return 1;

}

上述代码段为初始化代码。将N个空闲磁盘块组织成group\_num个组，每组由N=16个块组成。每组第一个块登记下一个组的信息。每个登记信息的磁盘块中，data[0]记录下一组空闲块的数目，data[1]，data[2]……记录下一组空闲块块号。

int\* allocateDisk(Disk\_block\* disk, MA\* ma, int size)

{

int k=0;

int\* rtr = (int\* )malloc(size\*sizeof(int));

while (true)

{

if (ma->data[0]>1)

{

int block\_num;

block\_num = ma->data[ma->data[0]];

(disk+block\_num)->occupied = 1;

rtr[k++] = block\_num;

ma->data[0]--;

}

else if (ma->data[0]==1)

{

int i;

int new\_sb\_offset = ma->data[1]; //new super block

Disk\_block\* new\_sb = disk + new\_sb\_offset;

for (i=0; i<N+1; i++)

disk->data[i] = new\_sb->data[i];//set super\_block

new\_sb->occupied = 1;

rtr[k++] = new\_sb\_offset;

ma->data[0]--;

writeMA(ma, disk);

}

if (k>=size) break;

}

return rtr;

}

上述代码实现对磁盘块的分配。每次分配ma所指的组中的一块，即第一组中的一块。若分配该块时，该块为第一组最后一个空闲块，需将该块中登记的第二组的信息，即该块的data[0]，data[1]……转移入专用块与ma中，使第二组成为新的ma所指的第一组。

总共分配的块数为size块，被分配的各个块的块号被写入rtr数组中。rtr起始地址作为该函数的返回值。

int releaseDisk(Disk\_block\* disk, MA\* ma, int block\_num)

{

Disk\_block\* target = disk + block\_num;

if (target->occupied==0)

return 0;

else

{

if(ma->data[0]<N)

{

target->occupied = 0;

ma->data[0]++;

ma->data[ma->data[0]] = block\_num;

}

else

{

int i;

target->occupied = 0;

//create a new group with target itself being the first block

for (i=0; i<N+1; i++)//ma into the target

target->data[i] = ma->data[i];

ma->data[0] = 1;

ma->data[1] = block\_num;

writeSB(disk, ma);

}

return 1;

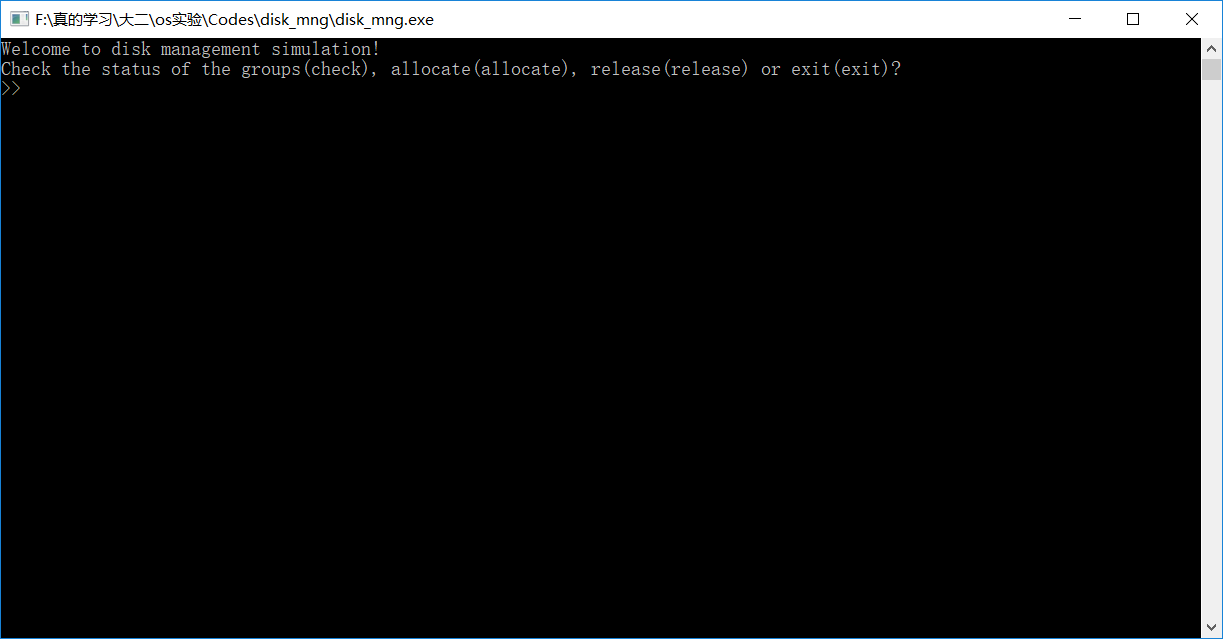
}

}

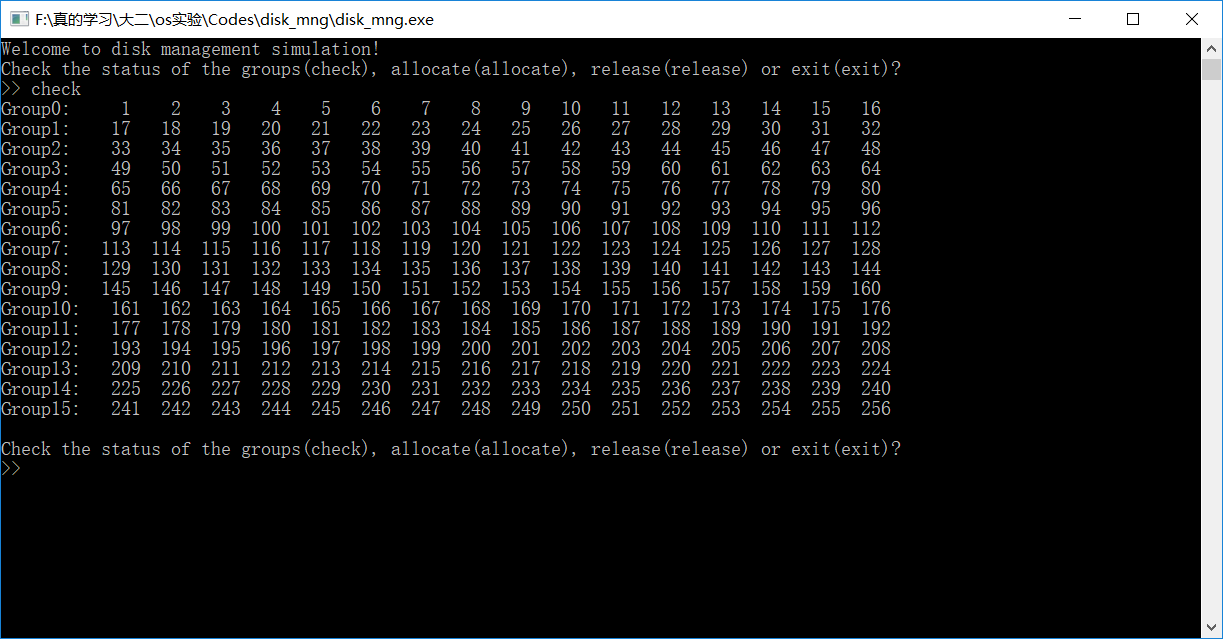
上述代码实现对编号为block\_num的磁盘块的释放。若当前ma所指的第一组未满（块数不足N=16个），则将该块直接加入该组，在专用块与ma中登记该块即可。若当前ma所指的第一组已满，则将该块作为一个新的组的第一块，将当前专用块与ma中的信息转移入该块，并在专用块与ma中登记该块。

