

西北民族大学

计算机组成原理与系统结构 课程设计报告



2022 年 6 月

题 目:	段页式虚拟存储管理系统
设计者:	
学 院:	数学与计算机科学学院
专 业:	计算机科学与技术
班 级:	计算机科学与技术班

操作系统原理课程设计任务书

学 院：	数学与计算机 科学学院	专业班 级：	
设计人：			
题目：段页式虚拟存储管理系统			
<p>课程设计内容：段页式虚拟存储管理系统：建立一个段页虚拟存储管理系统的模型^[1]</p> <p>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；</p> <ul style="list-style-type: none"> ● 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； ● 建立管理段页的基本数据结构与算法； ● 设计存储空间的分配与回收算法； ● 实现缺页中段支持的逻辑地址到物理地址转换，实现虚拟存储器； <p>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。</p>			
要求：按照的有关要求完成算法设计、代码编写与调试以及课设报告的撰写 ^[1] 。			
<p>环境：硬件：Dell G3579 笔记本电脑；</p> <p>软件：Visual Studio 2019 Enterprise、gcc、Notepad++、Qt Creator 4.11.1 (Community)</p>			
<p>目的：在掌握程序的设计技能、专业基础课程和《操作系统》课程的理论知识的基础上，设计和实现操作系统的基本算法、模块与相关的资源管理功能，旨在加深对计算机硬件结构和系统软件的认识，初步掌握操作系统组成模块和应用接口的使用方法，提高进行工程设计和系统分析的能力，为毕业设计和以后的工程实践打下良好的基础^[1]。</p>			
任务下达日期 2012 年 6 月 18 日		完成日期 2012 年 7 月 7 日	

1 相关原理及算法

段页式系统既具有分段系统的便于实现、分段可共享、易于保护、可动态链接等一系列优点，又能像分页系统那样，很好地解决内存的外部碎片问题。

1.1 基本原理

段页式系统的基本原理是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。在段页式系统中，其结构由段号、段内页号和段内地址三部分组成，如图 3.1.1 所示^[2]。

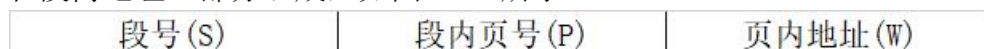


图 3.1.1 段页式地址结构

在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置段表和页表。段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度。图 3.1.2 示出了利用段表和页表进行从用户地址空间到物理（内存）空间的映射。

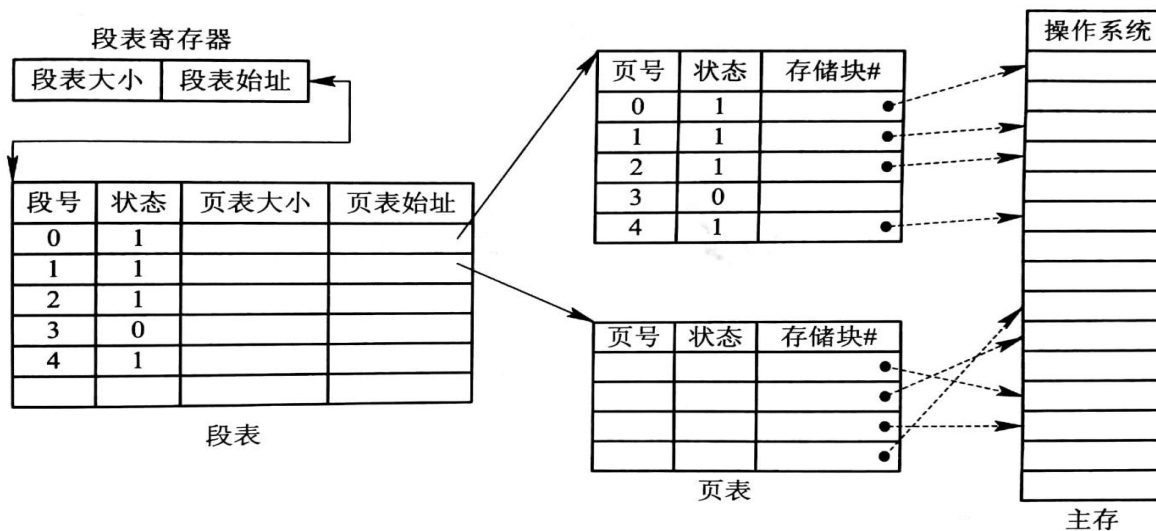


图 3.1.2 利用段表和页表实现地址映射

1.2 地址变换过程

在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段长 TL^[2]。进行地址变换时，首先利用段号 S，将它与段长 TL 进行比较。若 $S < TL$ ，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号 P 来获得对应页的页表项位置，从中读出该页所在的物理块号 b，再利用块号 b 和页内地址来构成物理地址。图 3.2.1 示出了段页式系统中的地址变换机构。

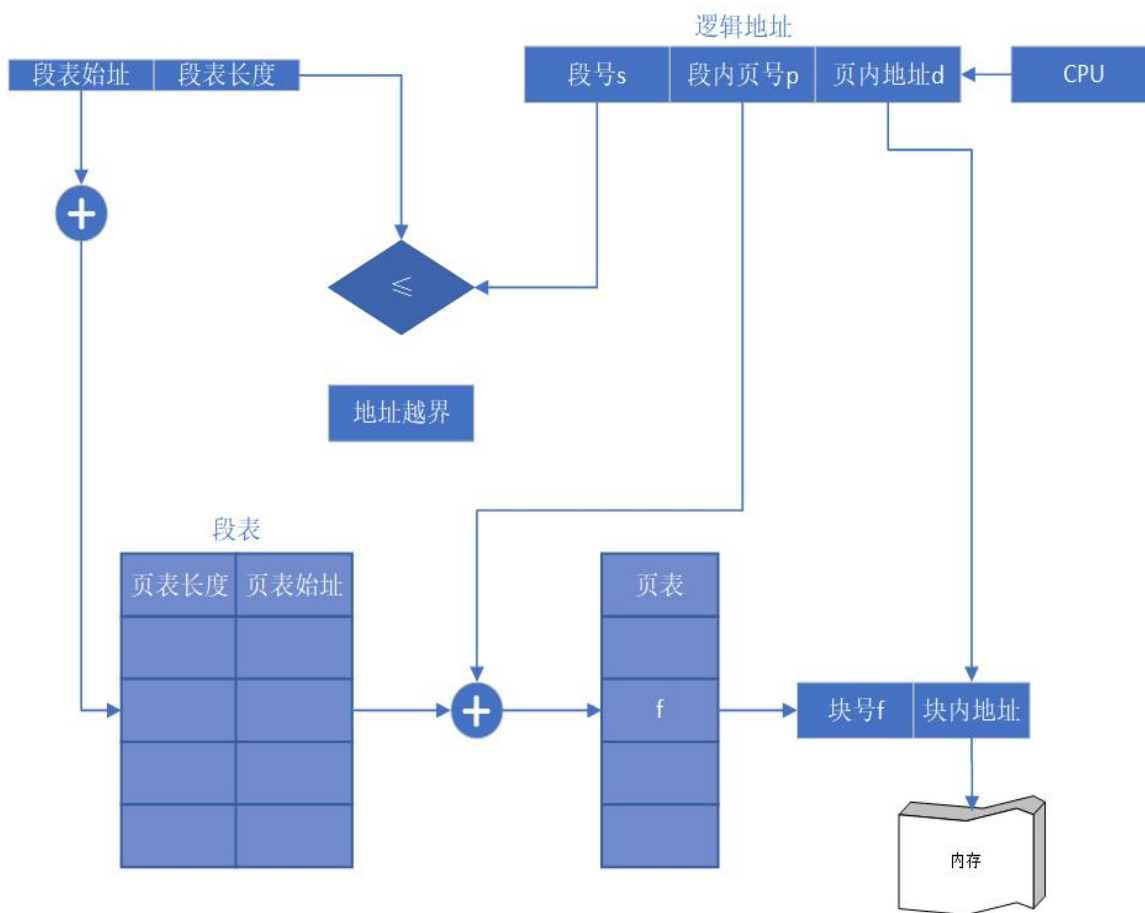


图 3.2.1 段页式系统中的地址变换机构

在段页式系统中，为了获得一条指令或数据，须三次访问内存。第一次访问是访问内存中的段表，从中取得页表始址；第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与页内地址一起形成指令或数据的物理地址；第三次访问才是真正从第二次访问所得的地址中取出指令或数据。

