**武汉大学计算机学院**

**本科生实验报告**

**操作系统设计与实现**

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

学 生 学 号 ：

学 生 姓 名 ：

二○二○年五月

**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

操作系统实验的实验目的是通过模拟操作系统在运行中的各个过程并实现相关的算法来加深对操作系统理论的理解。操作系统实验设计在抽象上主要遵循相应的操作系统理论知识，在实践上遵循模块化编程的思想。实验内容主要包括：模拟处理机调度，模拟主存空间的分配和回收（首次适应算法），模拟磁盘存储空间的分配和回收（位示图法），进程创建（fork函数）和进程同步。经过调试和验收，实验结论基本符合预期。

**关键词：**处理器调度，主存空间，磁盘存储空间，分配和回收，进程创建，进程同步；

目录

[1 处理机调度 1](#_Toc40983395)

[1.1 实验内容及上机实验所用平台 1](#_Toc40983396)

[1.2 数据结构及代码段分析 1](#_Toc40983397)

[1.3 调试过程 5](#_Toc40983398)

[1.4 实验总结 6](#_Toc40983399)

[2 主存空间的分配和回收 7](#_Toc40983400)

[2.1 实验内容及上机实验所用平台 7](#_Toc40983401)

[2.2 数据结构及代码段分析 7](#_Toc40983402)

[2.3 调试过程 11](#_Toc40983403)

[2.4 实验总结 12](#_Toc40983404)

[3 磁盘存储空间的分配和回收 14](#_Toc40983405)

[3.1 实验内容及上机实验所用平台 14](#_Toc40983406)

[3.2 数据结构及代码段分析 14](#_Toc40983407)

[3.3 调试过程 17](#_Toc40983408)

[3.4 实验总结 19](#_Toc40983409)

[4 进程创建 19](#_Toc40983410)

[4.1 实验内容及上机实验所用平台 19](#_Toc40983411)

[4.2 数据结构及代码段分析 19](#_Toc40983412)

[4.3 调试过程 20](#_Toc40983413)

[4.4 实验总结 21](#_Toc40983414)

[5 进程同步 22](#_Toc40983415)

[5.1 实验内容及上机实验所用平台 22](#_Toc40983416)

[5.2 数据结构及代码段分析 22](#_Toc40983417)

[5.3 调试过程 30](#_Toc40983418)

[5.4 实验总结 31](#_Toc40983419)

[参考文献 32](#_Toc40983420)

[附录 33](#_Toc40983421)

# 处理机调度

## 实验内容及上机实验所用平台

实验内容：本实习模拟在单处理器环境下的处理器调度，加深了解处理器调度的工作。

上机实验所用平台：JetBrains Clion on Windows10。编程语言：C++。

## 数据结构及代码段分析

数据结构：

**PCB块：**保存进程名，进程要求运行时间，进程优先数和进程状态。成员函数功能主要有创建进程和运行进程。

class PCB**{**

string name**;**//进程名

int time**;**//要求运行时间

int priority**;**//优先数

bool status**;**//状态 0为完成 1为就绪

public**:**

bool Run**();**//运行 返回是否成功运行

bool Input**();**//输入PCB信息 返回是否成功创建

PCB**(){**name**=**"none"**;**time**=**0**;**priority**=**0**;**status**=** **false;}**//默认构造函数

PCB**(**string n**,**int t**,**int p**,**bool s**):**name**(**n**),**time**(**t**),**priority**(**p**),**status**(**s**){}**//构造函数

int GetPriority**(){return** priority**;}**//返回优先数

bool GetStatus**(){return** status**;}**//返回状态

bool **operator<(**PCB pcb**){return** priority**>**pcb**.**priority**;}**//重载小于以使用sort 因为从大到小排序 所以返回大与

bool **operator==(**PCB pcb**){return** priority**==**pcb**.**priority**;}**//重载等于以使用sort

void Show**(**int color **=** GREEN**)**//打印进程信息

**{**

colour**(**color**);**

cout **<<** name **<<** " \t" **<<** time **<<** " \t" **<<** priority **<<** "\t\t" **<<** **(**status**==false?**'E'**:**'R'**)** **<<** endl**;**