**3.3 调试过程**

进入程序，初始界面截图如下：

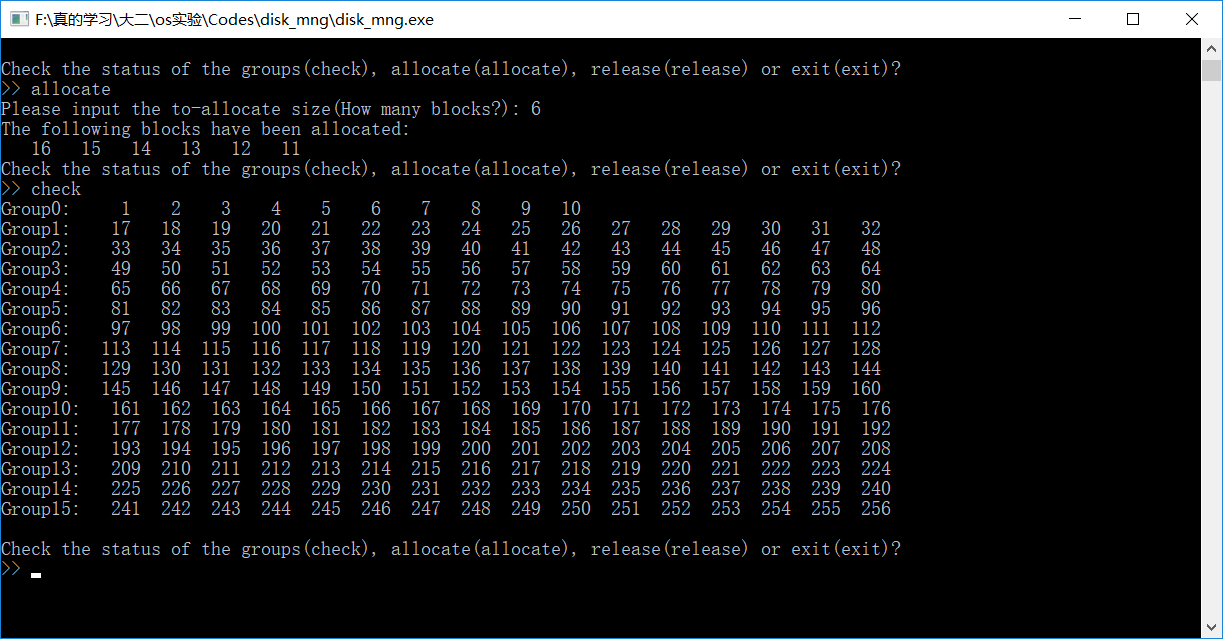


等待用户输入命令。首先输入check，查看当前空闲磁盘块成组链接的情况，如下所示：



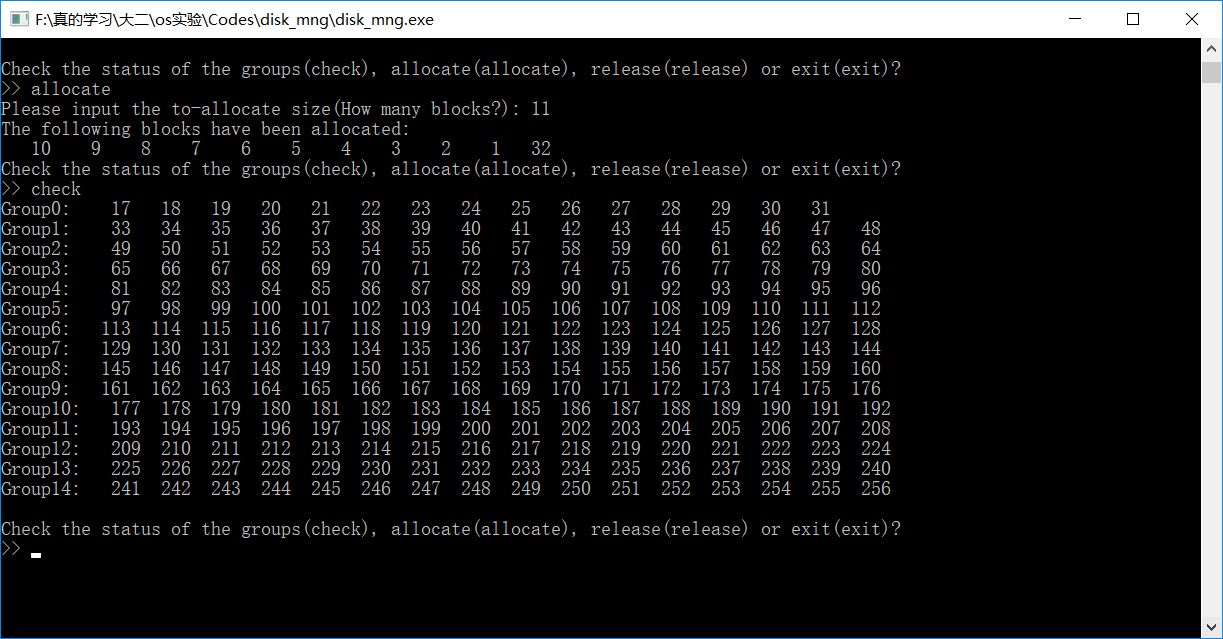
当前磁盘共有256个磁盘块，被分成16个组，每组有16个磁盘块。

输入allocate命令，分配6个磁盘块。然后再次输入check命令，查看分配6个磁盘块后，空闲磁盘块成组链接的情况：



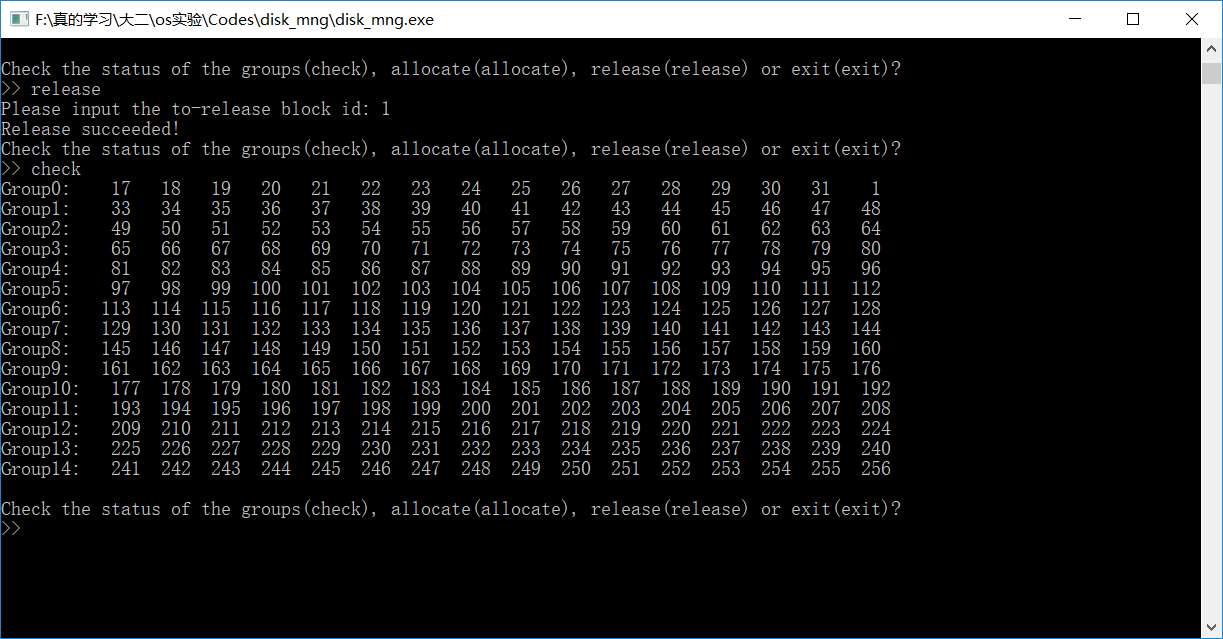
可看到，编号11-16的磁盘块被分配。这些磁盘块不再是空闲磁盘块，从第一组空闲磁盘块组（Group0）中被移除。