显然，这使访问内存的次数增加了近两倍。为了提高执行速度，在地址变换机构中增设一个高速缓冲寄存器。每次访问它时，都须同时利用段号和页号去检索高速缓存，若

找到匹配的表项，便可从中得到相应页的物理块号，用来与页内地址一起形成物理地址；若未找到匹配表项，则仍需第三次访问内存。

1.3 有关算法

①分页系统中逻辑地址到物理地址的转换

本设计需要根据内存中的逻辑地址计算对应的物理地址，段页式系统的逻辑地址由页地址与段地址共同表示^[2]。而在分页管理中，对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间的地址为 A ，页面的大小为 L ，则页号 P 和页内地址 d 可按下式求得：

$$P = \text{INT} \left[\frac{A}{L} \right], d = [A] \text{MOD } L$$

其中，INT 是整除函数，MOD 是取余函数。

②最近最久未使用（Least Recently Used, LRU）置换算法

本设计的目标是构建一个允许内存与外存进行数据置换的调度系统，因此合适的虚拟地址到物理地址的算法，即虚拟存储地址变换，是应当被考虑的。值得一提的是，虚拟存储器地址变换基本上有 3 种形式：虚拟存储器工作过程式：全联想变换、直接变换、组联想变换^[3]。其中，任何逻辑空间页面能够变换到物理空间任何页面位置的方式称为全联想变换；每个逻辑空间页面只能变换到物理空间一个特定页面的方式称为直接变换；组联想变换是指各组之间是直接变换，而组内各页间则是全联想变换。替换规则用来确定替换主存中哪一部分，以便腾空部分主存，存放来自辅存要调入的那部分内容。当前流行的替换算法有 4 种：

①随机算法：用软件或硬件随机数产生器确定替换的页面；

②先进先出：先调入主存的页面先替换；

③近期最少使用算法（LRU, Least Recently Used）：替换最长时间不用的页面；

④最优算法：替换最长时间以后才使用的页面。这是理想化的算法，只能作为衡量其他各种算法优劣的标准。

综合算法性能、鲁棒性与编程复杂度等因素考量，本设计采用 LRU 算法作为内存与外存的内容置换算法。

2 系统结构和主要的算法设计思路

本设计所构建的虚拟段页式系统主要分为 3 个模块：内存管理模块、请求调度模块与异常处理模块，其中内存管理模块又细分为：请求分页的页表机制、请求分段的段表机制以及内存分配/回收模块，异常处理模块分为：缺页中断机构、缺段中断机构。系统的 UML 类图如图 4.1 所示。本设计在完成虚拟段页式系统的构建的基础上，使用 Qt 创建了用户良好、操作简洁的可视化操作界面。

要构建虚拟段页式系统，本设计必须提供内存置换的数据结构与算法支持，为了保证内存的有效分配及高效管理，应当考虑采用链式结构进行存储。内存置换，会带来缺页中断与缺段中断的问题，这是本设计重点探究的症结。

接下来详细介绍本设计所构建的系统以及使用的有关算法。

2.1 系统结构

2.1.1 内存管理模块

本模块主要负责系统的内存申请、初始化、分配与回收，根据其操作对象的不同有：请求分页的页表机制、请求分段的段表机制以及内存分配/回收模块。下面逐一介绍。

①请求页表机制

为了实现请求分页，系统必须提供一定的硬件支持。本设计构建的虚拟段页式系统可以完全模拟标准 OS 中相关模块的工作机制，因此，本系统不仅具备一定容量的内存和外存，还配置了请求页表机制^[2]。

在请求分页系统中需要的主要数据结构是请求页表，其基本作用仍然是将用户地址空间中的逻辑地址映射为内存空间中的物理地址。为了满足页面换进换出的需要，在请求页表中又增加了四个字段。这样，在请求分页系统中的每个页表应含以下诸项：

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址
----	------	-------	--------	-------	------

现对其中各字段说明如下：

(1) 状态位(存在位)P：由于在请求分页系统中，只将应用程序的一部分调入内存，还有一部分仍在外存磁盘上，故须在页表中增加一个存在位字段。由于该字段仅有一位，

故又称位字。它用于指示该页是否已调入内存，供程序访问时参考。

(2) 访问字段 A：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，提供给置换算法(程序)在选择换出页面时参考。

(3) 修改位 M：标识该页在调入内存后是否被修改过。由于内存中的每一页都在外存上保留一份副本，因此，在置换该页时，若未被修改，就不需再将该页写回到外存上，以减少系统的开销和启动磁盘的次数；若已被修改，则必须将该页重写到外存上，以保证外存中所保留的副本始终是最新的。简而言之，M 位供置换页面时参考。

(4) 外存地址：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

②请求段表机制

为了实现请求分段式存储管理，应在系统中配置多种硬件机构，以支持快速地完成请求分段功能。与请求分页系统相似，本设计所构建的虚拟段页式系统也配备了请求段表机制^[2]。

在请求分段式管理所需的主要数据结构是请求段表。在该表中除了具有请求分页机制中有的访问字段 A、修改位 M、存在位 P 和外存始址四个字段外，还增加了存取方式字段和增补位。这些字段供程序在调进、调出时参考。下面给出请求分段的段表项。

段名	段长	段基址	存取方式	访问字段 A	修改位 A	存在位 P	增补位	外存地址
----	----	-----	------	--------	-------	-------	-----	------

在段表项中，除了段名(号)、段长、段在内存中的起始地址(段基址)外，还增加了以下字段：

(1) 存取方式。由于应用程序中的段是信息的逻辑单位，可根据该信息的属性对它实施保护，故在段表中增加存取方式字段，如果该字段为两位，则存取属性是只执行、只读和允许读/写。

(2) 访问字段 A。其含义与请求分页的相应字段相同，用于记录该段被访问的频繁程度。提供给置换算法选择换出页面时参考。

(3) 修改位 M。该字段用于表示该页在进入内存后是否已被修改过，供置换页面时参考。

(4) 存在位 P。该字段用于指示本段是否已调入内存，供程序访问时参考。

(5) 增补位。这是请求分段式管理中所特有的字段，用于表示本段在运行过程中是

否做过动态增长。

(6) 外存始址。指示本段在外存中的起始地址，即起始盘块号。

③内存分配/回收模块

存储管理是 OS 管理主存储器的软件部分，而在段页式存储管理中，段式二维逻辑地址空间的程序占用多个主存页架区^[4]。本设计所构建虚拟段页式系统的内存是以内存块留驻页面为主要存储方式，页表嵌套在段表中，而若干个段表分属不同的进程。本设计针对页表、段表、进程、内存块分别使用了合适的数据结构进行内存分配，为了提高存储效率，重点关注了内存的管理与回收。譬如：当内存申请不合理时，向用户发出警告信息；当在内存一定的情况下申请过大的段表空间时，允许用户扩充内存；每个内存空间在被使用后，均会被重新回收至主存中。

下面介绍本设计所使用的有关算法。

2.2 算法设计思路

本设计所构建的虚拟段页式系统主要需以下 4 个算法的支持：页表虚拟地址变换机构、段表虚拟地址变换机构、LRU 页面置换算法和进程随机初始化分配内存算法。

2.2.1 页表虚拟地址变换机构

请求分页系统中的地址变换机构是在分页系统地址变换机构的基础上，为实现虚拟存储器，再增加了某些功能所形成的，如产生和处理缺页中断，以及从内存中换出一页的功能等等^[2]。图 4.2.1.1 示出了请求分页系统中的地址变换过程。