**}**

**};**

**顶层结构：**

保存和更新就绪和完成进程的队列。成员函数功能主要有初始化，输入进程，运行就绪队列中的进程和打印进程信息。

class OS**{**

deque**<**PCB**>** ReadyDeque**;**//就绪队列

deque**<**PCB**>** EndDeque**;**//完成队列

public**:**

OS**(){}**

void Initial**();**//初始化

void Input**();**//输入信息

void Run**();**//运行就绪队列中的进程

void Show**(**bool ready**);**//打印队列进程信息 ready为true打印就绪队列 false打印完成队列

**};**

主要代码段分析：

**进程运行函数：**

每次运行时先检查PCB是否合法，如果合法则正常运行，将时间和优先级减一。

bool PCB**::**Run**()**

**{**

**if** **(**time**<**1**||!**status**)**//参数错误

**return** **false;**

**else**

**{**

//\*\*\*运行

time**--;**

priority**--;**

**if** **(**time**==**0**)** status **=** **false;**//更新状态

cout**<<**name**<<**" is running."**<<**endl**;**

**return** **true;**

**}**

**}**

**顶层模块进程调度函数：**

当输入回车时开始进行进程调度，在就绪队列中按优先级排序，然后运行优先级最高的进程。如果进程完成，则将进程加入完成队列。

void OS**::**Run**()** **{**

getchar**();**//吞入上一个回车输入

int num **=** 1**;**//记录轮数

**while** **(!**ReadyDeque**.**empty**())**//就绪队列非空

**{**

colour**(**RED**);**cout **<<** "Turn " **<<** num **<<**": "**;**

ReadyDeque**.**begin**()->**Run**();**//运行优先度最高的进程 此进程优先级和时间均减一

**if** **(**ReadyDeque**.**begin**()->**GetStatus**()** **==** **false)**//队首进程完成则出队并添加入完成队列

**{**

EndDeque**.**push\_back**(**ReadyDeque**[**0**]);**

ReadyDeque**.**pop\_front**();**

**}**

**else**//否则再次加入队列尾部

**{**

ReadyDeque**.**push\_back**(**ReadyDeque**[**0**]);**

ReadyDeque**.**pop\_front**();**

**}**

sort**(**ReadyDeque**.**begin**(),**ReadyDeque**.**end**());**//再次排序就绪队列

Show**(true);**//打印排序后的就绪队列

**if** **(!**EndDeque**.**empty**())**//如果完成队列非空

Show**(false);**//打印完成队列

getchar**();**//等待输入后继续

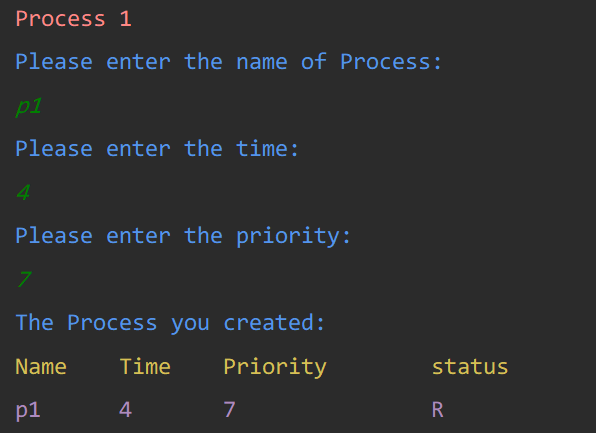
num**++;**//轮数加一

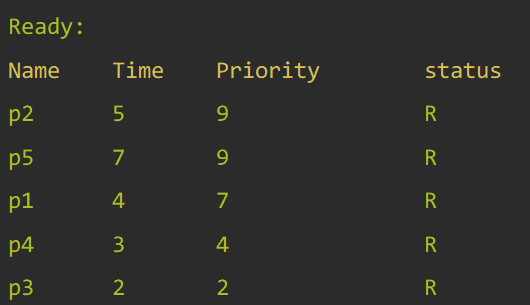
**}**

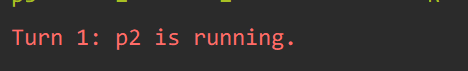
cout**<<**"All end."**;**//全部完成

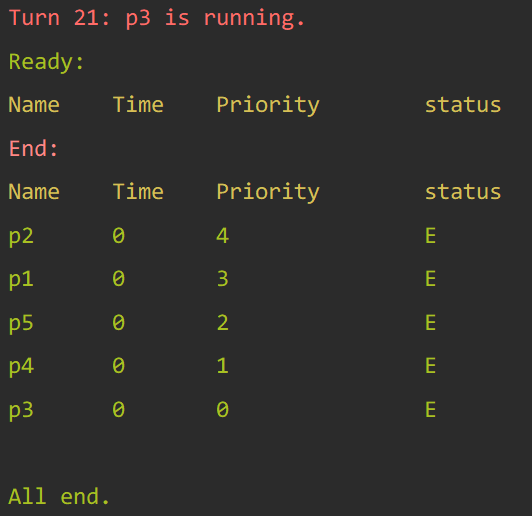
**}**

## 调试过程

进程创建：输入进程名，运行时间和优先级。

创建完后按优先级排序并显示就绪进程队列。

然后按顺序运行。

最终运行结果：与预期相符。

## 实验总结

在实验中经常出现由于一个进程运行后的优先数和另一个进程并列第一，而导致一个进程连续运行两次的问题。经过排查，是因为队列排序使用的稳定排序。因此我将每次运行后的进程出队后再进队然后再排序。这样就避免了由于优先数并列第一导致的连续运行。

通过本实验，我加深了对进程调度基本原理的理解，锻炼了设计模式的能力。

# 主存空间的分配和回收

## 实验内容及上机实验所用平台

实验内容：存储空间的分配和回收。

上机实验所用平台：JetBrains Clion on Windows10。编程语言：C++。

## 数据结构及代码段分析

数据结构：

**空闲分区链节点：**

存储空闲分区的起始地址、占用空间和下一个节点的地址。支持分配、合并、显示功能。

class Node**{**//空闲分区链的节点

int address**;**//空闲分区起始地址

int memory**;**//占用空间

public**:**

Node **\*** next**;**//下一个节点

Node**():**address**(**0**),**memory**(**0**),**next**(nullptr){}**

Node**(**int ad**,** int mem**):**address**(**ad**),**memory**(**mem**),**next**(nullptr){}**

int GetMem**(){return** memory**;}**//返回剩余空间

int GetAddr**(){return** address**;}**//返回地址

void Allocate**(**int mem**){**address**+=**mem**;**memory**-=**mem**;}**//分配mem大小的空间

void Merge**(**int mem**){**memory**+=**mem**;}**//合并mem大小的空间

void Show**(**int n **=** 0**)**

**{**

colour**(**BLUE**);**cout**<<**n**<<**"\t"**<<**address**<<**"\t"**<<**memory**<<**"\t"**<<**endl**;**

