

合肥工业大学

系统软件综合设计报告

操作系统分册

设计题目	段式虚拟存储管理系统
学生姓名	郭子煜
学 号	2018217845
专业班级	物联网 18-1 班
指导教师	田卫东
完成日期	2021.6.29

目录

1. 课程设计任务、要求、目的	3
1.1. 课程设计任务	3
1.2. 课程设计目的和要求	3
2. 开发环境	3
3. 相关原理及算法	3
3.1. 段式虚拟存储管理	3
3.1.1. 基本分段存储管理	3
3.1.2. 请求分段系统（段式虚拟存储器，分段请求系统）	6
3.2. 段面置换算法	8
4. 系统结构和主要的算法设计思路	9
4.1 内存的空间管理	9
4.2 段面置换算法	14
4.3 内存地址映射	14
5. 程序实现——主要数据结构	16
6. 程序实现——主要程序清单	17
7. 程序运行的主要界面和结果截图	17
1) 创建进程	18
2) 地址映射（中断功能及段面置换算法 FIFO 的实现）	19
8. 总结和感想体会	23
参考文献	23

1. 课程设计任务、要求、目的

1.1. 课程设计任务

依据操作系统课程所介绍的固定分区分配的存储管理方案,按照内河代码的实现原则,设计和实现一个固定分区分配存储管理系统。

1.2. 课程设计目的和要求

系统应该包含两个部分,一个部分是按内核代码原则设计的固定分区分配存储管理系统,由一系列的函数组成;另一个部分是演示系统,调用固定分区分配存储管理系统的相应函数,以让其运行,同时提供系统的展示界面,可以是 GUI 或者字符界面,以展示系统的运行状态,显示系统的关键数据结构的内容。

具体包括:

- 首先分配一片较大的内存空间和一段磁盘空间,作为程序运行的可用存储空间和外存交换区;
- 建立应用程序的模型,包括分段结构在内;
- 建立进程的基本数据结构及相应算法
- 建立管理存储空间的基本存储结构。
- 建立管理段的基本数据结构与算法。
- 设计存储空间的分配与回收算法;
- 实现缺段中段支持的逻辑地址到物理地址转换,实现虚拟存储器;
- 提供信息转储功能,可将存储信息存入磁盘,也可从磁盘读入;

2. 开发环境

系统环境: Win10

软件环境: visual studio 2019

3. 相关原理及算法

3.1. 段式虚拟存储管理

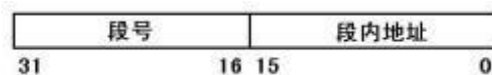
3.1.1. 基本分段存储管理

■ 分段

在分段存储管理方式中,作业的地址空间被划分为若干个段,每个

段定义了一组逻辑信息。例如，有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等，如图所示。每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从 0 开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间由于是分成多个段，因而是二维的，亦即，其逻辑地址由段号(段名)和段内地址所组成。

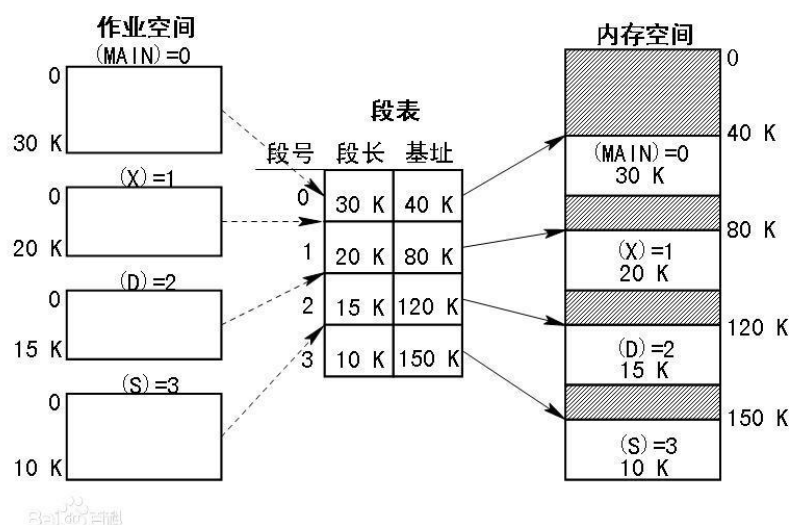
分段地址中的地址具有如下结构：



在该地址结构中，允许一个作业最长有 64 K 个段，每个段的最大长度为 64 KB。分段方式已得到许多编译程序的支持，编译程序能自动地根据源程序的情况而产生若干个段。例如，Pascal 编译程序可以为全局变量、用于存储相应参数及返回地址的过程调用栈、每个过程或函数的代码部分、每个过程或函数的局部变量等等，分别建立各自的段。类似地，Fortran 编译程序可以为公共块(Common block)建立单独的段，也可以为数组分配一个单独的段。装入程序将装入所有这些段，并为每个段赋予一个段号。

■ 段表

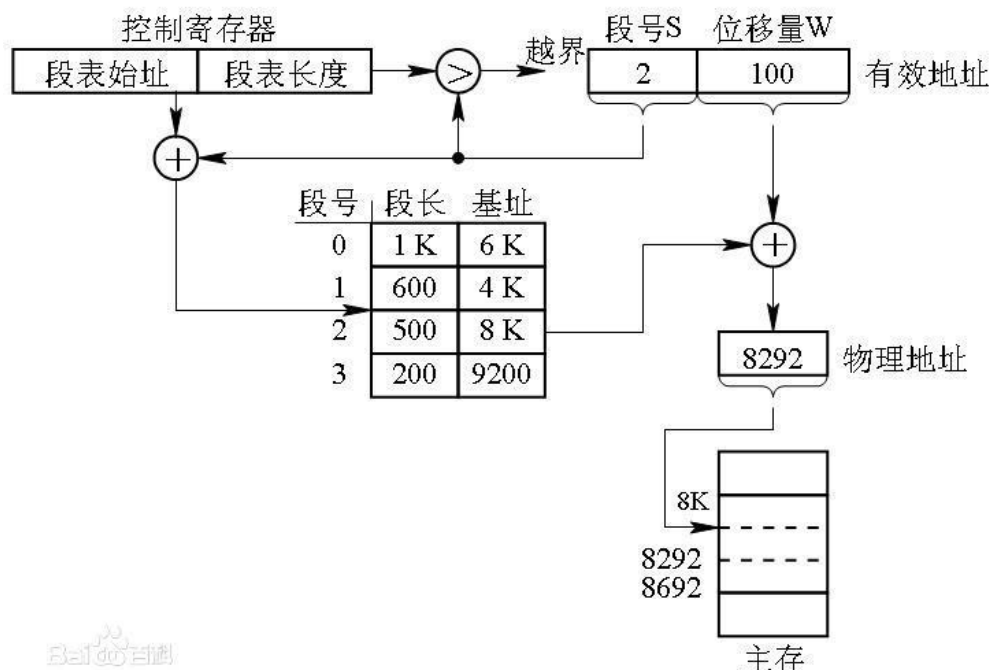
在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区，而进程中的各个段可以离散地移入内存中不同的分区中。



