

**系统软件综合设计报告**

**操作系统分册**

**实验题目**  虚拟页式存储模型

**学生姓名**  余梓俊

**学 号**  2018211991

**专业班级**  计算机科学与技术18-3班

**指导教师**  田卫东 李琳 刘晓平 孙晓 罗月童 周红鹃

**完成日期**  2021年 7月9日

**合肥工业大学 计算机与信息学院**

目录

[1. 课程设计任务、要求、目的 3](#_Toc6014)

[1.1 课程设计目的和要求 3](#_Toc17474)

[1.2 课程设计任务 3](#_Toc13967)

[2. 开发环境 4](#_Toc19983)

[3. 相关原理及算法 4](#_Toc29640)

[3.1 物理地址与虚拟地址 4](#_Toc22429)

[3.2 页式存储 5](#_Toc8206)

[3.3 页表 6](#_Toc21021)

[3.4 地址转换 7](#_Toc9280)

[3.5 请求调页与Clock页面置换算法 8](#_Toc5525)

[4. 系统结构和主要的算法设计思路 10](#_Toc26696)

[4.1 控制台版本项目结构 10](#_Toc8772)

[4.2 GUI版本项目结构 12](#_Toc11139)

[4.3 算法流程图 13](#_Toc16865)

[5. 程序实现——主要数据结构 17](#_Toc8553)

[6. 程序实现——主要程序清单 19](#_Toc12309)

[6.1 创建和执行进程 19](#_Toc31922)

[6.2 访存和请求调页 22](#_Toc27565)

[6.3 内存可视化类的实现 23](#_Toc9273)

[7. 程序运行的主要界面和结果截图 26](#_Toc7752)

[7.1 控制台版本 26](#_Toc26741)

[7.2 GUI版本 31](#_Toc13970)

[8. 总结和感想体会 37](#_Toc1344)

[参考文献 38](#_Toc29009)

[附录 38](#_Toc23359)

# 课程设计任务、要求、目的

## 课程设计目的和要求

**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]。

**要求**

按照《系统软件综合设计指导书——操作系统分册》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]。

## 课程设计任务

页式虚拟存储管理系统：建立一个请求分页存储管理系统的模型。（1-2 人，难度：4）:

* 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兑换区；
* 建立应用程序的模型，包括分页结构在内；
* 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；
* 实现缺页中断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；

# 开发环境

* OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
* IDE: Visual Studio Code, Qt Creator 4.15.1 based on Qt 5.15.2
* Compiler: GCC 9.3.0

# 相关原理及算法

## 物理地址与虚拟地址

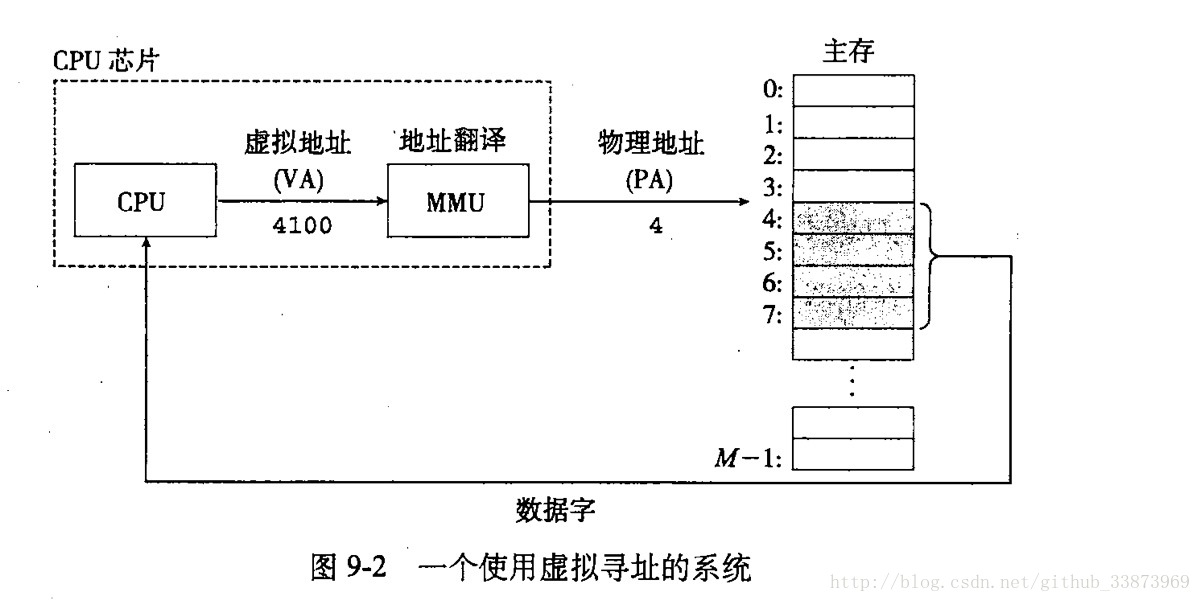
主存被组织成一个由M个连续的字节大小的单元组成的数组。而每一个字节都有一个对应的地址，这样的地址就被称作是物理地址。但是，我们的程序是不可以直接可以接触到物理地址的，所有的进程直接访问同一块连续的物理地址存在若干弊端：

1. 主存的容量有限。虽然我们现在的主存容量在不断上升，4G，8G，16G的主存都出现在市面上。但是我们的进程是无限，如果计算机上的每一个进程都独占一块物理存储器(即物理地址空间)。那么，主存就会很快被用完。但是，实际上，每个进程在不同的时刻都是只会用同一块主存的数据，这就说明了其实只要在进程想要主存数据的时候我们把需要的主存加载上就好，换进换出。针对这样的需求，直接提供一整块主存的物理地址就明显不符合。

2. 进程间通信的需求。如果每个进程都 独占一块物理地址，这样就只能通过socket这样的手段进行进程通信，但如果进程间能使用同一块物理地址就可以解决这个问题。

3. 主存的保护问题。对于主存来说，需要说明这段内存是可读的，可写的，还是可执行的。针对这点，光用物理地址也是很难做到的。

针对物理地址的直接映射的许多弊端，计算机的设计中就采取了一个虚拟化设计，就是虚拟内存。CPU通过发出虚拟地址，虚拟地址再通过MMU翻译成物理地址，最后获得数据，具体的操作如下所示：



利用了虚拟内存就可以比较有效的解决以上三个问题，在每一个进程开始创建的时候，都会分配一个虚拟存储器（就是一段虚拟地址）然后通过虚拟地址和物理地址的映射来获取真实数据，这样进程就不会直接接触到物理地址，甚至不知道自己调用的那块物理地址的数据。

## 页式存储

对于一整块连续的内存，直接连续使用也是不太符合实际的。于是，就有分页的概念。将1024个地址分成一页，通过访问页来访问数据。

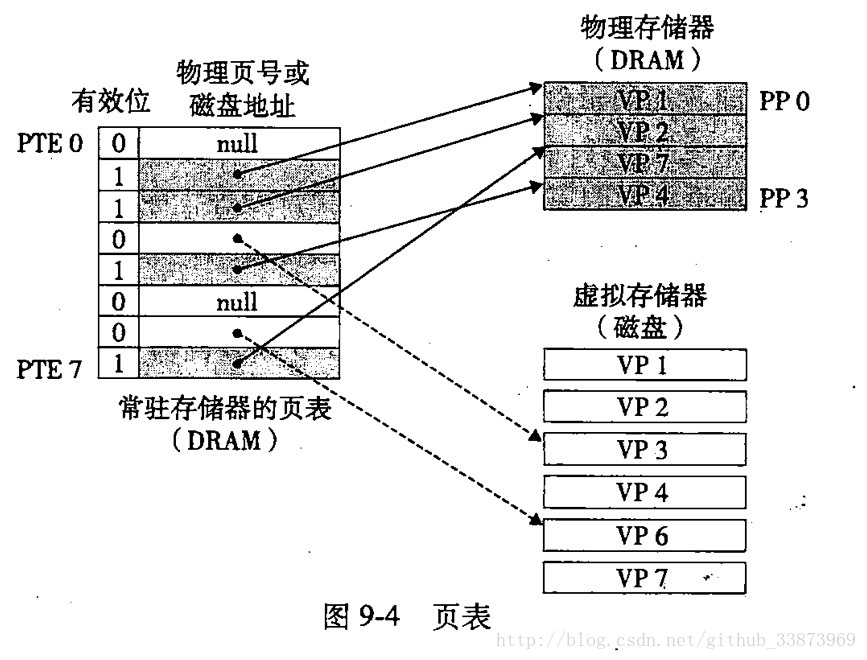
①将整个系统的内存空间划分成一系列大小相等的块，每一块称为一个物理块、物理页或实页，页架或页帧（frame），可简称为块（block）。所有的块按物理地址递增顺序连续编号为0、1、2、……。