**}**

void Swap**(**Node **\*** n**)**

**{**

swap**(**address**,**n**->**address**);**

swap**(**memory**,**n**->**memory**);**

**}**

**};**

**分区链表：**

存放链表头结点，成员函数主要负责完成首次适应算法分配和合并空闲分区。

class LinkList**{**//分区链

Node **\*** head**;**//链表头结点

public**:**

LinkList**(){**head**=new** Node**(-**1**,-**1**);}**

void AddNode**(**int ad**,**int mem**);**//在尾部添加节点

Node **\*** FindMem**(**int mem**);**//找到第一个空间大于mem的节点

Node **\*** FindNum**(**int num**);**//返回第num个节点

Node **\*** FindAddr**(**int addr**);**//返回地址为addr的节点

void Delete**(**Node **\*** Dnode**);**//删除一个节点

void Show**();**//打印信息

void Sort**();**//从低地址向高地址排序 地址交换 非值交换

void Merge**(**Node **\*** node**);**//用于合并空闲分区前后的空闲分区

**};**

**顶层模块：**

存储空闲分区链表，获取输入，调用分配和回收函数。

class OS**{**

LinkList FreeLinkList**;**//空闲分区链

LinkList AllocLinkList**;**//存储分配的分区

public**:**

void Initial**(){**FreeLinkList**.**AddNode**(**0**,** MaxMemory**);}**//初始化

OS**(){}**

void Input**();**//获取输入

bool Allocate**();**//分配

bool Recycle**();**//回收

**};**

主要代码段分析：

**首次适应算法：**

从前向后遍历空闲分区链，如果找到一个满足大小要求的节点，则返回该节点。

Node **\***LinkList**::**FindMem**(**int mem**)** **{**

Node **\*** node **=** head**->**next**;**

**while** **(**node**!=** **nullptr)**

**{**

**if** **(**node**->**GetMem**()>=**mem**)**//如果空间大于要求

**{**

**return** node**;**//返回该节点

**}**

node**=**node**->**next**;**

**}**

**return** **nullptr;**//没找到返回空指针

**}**

**合并空闲分区：**

如果回收的节点前面或后面有连续的空闲分区，则合并空闲分区。合并顺序：先尝试合并后面的节点与当前节点，然后尝试合并当前节点与前面的节点。

void LinkList**::**Merge**(**Node **\***node**)** **{**

**if** **(**node**==** **nullptr)** **return;**

Node**\***FrontNode**=**head**;**//前节点

**while(**FrontNode**->**next**!=**node **&&** FrontNode**!=** **nullptr)**//找到node前面的节点

FrontNode**=**FrontNode**->**next**;**

Node**\***BackNode**=**node**->**next**;**//后节点

**if** **(**BackNode**!=nullptr)**//尝试合并后面与当前的节点

**{**

**if** **(**node**->**GetAddr**()+**node**->**GetMem**()==**BackNode**->**GetAddr**())**//满足合并条件

**{**

node**->**Merge**(**BackNode**->**GetMem**());**//合并节点

Delete**(**BackNode**);**//删除后节点

**}**

**}**

**if** **(**FrontNode**->**GetAddr**()+**FrontNode**->**GetMem**()==**node**->**GetAddr**())**//满足合并前节点与当前节点

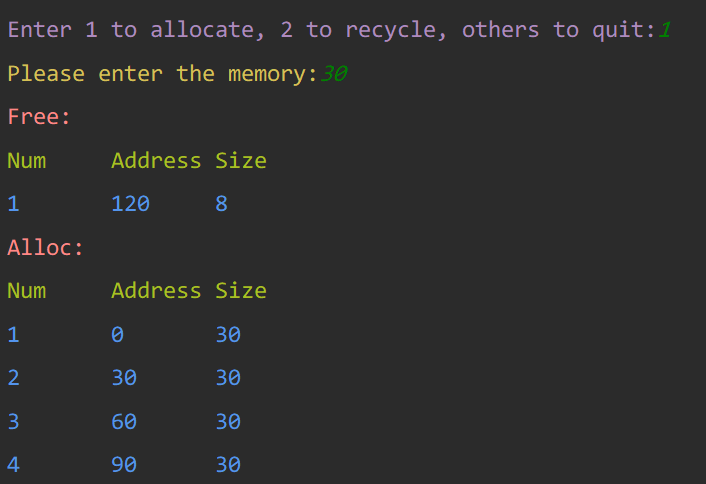
**{**

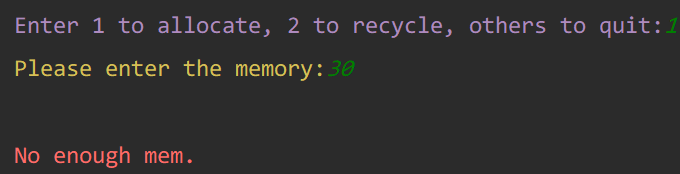
FrontNode**->**Merge**(**node**->**GetMem**());**//合并节点