再次输入allocate命令，分配11个磁盘块；输入check命令查看各组情况：



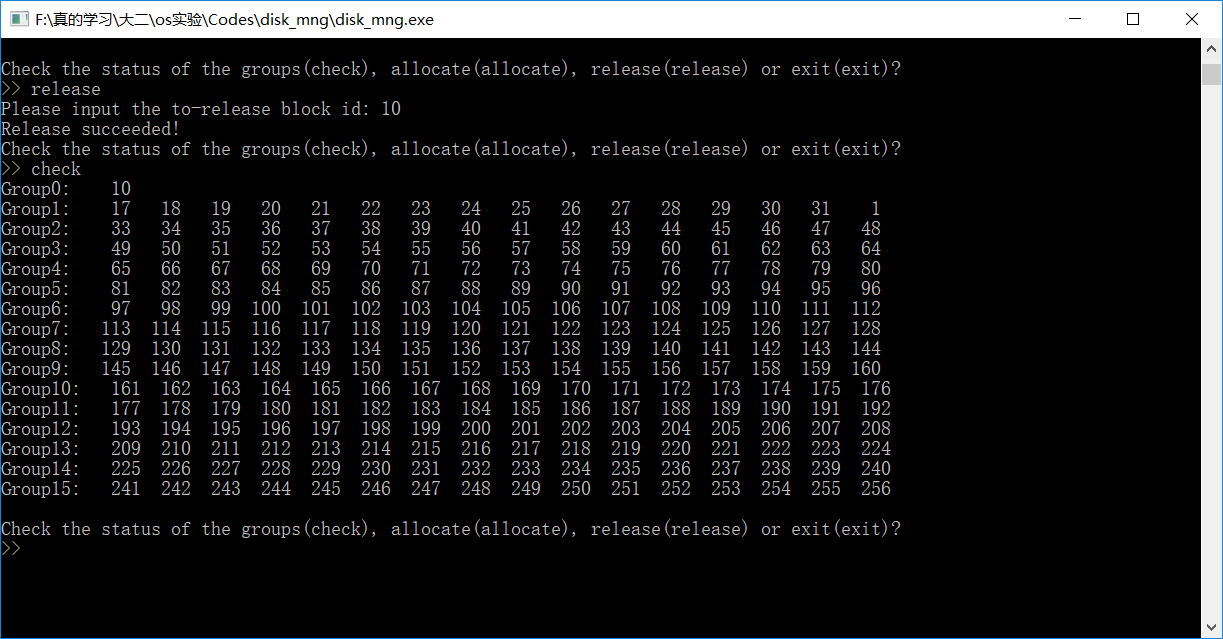
可见，不仅第一组剩余的空闲磁盘块，即编号1-10的磁盘块被分配，原第二组中的编号为32的一块也被分配。之后原第二组成为新的第一组（Group0）。

下面测试release命令释放磁盘块。输入release命令，释放编号为1的磁盘块：



可见1号磁盘块被加入了当前第一组Group0中。

再次release编号为10的磁盘块：



由于原第一组（现第二组，上图中Group1）已满，建立一个新的组作为Group0，并将第10号磁盘块加入这一新的组里，完成释放。

**3.4 总结**

通过模拟UNIX系统的空闲块成组链接法，实现磁盘空间的分配和回收，加深了对操作系统磁盘管理的理解。

本次实习的关键有：

（1）将空闲磁盘块链接成组的操作。这个操作的实质是：找到各组的第一块，在该块中登记下一组的信息。各组第一块的块号可以用一个等差数列表示，第一项为1，以后每项为前一项加上N=16。需要注意的细节是：倒数第二组的第一块中登记的下一组空闲块数很可能不是N=16，而是小于N的数，因为最后一组很可能是一个不满的组；

（2）取出一个空闲块并将之分配给进程的操作和回收一个空闲块的操作。细节已经在3.1.2节中阐明。

**4 进程同步：模拟P、V操作实现同步机构、用P、V操作解决生产者-消费者问题（选做题）**

**4.1 实习内容及上机实验所用平台**

本实验模拟实现同步机构，以避免发生进程执行时可能出现的与时间有关的错误，加深对进程同步的了解。

**4．1．1** **实习内容：进程同步**

模拟P、V操作实现同步机构、用P、V操作解决生产者-消费者问题。

**4．1．2** **设计思路**

(1) P、V操作同步机构，由P操作原语和V操作原语组成，它们的定义如下：

P操作原语P (s)：将信号量s减去1，若结果小于0，则执行原语的进程被置成等待信号量s的状态。

V操作原语V (s)：将信号量s加1，若结果不大于0，则释放一个等待信号量s的进程。

这两条原语是如下的两个过程：

procedure p (var s: semaphore);

begin s: = s-1;

if s<0 then W (s)

end {p}

procedure v (var s: semaphore);

egin s: = s+1;

if s0 then R (s)

end {v}

其中W（s）表示将调用过程的进程置为等待信号量s的状态；R（s）表示释放一个等待信号量s的进程。

(2) 生产者——消费者问题。

假定缓冲器内可同时存放10件产品。那么，用PV操作来实现生产者和消费者之间的同步，生产者和消费者两个进程的程序如下：

B: array [0..9] of products;

s1, s2; semaphore;

s1: =10, s2: =0;

IN, out: integer;

IN: =0; out: =0;

cobegin

procedure producer;

c: products;

begin

L1:

Produce (c);

P (s1);

B[IN]: =C;

IN: =(IN+1)mod 10;

V (s2);

goto L1

end;

procedure consumer;

x: products;

begin

L2:

p (s2);

x: =B[out];

out: =(out+1) mod10;

v (s1);

consume (x);

goto L2

end;

coend.