为使程序能正常运行，亦即，能从物理内存中找出每个逻辑段所对应的位置，在系统中为每个进程建立一张段映射表，简称“段表”。每个段在表中占有一个表项，其中记录了该段在内存中的起始地址(又称为“基址”)和段的长度，如上图所示。段表可以存放在一组寄存器中，这样有利于提高地址转换速度，但更常见的是将段表放在内存中。

■ 地址变换

为了实现从进程的逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表始址和段表长度 TL。在进行地址变换时，系统将逻辑地址中的段号与段表长度 TL 进行比较。若 $S > TL$ ，表示段号太大，是访问越界，于是产生越界中断信号；若未越界，则根据段表的始址和该段的段号，计算出该段对应段表项的位置，从中读出该段在内存的起始地址，然后，再检查段内地址 d 是否超过该段的段长 SL。若超过，即 $d > SL$ ，同样发出越界中断信号；若未越界，则将该段的基址 d 与段内地址相加，即可得到要访问的内存物理地址。



上图示出了分段系统的地址变换过程。当段表放在内存中时，每要访问一个数据，都须访问两次内存，从而极大地降低了计算机的速率。

解决的方法：再增设一个联想存储器，用于保存最近常用的段表项，便可以显著地减少存取数据的时间。

3.1.2. 请求分段系统（段式虚拟存储器，分段请求系统）

在基本分段系统基础上，增加以分段为单位的请求调入和自动置换功能。在请求分段存储管理系统中，作业运行之前，只要求将当前需要的若干个分段装入内存，便可启动作业运行。在作业运行过程中，如果要访问的分段不在内存中，则通过调段功能将其调入，同时还可以通过置换功能将暂时不用的分段换出到外存，以便腾出内存空间。

为实现请求分段系统，系统应该配置段表机制、缺段中断机构、机制变换机构硬件支持。

■ 段表机制

段名	段长	段的基址	存取方式	访问字段A	修改位M	存在位P	增补位	外存始址
----	----	------	------	-------	------	------	-----	------