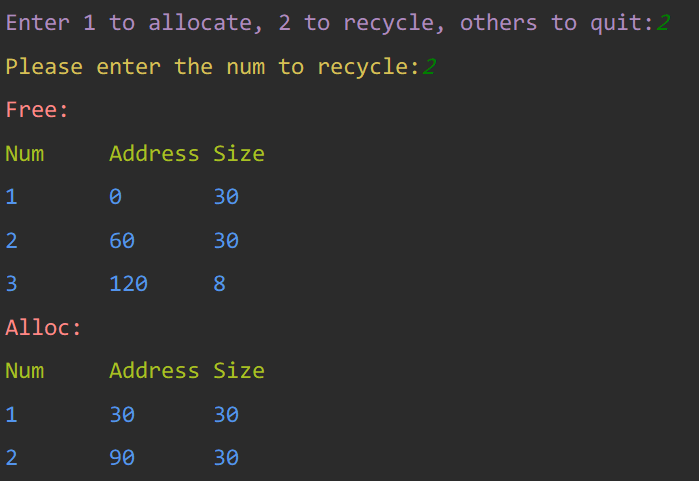
Delete**(**node**);**//删除当前节点

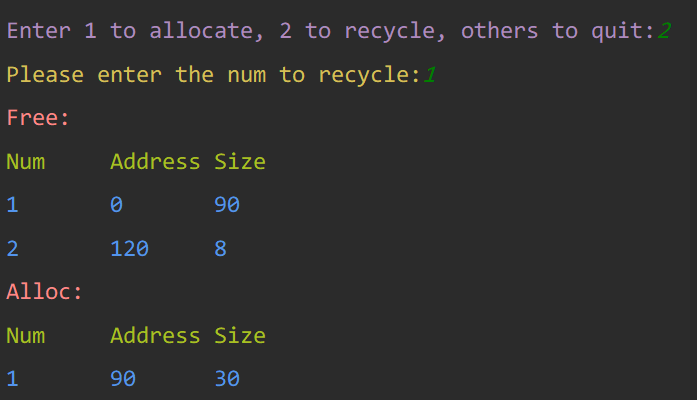
**}**

## 调试过程

尝试分配4次，每次分配30KB：

再次分配30KB，空间不足，分配失败：

回收两块不连续的空间：

回收中间那块空间，观察合并结果：

结果与预期相符。

## 实验总结

在处理合并空闲分区节点的时候，我一开始也和很多同学一样，把情况分成了四种：该节点前后均没有空闲分区，该节点前后均有空闲分区，该节点前面有空闲分区后面没有空闲分区和该节点前面没有空闲分区，后面有空闲分区。对于每种情况使用不同的合并算法。但我后来发现只需要依序处理两次即可：先查看后面是否有空闲分区，如果有，则和当前分区合并；再查看前面是否有空闲分区，如果有，则和当前分区合并。这样只经过两步操作，即可包括四种情况，简化了代码和流程。

我通过本实验加深了对空闲分区分配的首次适应算法的理解，锻炼了自己的思考能力和数据结构思想。

# 磁盘存储空间的分配和回收

## 实验内容及上机实验所用平台

实验内容：模拟磁盘空闲空间的表示方法，以及模拟实现磁盘空间的分配和回收。上机实验所用平台：JetBrains Clion on Windows10。编程语言：C++。

## 数据结构及代码段分析

数据结构：

**磁盘：**

保存位示图，标记任务，成员函数负责分配和回收空间。

class Disk**{**//磁盘

int Memory**[**TotalRecordsNum**];**//记录磁盘空间使用情况 0为空闲 非0为占用标志

int FreeNum**;**//总空闲大小

int WorkNum**;**//递增标志 同一次分配的块具有相同的标志

public**:**

Disk**(){**memset**(**Memory**,**0**,** **sizeof(**Memory**)/** **sizeof(**int**));}**

void Initial**();**//磁盘初始化

void ShowAll**();**//打印磁盘状态

void ShowByNum**(**int n**);**//打印对应标志位的地址

int Allocate**(**int blockNum**);**//分配函数 返回标志号 0代表分配失败

bool Recycle**(**int workNum**);**//回收函数，返回是否回收成功

**};**

主要代码段分析：

**分配函数：**

首先判断总空闲大小是否满足，如果满足则从开头开始分配空间。

int Disk::Allocate(int blockNum)//分配函数 返回标志号 0代表分配失败

{

if (blockNum>FreeNum)//空间不足

{

return 0;

}

else

{

FreeNum-=blockNum;

WorkNum++;//标志加一

for (int i=0;i<TotalRecordsNum;i++)

{

if (Memory[i]==0)

{

Memory[i]=WorkNum;

blockNum--;

}

if (blockNum==0) break;

}

return WorkNum;