(3) 进程控制块PCB。

为了记录进程执行时的情况，以及进程让出处理器后的状态，断点等信息，每个进程都有一个进程控制块PCB。在模拟实验中，假设进程控制块的结构如图4-1。其中进程的状态有：运行态、就绪态、等待态和完成态。当进程处于等待态时，在进程控制块PCB中要说明进程等待原因（在模拟实验中进程等待原因是为等待信号量s1或s2）；当进程处于等待态或就绪态时，PCB中保留了断点信息，一旦进程再度占有处理器则就从断点位置继续运行；当进程处于完成状态，表示进程执行结束。

|  |
| --- |
| 进程名 |
| 状态 |
| 等待原因 |
| 断点 |

图4-1 进程控制块结构

(4) 处理器的模拟。

计算机硬件提供了一组机器指令，处理器的主要职责是解释执行机器指令。为了模拟生产者和消费者进程的并发执行，我们必须模拟一组指令和处理职能。

模拟的一组指令见图2-2，其中每条指令的功能由一个过程来实现。用变量PC来模拟“指令计数器”，假设模拟的指令长度为1，每执行一条模拟指令后，PC加1，提出下一条指令地址。使用模拟的指令，可把生产者和消费者进程的程序表示为图2-3的形式。

定义两个一维数组PA[0..4]和SA[0..4]，每一个PA[i]存放生产者程序中的一条模拟指令执行的入口地址；每个SA[i]存放消费者程序中的一条模拟指令执行的入口地址。于是模拟处理器执行一条指令的过程为：取出PC之值，按PA[PC]或SA[PC]得模拟指令执行的入口地址，将PC之值加1，转向由入口地址确定的相应的过程执行。

(5) 程序设计

本实验中的程序由三部分组成：初始化程序、处理器调度程序、模拟处理器指令执行程序。各部分程序的功能及相互间的关系由图2-4至图2-7指出。

|  |  |
| --- | --- |
| 模拟的指令 | 功 能 |
| p (s) | 执行P操作原语 |
| v (s) | 执行V操作原语 |
| put | B[IN]: =product; IN: = (IN+1) mod 10 |
| GET | x: =B[out]; out: =(out+1) mod 10 |
| produce | 输入一个字符放入C中 |
| consume | 打印或显示x中的字符 |
| GOTO L | PC: =L |
| NOP | 空操作 |

图4-2 模拟的处理器指令

|  |  |  |
| --- | --- | --- |
| 序号 | 生产者程序 | 消费者程序 |
| 0 | produce | p (s2) |
| 1 | p (s1) | GET |
| 2 | PUT | v (s1) |
| 3 | v (s2) | consume |
| 4 | goto 0 | goto 0 |

图4-3 生产者和消费者程序

·初始化程序：模拟实验的程序从初始化程序入口启动，初始化工作包括对信号量s1、s2赋初值，对生产者、消费者进程的PCB初始化。初始化后转向处理调度程序，其流程如图4-4。

5

图4-4 初始化流程

**4．1．3** **上机实验所用平台**

模拟算法所用语言：c语言。编辑器为Dev C++。

**4.2数据结构及代码段分析**

**4．2．1** **部分主要数据结构**

typedef int Product; //产品定义为int类型

typedef struct PCB {

char proc\_name[20]; //进程名（producer或consumer）

int status; //进程状态：ready，running，waiting，terminated

int need; //等待的信号量s1 or s2

int breakpoint; //断点

struct PCB\* next;

} PCB;

信号量定义如下：

typedef struct Semaphore {

int value; //信号量值

PCB \*pending; //等待该信号量的进程

} Semaphore;

**4．2．2** **主要代码段分析**

p，v操作如下：

int p(Semaphore\* s)

{

s->value--;

if (s->value<0)

{

running\_proc->status = WAITING;

running\_proc->need = (s==s1)? S1 : ((s==s2)? S2 : NONEED);

//insert to tail

s->pending = running\_proc;

}

else

running\_proc->status = READY;

}

int v(Semaphore\* s)

{

s->value++;

if (s->value<=0)

{

s->pending->status = READY;

}

running\_proc->status = READY;

}

每次处理器调度进程的函数如下（有所省略）：

int schedule()

{

while (true)

{

……

if (producer->status==READY && consumer->status==READY)

{

srand((unsigned)time(NULL));

(rand()%2==0)?setRunningProc(producer):setRunningProc(consumer);

}

……

PC = running\_proc->breakpoint;

execute();

}

return 0;

}

每次调度，若生产者、消费者都不用等待（生产线既不满又不空），则随机选择一个为处理器马上将运行的进程running\_proc。每次调度，PC设为马上将运行的进程的断点值，随后转到execute()，运行该进程。

execute()如下：

int execute()

{

if (running\_proc==producer)

{

switch(PC) //PA

{

case 0: PC++; produce(); running\_proc->status=READY; break;

case 1: PC++; p(s1); break;

case 2: PC++; put(); running\_proc->status=READY; break;

case 3: PC++; v(s2); to\_produce--; break;

case 4: PC++; PC=0; running\_proc->status=READY; break;

}

}

else if (running\_proc==consumer)

{

switch(PC) //SA

{

case 0: PC++; p(s2); break;

case 1: PC++; get(); running\_proc->status=READY; break;

case 2: PC++; v(s1); break;

case 3: PC++; consume(); running\_proc->status=READY; break;

case 4: PC++; PC=0; running\_proc->status=READY; break;

}

}

if (to\_produce<=0)

producer->status=TERMINATED;

if (to\_consume<=0)

consumer->status=TERMINATED;

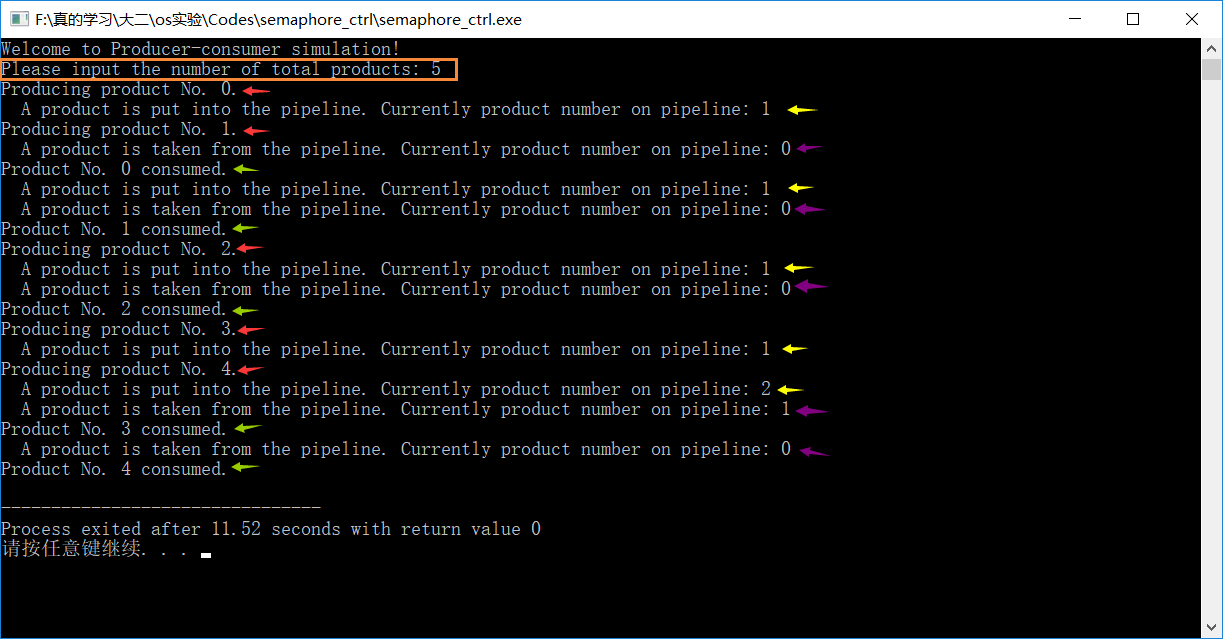
return 0;