存取方式：存取属性（执行、只读、允许读/写）

访问字段 A：记录该段被访问的频繁程度

修改位 M：表示该段在进入内存后，是否被修改过。

存在位 P：表示该段是否在内存中。

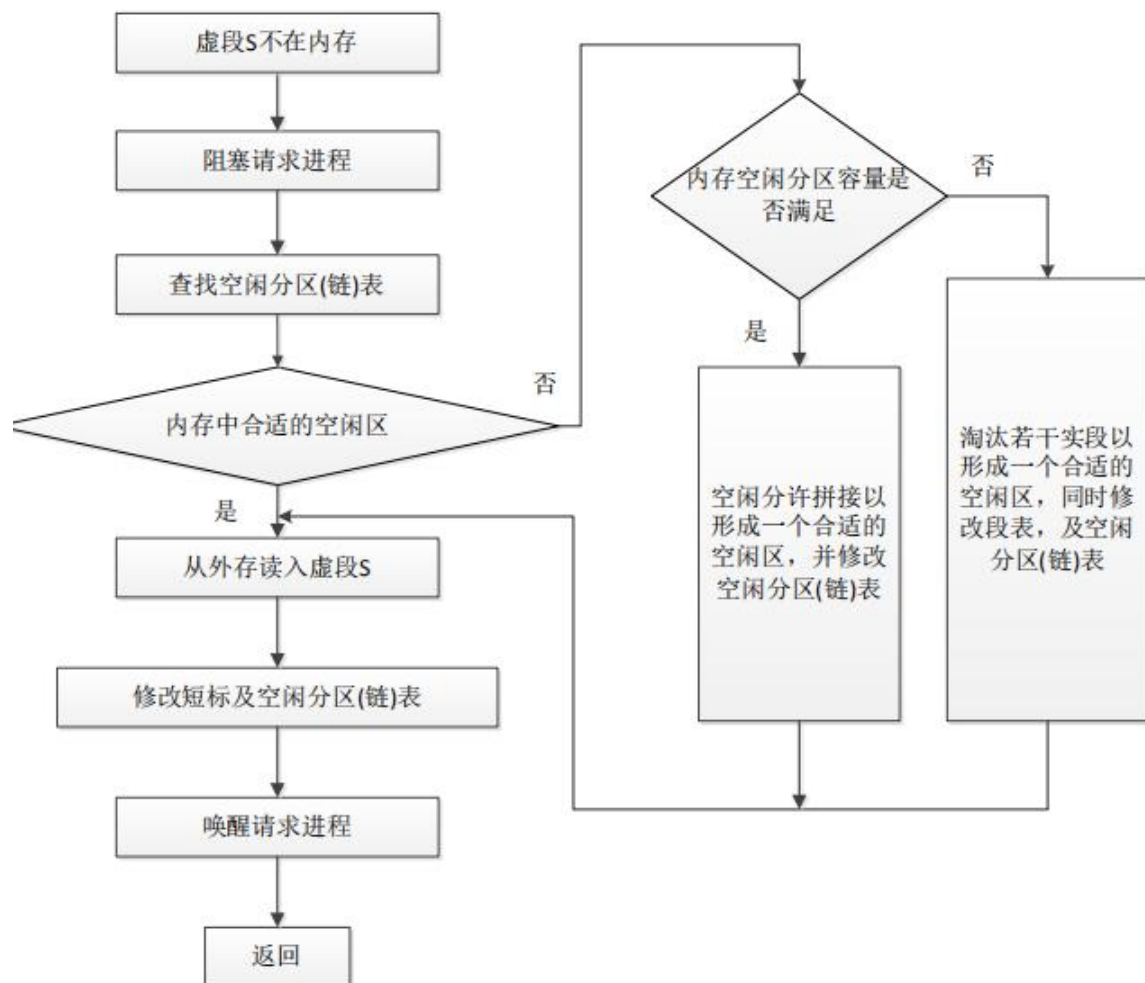
增补位：表示在运行过程中，该段是否做过动态增长。

外存地址：表示该段在外存中的起始地址。

■ 缺段中断机制

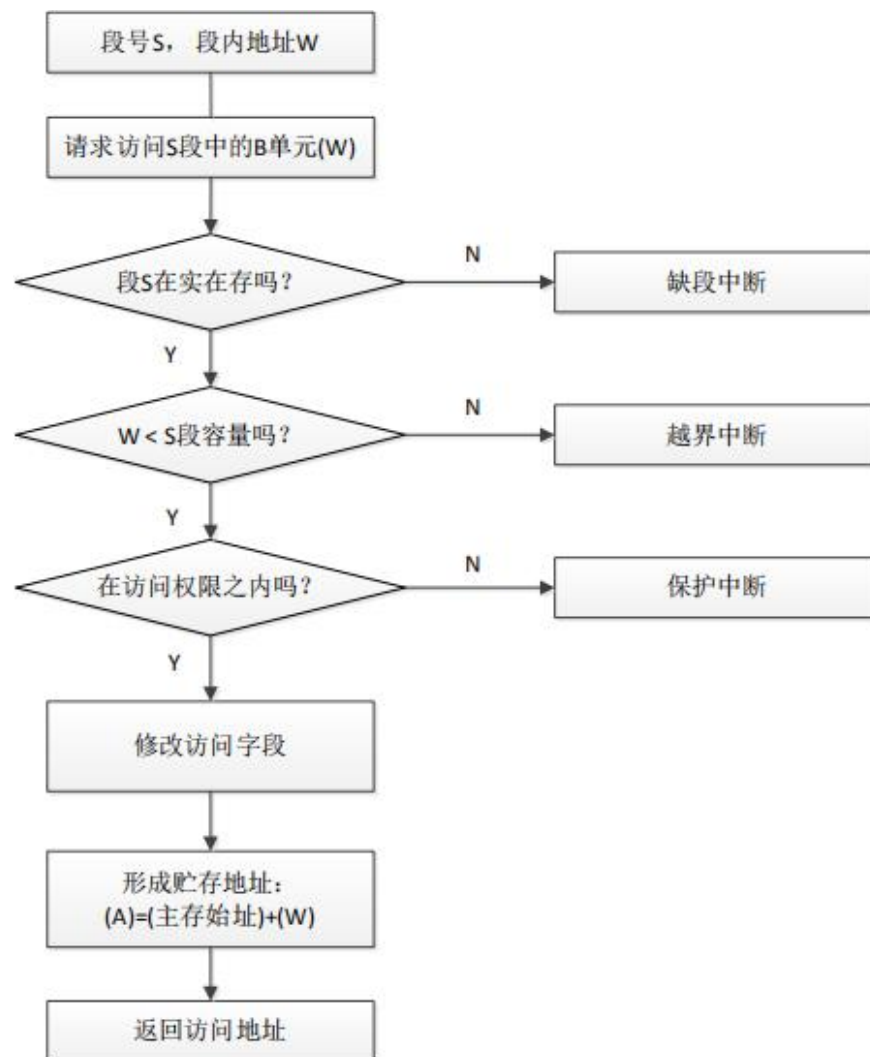
缺段中断机构与缺页中断机构类似，它同样需要在一条指令执行期间，产生和处理中断，以及一条指令执行期间可能产生多此缺段中断。由于分段是信息的逻辑单元，因而不可能出现一条指令被分割在两个分段中和一个信息被分割在两个分段中的情况。

缺段中断的处理过程如图：



■ 地址变换

请求分段系统中地址变换机构是在分段系统地址变换机构的基础上形成的。因为被访问的段并非全在内存，所以在地址变换时，若发现所要访问的段不在内存，必须先将所缺的段调入内存，并修改段表，然后才能再利用段表进行地址变换。



3.2. 段面置换算法

■ 先进先出置换算法（FIFO）

当需要淘汰一个段面时，总是选择驻留主存时间最长的段面进行淘汰，即先进入主存的段面先淘汰。

■ 最近最久未使用（LRU）算法

利用局部性原理，根据一个作业在执行过程中过去的段面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的段面，在最近的将来可能也不会再被访问。所以当需要淘汰一个段面时，总是选择在最近一段时间内最久不用的段面予以淘汰。

■ 时钟（CLOCK）置换算法

给每个段面关联一个使用位。当该段第一次装入内存或者被重新访问到时，将使用位置为 1。每次需要替换时，查找使用位被置为 0 的第一个段进行替换。在扫描过程中，如果碰到使用位为 1 的段，将使用位置为 0，在继续扫描。如果所谓段的使用位都为 0，则替换第一个段。

4. 系统结构和主要的算法设计思路

4.1 内存的空间管理

常见的内存分配与回收算法包括：首次拟合法、最佳拟合法、最差拟合法等，但是三种策略都会造成一些内存非常小的空间，从而使这些内存空间无法被分配。为了更有效的利用内存，需要将相邻的空闲的内存合并成一个更大的内存。因此本系统采用边界标识法来解决系统中内存碎片过多而无法使用的问题。