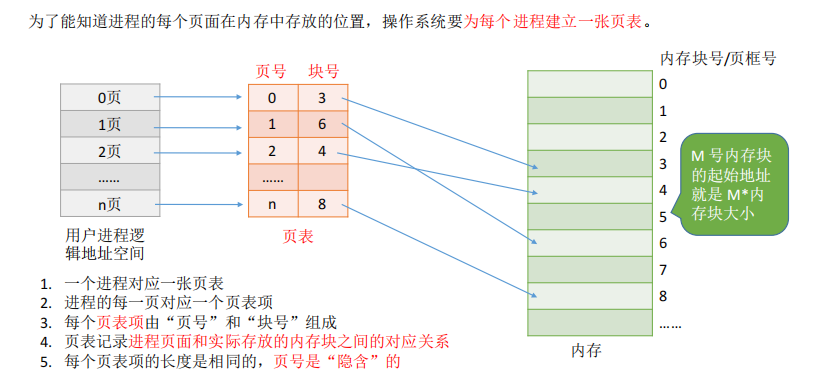
②每个作业的地址空间也划分成一系列与内存块一样大小的块，每一块称为一个逻辑页或虚页，也有人叫页面，可简称为页（page）。所有的页按照逻辑地址递增顺序连续编号为0、1、2、……。

③一个作业，只要它的总页数不大于内存中的可用块数，系统就可以对它实施分配。系统装入作业时，以页为单位分配内存，一页分配一个块，作业所有的页所占的块可以不连续。系统同时为这个作业建立一个页号与块号的对照表，称为页表。

## 页表

在分页系统中，允许将作业（进程）的任一页装入到内存中的任一可用的物理块中，但进程的地址空间本来是连续的，若把他分页后装入到不相邻的物理块中，要保证系统仍能正确运行，就要实现从进程的逻辑地址变换为内存的物理地址。所以，系统为每个进程建立一张页面映射表，简称页表。



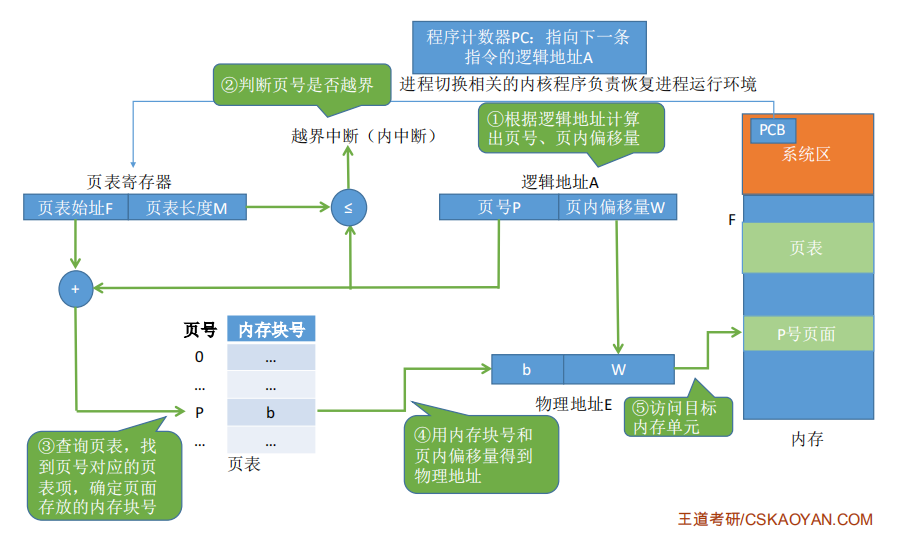


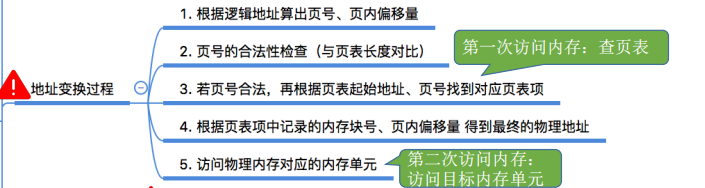
## 地址转换

在系统中设置地址变换机构，能将用户进程地址空间中的逻辑地址变为内存空间中的物理地址。

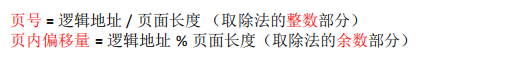
由于页面和物理块的大小相等，页内偏移地址和块内偏移地址是相同的。无须进行从页内地址到块内地址的转换。

地址变换机构的任务，关键是将逻辑地址中的页号转换为内存中的物理块号。物理块号内的偏移地址就是页内偏移地址。页表的作用就是从页号到物理块号的转换，所以地址变换的任务借助于页表来完成的。