}

基本思想与4.1.2的（4）中一致，在此不予赘述。

**4.3 调试过程**

程序运行截图如下：



上图中，橘色部分为用户输入要生产/消费的产品数量，本例中为5；

红色箭头所指的行：生产者每次生产一个产品，即produce()执行时打印的提示信息；

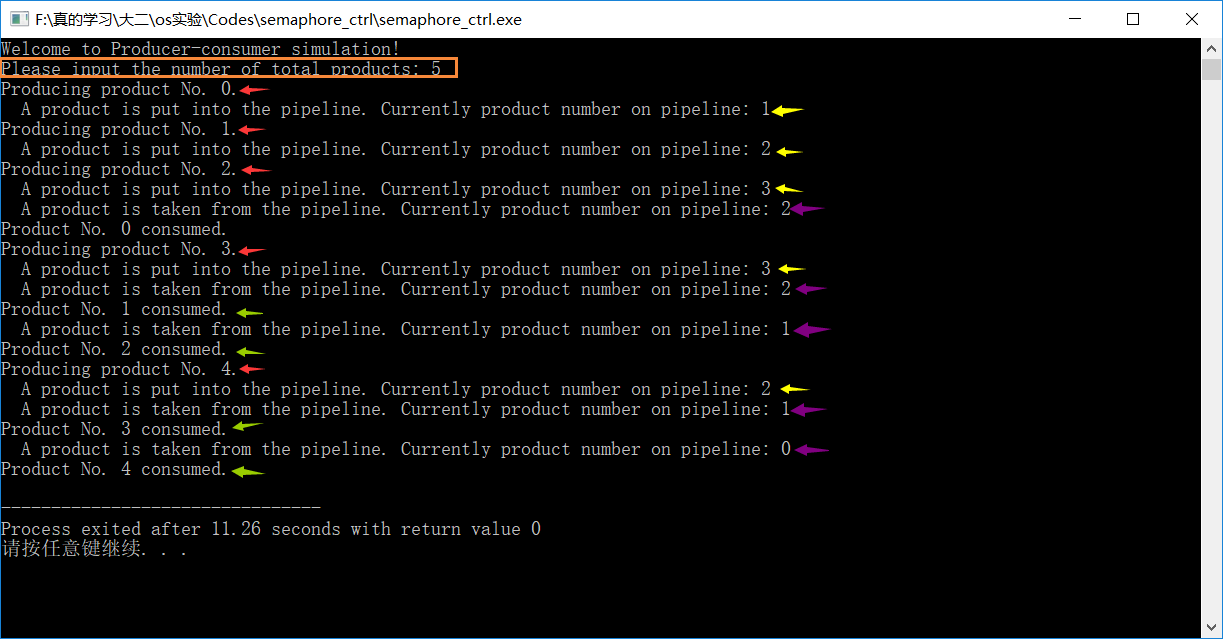
黄色箭头所指的行：生产者每次将一个已经生产好、但未放入生产线缓冲区的产品放入生产线缓冲区，即put()执行时打印的提示信息；

绿色箭头所指的行：消费者每次消费一个产品，即consume()执行时打印的提示信息；

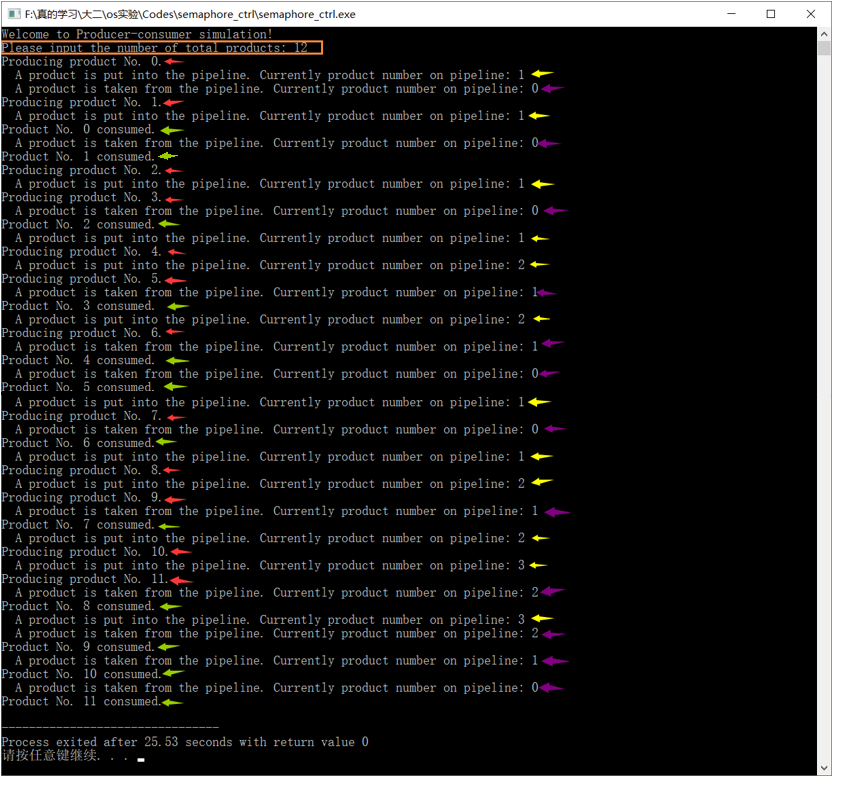
紫色箭头所指的行：消费者每次将一个已经从生产线缓冲区中取出、但还未被消费的产品消费掉，即get()执行时打印的提示信息；

从提示信息可见生产者-消费者互相协作的具体运行步骤。

为证明在生产者与消费者同时就绪时，处理器调度的随机性，再运行一次模拟程序，结果如下图：



改变生产/消费的总共产品数量为12（大于缓冲区大小10），运行结果如下：



运行结果仍正确，很好地体现了进程同步。

**4.4 总结**

通过模拟P、V操作实现同步机构、用P、V操作解决生产者-消费者问题，模拟进程的并发执行，了解进程并发执行时同步机构的作用。

本实习有两个问题需要注意：

（1）在4.1.2中，处理器调度时，若生产者与消费者均在就绪队列中，则处理器随机选择一个进程执行。但是若不是两者均就绪的情况，则不能随机选择一个进程执行。具体应如下：若两个进程均没有就绪，说明两进程均已执行完毕。若一个就绪，一个未就绪，则选择就绪的进程执行。4.1.2中并未阐释清楚，若忽视这些必要的判定步骤，代码实现时则会有问题出现。