● 结点构成

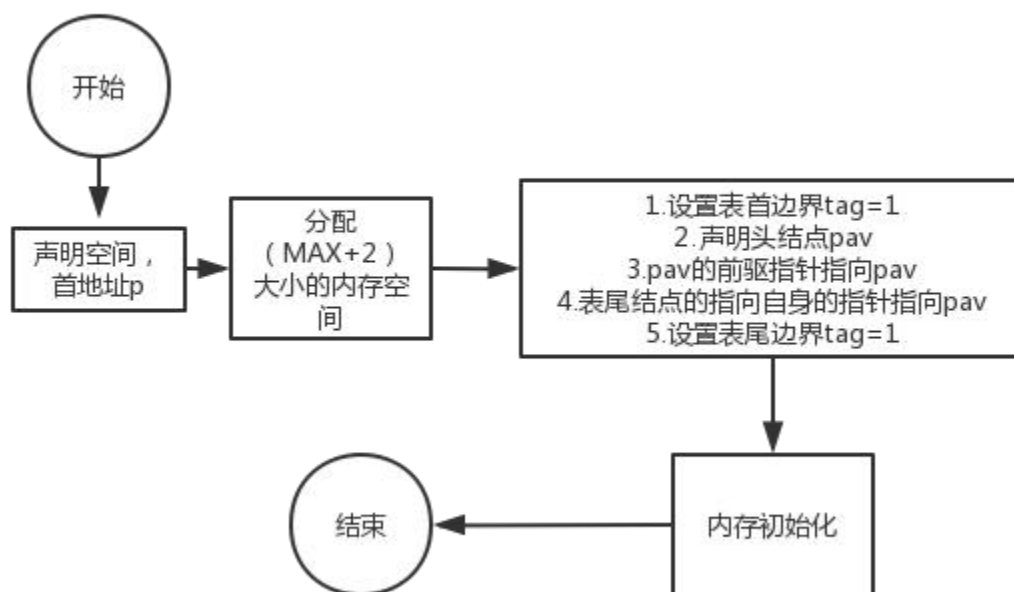
每个结点中包含 3 个区域，head 域、space 域和 foot 域：

- head 域：
llink 和 rlink 分别表示指向当前内存块结点的直接前驱和直接后继。tag 值用于标记当前内存块的状态，是占用块（用 1 表示）还是空闲块（用 0 表示）。size 用于记录该内存块的存储大小
- space 域：
该内存块的大小，它的大小通过 head 域中的 size 值表示
- foot 域：
uplink 是指针域，用于指向内存块本身，通过 uplink 就可以获取该内存块所在内存的首地址。tag 同 head 域中的 tag 相同，都是记录内存块状态的



● 内存初始化

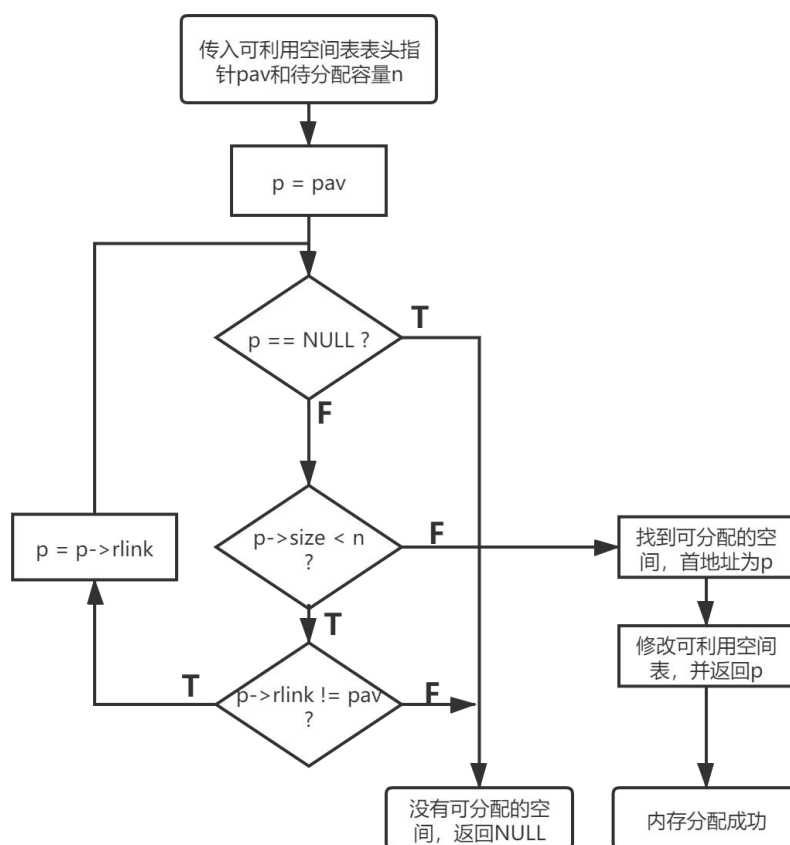
如图 8，首先声明地址空间，首地址为 p，然后分配内存空间（其中 MAX 初始化为 128, 即代表初始化申请一片占用 128+2 个内存字大小的内存空间，其中 2 代表表首边界和表尾边界，以防查找左右邻块时出错），然后设置表首边界、头结点、表尾边界，从而完成初始化工作。



● 内存分配

我们采用首次拟合法进行分配，则只要从表头指 pav 所指结点起，在可利用空间表中进行查找，找到第一个容量不小于请求分配的存储量 (n) 的 空闲块时，即可进行分配。为了使整个系统更有效地运行，在边界标识法中还作了如下两条约定：

- A. 选定一个常量 e，每次分配空间时，判断当前内存块向用户分配空间后，如果剩余部分的容量比 e 小，则将整个内存块全部分配给用户。
- B. 采用头部拟合法进行分配时，如果每次都从 pav 指向的结点开始遍历，在若干次后，会出现存储量小的结点密集地分布在 pav 结点附近的情况，严重影响遍历的时间。解决办法就是：在每次分配空间后，让 pav 指针指向该分配空间结点的后继结点，然后从新的 pav 指向的结点开始下一次的分配。



● 内存回收

一旦用户释放占用块，系统需立即回收以备新的请求产生时进行再分配。为了使物理地址毗邻的空闲块结合成一个尽可能大的结点，则首先需要检查刚释放的占用块的左、右紧邻是否为空闲块。由于本系统在每个内存区(无论是占用块或空闲块)的边界上都设有标志值，则很容易辨明这一点。