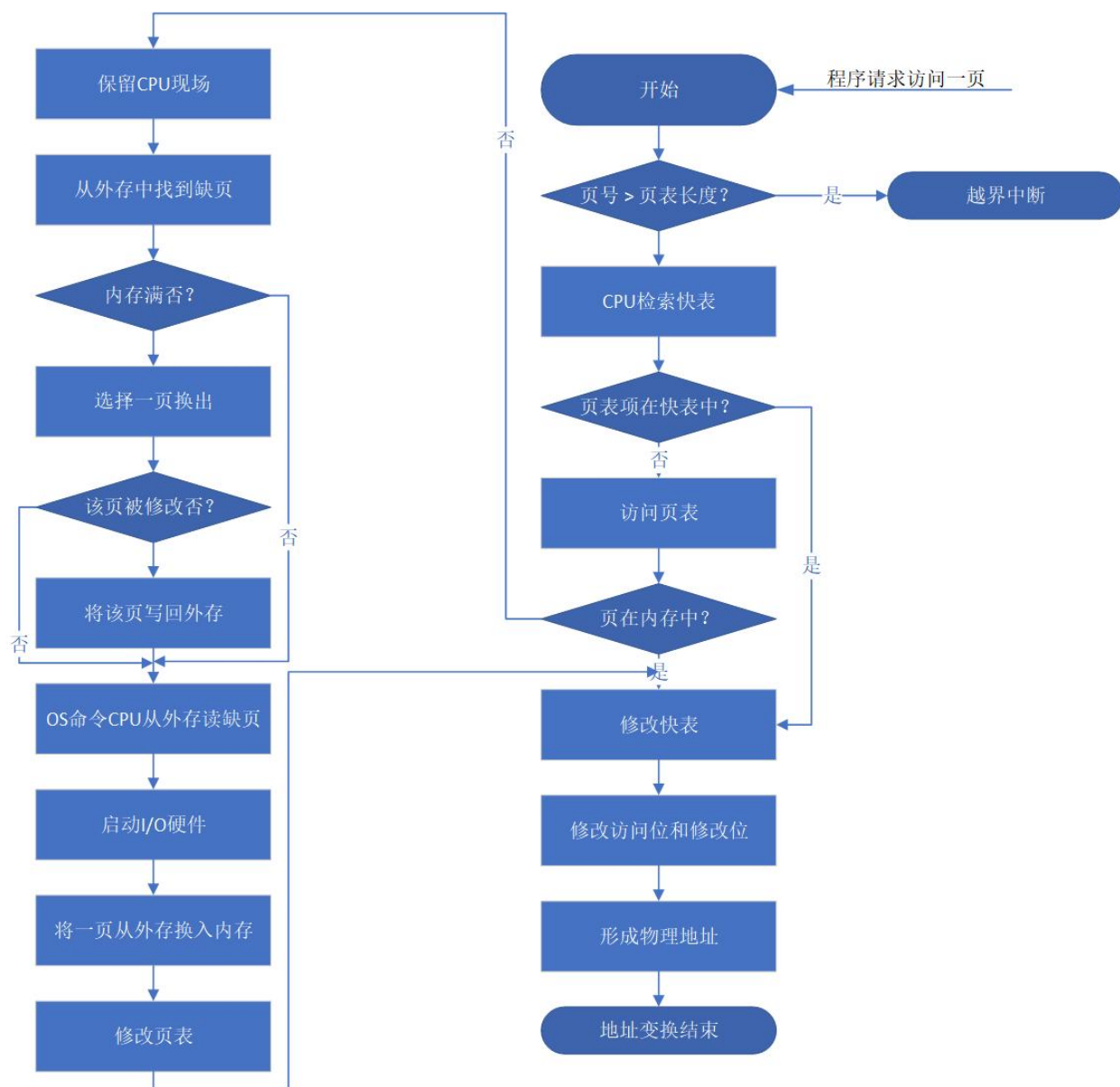


图 4.2.1 请求分页中的地址变换过程

在进行地址变换时，首先检索快表，试图从中找出所要访问的页。若找到，便修改页表项中的访问位，供置换算法选换出页面时参考。对于写指令，还须将修改位置成“1”，表示该页在调入内存后已被修改。然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。

如果在快表中未找到该页的页表项，则应到内存中去查找页表，再从找到的页表项中的状态位 P 来了解该页是否已调入内存。若该页已调入内存，这时应将该页的页表项写入快表。当快表已满时，则应先调出按某种算法所确定的页的页表项，然后再写入该页的

页表项;若该页尚未调入内存,这时应产生缺页中断,请求 OS 从外存把该页调入内存。

2.2.2 段表虚拟地址变换机构

在请求分段系统中采用的是请求调段策略^[2]。每当发现运行进程所要访问的段尚未调入内存时,便由缺段中断机构产生一缺段中断信号,进入 OS 后,由缺段中断处理程序将所需的段调入内存。与缺页中断机构类似,缺段中断机构同样需要在一条指令的执行期间产生和处理中断,以及在一条指令执行期间,可能产生多次缺段中断。但由于分段是信息的逻辑单位,因而不可能出现一条指令被分割在两个分段中,和一组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 4.2.2.1 所示。由于段不是定长的,这使对缺段中断的处理要比对缺页中断的处理复杂。

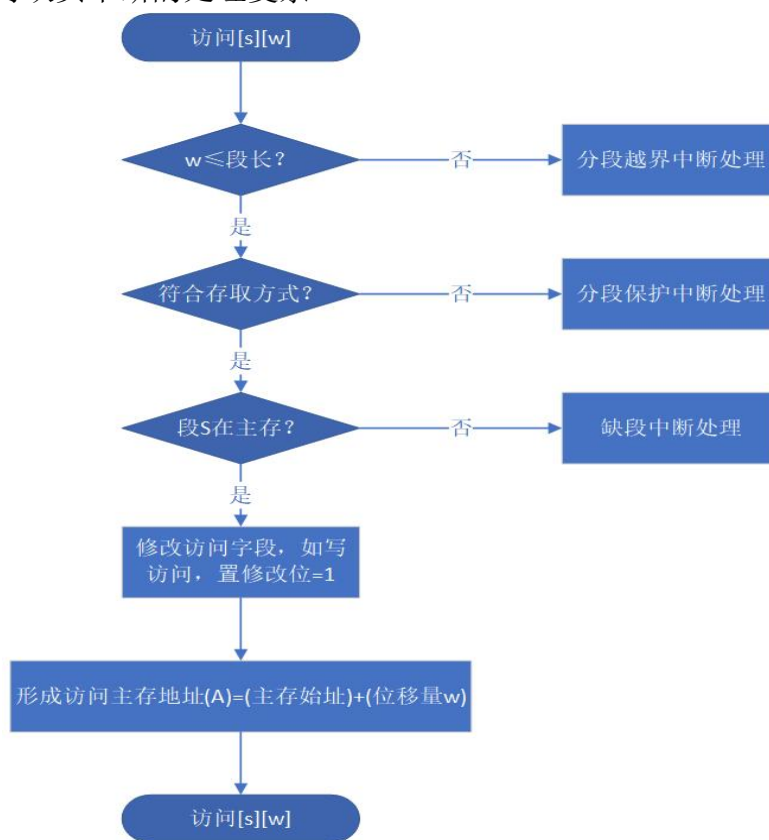


图 4.2.2 请求分段系统的地址变换过程

2.2.3 LRU 页面置换算法

本设计采用 LRU 作为页面置换算法。最近最久未使用(LRU)的页面置换算法是根据

页面调入内存后的使用情况做出决策的^[2]。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，故 LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t 。当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。LRU 算法的流程图如图 4.2.3.1 所示。

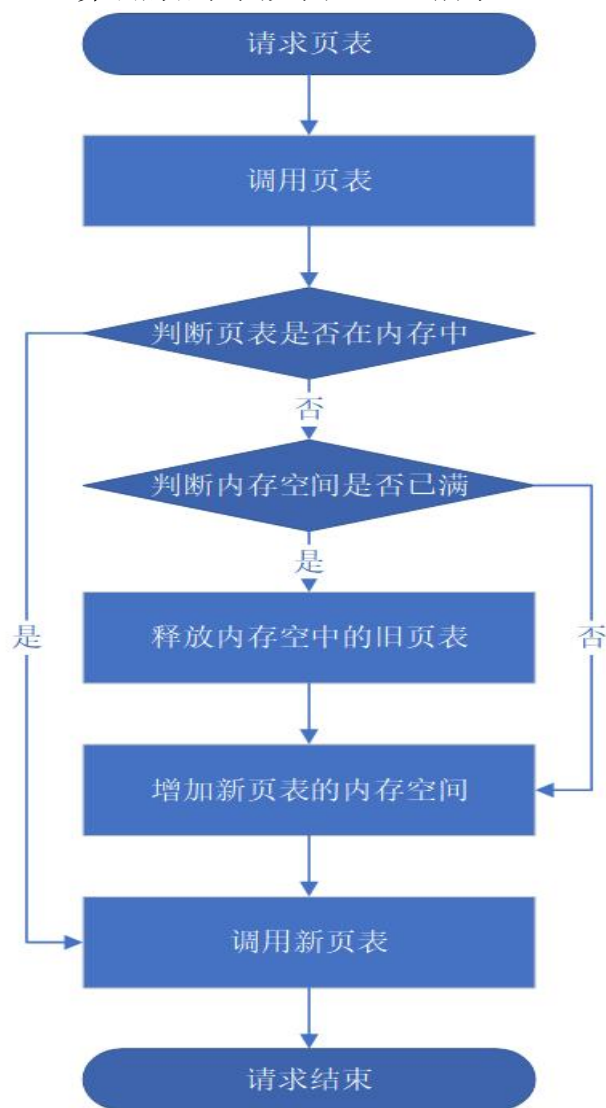


图 4.2.3 LRU 算法流程

本设计的程序实现是接下来的介绍重点。

3 程序实现

C 是一种通用的程式语言，广泛用于系统软件与应用软件的开发。于 1969 年至 1973 年间，为了移植与开发 UNIX 作业系统，由丹尼斯·里奇与肯·汤普逊，以 B 语言为基础，在贝尔实验室设计、开发出来^[5]。

C 语言具有高效、灵活、功能丰富、表达力强和较高的可移植性等特点，在程式设计中备受青睐，成为最近 25 年使用最为广泛的编程语言^[6]。目前，C 语言编译器普遍存在于各种不同的操作系统中，例如 Microsoft Windows、macOS、Linux、Unix 等。C 语言的设计影响了众多后来的程式语言，例如 C++、Objective-C、Java、C# 等。

