合肥工業大學

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

目录

1. 课程设计任务、要求、目的

1.1 课程设计目的和要求

①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]。

②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]。

1.2 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；

2. 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

3. 相关原理及算法

3.1 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：

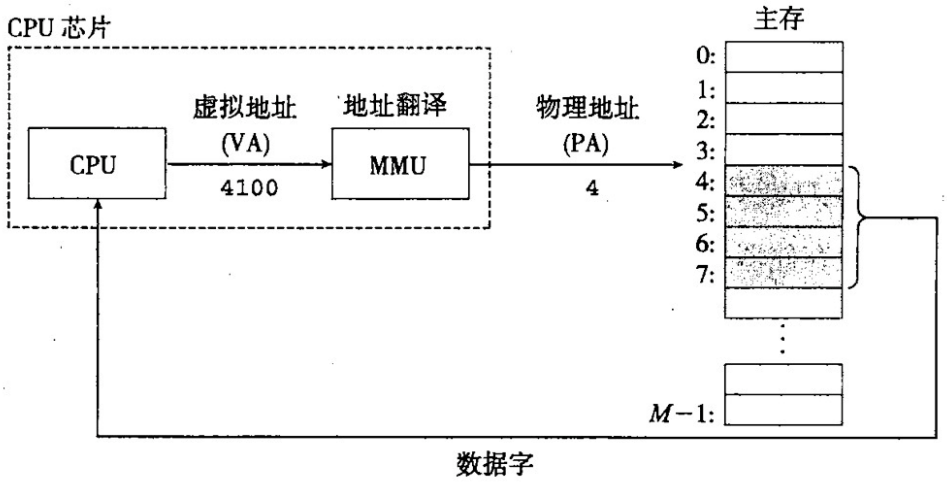


图 9-2　一个使用虚拟寻址的系统

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。

3.2 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。

3.3 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。

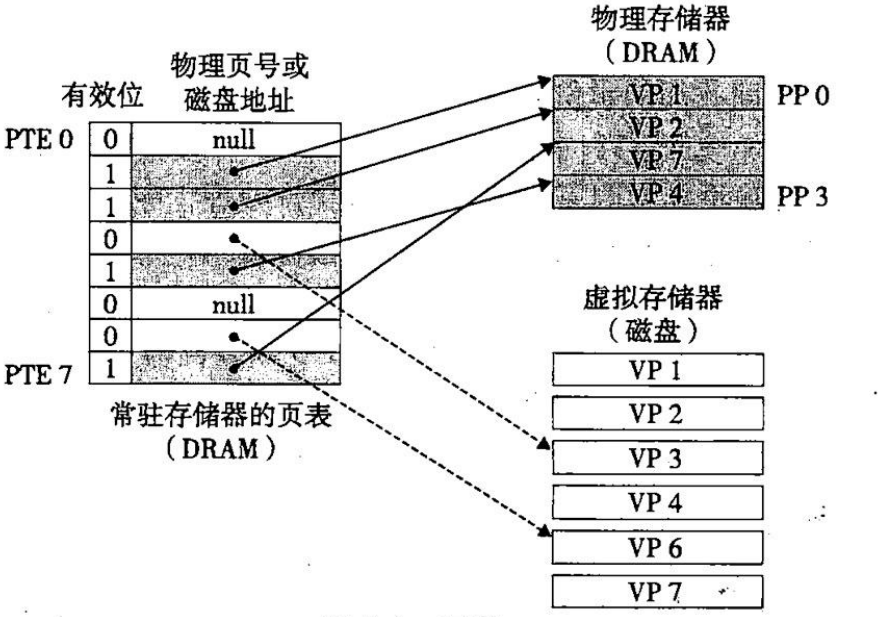
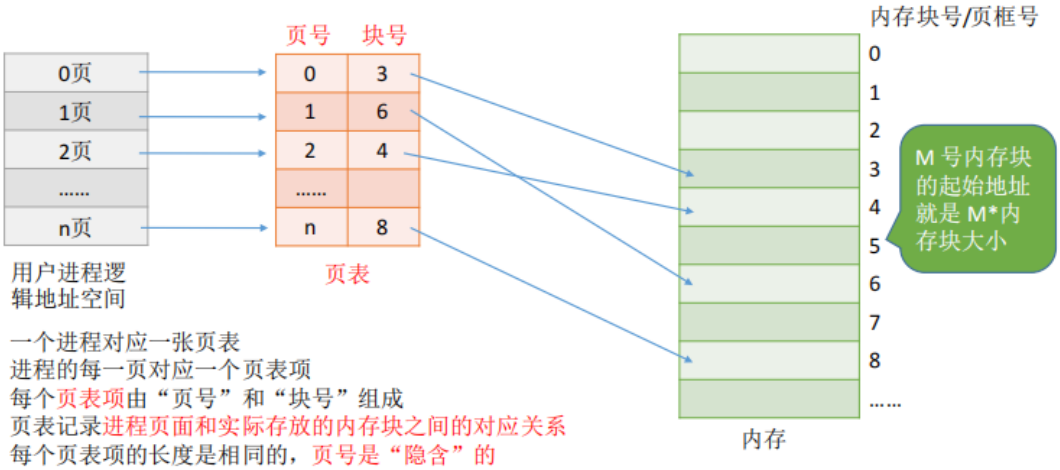