假设用户释放的内存区的头部地址为 p ，则与其低地址紧邻的内存区的底部地址为 $p-1$ ，与其高地址紧邻的内存区的头部地址为 $p + p \rightarrow \text{size}$ ，它们中的标志域就表明了这两个邻区的使用状况：

- ✧ 若 $(p - 1) \rightarrow \text{tag} = 0$ ；则表明其左邻为空闲块
- ✧ 若 $(p + p \rightarrow \text{size}) \rightarrow \text{tag} = 0$ ；则表明其右邻为空闲块。

下面分四种情况进行讨论：

- a) 当前空闲块的左右两侧都不是空闲块，而是占用块此种情况下只需要将新的空闲块按照相应的规则（头部拟合法随意插入）插入到可利用空间表中即可。

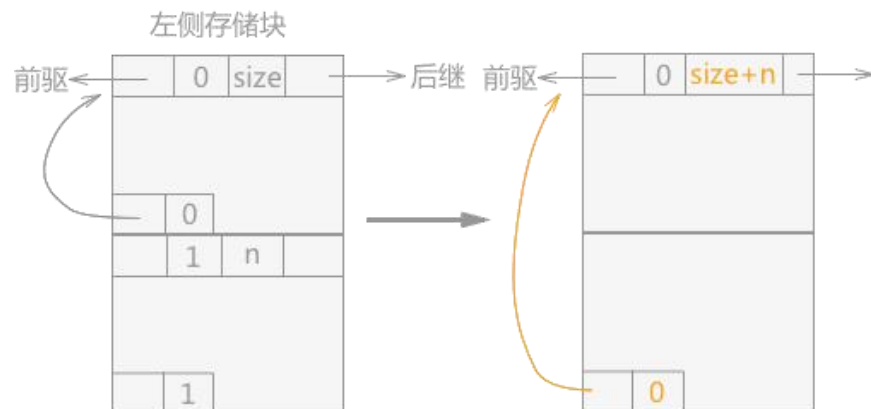
```

// 设定 p 指针指向的为 用户释放的空闲块
p->tag=0;
// f 指针指向 p 空闲块的 foot 域
Space f = FootLoc(p);
f->uplink = p;
f->tag = 0;
// 如果 pav 指针不存在，证明可利用空间表为空，此时设置 p 为头指针，并
// 重新建立双向循环链表
if (!pav) {
    pav = p->llink = p->rlink = p;
}else{
    // 否则，在 p 空闲块插入到 pav 指向的空闲块的左侧
    Space q = pav->llink;
    p->rlink = pav;
    p->llink = q;
    q->rlink = pav->llink = p;
    pav = p;
}

```

- b) 当前空闲块的左侧相邻的块为空闲块，右侧为占用块

只需要更改左侧空闲块中的 size 的大小，并重新设置左侧空闲块的 foot 域即可。



// 常量 n 表示当前空闲块的存储大小

```

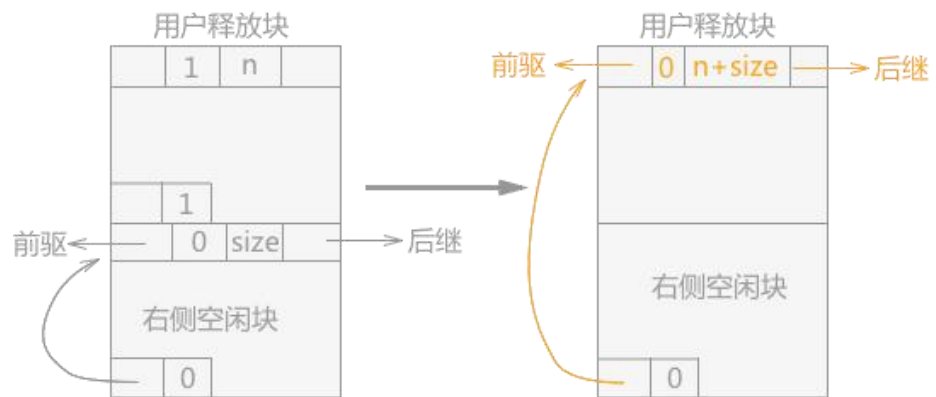
int n = p->size;
Space s = (p-1)->uplink; // p-1 为当前块的左侧块的 foot 域，foot 域中的
// uplink 指向的就是左侧块的首地址，s 指针代表的是当前块的左侧存储块
s->size += n; // 设置左侧存储块的存储容量
Space f = p + n - 1; // f 指针指向的是空闲块 p 的 foot 域
f->uplink = s; // 这是 foot 域的 uplink 指针重新指向合并后的存储
// 空间的首地址
f->tag = 0; // 设置 foot 域的 tag 标记为空闲块

```

- c) 当前空闲块的右侧相邻的块为空闲块，左侧为占用块

将用户释放掉的存储块替换掉右侧的空闲块，同时更改存储块的

size 和右侧空闲块的 uplink 指针的指向。



```
Space t = p + p->size;    // t 指针指向右侧空闲块的首地址
p->tag = 0;                // 初始化 head 域的 tag 值为 0
// 找到 t 右侧空闲块的前驱结点和后继结点，用当前释放的空闲块替换右侧
// 空闲块
Space q = t->llink;
p->llink = q; q->rlink = p;
Space q1 = t->rlink;
p->rlink = q1; q1->llink = p;
// 更新释放块的 size 的值
p->size += t->size;
// 更改合并后的 foot 域的 uplink 指针的指向
Space f = FootLoc(t);
f->uplink = p;
```

d) 当前空闲块，左右两侧的内存块全部为空闲块

需要将 3 个空闲块合并为一个更大的块，操作的过程为：更新左侧空闲块的 size 的值，同时在可利用空间表中摘除右侧空闲块，最后更新合并后的大的空闲块的 foot 域。

```
int n = p->size;
Space s = (p-1)->uplink;    // 找到释放内存块物理位置相邻的低地址的空
// 闲块
Space t = p + p->size;    // 找到物理位置相邻的高地址处的空闲块
s->size += n + t->size;    // 更新左侧空闲块的 size 的值
// 从可利用空间表中摘除右侧空闲块
Space q = t->llink;
Space q1 = t->rlink;
q->rlink = q1;
q1->llink = q;
// 更新合并后的空闲块的 uplink 指针的指向
Space f = FootLoc(t);
f->uplink = s;
```