（2）判定生产者执行完毕的时机。图4-3是生产者与消费者每次生产/消费的一次完整流程。对生产者来说，除去第五条对PC进行修改的指令没有实际意义外，前4条指令（produce()，p操作，put()，v操作）均为有意义的生产过程。在生产最后一个产品时，第生产者不应在前4条指令中的前3条终止，而是应该执行完这4条指令后，即v操作执行完后才可终止。to\_produce变量是生产者还需生产的产品数量，每生产一个产品，该变量减1。按照正常思维，应该在put()时执行一次to\_produce--。但是这会引起问题。因为每条指令执行完后，都会检查to\_produce变量是否为0来决定生产者是否终止，所以在生产最后一个产品时，若在put()时将to\_produce减1，put()执行完后，to\_produce等于0，生产者便终止了，最后的一次v操作并没有被执行。这导致对消费者而言，生产线内少了一个可供消费的产品，消费者消费最后一个产品时的p操作永远无法通过，消费者便无法进行get()与consume()操作消费最后一个产品。只有把在生产者生产产品时进行的to\_produce—操作放在v操作之后，才不会引起这个bug。

其它问题都较好解决，且4.1.2阐述较为清晰，在此不予赘述。

**附 录**

**1 处理器调度实习程序源码：**

/\*defines.h\*/

#ifndef \_\_DEFINES\_H\_\_

#define \_\_DEFINES\_H\_\_

#define READY 1

//Ready

#define END 0

//End

#endif

/\*declares.h\*/

#ifndef \_\_DECLARES\_H\_\_

#define \_\_DECLARES\_H\_\_

typedef struct PCB {

unsigned pid;

struct PCB \*next;

unsigned remain\_time;

unsigned priority;

int status;

} PCB;

PCB \* createProc(unsigned, unsigned, unsigned);

int insertProc(PCB \*, PCB \*);

int runProc(PCB \*, PCB \*);

#endif

/\*proc\_ctrl.h\*/

/\*CPU control with priority\*/

#include <stdio.h>

#include <stdlib.h>

#include "defines.h"

#include "declares.h"

PCB \* createProc(unsigned pid, unsigned remain\_time, unsigned priority)

{

PCB \*p;

p = (PCB \*)malloc(sizeof(PCB));

p->pid = pid;

p->priority = priority;

p->remain\_time = remain\_time; //remain\_time > 0

p->next = NULL;

p->status = READY;

return p;

}

int insertProc(PCB \*new\_proc, PCB \*queue\_head)

{

//insert to right position according to priority(decreasing)

PCB \*p, \*pre;

pre = queue\_head;

p = pre->next;

while (p!=NULL)

{

if (new\_proc->priority > p->priority)

break;

pre = p;

p = p->next;

}

new\_proc->next = pre->next;

pre->next = new\_proc;

return 1;

}

int runProc(PCB \*ready\_queue\_head, PCB \*terminated\_queue\_head)

{

if (ready\_queue\_head->next==NULL)

return 0;

else

{

/\*run\*/

PCB \*p = ready\_queue\_head->next;

ready\_queue\_head->next = p->next;

p->next = NULL; //remove p from ready queue

p->priority = p->priority==0 ? 0 : (p->priority)-1;

p->remain\_time = p->remain\_time==0 ? 0 : (p->remain\_time)-1;

printf("Current running process pid: %u, \

priority: %u, remaining time: %u.\n", p->pid, p->priority, p->remain\_time);

if (p->remain\_time==0)

{

p->status = END;

p->next = terminated\_queue\_head->next;

terminated\_queue\_head->next = p;

//insert p into terminated\_queue

}

else

{

insertProc(p, ready\_queue\_head);

}

return 1;

}

}

/\*test\_proc\_ctrl.h\*/

#include <stdio.h>

#include <stdlib.h>

#include "defines.h"

#include "declares.h"

int main()

{

unsigned proc\_num;

PCB \*ready\_queue\_head;

PCB \*terminated\_queue\_head;

PCB \*p;

int i;

ready\_queue\_head = (PCB \*)malloc(sizeof(PCB));

ready\_queue\_head->priority = 0-1; //MAX priority

ready\_queue\_head->next = NULL; //necessary! avoid bugs!

terminated\_queue\_head = (PCB \*)malloc(sizeof(PCB));

terminated\_queue\_head->next = NULL; //necessary! avoid bugs!

printf("Welcome to process control test!\n\

Please input the number of processes: ");

scanf("%u", &proc\_num);

for (i=0; i<proc\_num; i++) //establish link-table

{

unsigned time, priority;

printf("Please input the priority and \

to-run time of the process %d: ", i);

scanf("%u %u", &priority, &time);

p = createProc(i, time, priority);

insertProc(p, ready\_queue\_head);

}

while (runProc(ready\_queue\_head, terminated\_queue\_head)) ;

printf("Process control ends.\n");

getchar();

return 0;

}

**2 主存空间管理实习程序源码：**

/\*declares.h\*/

#ifndef \_\_DECLARES\_H\_\_

#define \_\_DECLARES\_H\_\_

#define MEM\_SIZE 64

typedef struct PageEntry

{

int page\_num;

int block\_num;

} PageEntry;

typedef struct PCB

{

unsigned pid;

unsigned mem\_size;

PageEntry\* pagetable;

struct PCB\* next;

} PCB;

typedef struct BLOCK

{

int block\_id;

} BLOCK;

typedef struct Bitmap

{

char data[MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1)];

} Bitmap;

int printPageTable(PageEntry\*, int);

PageEntry\* setPageEntry(PageEntry\*, int, int);

int printBitmap(Bitmap\*);

int setBitmap0(Bitmap\*, int);

int\* setBitmap1(Bitmap\*, int);

#endif

/\*mem\_mng.h\*/

#include <stdio.h>

#include <stdlib.h>

#include "declares.h"

int printPageTable(PageEntry\* pagetable, int size)

{

int i;

printf("\t+=============================+\n");

for (i=0; i<size; i++)

{

printf("\t| Page %4d in Block %4d |\n", pagetable->page\_num, pagetable->block\_num);

pagetable++;

}

printf("\t+=============================+\n");

return 1;

}

PageEntry\* setPageEntry(PageEntry\* pos, int page\_num, int block\_num)

{

pos->page\_num = page\_num;

pos->block\_num = block\_num;

}

int printBitmap(Bitmap\* bitmap)

{

int i, j , n=MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1);

char string[9];

char temp, mask;

printf("Bitmap is as follow:\n");

printf("\t+==========+\n");

for (i=0; i<n; i++)

{

mask = 1;

temp = bitmap->data[i];

for (j=0; j<8; j++)

{

if ((int)(temp & mask) == 0) string[j] = '0';

else string[j] = '1';

mask = mask << 1;

}

string[8] = '\0';

printf("\t| %s |\n", string);

}

printf("\t+==========+\n");

return 1;

}

int setBitmap0(Bitmap\* bitmap, int block\_id)

{

int index = block\_id/8;

int offset = block\_id%8;

char temp = bitmap->data[index];

temp = (temp & ~(1<<offset));

bitmap->data[index] = temp;

return 1;

}

int\* setBitmap1(Bitmap\* bitmap, int size)

{

/\*set lowest available bit to 1\*/

int i, j, k=0, n=MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1);

int \*rtr = (int\* )malloc(size\*sizeof(int));

char mask;

char temp;

for (i=0; i<n; i++)

{

mask = 1;

temp = bitmap->data[i];

for (j=0; j<8; j++)

{

if ((int)(temp & mask) == 0) // when bit == 0

{

temp = temp | mask; // set the bit to 1

rtr[k++] = 8\*i + j;

if (k>=size) break;