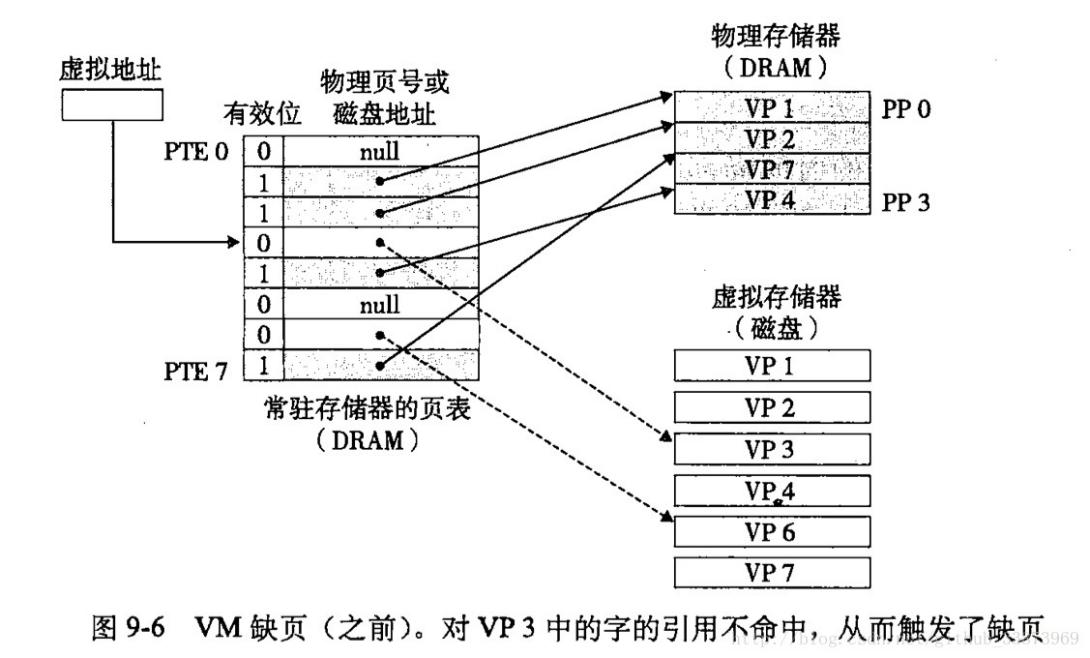


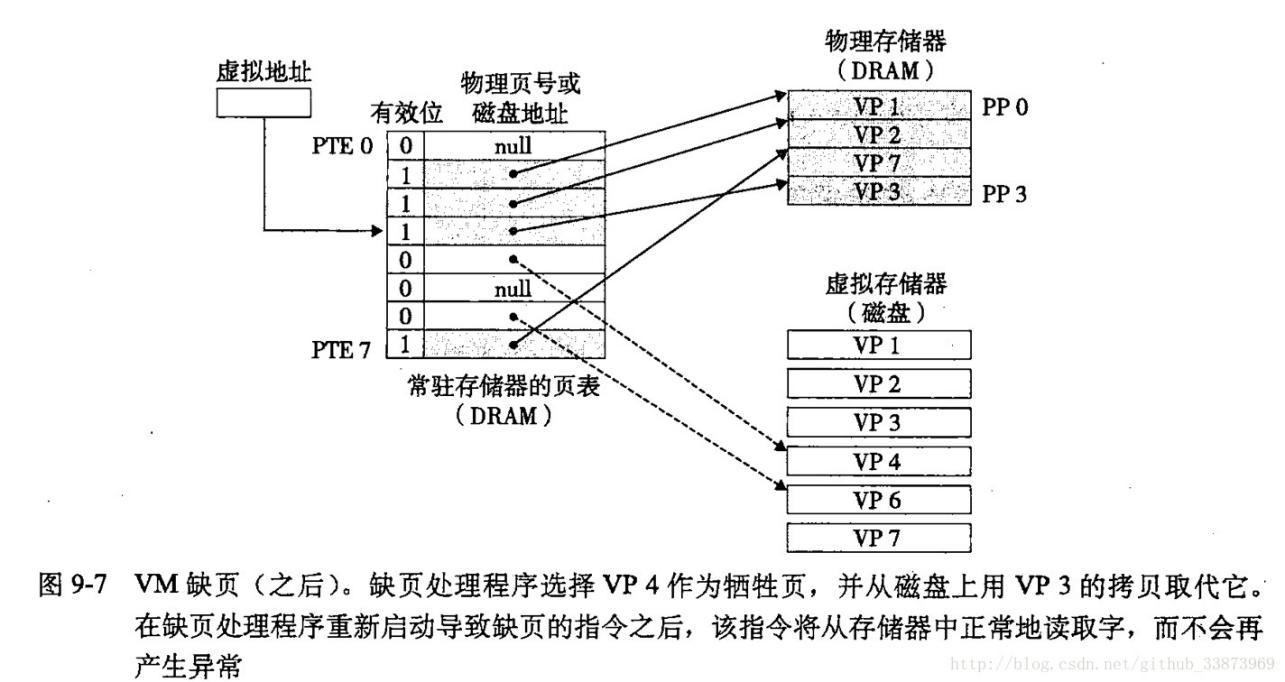
如果用十进制数表示逻辑地址，则：



## 3.5 请求调页与Clock页面置换算法

对于虚拟页式存储器来说：就有页命中，和页不命中两种情况。



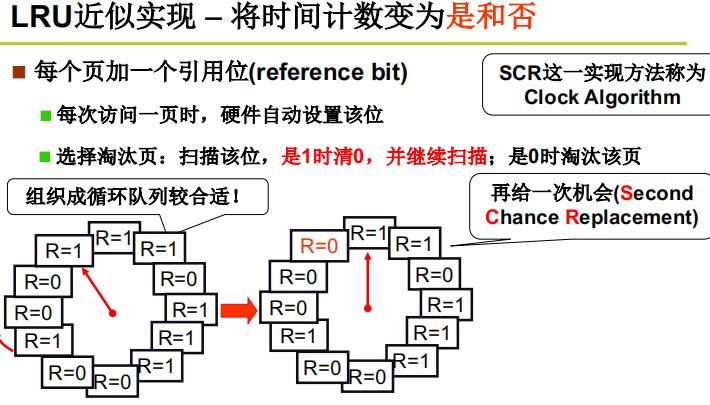


虚拟地址想访问VP3的时候，发现VP3未在缓存中，发生page fault。利用替换算法将物理存储器中的一个VP3从物理存储器中导出，VP4从磁盘导入DRAM中。此时,PTE3就变成了已缓存，PTE4变成了未缓存。这时候在进行地址翻译，就变成页命中了。

可见，page fault从磁盘导入的效率是非常低的，但是由于局部性原理，进程往往更多的在较小的活动页面上工作，很少有大跨度的访问内存，使得page fault产生的可能性降低。页命中的可能性提高。

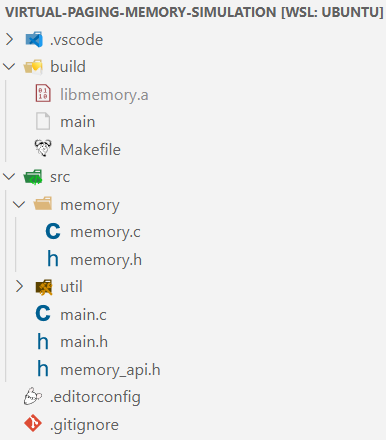
其中，页面替换算法有随机替换，先进先出（FIFO），最近最少使用（LRU）和时钟（Clock）置换算法。LRU能最好的利用局部性原理，但LRU的精确实现需要维护一个栈，并且会增加额外的访存次数，开销较大，在实际的操作系统中较少使用。

Clock算法是对LRU的近似，访存时只需要顺带将目标页表项的Reference Bit置为1即可，不需要额外的访存，并且这个动作可以设计为由硬件自动完成。



# 系统结构和主要的算法设计思路

## 控制台版本项目结构



Memory文件夹下的代码将编译成静态库libmemory.a。Main是控制台演示程序代码，util文件夹中的代码是main中用到的一些辅助函数。

Memory\_api头文件是memory模块提供的接口，包含了main中需要的一些结构体和函数声明。内容如下：

#ifndef MEMORY\_API\_H

#define MEMORY\_API\_H

#include <stdbool.h>

// Process page

typedef struct s\_Page {...

// Memory page frame

typedef struct s\_Frame {...

typedef struct s\_Program {...

typedef struct s\_PCB {...

void init\_memory();

int create\_program(int instructions[], int inst\_count);

int create\_process(int program\_id);

int release\_process(int process\_id);

int execute\_process(int process\_id);

Program\* find\_program\_by\_id(int program\_id);

PCB\* find\_process\_by\_id(int process\_id);

int get\_next\_instruction(PCB\* process);

int get\_page\_index(int address, int page\_size);

int get\_page\_offset(int address, int page\_size);

#endif

Makefile文件内容如下：

main: main.o util.o libmemory.a

**gcc -o main main.o util.o -static libmemory.a**

    rm \*.o

libmemory.a: ../src/memory/memory.c ../src/memory/memory.h

**gcc -c ../src/memory/memory.c**

**ar -rcs libmemory.a memory.o**

    rm memory.o

main.o: ../src/main.c ../src/main.h ../src/memory\_api.h

    gcc -c ../src/main.c

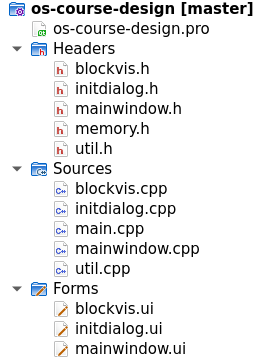
util.o: ../src/util/util.c ../src/util/util.h

    gcc -c ../src/util/util.c

clean:

    rm main libmemory.a \*.o

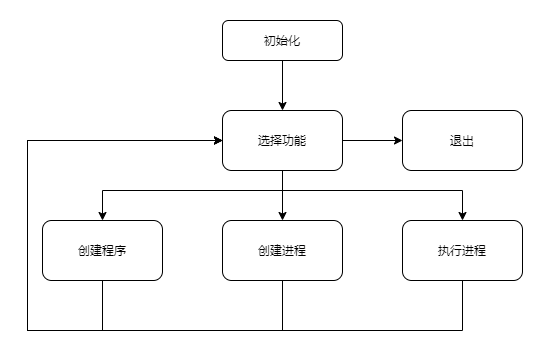
## GUI版本项目结构



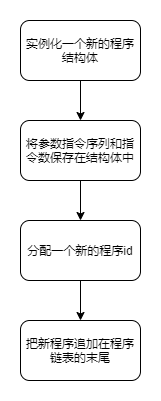
界面类有三个：initdialog是初始化对话框；mainwindow是主窗口，包括创建程序，创建进程，执行进程以及显示当前有哪些程序和进程；blockvis是物理内存和虚拟内存的可视化类，主窗口中创建进程和执行进程都会引起可视化实例的变化。