为了使程序更符合现代的 OS 代码标准，本设计选择 C 语言作为虚拟段页式系统的编写语言。

3.1 主要数据结构

如 4.1.1 节所述，本设计根据不同内存留驻模型与状态信息创建相应的数据结构进行存储管理，涉及的主要模型有：页表结构、段表结构、进程信息与内存块信息；依据 3.1、4.1、4.2 节所述的基本原理，使用 C 语言设计实现的数据结构如下：

```
typedef struct Page //页结构
{
    int ID;          //页号
    char in_out;     //是否在内存
    int pageOffset;  //页内偏移
    int block_num;   //对应块号
    int time;        //在内存的时间
    int Process_ID;  //所属进程ID
    char name[10];   //所属进程名
    int Segment_ID;  //所属进程段号
    struct Page* next;
} Pa, * PaNode;

typedef struct Segment //段结构
{
    int ID;          //段号
    Page* page;      //段中的页结构
    int size;        //段大小
    char in_out;     //是否在内存中
    struct Segment* next;
```

```

}Seg, * SegNode;

typedef struct PCB //进程信息
{
    Segment* segm; //进程段信息
    int total_segmet; //段总数
    int ID; //进程ID
    char name[10]; //进程名
    struct PCB* next;
}PCB, * pcb;

typedef struct Memory //内存块信息
{
    int ID; //页框号
    char allocated; //标记是否已分配
    int block_size; //页框大小
    int Process_ID; //占用进程ID
    char name[10]; //占用进程名
    int Segment_ID; //占用进程段号
    int Page_ID; //占用进程的页号
    int time_in_memory; //页在内存中的时间
    int recently; //最近访问的时间
    struct Memory* next;
}*memory;

```

可以看出，页表、段表、进程信息与内存块信息数据结构的设置在符合上文所述基本原理的同时，兼顾了编程实现的便捷与效率。本设计使用单链表作为信息存储的基本单元，极大地方便了对内存空间的频繁开辟与回收。

值得一提的是，本设计借助 Qt 开发了虚拟式段页系统的可视化 UI 界面。Qt 是一个跨平台个 C++ 应用程序开发框架^[7]。广泛用于开发 GUI 程序，某种情况下又被称为部件工具箱。也可用于开发非 GUI 程序，比如控制台工具搭服务器^[8]。

信号与槽（Signal & Slot）是 Qt 编程的基础，也是 Qt 的一大创新。因为有了信号与槽的编程机制，在 Qt 中处理界面各个组件的交互操作时变得更加直观和简单。

信号（Signal）就是在特定情况下被发射的事件，例如 PushButton 最常见的信号就是鼠标单击时发射的 clicked() 信号，一个 ComboBox 最常见的信号是选择的列表项变化时发射的 CurrentIndexChanged() 信号。

GUI 程序设计的主要内容就是对界面上各组件的信号的响应，只需要知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。

槽（Slot）就是对信号响应的函数。槽就是一个函数，与一般的 C++ 函数是一样的，

可以定义在类的任何部分（public、private 或 protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函数不同的是：槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

本设计构建 GUI 程序所使用的类如下所示（内存管理的各个模块不再罗列）：

```
class ProcessUnit
{
public:
    QString name;
    int totalSeg;
    QStringList segSizeList;
};

class MemUnit
{
public:
    QString name;
    int pid;
    int seg_id;
    int page_id;
    int block_id;
};

class SegUnit
{
public:
    int ID;
    int size;
    QString in_out;
};

class PageUnit
{
public:
    int ID;
    int pageOffset;
    int block_num;
    QString in_out;
};

class QShowEvent;
class MemInitWindow : public QDialog
{
    Q_OBJECT
public:
    MemInitWindow(QWidget *pParent = NULL);
    virtual ~MemInitWindow();
public:
    void Enter();
};
```

```

        void Exit();

private:
    void Init();
    void InitUI();
    void InitConnections();
    void LoadData();
    void CreateProcessList();
    void UpdateTable();

public slots:
    void OnCreatePS();
    void OnMemMng();
    void OnSegNum();

private:
    Ui::MemInitWindow m_ui;
    vector<ProcessUnit> m_processList;
};

```

本设计构建 GUI 程序所使用的名称空间如下所示：

```

class QComboBox;
class QLineEdit;
class QPushButton;
class QLabel;
class QTextEdit;
class QCheckBox;
class QRadioButton;
class QSpinBox;
class QTableWidget;

namespace UIQuery
{
    void SetComboboxValue(QComboBox * pWidget,QString value,QStringList & range,int
defaultValue);
    QString GetValue(QComboBox * pWidget);
    QString GetValue(QLineEdit * pWidget);
    QString GetValue(QPushButton * pWidget);
    QString GetValue(QLabel * pWidget);
    QString GetValue(QTextEdit * pWidget);
    int GetValue(QSpinBox * pWidget);
    bool GetValue(QCheckBox * pWidget);
    bool GetValue(QRadioButton * pWidget);

    void SetValue(QCheckBox * pWidget,bool state);
    void SetValue(QLineEdit * pWidget,QString value);
    void SetValue(QRadioButton * pWidget,bool state);
    void SetValue(QComboBox * pWidget,std::vector<std::string> & valList);
    void SetValue(QComboBox * pWidget,QStringList & valList);
    void SetValue(QTextEdit * pWidget,QString value);
    void SetValue(QLabel * pWidget,QString value);
}

```

```

void SetValue(QSpinBox * pWidget,int value);

void InitTableOption(QTableWidget* pTable);

void Alert(QString msg);
bool VerifyEmpty(QString input,QString tip);

void AppendText(QTextEdit* pTextEdit,const QString& qsrText, QColor c);
}

```

本设计构建 GUI 程序所使用的信号&槽如下所示：

```

class QShowEvent;
class MemoryWindow : public QDialog
{
    Q_OBJECT
public:
    MemoryWindow(QWidget *pParent = NULL);
    virtual ~MemoryWindow();
public:
    void Enter();
    void Exit();

private:
    void Init();
    void InitUI();
    void InitConnections();
    void LoadData();
    void Apply(ProcessUnit ps);
    void Recycle(int ID);
    void PrintProcessSegment(int j);
    QString InterruptHandling(int id, int i, int j);

public slots:
    void OnCreatePS();
    void OnRecyle();
    void OnQueryPSInfo();
    void OnPrintMemory();
    void AddressExchange();
    void OnViewPageInfo(QTableWidgetItem *);
    void OnSegNum();

protected:
    void showEvent(QShowEvent *event);

private:
    Ui::MemoryWindow m_ui;
    vector<SegUnit> m_segList;
    map<int,vector<PageUnit>> m_blockMap;
};

```

上述 GUI 程序中的方法与变量名大都见名知意，再次便不作过多赘述。

3.2 主要程序清单

3.2.1 核心程序

下面描述本设计所构建的虚拟段页式系统的核心程序。

本设计的编码工作并不复杂，代码量较小，故为了方便各个函数间的参数传递，本设计在代码中使用了 8 个全局变量，以此作为系统状态的监控标志与信息共享接口。这些全局变量如下所示：

```
int Register; //段表的起始地址
int Flash; //内存大小
int BLOCK; //页框大小
int remained; //剩余的内存
pcb process; //申明一个进程链表
int total_process = 0; //进程总数
memory memory_info; //申明一个记录内存信息的链表
int InterruptType = -1; //中断类型, 1 表示缺段中断, 0 表示缺页中断
```

这 8 个全局变量在内存替换中起到了重要作用，接下来做概要介绍。

虚拟页表与虚拟段表，它们的核心都在于内存申请与置换，当用户所需要的内存不在内存中时，程序应当根据预置的算法自动进行页/段表置换。本设计用于实现内存申请和置换的模块是异常处理模块，该模块涉及的主要局部变量如下所示：

```
char name[10]; //置换出来的进程名
int ID; //置换出来的进程ID
int segment_ID; //置换出来的段号ID
int page_ID; //置换出来的页号
int enough = 0; //标记内存空间是否足够
int min;
int address;
PCB* p;
Seg* s;
Pa* q;
Memory* mer, * temp;
```

程序首先根据全局变量 InterruptType 判断是否发生中断以及中断类型，默认值为-1，1 表示缺段中断，0 表示缺页中断。而后将下一个进程的地址赋值给局部变量 p，通过 while 循环找到发生中断的进程；将下一个段表的地址赋值给局部变量 s，通过 while 循环找到发生中断的段表，将该段置于内存中的标志设置为真；将下一个页表的地址赋值给局部变量 q，通过 while 循环找到发生中断的页表。若发生了缺段中断，在将发生中断的页

表调入内存前，预先判断剩余的内存是否足够，若没有足够的内存空间，用 LRU 法进行段表置换。在实现 LRU 算法时，必须修改当前进程、段表和页表的相关属性，以及修改最近最久未使用页面，保证时空一致性：内存中所有页的时间均加 1，修改进程链表。处理缺页中断的方法类似。

介绍核心函数后，接下来给出本设计所构建虚拟段页式系统的控制台程序与 GUI 程序的界面转换图。

3.2.2 函数清单与类图

如表 5.2.2.1 所示为控制台程序的函数清单。

表 5.2.2.1 控制台程序函数清单

函数名	参数	返回值	作用
CreateSegment	无	void	创建段表空间
CreateMemory	无	void	创建页表空间
CreateProcess	无	void	建立进程信息
InitPage	页表结构地址 PaNode* p	void	初始化页表空间
InitSegment	段表结构地址 SegNode* s	void	初始化段表空间
InitProcess	进程地址 pcb* pro	void	初始化进程
ApplyMemory	无	void	随机初始化分配内存
AddressExchange	无	void	虚拟地址向物理地址转换
InterruptHandling	进程号 id，页表号 i，段表号 j	void	缺段/页中断异常处理
Apply	无	void	手动申请内存
Recycle	无	void	回收内存