}

mask = mask << 1;

}

bitmap->data[i] = temp;

if (k>=size) break;

}

return rtr; // block\_num array

}

/\*test\_mem\_mng.h\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "declares.h"

int main()

{

int i;

int exit=0;

unsigned max\_pid=0;

int block\_remain = MEM\_SIZE;

char command[20];

PCB\* PCB\_head = (PCB\* )malloc(sizeof(PCB));

Bitmap\* bitmap = (Bitmap\* )malloc(sizeof(Bitmap));

PCB\_head->next=NULL;

for (i=0; i<MEM\_SIZE/8 + (MEM\_SIZE%8==0 ? 0 : 1); i++)

{

bitmap->data[i]='\0';

}

printf("Welcome to memory management simulation!\n");

do

{

printf("Current memory condition: %d blocks remaining.", block\_remain);

printBitmap(bitmap);

printf("Check pagetable(check), create a new process(add), \

release the memory occupied by a process(release) or exit(exit)?\n");

do

{

printf(">> ");

scanf("%s", &command);

if (strcmp(command, "exit")==0)

{

exit = 1;

break;

}

else if(strcmp(command, "check")==0)

{

int found=0; unsigned pid; PCB\* p=PCB\_head->next;

printf("Please input the to-check pid: ");

scanf("%u", &pid);

while(p!=NULL)

{

if (p->pid==pid)

{

printPageTable(p->pagetable, p->mem\_size);

found = 1;

break;

}

p = p->next;

}

if(!found) printf("Invalid pid!\n");

break;

}

else if(strcmp(command, "add")==0)

{

int mem\_size;

printf("Please input the to-occupy \

memory blocks of the new process: ");

scanf("%d", &mem\_size);

if (mem\_size > block\_remain)

printf("No enough memory blocks!\n");

else

{

PCB\* new\_proc = (PCB\* )malloc(sizeof(PCB));

new\_proc->pid = max\_pid;

new\_proc->mem\_size = mem\_size;

new\_proc->next = PCB\_head->next;

PCB\_head->next = new\_proc;

int\* blocks;

PageEntry\* pagetable = (PageEntry\* )malloc(mem\_size\*sizeof(PageEntry));

new\_proc->pagetable = pagetable;

blocks = setBitmap1(bitmap, mem\_size); // set bitmap

for (i=0; i<mem\_size; i++) //set pagetable

setPageEntry(pagetable++, i, blocks[i]);

free(blocks);

block\_remain -= mem\_size; // decrease available block number

printf("Process %d created, pagetable is as follow: \n", max\_pid++);

printPageTable(new\_proc->pagetable, mem\_size);

}

break;

}

else if(strcmp(command, "release")==0)

{

unsigned pid;

PCB\* p=PCB\_head->next;

PageEntry\* pagetable;

printf("Please input the to-release process pid: ");

scanf("%u", &pid);

while(p!=NULL)

{

if (p->pid==pid) break; // find the PCB

p = p->next;

}

if (p==NULL) printf("Invalid pid! Please input a correct pid.\n");

else

{

int i, size;

size = p->mem\_size;

pagetable = p->pagetable;

for (i=0; i<size; i++)

setBitmap0(bitmap, pagetable++->block\_num);

free(p->pagetable);

PCB\_head->next = p->next;

free(p);

block\_remain += size;

}

break;

}

else printf("Invalid command! Please input once again!\n");

} while(true); //give a command

} while(!exit);

getchar();

}

**3 磁盘空间管理实习程序源码：**

/\*declares.h\*/

#ifndef \_\_DECLARES\_H\_\_

#define \_\_DECLARES\_H\_\_

#define N 16

//group\_size

#define M 256

//block\_num

#define BLK\_SIZE 63

//block\_size

typedef struct Disk\_block

{

int occupied;

int data[BLK\_SIZE];

} Disk\_block;

typedef Disk\_block MA;

int disk\_init(Disk\_block\*);

int writeMA(MA\*, Disk\_block\*);

int writeSB(Disk\_block\*, MA\*);

int\* allocateDisk(Disk\_block\*, MA\*, int);

int releaseDisk(Disk\_block\*, MA\*, int);

int printGroups(MA\*, Disk\_block\*);

#endif

/\*disk\_mng.h\*/

#include "declares.h"

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

int disk\_init(Disk\_block\* disk)

{

int i, j , group\_num=(M-1)/N+((M-1)/N==0?0:1);

Disk\_block \*p = disk+1;

disk->data[0] = (M<N)? M : N; //super\_block

for (i=1; i<disk->data[0]+1; i++)

disk->data[i] = i;

for (i=1; i<group\_num+1; i++)

{

if (i<group\_num-1)

{

p->data[0] = N;

for (j=1; j<N+1; j++)

p->data[j] = N\*i + j;

}

else if (i==group\_num-1)

{

p->data[0] = N - M%N; // descripting the last group

for (j=1; j<(M-1)%N+2; j++) //1-16 ! not {1-15, 0}

p->data[j] = N\*i + j;

}

else if (i==group\_num)

p->data[0] = 0;

p += N;

}

return 1;

}

int writeMA(MA\* ma, Disk\_block\* super\_block)

{

int i;

for (i=0; i<super\_block->data[0]+1; i++)

ma->data[i] = super\_block->data[i];

return 1;

}

int writeSB(Disk\_block\* super\_block, MA\* ma)

{

int i;

for (i=0; i<ma->data[0]+1; i++)

super\_block->data[i] = ma->data[i];

return 1;

}

int\* allocateDisk(Disk\_block\* disk, MA\* ma, int size)

{

int k=0;

int\* rtr = (int\* )malloc(size\*sizeof(int));

while (true)

{

if (ma->data[0]>1)

{

int block\_num;

block\_num = ma->data[ma->data[0]];

(disk+block\_num)->occupied = 1;

rtr[k++] = block\_num;

ma->data[0]--;

}

else if (ma->data[0]==1)

{

int i;

int new\_sb\_offset = ma->data[1]; //new super block

Disk\_block\* new\_sb = disk + new\_sb\_offset;

for (i=0; i<N+1; i++)

disk->data[i] = new\_sb->data[i];//set super\_block

new\_sb->occupied = 1;

rtr[k++] = new\_sb\_offset;

ma->data[0]--;

writeMA(ma, disk);

}

if (k>=size) break;

}

return rtr;

}

int releaseDisk(Disk\_block\* disk, MA\* ma, int block\_num)

{

Disk\_block\* target = disk + block\_num;

if (target->occupied==0)

return 0;

else

{

if(ma->data[0]<N)

{

target->occupied = 0;

ma->data[0]++;

ma->data[ma->data[0]] = block\_num;

}

else

{

int i;

target->occupied = 0;

//create a new group with target itself being the first block

for (i=0; i<N+1; i++)//ma into the target

target->data[i] = ma->data[i];

ma->data[0] = 1;

ma->data[1] = block\_num;

writeSB(disk, ma);

}

return 1;

}

}

int printGroups(MA\* ma, Disk\_block\* disk)

{

int i, gid=0, num;

Disk\_block\* p=ma;

while (true)

{

num=p->data[0];

if (num==0) break;

printf("Group%d: ", gid++);

for (i=1; i<num+1; i++)

printf("%5d",p->data[i]);

printf("\n");

p = disk + p->data[1];