图 9-4　页表

为了能知道进程的每个页面在内存中存放的位置，操作系统要为每个进程建立一张页表。



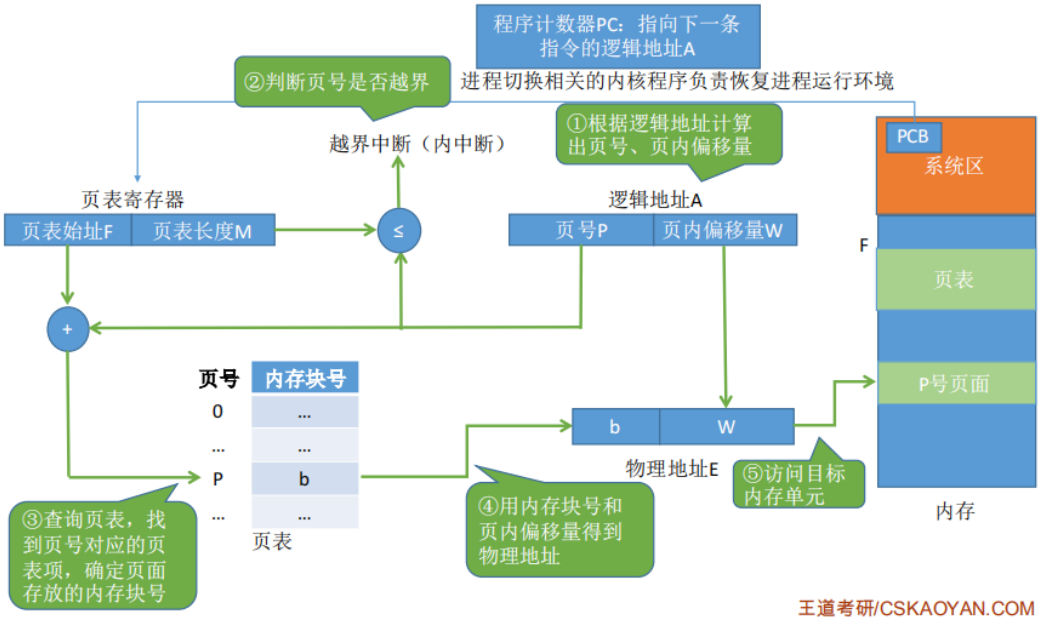
1. 一个进程对应一张页表
2. 进程的每一页对应一个页表项
3. 每个页表项由“页号”和“块号”组成
4. 页表记录进程页面和实际存放的内存块之间的对应关系
5. 每个页表项的长度是相同的，页号是“隐含”的