}

}

**回收函数：**

找到磁盘中所有该文件占用的块，然后释放这些块，并更新空闲大小。

bool Disk**::**Recycle**(**int workNum**)**//回收函数，返回是否回收成功

**{**

**if** **(**workNum**<=**0**)**

**return** **false;**

bool ReturnFlag **=** **false;**//返回值

ShowByNum**(**workNum**);**

**for** **(**int i**=**0**;**i**<**TotalRecordsNum**;**i**++)**

**{**

**if** **(**Memory**[**i**]==**workNum**)**

**{**

FreeNum**++;**//空闲块加一

ReturnFlag**=true;**

Memory**[**i**]=**0**;**//回收空间

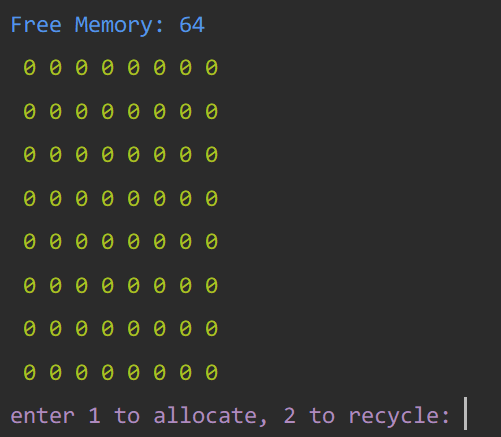
**}**

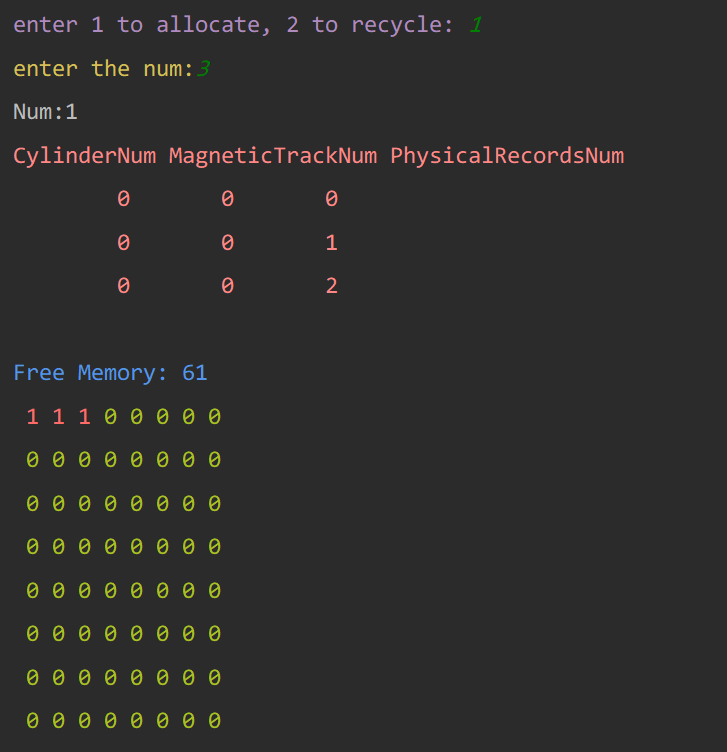
**}**

**return** ReturnFlag**;**

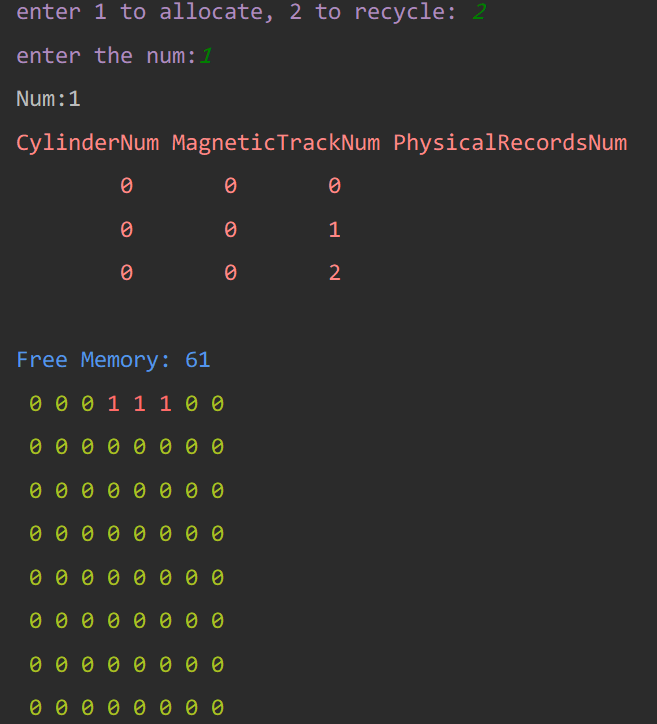
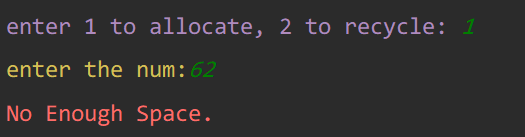
**}**