}

printf("\n");

return 1;

}

/\*test\_disk\_mng.h\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "declares.h"

int main()

{

int remain=M, exit=0;

char command[20];

Disk\_block\* disk = (Disk\_block\* )malloc(M\*sizeof(Disk\_block));//Disk\_block disk[M]

MA\* ma = (MA\* )malloc(sizeof(MA)); //memory access

disk\_init(disk);

writeMA(ma, disk);

remain--;

printf("Welcome to disk management simulation!\n");

do

{

printf("Check the status of the groups(check), \

allocate(allocate), release(release) or exit(exit)?\n");

do

{

printf(">> ");

scanf("%s", &command);

if (strcmp(command, "check")==0)

{

printGroups(ma, disk);

break;

}

else if (strcmp(command, "allocate")==0)

{

int i, size;

int\* blocks;

printf("Please input the to-allocate size(How many blocks?): ");

scanf("%d", &size);

blocks = allocateDisk(disk, ma, size);

printf("The following blocks have been allocated:\n");

for (i=0; i<size; i++)

printf("%5d", blocks[i]);

printf("\n");

free(blocks);

break;

}

else if (strcmp(command, "release")==0)

{

int block\_id;

printf("Please input the to-release block id: ");

scanf("%d", &block\_id);

if (block\_id>M-1) printf("Illegal block address! Please input again!\n");

if (releaseDisk(disk, ma, block\_id)) printf("Release succeeded!\n");

else printf("Release failed!\nThe block No. %d is already empty. No release opertion on it can be performed!\n");

break;

}

else if (strcmp(command, "exit")==0)

{

exit = 1;

break;

}

else

{

printf("Invalid command! Please input again!");

}

} while (true);

}while (!exit);

getchar();

}

**4 进程同步实习程序源码：**

/\*declares.h\*/

#ifndef \_\_DECLARES\_H\_\_

#define \_\_DECLARES\_H\_\_

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <windows.h>

#define PIPELINE\_SIZE 10

#define RUNNING 0

#define READY 1

#define WAITING 2

#define TERMINATED 3

#define NONEED 0

#define S1 1

#define S2 2

typedef int Product;

typedef struct PCB {

char proc\_name[20];

int status;

int need; //s1 or s2

int breakpoint;

struct PCB\* next;

} PCB;

typedef struct Semaphore {

int value;

PCB \*pending;

} Semaphore;

extern Semaphore\* s1; //empty

extern Semaphore\* s2; //occupied

extern int PC;

extern int product\_num;

extern int to\_produce;

extern int to\_consume;

extern int product\_id;

extern int in;

extern int out;

extern Product C;

extern Product X;

extern Product products[PIPELINE\_SIZE];

extern PCB\* running\_proc;

extern PCB\* producer;

extern PCB\* consumer;

int setRunningProc(PCB\* );

int init();

int schedule();

int execute();

int p(Semaphore\* );

int v(Semaphore\* );

int produce();

int consume();

int put();

int get();

#endif

/\*functions.h\*/

#include "declares.h"

int setRunningProc(PCB\* loaded)

{

loaded->status=RUNNING;

running\_proc = loaded;

}

int init()

{

in = 0;

out = 0;

s1->value = 10;

s1->pending = NULL;

s2->value = 0;

s2->pending = NULL;

PC = 0;

product\_id = 0;

product\_num = 0;

setRunningProc(producer);

strcpy(producer->proc\_name, "producer");

producer->status = READY;

producer->need = NONEED;

producer->breakpoint = 0;

producer->next = NULL;

strcpy(consumer->proc\_name, "consumer");

consumer->status = READY;

consumer->need = NONEED;

consumer->breakpoint = 0;

consumer->next = NULL;

return 0;

}

int schedule()

{

while (true)

{

Sleep(200);

running\_proc->breakpoint = PC;

if (!(producer->status==READY || consumer->status==READY)) break;

if (producer->status==READY && consumer->status==READY)

{

srand((unsigned)time(NULL));

(rand()%2==0) ? setRunningProc(producer) : setRunningProc(consumer);

}

else if (producer->status==WAITING || producer->status==TERMINATED)

{

setRunningProc(consumer);

}

else if (consumer->status==WAITING || consumer->status==TERMINATED)

{

setRunningProc(producer);

}

PC = running\_proc->breakpoint;

execute();

}

return 0;

}

int execute()

{

if (running\_proc==producer)

{

switch(PC) //PA

{

case 0: PC++; produce(); running\_proc->status=READY; break;

case 1: PC++; p(s1); break;

case 2: PC++; put(); running\_proc->status=READY; break;

case 3: PC++; v(s2); to\_produce--; break;

case 4: PC++; PC=0; running\_proc->status=READY; break;

}

}

else if (running\_proc==consumer)

{

switch(PC) //SA

{

case 0: PC++; p(s2); break;

case 1: PC++; get(); running\_proc->status=READY; break;

case 2: PC++; v(s1); break;

case 3: PC++; consume(); running\_proc->status=READY; break;

case 4: PC++; PC=0; running\_proc->status=READY; break;

}

}

if (to\_produce<=0)

producer->status=TERMINATED;

if (to\_consume<=0)

consumer->status=TERMINATED;

return 0;

}

int p(Semaphore\* s)

{

s->value--;

if (s->value<0)

{

running\_proc->status = WAITING;

running\_proc->need = (s==s1)? S1 : ((s==s2)? S2 : NONEED);

//insert to tail

s->pending = running\_proc;

}

else

running\_proc->status = READY;

}

int v(Semaphore\* s)

{

s->value++;

if (s->value<=0)

{

s->pending->status = READY;

}

running\_proc->status = READY;

}

int produce()

{

C = product\_id;

printf("Producing product No. %d.\n", product\_id);

product\_id++;

return 0;

}

int consume()

{

to\_consume--;

printf("Product No. %d consumed.\n", X);

return 0;

}

int put()

{

products[in] = C;

in = (in+1)%PIPELINE\_SIZE;

product\_num++;

printf(" A product is put into the pipeline. Currently product number on pipeline: %d\n", product\_num);

return 0;

}

int get()

{

X = products[out];

out = (out+1)%PIPELINE\_SIZE;

product\_num--;

printf(" A product is taken from the pipeline. Currently product number on pipeline: %d\n", product\_num);

return 0;

}

/\*main.h\*/

#include <iostream>

#include "declares.h"

Semaphore\* s1 = (Semaphore\* )malloc(sizeof(Semaphore));

Semaphore\* s2 = (Semaphore\* )malloc(sizeof(Semaphore));

int PC;

int product\_num;

int to\_produce;

int to\_consume;

int product\_id;

int in;

int out;

Product C;

Product X;

Product products[PIPELINE\_SIZE];

PCB\* running\_proc;

PCB\* producer = (PCB\* )malloc(sizeof(PCB));

PCB\* consumer = (PCB\* )malloc(sizeof(PCB));

int main(int argc, char\*\* argv) {

int tmp;

printf("Welcome to Producer-consumer simulation!\nPlease input the number of total products: ");

scanf("%d", &to\_produce);

to\_consume = to\_produce;

init();

schedule();

getchar();

return 0;

}

教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）