如图 5.2.2.1 所示为 GUI 程序的界面转换图。

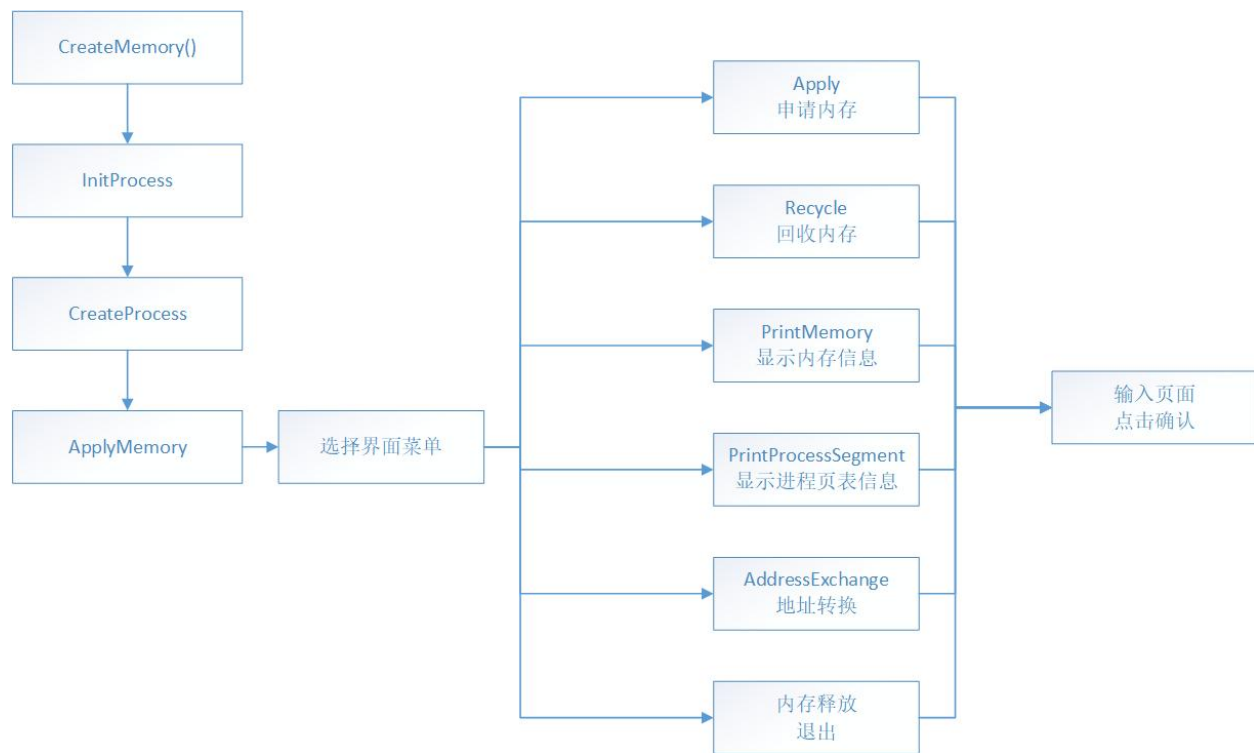


图 5.2.1 GUI 程序界面转换

4 程序运行的主要界面和结果截图

本节主要展示本设计所构建虚拟段页式系统的运行情况。

4.1 控制台程序的运行情况

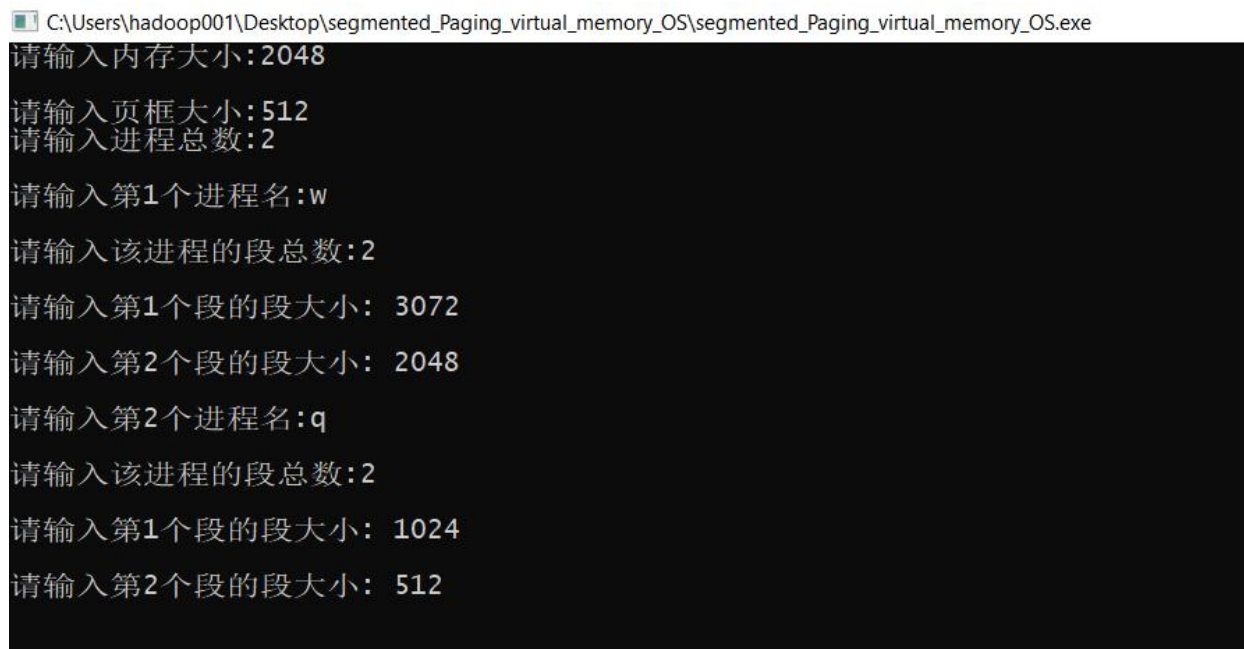


图 6.1.1 控制台程序输入初始化参数

如图 5.2.2.1 所示为使用控制台程序输入初始化参数。

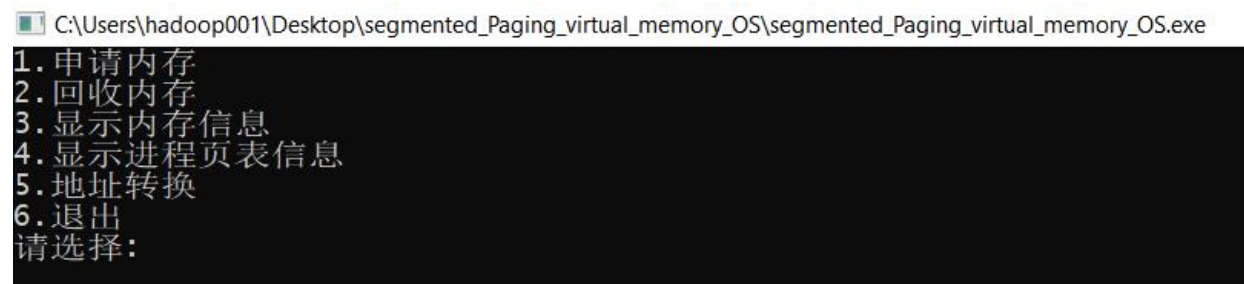


图 6.1.2 控制台程序菜单

如图 5.2.2.2 所示为控制台程序菜单。

```

C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS\segmented_Paging_virtual_memory_OS.exe

内存大小为:2048 总共分成了4块,每块为:512
已分配的内存块情况为:
进程名  进程号  进程段号      页号    块号
q        1      0            0       0

进程名  进程号  进程段号      页号    块号
q        1      0            1       1

进程名  进程号  进程段号      页号    块号
w        0      1            1       3

共占用3块内存块,剩余1块.
1.申请内存
2.回收内存
3.显示内存信息
4.显示进程页表信息
5.地址转换
6.退出
请选择:

```

图 6.1.3 控制台程序显示内存信息

如图 5.2.2.3 所示为控制台程序显示内存信息。

```

C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS\segmented_Paging_virtual_memory_OS.exe

请输入要显示的进程号信息:1

进程信息:
进程名:q
进程号:1

段号:0  段大小:1024      是否在内存:Y
段内页表:
页号:0  页内偏移:0      块号:0  是否存在内存:Y
页号:1  页内偏移:2      块号:1  是否存在内存:Y

段号:1  段大小:512      是否在内存:N
段内页表:
页号:0  页内偏移:0      块号:-1  是否存在内存:N

1.申请内存
2.回收内存
3.显示内存信息
4.显示进程页表信息
5.地址转换
6.退出
请选择:

```

图 6.1.4 台程序显示进程页表信息

如图 5.2.2.4 所示为控制台程序显示进程页表信息。

```
C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS\segmented_Paging_virtual_memory_OS.exe
请输入进程ID:1
请输入段表起始地址:0
请输入段号:0
请输入页号:1
物理地址为:513
1. 申请内存
2. 回收内存
3. 显示内存信息
4. 显示进程页表信息
5. 地址转换
6. 退出
请选择:
```