## 算法流程图

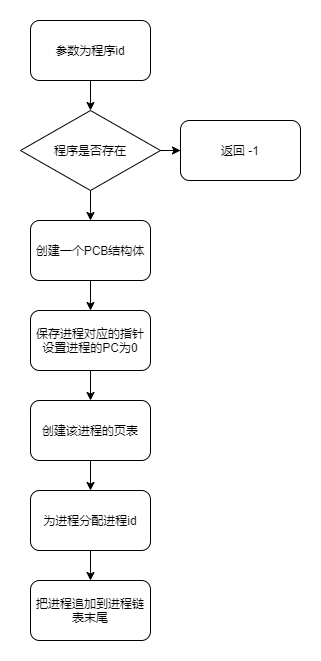
主函数流程：



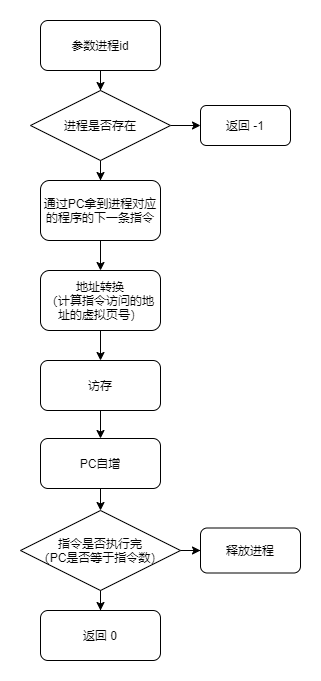
创建程序流程：



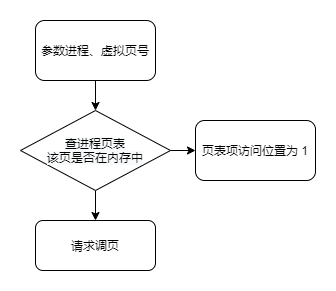
创建进程流程：



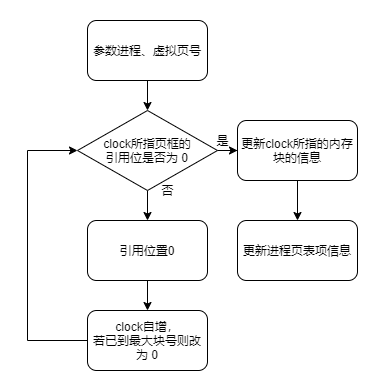
执行进程流程：



访存流程：



请求调页流程：



# 程序实现——主要数据结构

页框：

// Memory page frame

typedef struct s\_Frame {

    bool allocated;

    int process\_id;

    int page\_index;

    bool reference;

} Frame;

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 功能 |
| allocated | bool | 页框是否已分配 |
| process\_id | int | 分配给哪个进程 |
| page\_index | int | 对应进程的哪个虚拟页 |
| reference | bool | Clock算法引用位 |

页表项：

// Process page

typedef struct s\_Page {

    bool in\_mem;

    int frame\_index;

} Page;

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 功能 |
| in\_mem | bool | 是否在内存中 |
| frame\_index | int | 对应内存中哪个页框 |

程序：

typedef struct s\_Program {

    int id;

    int inst\_count;

    int instructions[128];

    struct s\_Program\* next;

} Program;

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 功能 |
| id | int | 程序id |
| inst\_count | int | 指令数 |
| instructions | int\* | 指令序列 |
| next | Program\* | 用于组织程序链表 |

进程：

typedef struct s\_PCB {

    int id;

    Program\* program;

    int inst\_executed;

    Page\* pages;

    struct s\_PCB\* next;

} PCB;

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 功能 |
| id | int | 进程id |
| program | Program\* | 对应的程序指针 |
| inst\_executed | int | 用于取下一条指令（PC） |
| pages | Page\* | 页表指针 |
| next | PCB\* | 用于组织进程链表 |

全局变量：

extern int VIRTUAL\_MEM\_PAGE;

extern int PHYSICAL\_MEM\_PAGE;

extern int PAGE\_SIZE;

extern Frame\* memory;

extern Program\* program\_list;

extern PCB\* pcb\_list;

extern int clock;

# 程序实现——主要程序清单

以下展示memory模块提供的接口中的5个较为重要的函数的实现。

## 创建和执行进程

memory\_api.h:

...

void init\_memory();

int create\_program(int instructions[], int inst\_count);

int create\_process(int program\_id);

int release\_process(int process\_id);

int execute\_process(int process\_id);

...

memory.c:

...

void init\_memory() {

    memory = (Frame\*) calloc(PHYSICAL\_MEM\_PAGE, sizeof(Frame));

    for (int i = 0; i < PHYSICAL\_MEM\_PAGE; ++i) {

        memory[i].allocated = false;

        memory[i].process\_id = -1;

        memory[i].page\_index = -1;

        memory[i].reference = false;

    }

}

int create\_program(int instructions[], int inst\_count) {

    Program\* program = (Program\*) malloc(sizeof(Program));

    program->inst\_count = inst\_count;

    memcpy(program->instructions, instructions, inst\_count \* sizeof(int));

    program->next = NULL;

    if (!program\_list) {

        program->id = 0;

        program\_list = program;

        return program->id;

    }

    Program\* p = program\_list;

    while (p->next) {

        p = p->next;

    }

    program->id = p->id + 1;

    p->next = program;

    return program->id;

}

int create\_process(int program\_id) {

    Program\* program = find\_program\_by\_id(program\_id);

    if (!program) {

        return -1;

    }

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

    pcb->program = program;

    pcb->inst\_executed = 0;

    pcb->next = NULL;

    pcb->pages = create\_process\_pages();

    if (!pcb\_list) {

        pcb->id = 0;

        pcb\_list = pcb;

        return pcb->id;

    }

    PCB\* p = pcb\_list;

    while (p->next) {

        p = p->next;

    }

    pcb->id = p->id + 1;

    p->next = pcb;

    return pcb->id;

}

int release\_process(int process\_id) {

    if (!pcb\_list) {

        return -1;

    }

    PCB\* p = pcb\_list;

    PCB\* prev = NULL;

    while (p) {

        if (p->id != process\_id) {

            prev = p;

            p = p->next;

            continue;

        }

        release\_process\_pages(p);

        if (!prev) {

            pcb\_list = pcb\_list->next;

        } else {

            prev->next = p->next;

        }

        free(p);

        return 0;

    }

    return -1;

}

int execute\_process(int process\_id) {

    PCB\* process = find\_process\_by\_id(process\_id);

    if (!process) {

        return -1;

    }

    Program\* program = process->program;

    if (process->inst\_executed >= program->inst\_count) {

        return -1;

    }

    int address = get\_next\_instruction(process);

    int page\_index = get\_page\_index(address, PAGE\_SIZE);

    int offset = get\_page\_offset(address, PAGE\_SIZE);

    visit\_page(process, page\_index);

    process->inst\_executed++;

    if (process->inst\_executed == program->inst\_count) {

        return release\_process(process\_id);