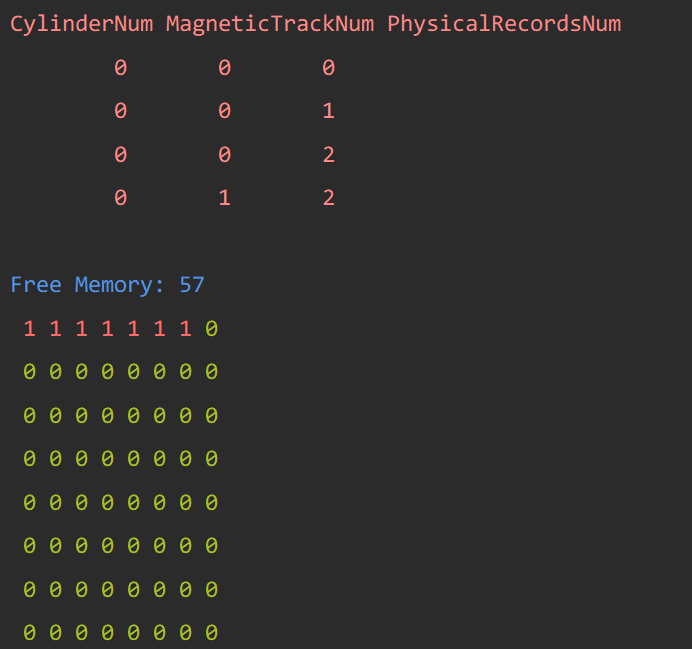
## 调试过程

初始状态：

尝试分配3块：

尝试分配62块，显示空间不足：

再分配3块后回收第一次分配的三块：

再分配4块：

与预期结果符合。

## 实验总结

在第一次验收时，由于错误采用了最坏适应算法，导致了分配结果与预期不符，经过了老师和同学的帮助后成功将其修改。

经过这次实验，加深了对位示图的理解。不过也引起了我对于该模拟程序的一些思考。我觉得我这种以数组内容标记同一个作业申请块的方法效率比较低，也许使用一些其他的数据结构来记录同一个作业申请的块地址效率会更高。

# 进程创建

## 实验内容及上机实验所用平台

实验内容：利用fork()系统调用创建进程。

上机实验所用平台：Visual Code on CentOS7。编程语言：C++。

## 数据结构及代码段分析

数据结构：无

代码段分析：

Fork函数创建子进程，其返回值<0代表创建失败，>0代表当前为父进程，=0代表为当前为子进程。下面这段代码创建了两个子进程，按照创建的先后顺序分别打印B和C还有其pid和ppid。父进程则打印A和pid。由于进程的调用没有严格的顺序，因此输出结果可能为ACB或ABC。

int main**()**

**{**

pid\_t pid1 **=** fork**();**

pid\_t pid2**;**

**if** **(**pid1**<**0**)**

cout**<<**"Error"**<<**endl**;**

**else** **if** **(**pid1**==**0**)**

cout**<<**"B,ppid:"**<<**getppid**()<<**",pid:"**<<**getpid**()<<**endl**;**

**else**

**{**

cout**<<**"A,pid:"**<<**getpid**()<<**endl**;**

pid2**=**fork**();**

**if** **(**pid2**<**0**)** cout**<<**"Error"**<<**endl**;**

**else** **if** **(**pid2**==**0**)**

cout**<<**"C,ppid:"**<<**getppid**()<<**",pid:"**<<**getpid**()<<**endl**;**

**}**

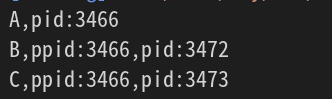
waitpid**(**pid1**,**0**,**0**);**

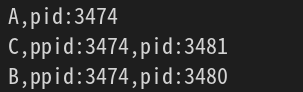
waitpid**(**pid2**,**0**,**0**);**

**return** 0**;**

**}**

## 调试过程

两种输出结果：由于子进程调用顺序的差异导致的输出不同。



## 实验总结

通过此次实验，加深了我对子进程创建和Linux系统与windows系统之间的差异的认识。在验收时，老师也问了个问题：如果把最外层的else语句去掉会打印什么输出。而我根据fork语句的特点，回答出了：由于没有else语句来判断当前是否为父进程，这导致父进程一开始创建的子进程会跟父进程进行在原else语句后一模一样的调用。这将导致最后输出一个B，两个A和两个C。

# 进程同步

## 实验内容及上机实验所用平台

实验内容：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。

上机实验所用平台：JetBrains Clion on Windows10。编程语言：C++。

## 数据结构及代码段分析

数据结构：

**进程控制块：**

负责记录进程名，进程状态，进程等待原因和进程断点。

class PCB**{**//进程控制块

public**:**

string name**;**//进程名

STATUS status**;**//进程状态

WAITINGREASON waitingReason**;**//进程等待原因

int Breakpoint**;**//进程断点

PCB**():**status**(**STATUS**::**READY**),**waitingReason**(**WAITINGREASON**::**None**){}**