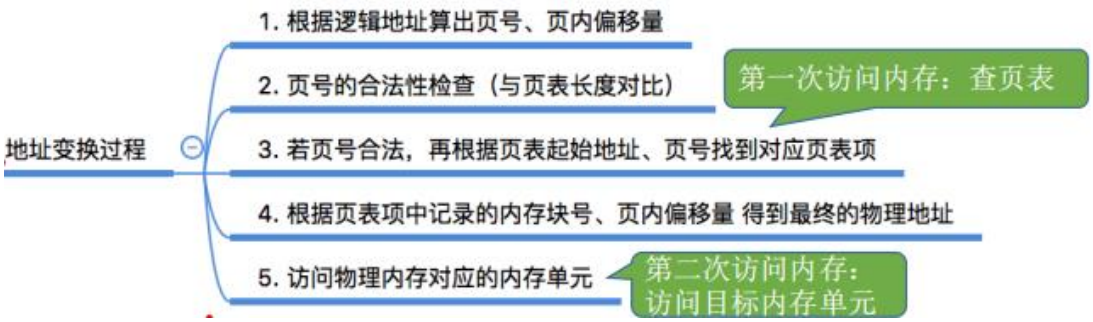
3.4 地址转换

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：

页号 = 逻辑地址 / 页面长度 （取除法的整数部分）
页内偏移量 = 逻辑地址 % 页面长度（取除法的余数部分）

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。

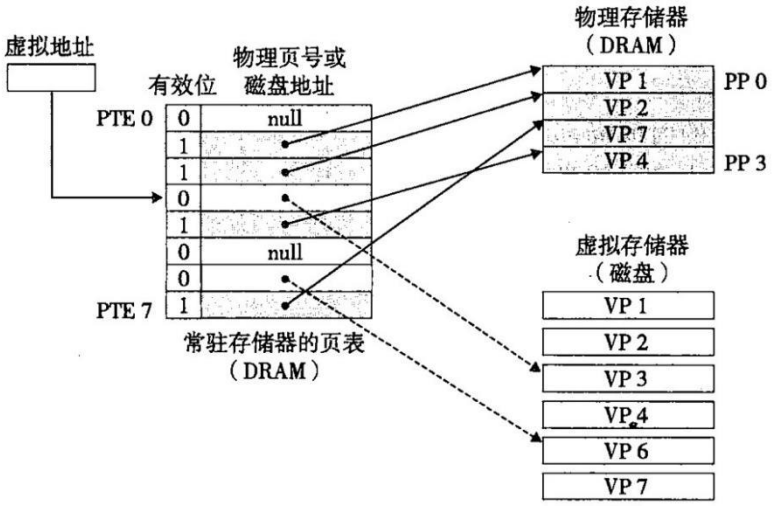


图 9-6 VM 缺页（之前）。对 VP 3 中的字的引用不命中，从而触发了缺页

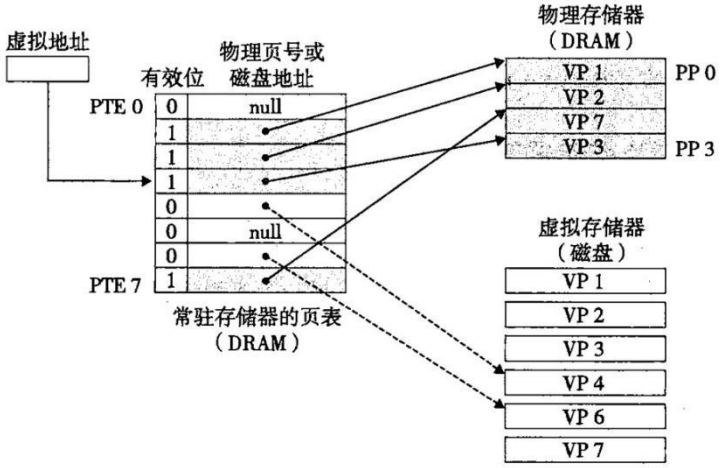


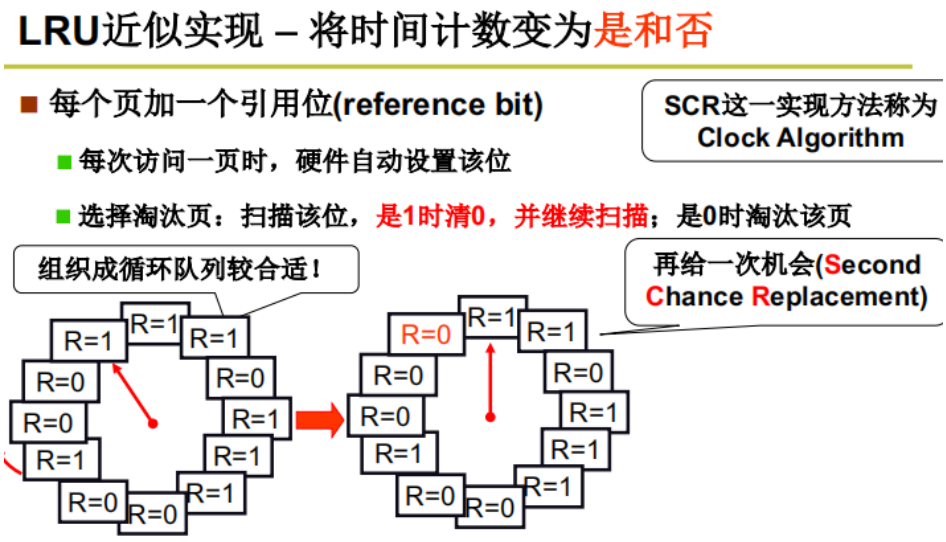
图 9-7 VM 缺页（之后）。缺页处理程序选择 VP 4 作为牺牲页，并从磁盘上用 VP 3 的拷贝取代它。在缺页处理程序重新启动导致缺页的指令之后，该指令将从存储器中正常地读取字，而不会再产生异常