    }

    return 0;

}

## 访存和请求调页

其中execute\_process中用到的访存和请求调页函数如下:

memory.c:

void visit\_page(PCB\* process, int page\_index) {

    Page page = process->pages[page\_index];

    if (page.in\_mem) {

        // hit

        memory[page.frame\_index].reference = true;

        return;

    }

    // miss

    int frame\_index = get\_free\_frame();

    swap\_in(process, page\_index, frame\_index);

}

void swap\_in(PCB\* process, int page\_index, int frame\_index) {

    process->pages[page\_index].in\_mem = true;

    process->pages[page\_index].frame\_index = frame\_index;

    memory[frame\_index].allocated = true;

    memory[frame\_index].process\_id = process->id;

    memory[frame\_index].page\_index = page\_index;

    memory[frame\_index].reference = true;

}

void swap\_out(int frame\_index) {

    Frame frame = memory[frame\_index];

    PCB\* process = find\_process\_by\_id(frame.process\_id);

    Page page = process->pages[frame.page\_index];

    page.in\_mem = false;

    page.frame\_index = -1;

    frame.allocated = false;

    frame.process\_id = -1;

    frame.page\_index = -1;

    frame.reference = false;

}

int get\_free\_frame() {

    for (int i = 0; i < PHYSICAL\_MEM\_PAGE; ++i) {

        if (!memory[i].allocated) return i;

    }

    int frame\_index = find\_frame\_to\_swap\_out();

    swap\_out(frame\_index);

    return frame\_index;

}

int find\_frame\_to\_swap\_out() {

    while (1) {

        if (clock == PHYSICAL\_MEM\_PAGE) clock = 0;

        if (!memory[clock].allocated) {

            clock++;

            continue;

        }

        if (!memory[clock].reference) {

            return clock;

        }

        memory[clock].reference = false;

        clock++;

    }

}

## 内存可视化类的实现

blockvis.h:

...

class BlockVis : public QMainWindow

{

    Q\_OBJECT

public:

    explicit BlockVis(QWidget \*parent = nullptr);

    explicit BlockVis(QWidget \*parent = nullptr, int block\_num = 8, int id = 0, QColor color = Qt::white);

    QColor get\_color();

    void set\_block(int index, int value, QColor color);

    void unset\_block(int index);

    ~BlockVis();

private:

    Ui::BlockVis \*ui;

    int block\_num;

    int id;

    QColor color;

    QStandardItemModel \*block\_table\_model;

};

...

在图形界面版本中，初始化对话框完成后，进入主程序，同时创建一个Blockvis类实例，弹出一个对物理内存可视化的新窗口。

每创建一个进程，同时会创建一个blockvis实例，弹出一个该进程看到的虚拟内存的可视化新窗口。

Blockvis类提供了两个重要的接口，set\_block和unset\_block。在执行进程时，如果发生请求调页，将会调用这两个接口，更新虚拟内存和物理内存的可视化窗口。

Blockvis类的实现如下：

...

BlockVis::BlockVis(QWidget \*parent, int block\_num, int id, QColor color) :

    QMainWindow(parent),

    ui(new Ui::BlockVis)

{

    ui->setupUi(this);

    this->id = id;

    this->block\_num = block\_num;

    this->color = color;

    int block\_height = 480 / block\_num;

    block\_height = block\_height < 30 ? 30 : block\_height;

    this->block\_table\_model = new QStandardItemModel;

    this->block\_table\_model->setColumnCount(1);

    this->block\_table\_model->setRowCount(block\_num);

    this->ui->block\_table->setModel(block\_table\_model);

    this->ui->block\_table->setColumnWidth(0, 178);

    this->ui->block\_table->verticalHeader()->setDefaultSectionSize(block\_height);

    this->ui->block\_table->verticalHeader()->setHidden(true);

    this->ui->block\_table->horizontalHeader()->setHidden(true);

}

BlockVis::~BlockVis()

{

    delete ui;

}

QColor BlockVis::get\_color() {

    return this->color;

}

void BlockVis::set\_block(int index, int value, QColor color) {

    auto model = this->block\_table\_model;

    model->setItem(index, 0, new QStandardItem(QString::number(value)));

    model->item(index, 0)->setTextAlignment(Qt::AlignCenter);

    model->item(index, 0)->setFont(QFont("Times", 14, QFont::Black));

    model->item(index, 0)->setBackground(color);

}

void BlockVis::unset\_block(int index) {

    auto model = this->block\_table\_model;

    model->item(index, 0)->setBackground(Qt::white);

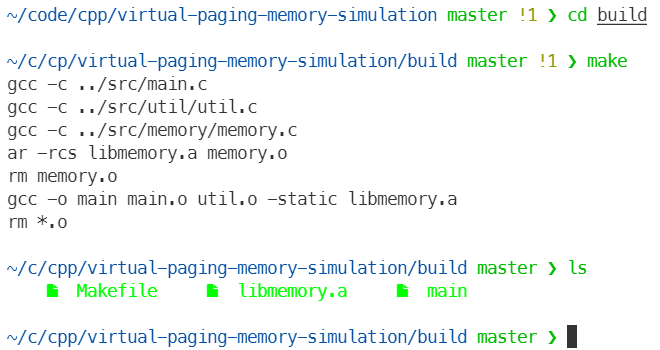
    model->setItem(index, 0, new QStandardItem(""));