4.2 段面置换算法

本系统采用 FIFO 策略来实现段面的置换，具体实现：建立一个结构体记录已经分配内存的进程和内存块（即段）的首址。

```
typedef struct
{
    Space s; //占用块指针循环队列(初始化为空)，配合全局变量 number 实现
    int num; //进程编号
} S_Space;
```

然后声明一个 S_Space 数组构成占用块指针循环队列，并且声明 2 个全局变量：front（记录队列首元素位置）、number（第 number 个进入内存的段）

```
S_Space v[MAX / e] = { NULL };
int front = 0;           //记录队列首元素位置
int number = 0;          //全局变量，第 number 个进入内存的段
```

利用队列 v 的队头元素(FIFO)，再与存储所有进程信息中的 A.seg[i].start 进行比较，相同则对其中信息进行修改

4.3 内存地址映射

内存地址映射需要输入进程号、段的编号和段内偏移，首先将输入的段内偏移与段长进行比较，如果输入的段内偏移大于段长，则产生**越界中断**。

根据段是否在内存中，可以分为两种情况：

- 要进行地址映射的段在内存中

此时直接计算物理地址：段的起始地址+偏移

```
q = A[processId].seg[segId].start + offset * sizeof(WORD);
```

- 要进行地址映射的段不在内存中

此时会产生缺段中断，在缺段中断处理子程序中，需要将该段调入内存中，根据“该段所需的内存大小+已分配的内存大小之和是否大于系统最大内存”分为两种情况：

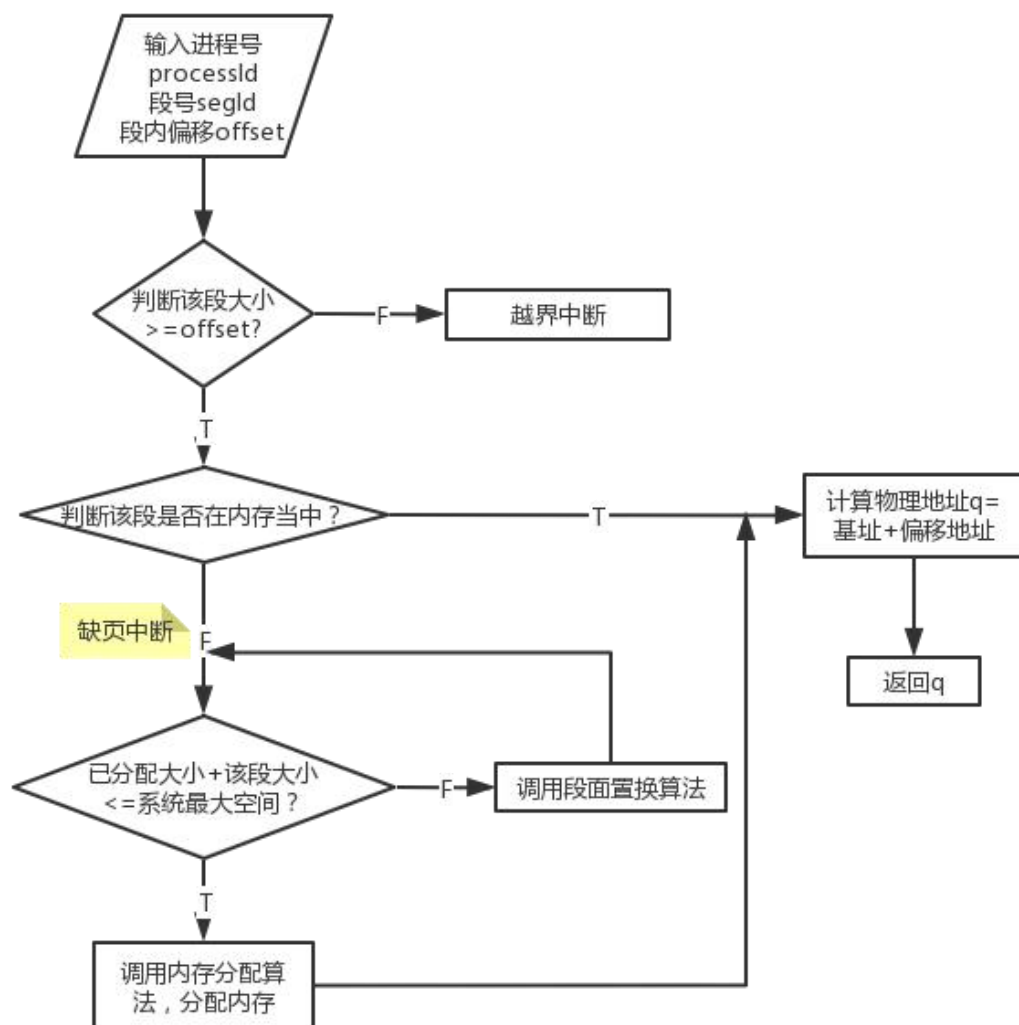
◆ 其和小于等于系统最大内存

直接调用内存分配算法，将该段调入内存中。

◆ 其和大于系统最大内存

首先调用 **FIFO** 子程序，循环替换出队首的段，直到该段所需的内存大小+已分配的内存大小之和小于系统最大内存。将该段调入内存之后，即可以计算物理地址:段的起始地址+偏移

```
q = A[processId].seg[segId].start + offset * sizeof(WORD);  
return q;
```



5. 程序实现---主要数据结构

主要数据结构包括:内存字 WORD、段 Segment、进程 Process、已分配内存队列元素 S_Space (S_Space 数组兼有段表与配合 FIFO 算法双重作用) 本系统规定,内存字 (WORD) 大小为 128 个,最小可分配段大小 e 为 1

✧ 内存字:

```
typedef struct WORD
{
    union {
        WORD *llink;
        WORD *uplink;
    } a;
    int tag;
    int size;
    WORD *rlink;
} WORD, head, foot, *Space;
```