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。

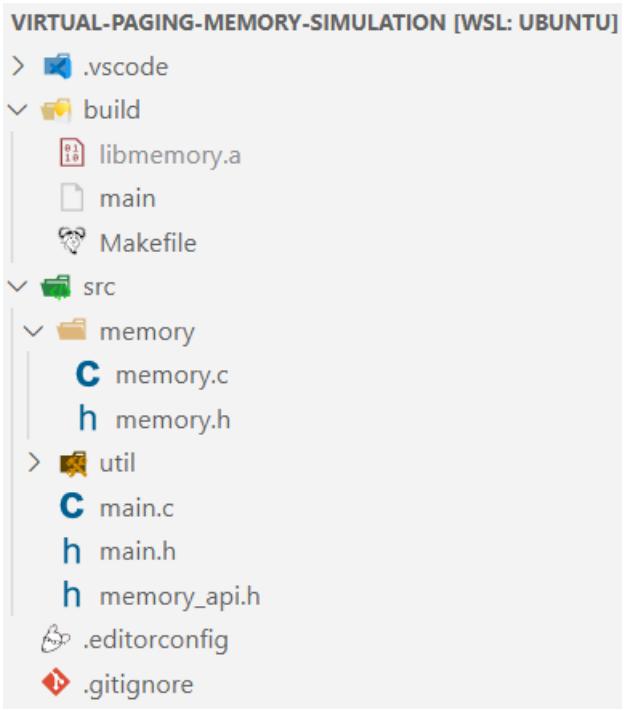
其中，页面替换算法有随机替换，先进先出（FIFO），最近最少使用（LRU）和时钟（Clock）置换算法。LRU 能最好的利用局部性原理，但 LRU 的精确实现需要维护一个栈，并且会增加额外的访存次数，开销较大，在实际的操作系统中较少使用。

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。



4. 系统结构和主要的算法设计思路

4.1 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：

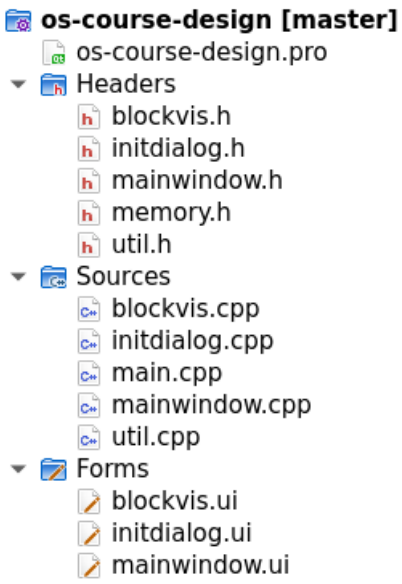
```
#ifndef MEMORY_API_H
#define MEMORY_API_H
#include <stdbool.h>

// 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
```