}

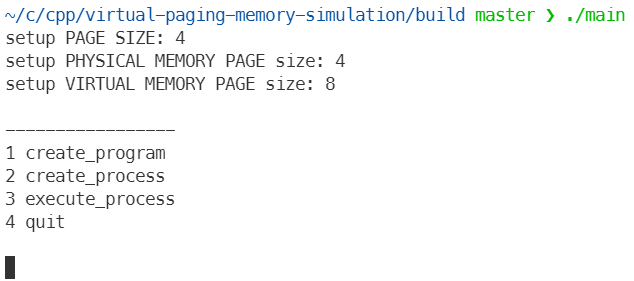
# 程序运行的主要界面和结果截图

## 控制台版本

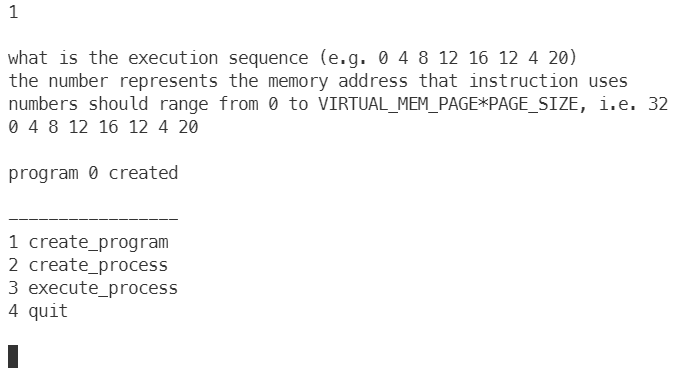
编译：



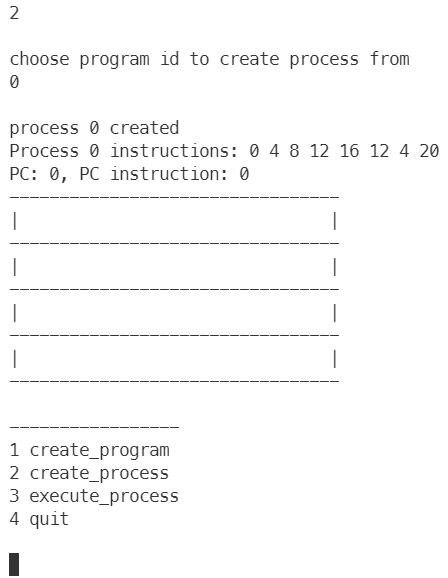
初始化：



创建程序：

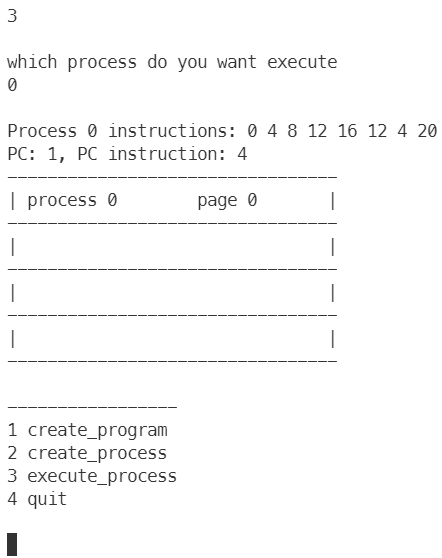


创建进程：

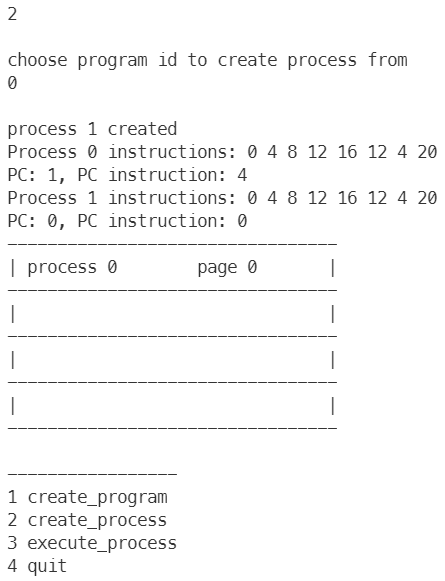


图中的方格为物理内存状态的展示，可以看出，当前物理内存中没有放入任何一个进程的页面。

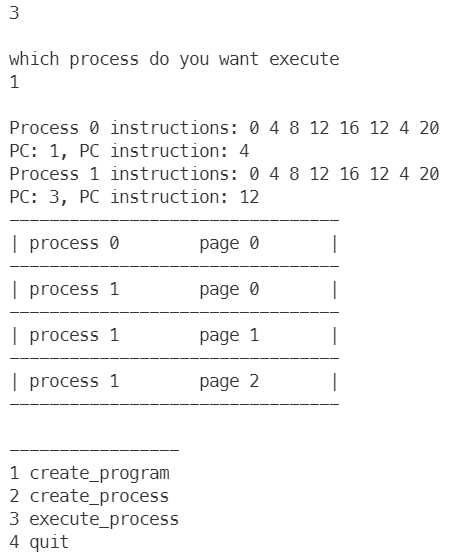
执行进程：



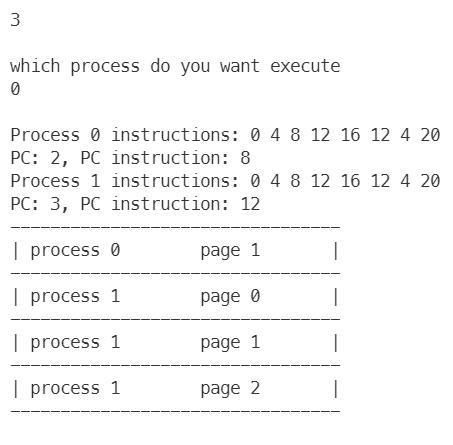
再次用0号程序创建一个进程：



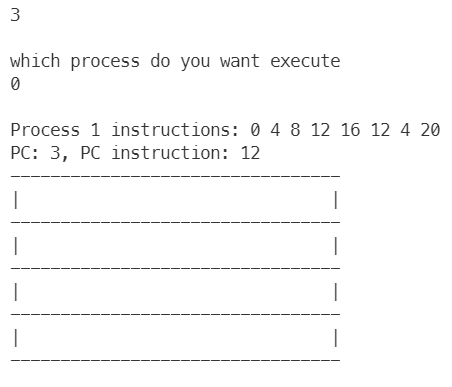
执行新的进程若干次：



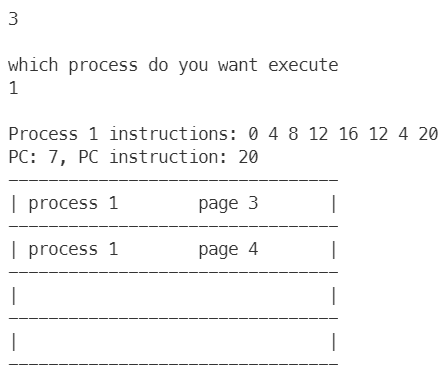
在物理内存已满的情况下再次执行进程0，将发生页面换入换出：

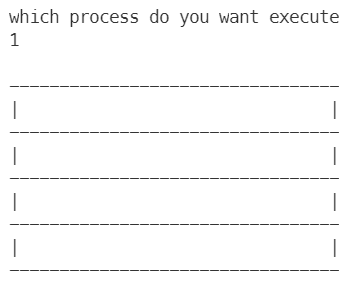


继续执行进程0，直至进程0退出：



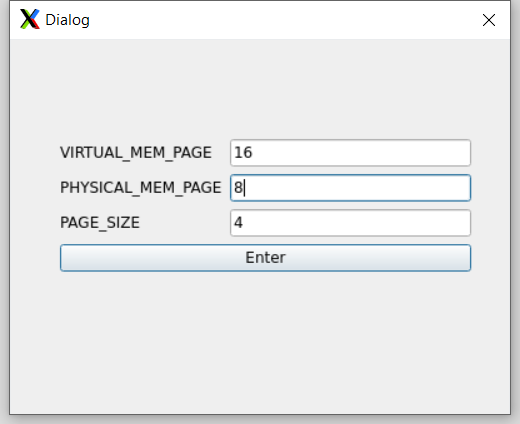
继续执行进程1，直至进程1退出：

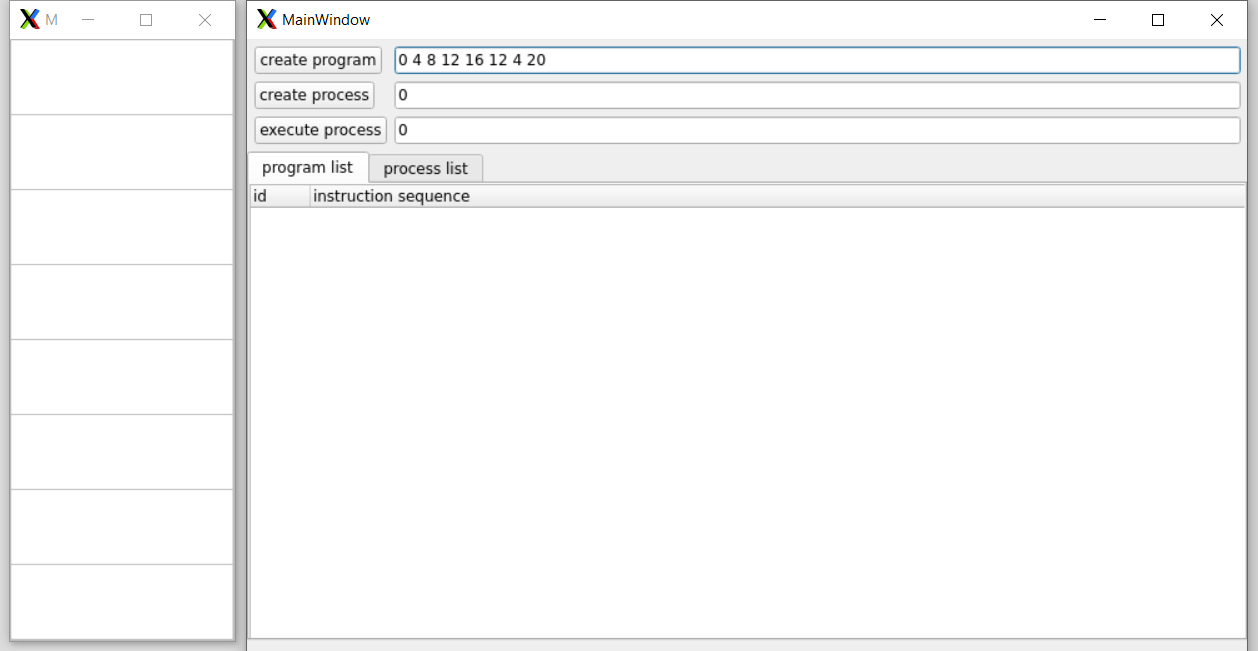




## GUI版本

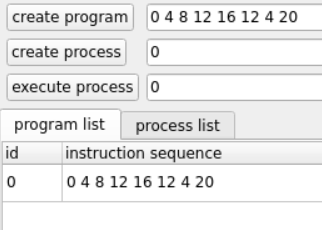
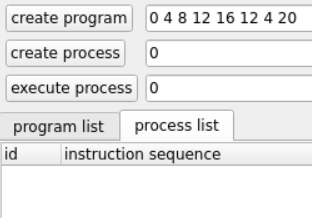
初始化：