✧ 段:

```
typedef struct Segment
{
    WORD *start; //物理段起始地址
    int length; // 段长
    int status; //是否在内存中, 0 表示不在, 1 表示在
    int time; //存储进程进入时间
} Segment;
```

✧ 进程:

```
typedef struct Process
{
    int ID; //进程号
    Segment seg[Max_Segment];
    //seg 中存储的是每段的信息, 每个进程最多分配 10 个段
    char name[20]; //进程名
    int seg_size; //进程包含的段数
    int has_allocated; //已分配内存的段数
} Process;
```

✧ 已分配内存队列元素 S_Space 及其数组:

```
typedef struct
{
    Space s; //占用块指针循环队列(初始化为空), 配合全局变量 number 实现
    int num; //进程编号
} S_Space;
S_Space v[MAX / e] = { NULL };
```


6. 程序实现---主要程序清单

● 内核函数

✧ 内存字：

```
void initMemory(Space &pav); //初始化内存
Space AllocBoundTag(Space &pav, int n); //分配内存
void Reclaim(Space &pav, Space p); //回收内存
```

✧ 进程：

```
void initProcess(); //初始化进程
int CreateProcess(char name[20], int segnum, int segsize[]); //创建进程
```

✧ 已分配内存队列元素 S_Space 及其数组：

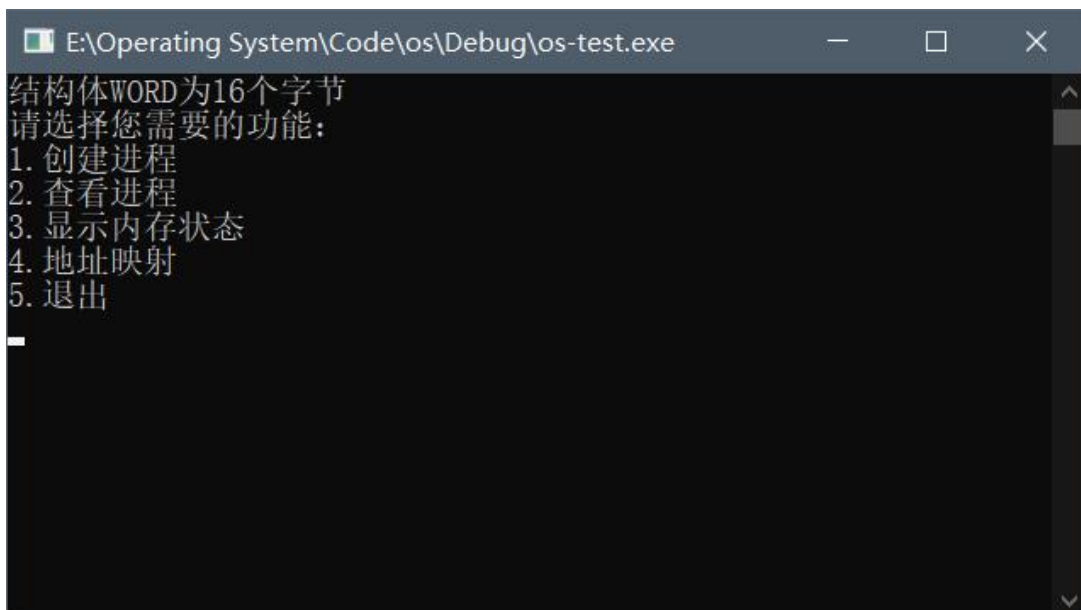
```
bool FIFO(Space &pav, Space &p); //替换算法, FIFO
WORD* PrintAddress(Space &pav, Space &a, int processId, int segId, int offset);
//地址映射
WORD* MissSegInt(Space &pav, Space &a, int processId, int segId, int offset);
//缺段中断处理
```

● 演示函数

```
void Print_free(Space p); //输出内存的空闲块状态
void Print_used(S_Space p[]); //输出 p 数组所指的已分配空间
void Print_Process(); //输出段表信息;
void createP(); //创建进程
void printA(Space &pav, Space &p); //内存地址映射演示
```

7. 程序运行的主要界面和结果截图

● 主界面：



```
E:\Operating System\Code\os\Debug\os-test.exe
结构体WORD为16个字节
请选择您需要的功能：
1. 创建进程
2. 查看进程
3. 显示内存状态
4. 地址映射
5. 退出
```

● 创建进程:

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入要创建的进程的个数:2
请输入要创建的第1个进程的名称:os1
请输入要创建的第1个进程的段数:2
请输入要创建的第1个进程的第1个段的长度:30
请输入要创建的第1个进程的第2个段的长度:57
第1个进程创建成功!
请输入要创建的第2个进程的名称:os2
请输入要创建的第2个进程的段数:4
请输入要创建的第2个进程的第1个段的长度:2
请输入要创建的第2个进程的第2个段的长度:6
请输入要创建的第2个进程的第3个段的长度:18
请输入要创建的第2个进程的第4个段的长度:35
第2个进程创建成功!
请按任意键继续. . .
```

● 查看进程:

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号: 1
进程名为: os1
虚拟段号 状态位 内存起始地址 长度 进程进入时间
0         0         0         30         -1
1         0         0         57         -1
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号: 2
进程名为: os2
虚拟段号 状态位 内存起始地址 长度 进程进入时间
0         0         0          2         -1
1         0         0          6         -1
2         0         0         18         -1
3         0         0         35         -1
请按任意键继续. . .
```

● 中断功能, 段面置换功能, 存储空间的分配与回收功能演示

1) 创建进程

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入要创建的进程的个数:1
请输入要创建的第1个进程的名称:os
请输入要创建的第1个进程的段数:6
请输入要创建的第1个进程的第1个段的长度:60
请输入要创建的第1个进程的第2个段的长度:50
请输入要创建的第1个进程的第3个段的长度:25
请输入要创建的第1个进程的第4个段的长度:20
请输入要创建的第1个进程的第5个段的长度:20
请输入要创建的第1个进程的第6个段的长度:97
第1个进程创建成功!
请按任意键继续. . .
```

2) 地址映射 (中断功能及段面置换算法 FIFO 的实现)

访问 1 号进程的第 0 段, 偏移地址为 0

第一次访问, 该段不在进程中, 会发生缺段中断, 则此时为其分配段长 60, 则空闲链节点为 68

68	60
----	----

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:0
请输入段内偏移:0
缺段中断!!!
物理地址为: 20950736
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20949648, 空闲块大小为68
已分配的空间:
段0的首地址=20950736 段的大小=60
请按任意键继续. . .
```