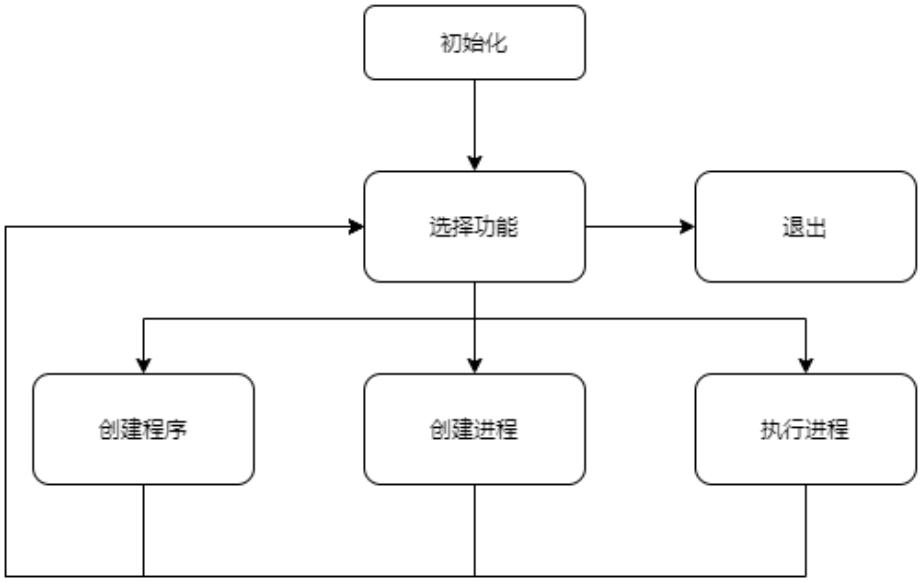
4.2 GUI 版本项目结构



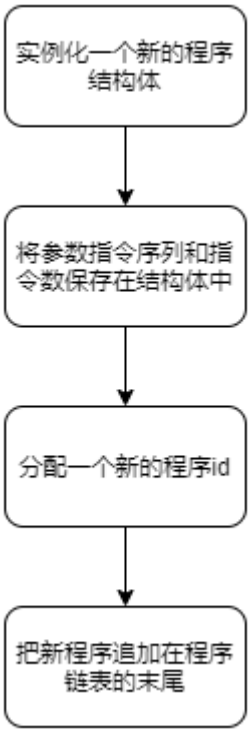
界面类有三个：initdialog 是初始化对话框；mainwindow 是主窗口，包括创建程序，创建进程，执行进程以及显示当前有哪些程序和进程；blockvis 是物理内存和虚拟内存的可视化类，主窗口中创建进程和执行进程都会引起可视化实例的变化。