**};**

**进程：**

记录调用次数，每个进程都有个PCB。成员函数用来判断进程是否结束。

class Process**{**

public**:**

PCB pcb**;**

int count**;**//记录调用次数

Process**(**string name**):**count**(**0**){**pcb**.**name**=**name**;}**

bool IsEnd**(){return** pcb**.**status**==**STATUS**::**END**;}**//判断进程是否结束

**};**

**生产者进程：**

继承进程类，存放生产出来的产品和生产者指令。成员函数包括生产者P操作、生产者V操作、Produce操作和Put操作。

class ProcessProducer**:**public Process**{**//生产者进程

int Product**;**//产品

public**:**

enum PRODUCER**{**//生产者指令 PA

PRODUCE**=**0**,**

P\_S1\_**,**

PUT**,**

V\_S2\_**,**

GO\_TO\_0

**};**

ProcessProducer**(**string name**):**Process**(**name**),**Product**(**0**)**

**{**pcb**.**Breakpoint**=**PRODUCER**::**PRODUCE**;}**//默认断点为第一条命令

virtual bool P**(**semaphore **&**s**);**//生产者P操作

virtual WAITINGREASON V**(**semaphore **&**s**);**//生产者V操作

void Produce**();**//生产函数

void Put**(**deque**<**int**>&** Buffer**);**//放入缓冲区

**};**

**消费者进程：**

继承进程类，存放取得的产品和消费者指令。成员函数包括消费者P操作、消费者V操作、Get操作和Consume操作。

class ProcessConsumer**:**public Process**{**//消费者进程

int Product**;**//产品

public**:**

enum CONSUMER**{**//消费者指令 SA

P\_S2\_**=**6**,**

GET**,**

V\_S1\_**,**

CONSUME**,**

GO\_TO\_0

**};**

ProcessConsumer**(**string name**):**Process**(**name**),**Product**(**0**)**

**{**pcb**.**Breakpoint**=**CONSUMER**::**P\_S2\_**;}**//默认断点为第一条命令

virtual bool P**(**semaphore **&**s**);**//消费者P操作

virtual WAITINGREASON V**(**semaphore **&**s**);**//消费者V操作

void Get**(**deque**<**int**>&** Buffer**);**//消费者取得产品

void Consume**();**//消费者消费产品

**};**

**CPU：**

根据进程断点运行程序，同时向操作系统返回进程调度的信号量。

class CPU**{**//负责运行进程，修改信号量，缓冲区和进程状态。

private**:**

semaphore s1**;**//信号量1

semaphore s2**;**//信号量2

deque**<**int**>** Buffer**;**//缓冲区

public**:**

void Execute**(**Process **\*** process**,**bool **&** block**,**WAITINGREASON **&** wake**);**//执行 阻塞和唤醒通知操作系统

CPU**():**s1**(**BufferSize**),**s2**(**0**){}**

**};**

**操作系统：**

负责维护进程的就绪队列和阻塞队列，将CPU资源分配给当前运行的进程。根据CPU返回的信号阻塞或唤醒进程。

class OS**{**

private**:**

CPU cpu**;**

deque**<**Process**\*>** ReadyQueue**;**//就绪队列

deque**<**Process**\*>** BlockQueue**;**//阻塞队列

public**:**

OS**()** **{}**

void Initial**(){**//初始化

ProcessConsumer **\*** processConsumer **=** **new** ProcessConsumer**(**"c1"**);**//创建消费者进程

ProcessProducer **\*** processProducer **=** **new** ProcessProducer**(**"p1"**);**//创建生产者进程

//将进程添加入就绪队列

ReadyQueue**.**push\_back**(**processProducer**);**

ReadyQueue**.**push\_back**(**processConsumer**);**

**}**

void Run**();**//运行

Process **\*** Dispatch**();**//进程调度

void Wake**(**WAITINGREASON wake**);**//进程唤醒

void Block**(**Process**\*** process**);**//进程阻塞

void End**(**Process**\*** process**);**//进程结束

**};**

代码段分析：

**CPU执行代码：**

根据进程断点所对应的指令，选择相应的函数运行。如果运行导致了进程阻塞或者需要唤醒进程，则通过返回值通知操作系统。

void CPU**::**Execute**(**Process **\*** process**,**bool **&** OutBlock**,**WAITINGREASON **&** OutWake**)** **{**//执行 阻塞和唤醒通知操作系统