访问 1 号进程的第 1 段, 段内偏移地址为 0

第一次访问该段, 该段不在内存, 发生缺段中断, 则此时分配段 0 大小为 60, 段 1 大小为 50, 空闲链节点为 18

18	50	60
----	----	----

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:1
请输入段内偏移:0
缺段中断!!!
物理地址为: 20949936
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20949648, 空闲块大小为18
已分配的空间:
段0的首地址=20950736 段的大小=60
段1的首地址=20949936 段的大小=50
请按任意键继续. . .
```

访问 1 号进程的第 2 段，段内偏移地址为 0

第一次访问该段，该段不在内存中，发生中断，但此时内存剩余空间不够，则发生段面置换，因此会由 FIFO 算法置换掉第一个分配的段（60），则此时分配段 1 大小为 50，段 2 大小为 25，空闲块大小分别为 18,35。



```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:2
请输入段内偏移:0
内存空间不足!!!
完成FIFO置换!!
物理地址为: 20951296
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20950736, 空闲块大小为18
空闲块首址为20949648, 空闲块大小为35
已分配的空间:
段1的首地址=20949936 段的大小=50
段2的首地址=20951296 段的大小=25
请按任意键继续. . .
```

访问第 1 号进程的第 3 段，段内偏移地址为 0

第一次访问该段，该段不在内存中，发生缺页中断，但是空闲块大小大于段大小，则分配段 1 大小为 50，段 2 大小为 25，段 3 大小为 20，从而形成两个空闲块，大小分别为 18,15。



```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:3
请输入段内偏移:0
缺段中断!!!
物理地址为: 20950976
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20950736, 空闲块大小为18
空闲块首址为20949648, 空闲块大小为15
已分配的空间:
段1的首地址=20949936 段的大小=50
段2的首地址=20951296 段的大小=25
段3的首地址=20950976 段的大小=20
请按任意键继续. . .
```

访问第 1 号进程的第 4 段，段内偏移地址为 0

第一次访问，该段不在内存中，会发生缺段中断，但由于此时内存空间剩余空闲块大小无法满足分配要求则会发生段面置换，从而置换掉段 1（50），而段 1 两边均为空闲块，所以进行合并产生一个大的空闲分区大小为 63，此时分配段 2 大小为 25，段 3 大小为 20，段 4 大小为 20。



```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:4
请输入段内偏移:0
内存空间不足!!!
完成FIFO置换!!
物理地址为: 20950656
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20949648, 空闲块大小为63
已分配的空间:
段2的首地址=20951296 段的大小=25
段3的首地址=20950976 段的大小=20
段4的首地址=20950656 段的大小=20
请按任意键继续. . .
```

访问第 1 号进程的第 5 段，段内偏移为 0

第一次访问，该段不在内存中，发生缺段中断，但由于当前内存空间剩余大小无法满足要求，则根据 FIFO 算法发生段面置换，从而置换掉段 2（25），段 3（20），段 4（20），分配段 5（97），并合并空闲块（31）



```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:5
请输入段内偏移:0
内存空间不足!!!
完成FIFO置换!!
物理地址为: 20950144
请按任意键继续. . .
```



```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20949648, 空闲块大小为31
已分配的空间:
段5的首地址=20950144 段的大小=97
请按任意键继续. . .
```

访问第 1 号程序的第 6 段, 段内偏移为 0

因为 1 号程序没有第 6 段, 则会发生越界中断, 对内存块的分配不会产生影响。

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号:1
请输入段号:6
请输入段内偏移:0
越界中断!!!
请按任意键继续. . .
```

```
E:\Operating System\Code\os\Debug\os-test.exe
内存的空闲块状态:
空闲块首址为20949648, 空闲块大小为31
已分配的空间:
段5的首地址=20950144 段的大小=97
请按任意键继续. . .
```

最后查看该程序, 则可知访问各段的顺序以及在内存中的位置。

```
E:\Operating System\Code\os\Debug\os-test.exe
请输入进程号: 1
进程名为: os
虚拟段号 状态位 内存起始地址 长度 进程进入时间
0         0         0         60         0
1         0         0         50         1
2         0         0         25         2
3         0         0         20         3
4         0         0         20         4
5         1       20950144       97         5
请按任意键继续. . .
```

8. 总结和感想体会

本次的操作系统课程设计对我而言是一个极为特殊的学习体验过程，本次课设我做的题目是关于段式的虚拟存储管理系统，它是基本分段存储方式和虚拟存储器的结合，用页面置换算法和边界标识法分配回收内存，从而对内存进行管理，区别于以往的课程设计，操作系统本次的课程设计不仅要教材上以及课堂中所学原理理解透彻，还需要查阅许多资料，以及一些算法的源码，并对其进行全面的理解才可以灵活的运用到自己的系统中来，高效地完成设计要求，提高系统的性能。

同时也要感谢田老师的细心讲解和课堂上举的实例，让我们可以对操作系统不仅仅是会做题，会编写内核代码，更体会到了它独特的魅力，另外还要感谢助教学长的帮助，解决了一些代码编写上的问题。

通过这门课我也了解到了国产的鸿蒙系统，它是一款全新的面向全场景的分布式操作系统，创造了一个超级虚拟终端互联的世界，将人、设备、场景有机地联系在一起，它的诞生拉开永久性改变操作系统全球格局的序幕，它的发展正是需要我们年轻一代的努力。所以，本次课程设计虽然已经结束了，但这仅仅是我对于操作系统学习的开始，还有更长更远的路等着我们去探索。

参考文献

- [1]. 张丽芬. 操作系统实验教程[M]. 北京: 清华大学出版社
- [2]. 汤小丹. 计算机操作系统(第四版)[M]. 西安: 电子科技大学出版社