4.3 算法流程图

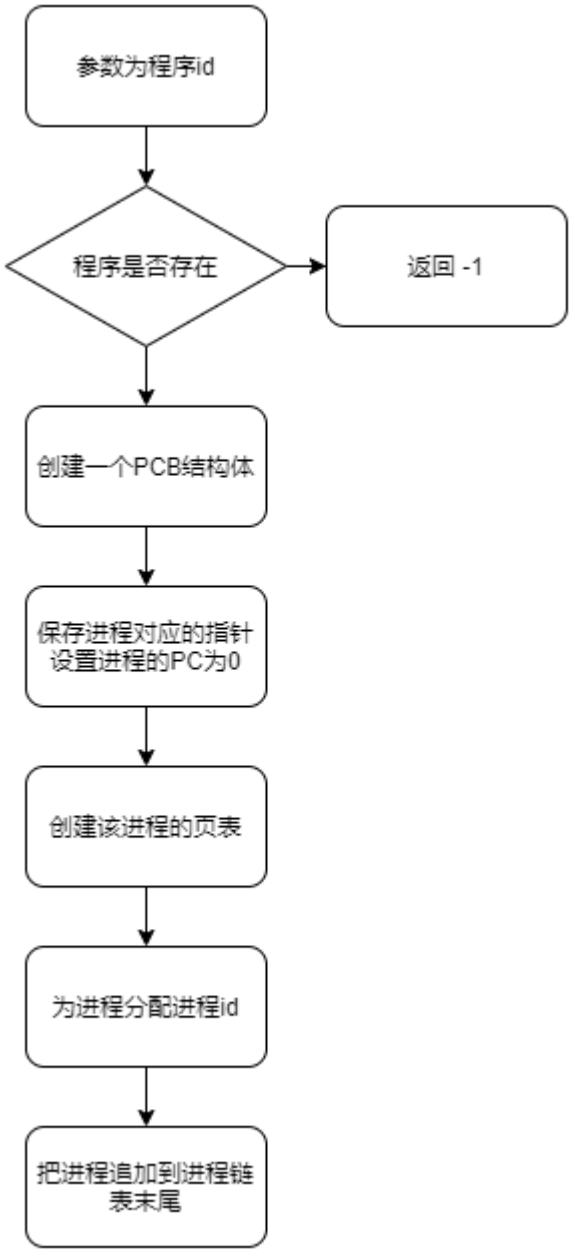
主函数流程：



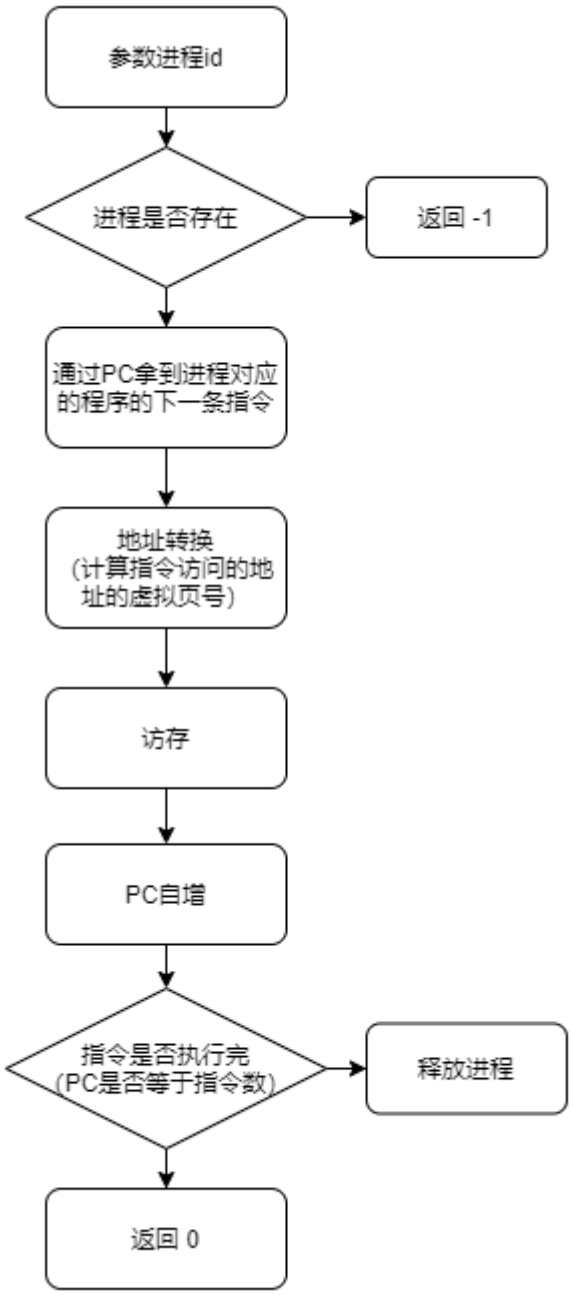
创建程序流程：



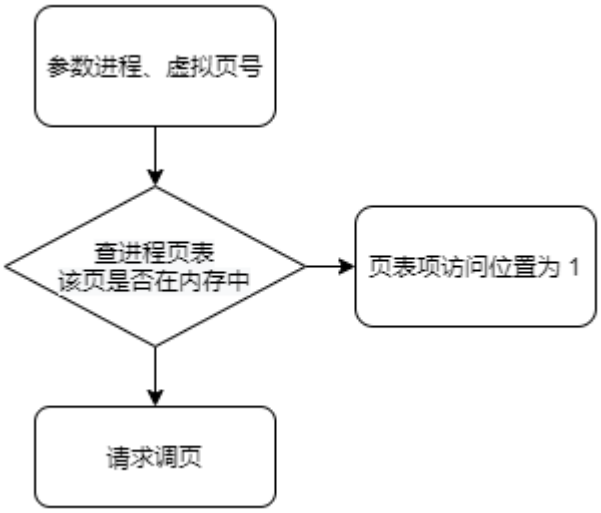
创建进程流程：


参数为程序id
程序是否存在
返回 -1
创建一个PCB结构体
保存进程对应的指针
设置进程的PC为0
创建该进程的页表
为进程分配进程id
把进程追加到进程链
表末尾