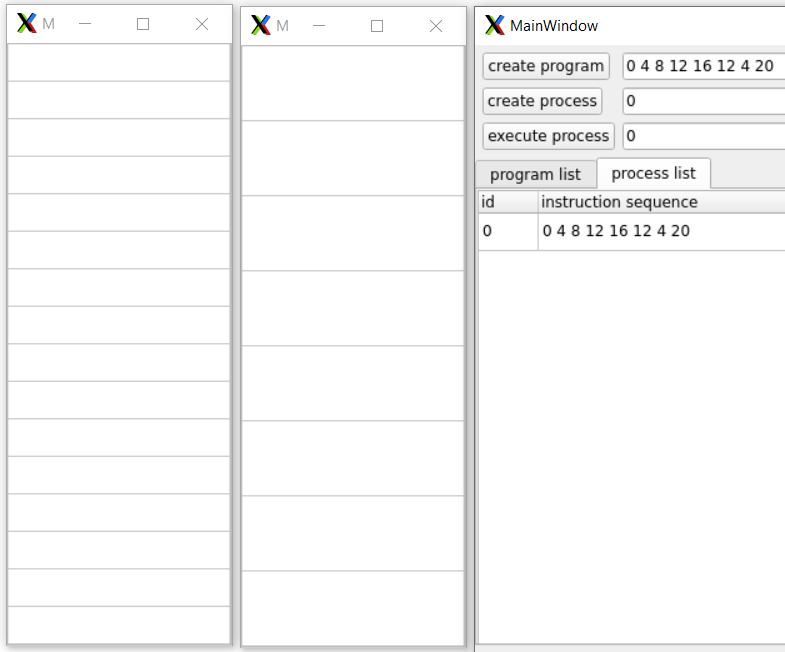


左侧的方格即为物理内存的可视化，可以看到总数为8块。

创建程序：

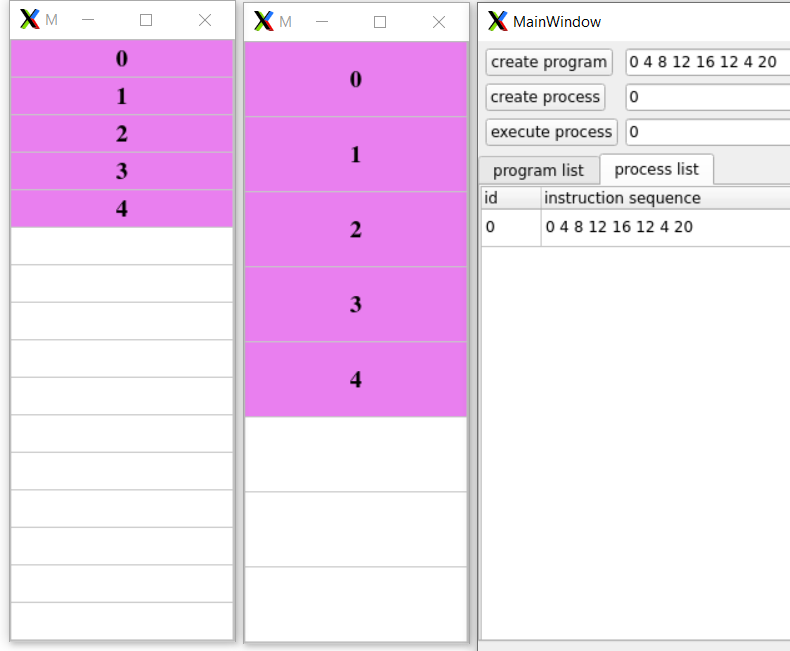
 

创建进程：



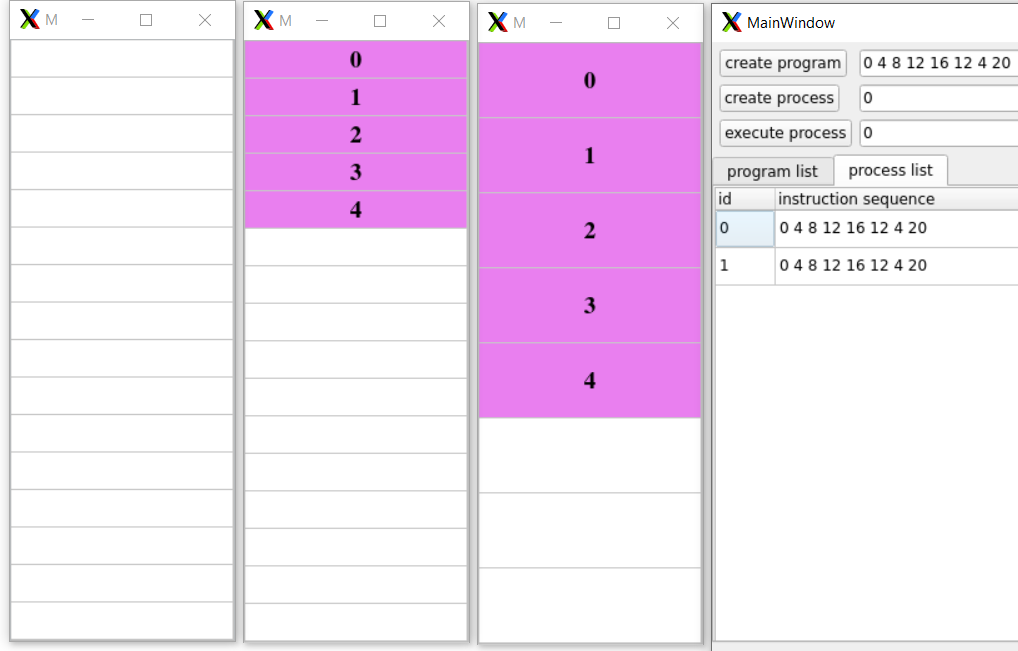
最左侧的方格为0号进程看到的虚拟内存的可视化，可以看到总数为16个页面。

执行进程0若干次：

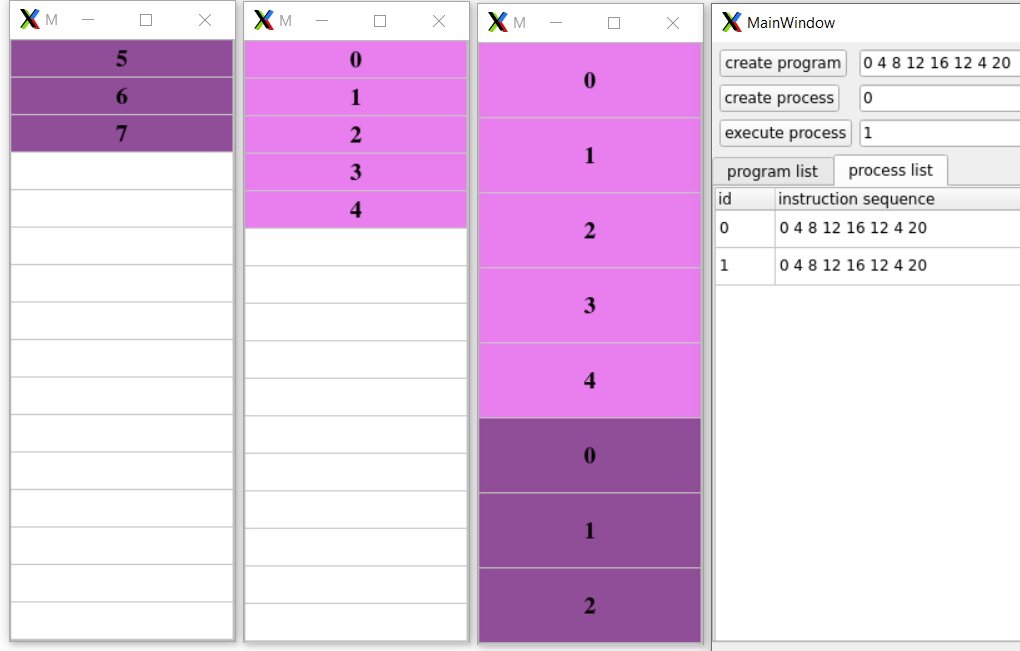


虚拟内存页上的数字为该页在内存中的页框号，物理内存页框上的数字为该页框对应的虚拟页号（用颜色来表示是哪个进程的页，这里为进程0随机分配的颜色为浅紫色）。

创建一个新的进程：

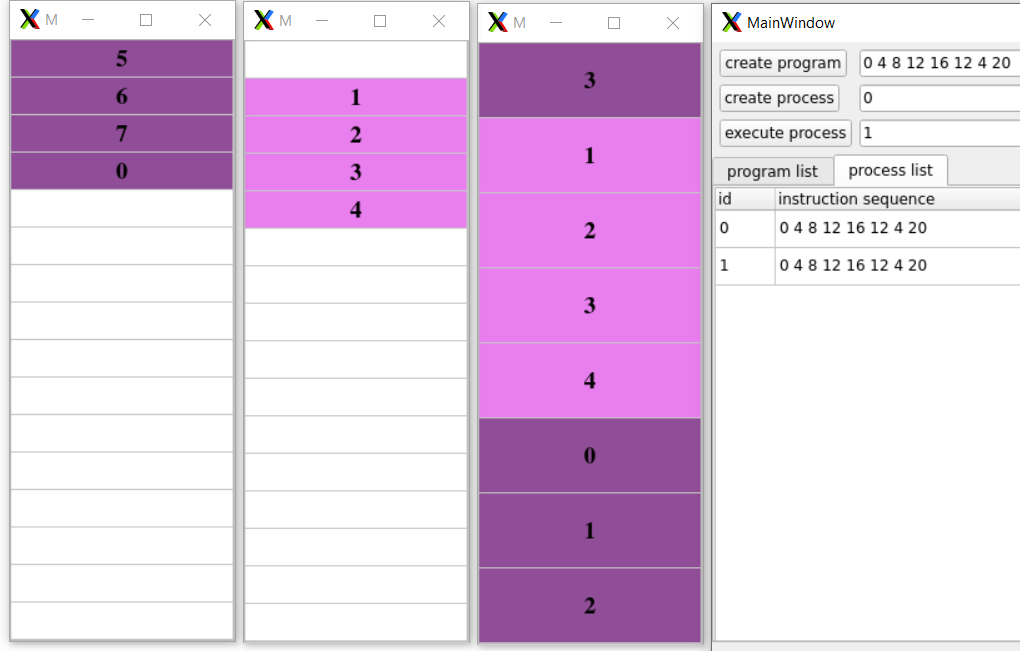


执行新进程（1号进程）若干次：

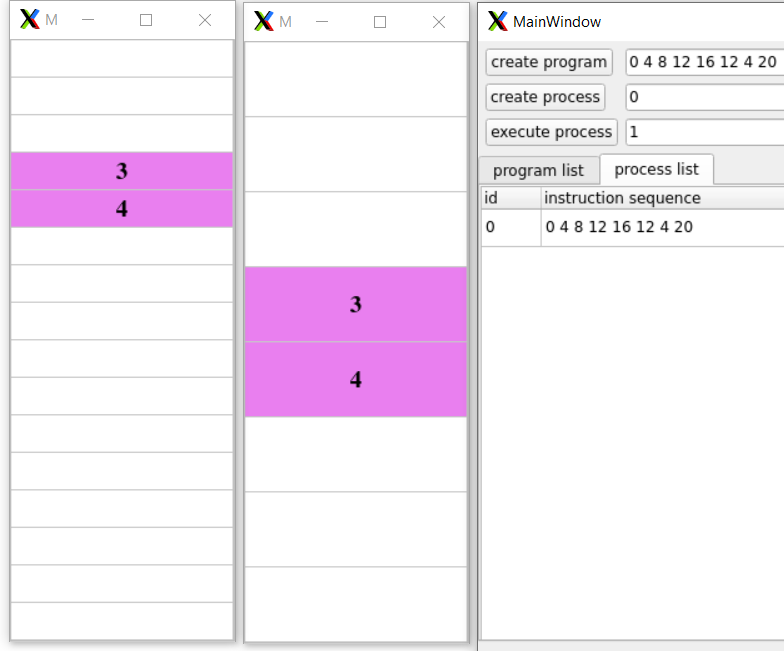


这里为进程1随机分配的颜色为深紫色。

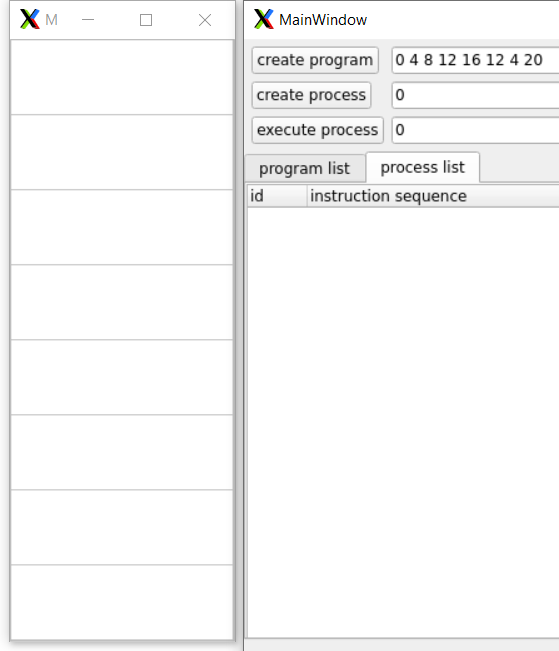
再次执行进程1，将发生页面换入换出：



继续执行进程1，直至退出：



继续执行进程0，直至退出：



# 总结和感想体会

本次设计让我加深了对虚拟页式系统以及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。将内存模块与演示程序的代码分离，将前者编译后打包为静态库文件，使我明白了如何用C语言编写功能模块供其他程序使用，也让我更加熟悉了Makefile的编写。使用Qt编写图形界面，让我掌握了通过Web前端以外的技术编写演示程序的技能。

# 参考文献

1. 田卫东，李琳，刘晓平，孙晓，罗月童，周红鹃. 系统软件综合设计指导书——操作系统分册（第三版）[M]. 合肥：合肥工业大学， 2018.
2. 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.
3. <https://zh.wikipedia.org/zh-hans/C%E8%AF%AD%E8%A8%80>
4. <https://wiki.qt.io/Qt_5>

# 附录

项目Github地址：

<https://github.com/wine99/virtual-paging-memory-simulation>