图 6.1.5 控制台程序进行虚拟地址至物理地址的转换

如图 5.2.2.5 所示为使用控制台程序进行虚拟地址至物理地址的转换。

```
C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS\segmented_Paging_virtual_memory_OS.exe
请输入进程ID:0
请输入段表起始地址:0
请输入段号:1
请输入页号:2
缺页中断!
该页已经调入内存!
地址为:1540
```

图 6.1.6 控制台程序引发缺页中断

如图 5.2.2.6 所示为使用控制台程序引发缺页中断。

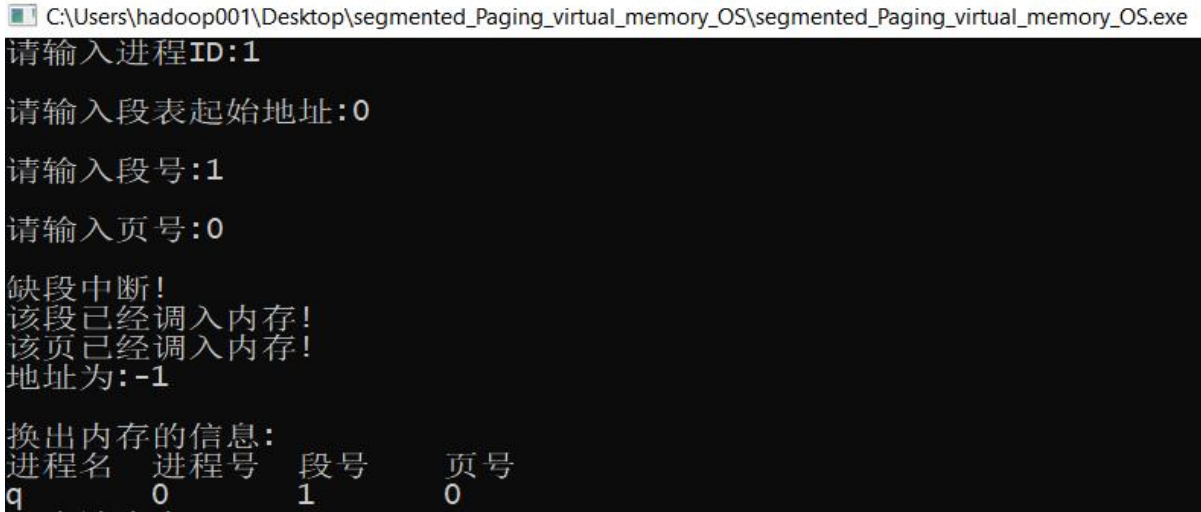


图 6.1.7 控制台程序引发缺段中断

如图 5.2.2.7 所示为使用控制台程序引发缺段中断。

4.2 GUI 程序的运行情况

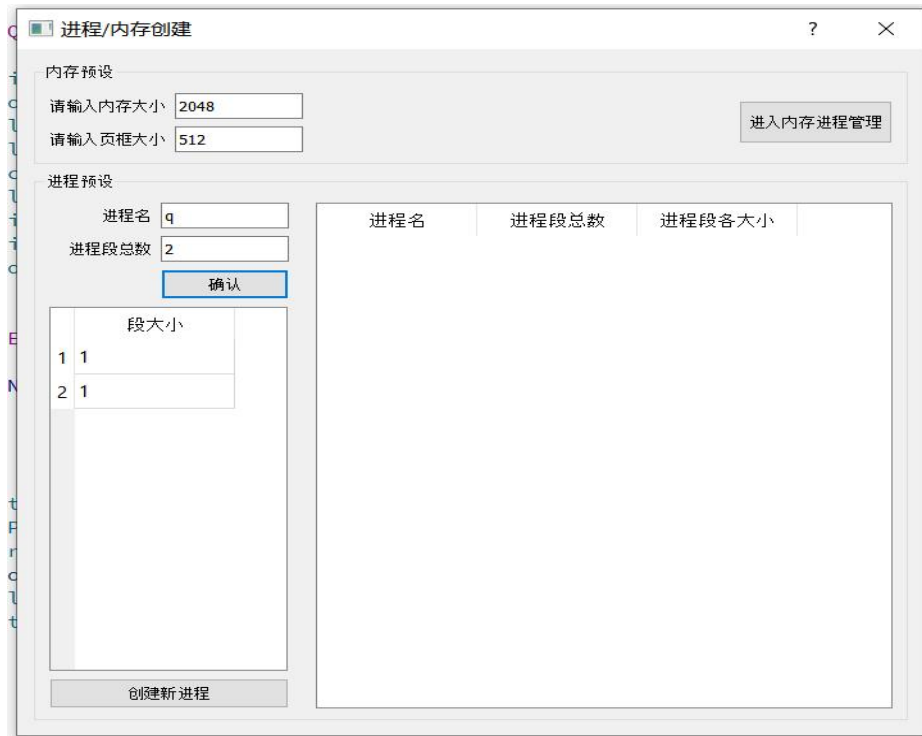


图 6.2.1 GUI 程序的开始界面

如图 6.2.1 所示为 GUI 程序的开始界面，出于用户友好考虑，输入框内置了 4 个输

入参数。

The GUI window titled "进程/内存创建" (Process/Memory Creation) contains the following elements:

- 内存预设 (Memory Preset):**
 - 请输入内存大小 (Please enter memory size): 2048
 - 请输入页框大小 (Please enter page frame size): 512
 - 按钮: 进入内存进程管理 (Enter memory process management)
- 进程预设 (Process Preset):**
 - 进程名 (Process name): q
 - 进程段总数 (Total number of process segments): 2
 - 按钮: 确认 (Confirm)
- 段大小表 (Segment Size Table):**

	段大小
1	3072
2	2048
- 主数据表 (Main Data Table):**

	进程名	进程段总数	进程段各大小
1	q	2	3072-2048
- 按钮:** 创建新进程 (Create new process)