**if** **(**process**==** **nullptr)**

**return;**

cout**<<**"Process :"**<<**process**->**pcb**.**name**<<**" "**;**

process**->**count**++;**//调用次数加一

ProcessConsumer**\*** processConsumer**;**

ProcessProducer**\*** processProducer**;**

bool block **=** **false;**//是否阻塞

WAITINGREASON wake **=** None**;**//唤醒信号量

**switch** **(**process**->**pcb**.**Breakpoint**)**//从断点开始执行 模拟取指令和运行

**{**

**case** ProcessProducer**::**PRODUCE**:{**

processProducer **=** **(**ProcessProducer **\*)** process**;**

processProducer**->**Produce**();**

**break;**

**}**

**case** ProcessProducer**::**P\_S1\_**:{**

processProducer **=** **(**ProcessProducer **\*)** process**;**

block **=** processProducer**->**P**(**s1**);**

**break;**

**}**

**case** ProcessProducer**::**PUT**:{**

processProducer **=** **(**ProcessProducer **\*)** process**;**

processProducer**->**Put**(**Buffer**);**

**break;**

**}**

**case** ProcessProducer**::**V\_S2\_**:{**

processProducer **=** **(**ProcessProducer **\*)** process**;**

wake **=** processProducer**->**V**(**s2**);**

**break;**

**}**

**case** ProcessProducer**::**GO\_TO\_0**:{**

OutBlock **=** **false;**

OutWake **=** None**;**

cout**<<**"Producer Go To 0."**<<**endl**;**

process**->**pcb**.**Breakpoint**=**ProcessProducer**::**PRODUCE**;**

**if** **(**process**->**count**>=**EndNum**)**

process**->**pcb**.**status**=**STATUS**::**END**;**

**return;**

**}**

**case** ProcessConsumer**::**P\_S2\_**:{**

processConsumer **=** **(**ProcessConsumer**\*)** process**;**

block **=** processConsumer**->**P**(**s2**);**

**break;**

**}**

**case** ProcessConsumer**::**GET**:{**

processConsumer **=** **(**ProcessConsumer**\*)** process**;**

processConsumer**->**Get**(**Buffer**);**

**break;**

**}**

**case** ProcessConsumer**::**V\_S1\_**:{**

processConsumer **=** **(**ProcessConsumer**\*)** process**;**

wake **=** processConsumer**->**V**(**s1**);**

**break;**

**}**

**case** ProcessConsumer**::**CONSUME**:{**

processConsumer **=** **(**ProcessConsumer**\*)** process**;**

processConsumer**->**Consume**();**

**break;**

**}**

**case** ProcessConsumer**::**GO\_TO\_0**:{**

OutBlock **=** **false;**

OutWake **=** None**;**

cout**<<**"Consumer Go To 0."**<<**endl**;**

process**->**pcb**.**Breakpoint**=**ProcessConsumer**::**P\_S2\_**;**

**return;**

**}**

**}**

process**->**pcb**.**Breakpoint**++;**//指令移往下一条

//向操作系统返回信号量

OutBlock**=**block**;**

OutWake**=**wake**;**

**}**

**操作系统运行进程的函数：**

每次从就绪队列中随机挑选一个进程执行，将CPU资源分配给该进程。如果CPU向操作系统返回了阻塞信号，则阻塞该进程，将其移出就绪队列并加入阻塞队列。如果CPU向操作系统返回了唤醒信号，则根据唤醒信号在阻塞队列中寻找需要被唤醒的进程，然后将这个进程从阻塞队列中移出并加入就绪队列。

void OS**::**Run**()** **{**

**while** **(true)**

**{**

bool block **=** **false;**//阻塞信号

WAITINGREASON wake **=** None**;**//唤醒信号

Process**\*** process **=** Dispatch**();**//选择一个进程开始运行

cpu**.**Execute**(**process**,**block**,**wake**);**//将CPU资源分配给进程，监听阻塞与唤醒

**if** **(**block**)**//发生阻塞 权限交给操作系统

**{**

Block**(**process**);**//阻塞该进程

**}**

**if** **(**wake **!=** None**)**//需要操作系统唤醒某进程

**{**

Wake**(**wake**);**

**}**

**if** **(**process**->**IsEnd**())**//结束进程

**{**

End**(**process**);**

**}**

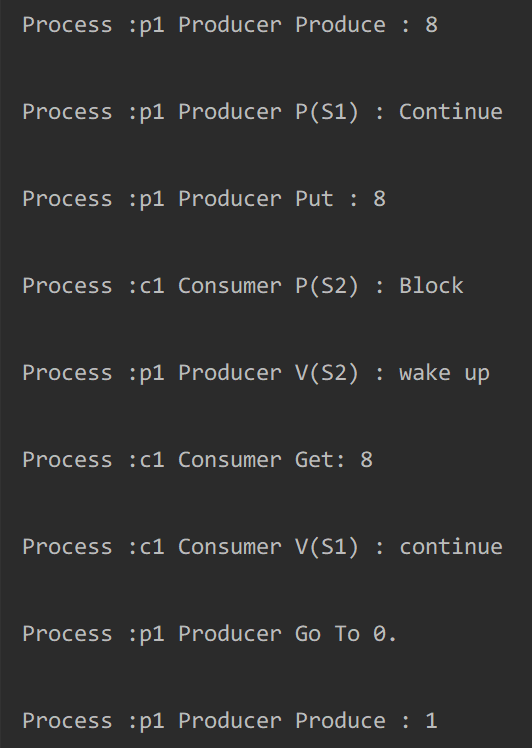
getchar**();**

**}**

**}**

## 调试过程

1种调试情况：生产者进程先生产产品8，然后P（S1），然后将产品8放入缓冲区。此时随机到消费者运行，消费者执行P（S2），由于S2为0，-1后为负数，故消费者进入阻塞态。此时只能调用生产者，生产者进行V（S2）操作，将S2由-1变为0，此时需要操作系统唤醒之前阻塞的消费者。接下来消费者取得产品8，然后V（S1）。生产者执行GoTo0操作，进入下一个循环。均与预期相符。



## 实验总结

在这次实验中，我完成了进程同步的模拟。在代码过程中，我一直在努力按照计算机各体系的功能来抽象各部分，包括底层的进程，进程控制块，CPU，高层的操作系统。比如在进程中保存数据、代码段和进程控制块，在进程控制块中保存进程状态，断点等信息，在CPU中根据进程断点运行不同的指令，在操作系统中调度进程等。但由于有些地方运用抽象和设计模式还不够熟练，导致各模块的耦合度还是有点略高。但经过此次试验，我仍然锻炼了自己的抽象设计能力，加深了对进程同步的理解。

# 参考文献

1. 郑鹏,曾平,金晶.计算机操作系统第2版[M]:武汉大学出版社,2014:28-29，86-87.
2. 郑鹏,曾平,金晶.计算机操作系统第2版[M]:武汉大学出版社,2014:113.
3. 郑鹏,曾平,金晶.计算机操作系统第2版[M]:武汉大学出版社,2014:189.
4. <https://www.cnblogs.com/dongguolei/p/8086346.html>
5. 郑鹏,曾平,金晶.计算机操作系统第2版[M]:武汉大学出版社,2014:56-58.

# 附录

全部代码详见<https://github.com/A-Salty-Fish/OSProject/>