图 6.2.2 GUI 程序输入有关参数建立段表与页表

如图 6.2.2 所示为使用 GUI 程序输入有关参数建立段表与页表。

进程/内存创建

内存预设

请输入内存大小2048

请输入页框大小512

进入内存进程管理

进程预设

进程名w

进程段总数2

确认

段大小

11024

2512

创建新进程

	进程名	进程段总数	进程段各大小
1	q	2	3072-2048
2	w	2	1024-512

图 6.2.3 GUI 程序输入有关参数建立段表与页表

如图 6.2.3 所示为使用 GUI 程序输入有关参数建立段表与页表。

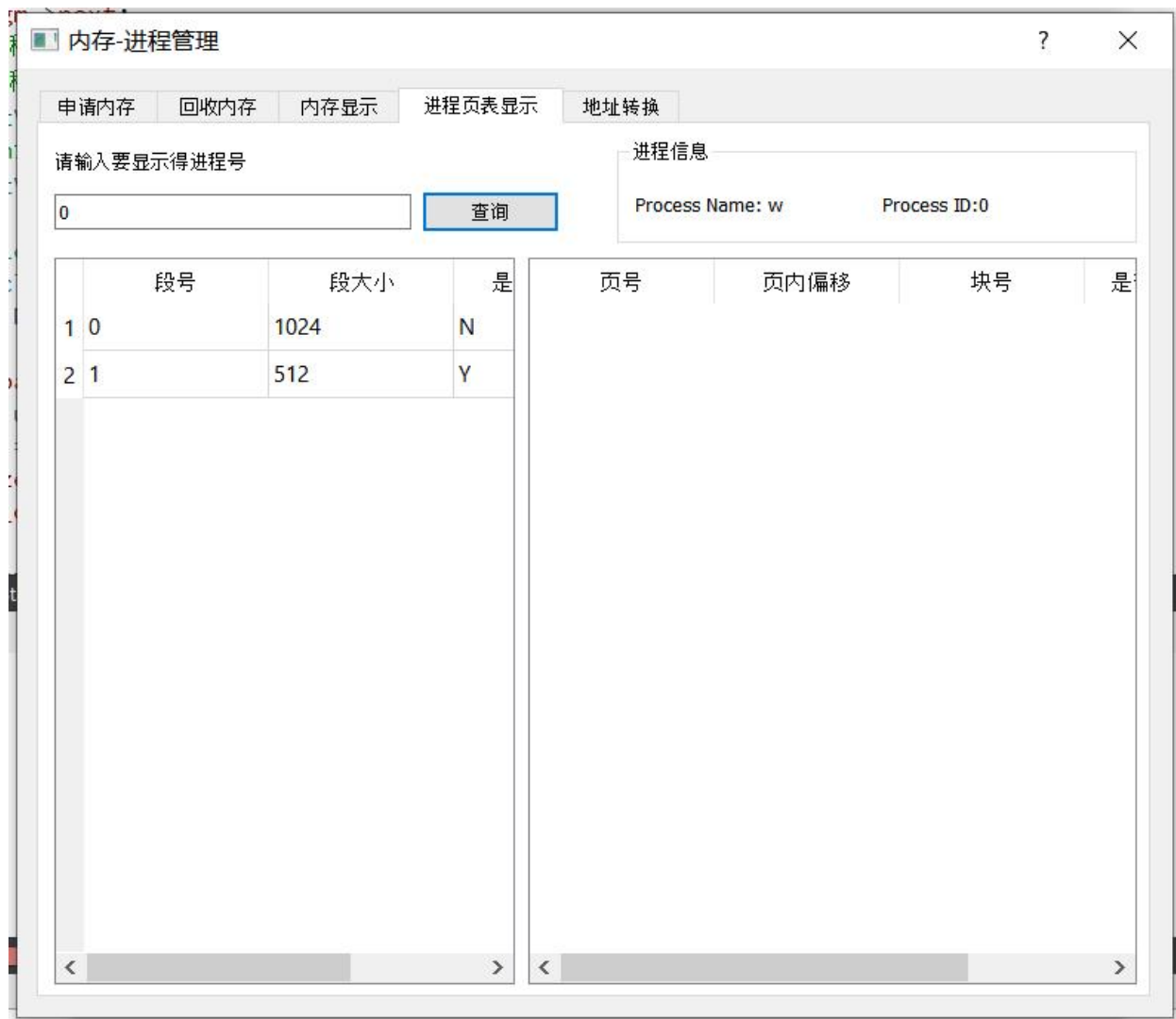


图 6.2.4 GUI 程序查询段表信息

如图 6.2.4 所示为使用 GUI 程序查询段表信息。



图 6.2.5 GUI 程序查询段表与页表信息

如图 6.2.5 所示为使用 GUI 程序查询段表与页表信息。



图 6.2.6 GUI 程序查询内存分配信息

如图 6.2.6 所示为使用 GUI 程序查询内存分配信息。

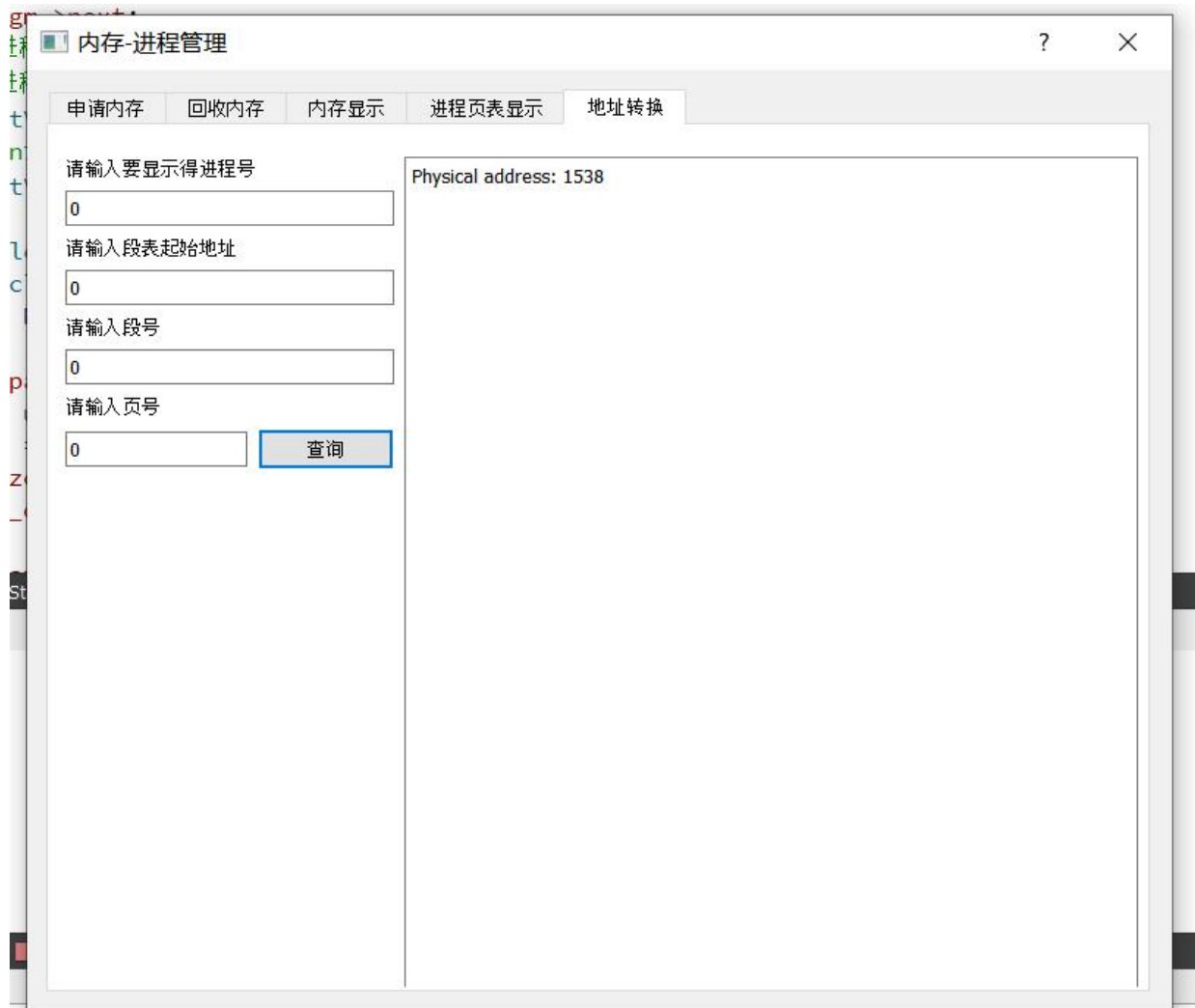


图 6.2.7 GUI 程序查询物理地址且成功

如图 6.2.7 所示为使用 GUI 程序查询物理地址且成功。

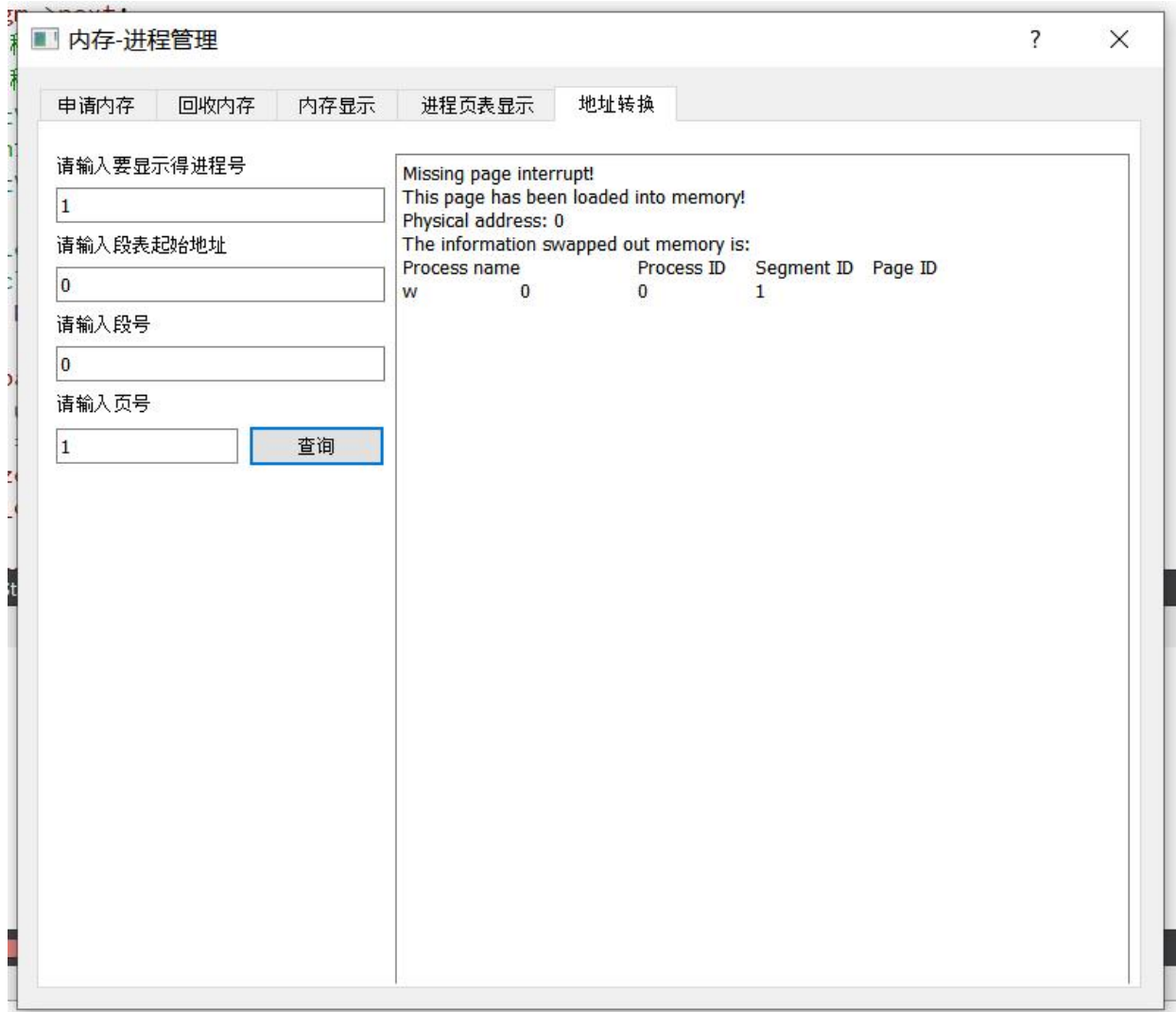


图 6.2.8 GUI 程序查询物理地址发生缺页中断

如图 6.2.8 所示为使用 GUI 程序查询物理地址但发生了缺页中断。

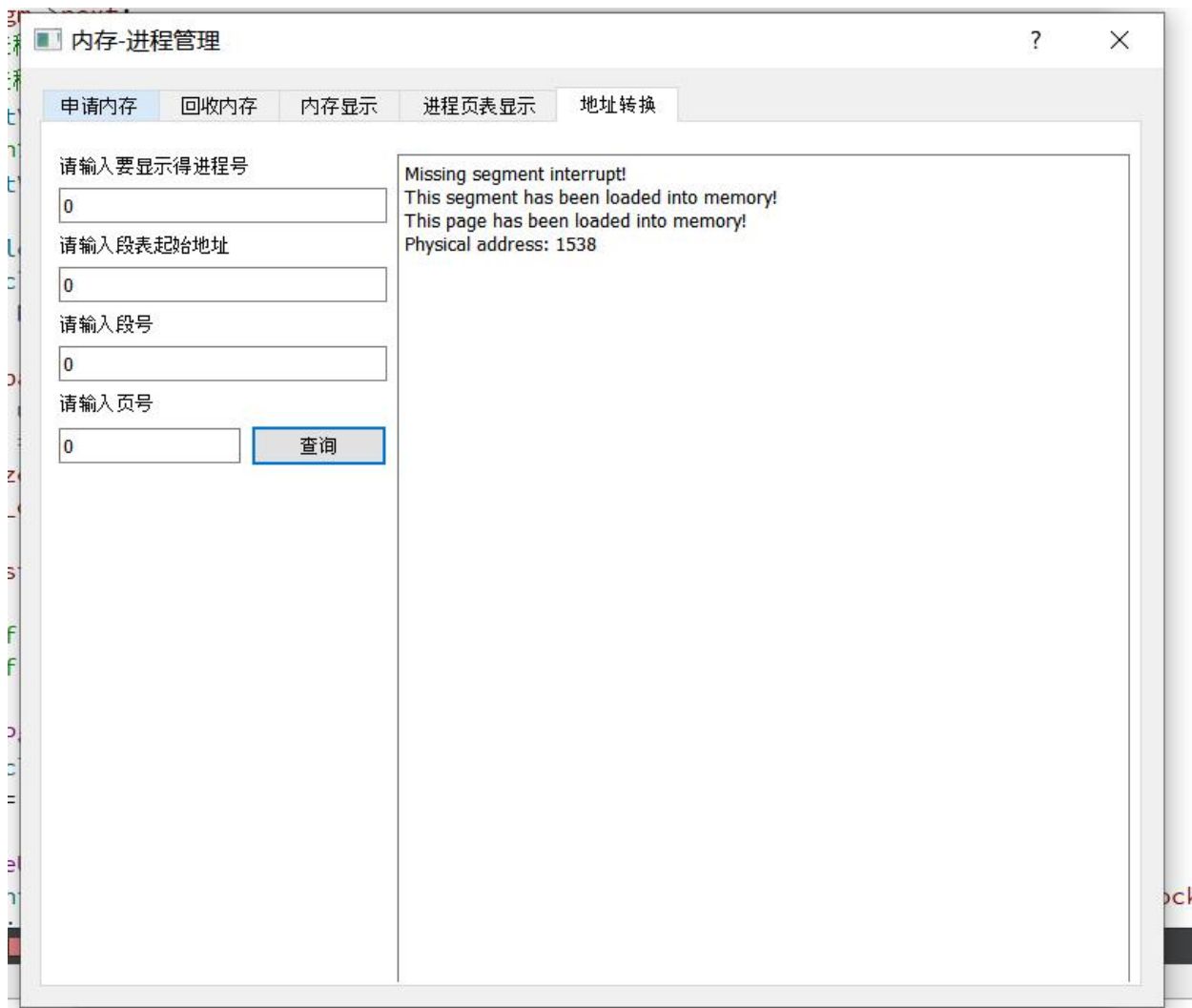


图 6.2.9 GUI 程序查询物理地址缺段中断

如图 6.2.9 所示为使用 GUI 程序查询物理地址但发生了缺段中断。

5 总结和感想体会

本次设计让我加深了对虚拟段页式系统以及 OS 内存管理方式的理解：虚拟存储器作为现代操作系统中存储器管理的一项重要技术，实现了内存扩充功能。但该功能并非是从物理上实际地扩大内存的容量，而是从逻辑上实现对内存容量的扩充，让用户所感觉到的内存容量比实际内存容量大得多。于是便可以让比内存空间更大的程序运行，或者让更多的用户程序并发运行。这样既满足了用户的需要，又改善了系统的性能。在完成设计的过程中，我也锻炼了自己的逻辑思考与运用所学知识解决实际问题的能力，当然，我的编程能力也得到了提高。

电子计算机从 1945 年诞生发展到今天，经历了 75 个春秋，期间出现了无数的名家大师，提出了很多划时代的思想与方法，极大地推动了计算机的发展，使其与人们的生活交织互印、深度融合，影响了人类文明的历史进程。没有哪一门现代学科像计算机这样发展迅猛，至今仍保持着旺盛的生命力，虚拟存储可谓是计算机历史上浓墨重彩的一笔，正所谓“万物皆可虚拟”，虚拟存储器的作用简单罗列如下：

①简化链接。独立的地址空间允许每个进程的存储器映像使用相同的基本格式，而不管代码和数据实际存放在物理存储器的何处。

②简化加载。虚拟存储器使得容易向存储器中加载可执行文件和共享文件对象。加载器从不实际从磁盘拷贝任何数据到存储器，虚拟存储器系统会按照需要自动地调入数据页。

③简化共享。一般而言，每个进程都拥有自己的独立地址空间，这是操作系统通过创建页表将相应的虚拟页映射到不同的物理页来实现的。但需要进程共享代码和数据时（如操作系统内核代码，标准库函数等），只需将不同进程中适当的虚拟页面映射到相同的物理页面，再安排多个进程共享这部分代码的一个拷贝，而不是在每个进程中都包括单独的内存和标准库的拷贝。

④简化存储器分配。虚拟存储器为用户进程提供了一个简单的分配额外存储器的机制。

参考文献

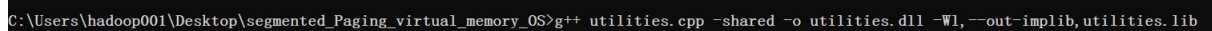
- [1]汤小丹, 梁红兵, 哲凤屏, 汤子瀛. 计算机操作系统 (第四版) [M]. 西安: 西安电子科技大学出版社, 2014.
- [2]https://blog.csdn.net/Bob__yuan/article/details/102584606
- [3]<https://www.cnblogs.com/xumaomao/p/12864406.html>
- [4]<https://zh.wikipedia.org/zh-hans/C%E8%AF%AD%E8%A8%80>
- [5]TIOBE Programming Community Index [TIOBE 编程社区指数]. 2012[2012-11-03] (英语).
- [6]<https://wu.wikipedia.org/wiki/Qt>

附录

1、Windows 10 使用 g++生成静态库文件与动态库文件

如图 1 所示, 在 Windows 控制台 cmd 使用如下命令生成静态库文件与动态库文件:

```
g++ utilities.cpp -shared -o utilities.dll -Wl,--out-implib,utilities.lib
```

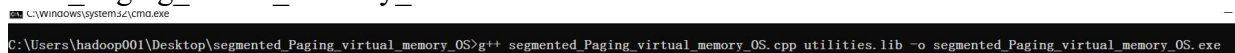


```
C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS>g++ utilities.cpp -shared -o utilities.dll -Wl,--out-implib,utilities.lib
```

图 1

如图 2 所示, 在 Windows 控制台 cmd 使用如下命令将测试程序与图 1 所生成的静态库文件与动态库文件链接得到完整的可执行文件:

```
g++ segmented_Paging_virtual_memory_OS.cpp utilities.lib -o  
segmented_Paging_virtual_memory_OS.exe
```



```
C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS>g++ segmented_Paging_virtual_memory_OS.cpp utilities.lib -o segmented_Paging_virtual_memory_OS.exe
```

图 2

如图 3 所示, 在 Windows 控制台 cmd 执行图 2 所得可执行程序。

```
C:\Windows\system32\cmd.exe
C:\Users\hadoop001\Desktop\segmented_Paging_virtual_memory_OS>segmented_Paging_virtual_memory_OS.exe
请输入内存大小:2048
请输入页框大小:512
请输入进程总数:1
请输入第1个进程名:w
请输入该进程的段总数:1
请输入第1个段的段大小: 1024
```

图 3