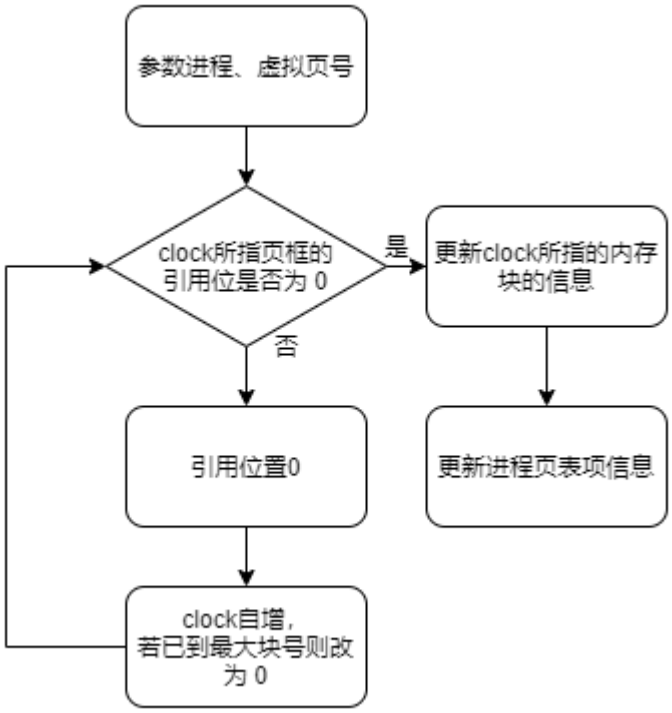
执行进程流程：


参数进程id
进程是否存在
返回 -1
通过PC拿到进程对应的程序的下一条指令
地址转换（计算指令访问的地址的虚拟页号）
访存
PC自增
指令是否执行完（PC是否等于指令数）
释放进程
返回 0

访存流程：


参数进程、虚拟页号
查进程页表
该页是否在内存中
页表项访问位置为 1
请求调页

请求调页流程：


参数进程、虚拟页号
clock所指页框的
引用位是否为 0
是
更新clock所指的内存
块的信息
否
引用位置0
更新进程页表项信息
clock自增，
若已到最大块号则改
为 0

5. 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;
```

6. 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。

6.1 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(i
nt));
    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;
}
```

6.2 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++;
    }
}
```

6.3 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, i
nt 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 col
or) :
    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(bl
ock_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(""));
}
```

7. 程序运行的主要界面和结果截图

7.1 控制台版本

编译：

```
~/code/cpp/virtual-paging-memory-simulation master !1 ❯ cd build

~/c/cp/virtual-paging-memory-simulation/build master !1 ❯ make
gcc -c ../src/main.c
gcc -c ../src/util/util.c
gcc -c ../src/memory/memory.c
ar -rcs libmemory.a memory.o
rm memory.o
gcc -o main main.o util.o -static libmemory.a
rm *.o

~/c/cpp/virtual-paging-memory-simulation/build master ❯ ls
    Makefile    libmemory.a    main

~/c/cpp/virtual-paging-memory-simulation/build master ❯
```

初始化：

```
~/c/cpp/virtual-paging-memory-simulation/build master ❯ ./main
setup PAGE SIZE: 4
setup PHYSICAL MEMORY PAGE size: 4
setup VIRTUAL MEMORY PAGE size: 8

-----------------
1 create_program
2 create_process
3 execute_process
4 quit
```

创建程序：

```
1

what is the execution sequence (e.g. 0 4 8 12 16 12 4 20)
the number represents the memory address that instruction uses
numbers should range from 0 to VIRTUAL_MEM_PAGE*PAGE_SIZE, i.e. 32
0 4 8 12 16 12 4 20

program 0 created

-----------------
1 create_program
2 create_process
3 execute_process
4 quit
```

创建进程：

```
2

choose program id to create process from
0

process 0 created
Process 0 instructions: 0 4 8 12 16 12 4 20
PC: 0, PC instruction: 0
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------

-----------------
1 create_program
2 create_process
3 execute_process
4 quit
```

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：

```
3

which process do you want execute
0

Process 0 instructions: 0 4 8 12 16 12 4 20
PC: 1, PC instruction: 4
--------------------------------
| process 0        page 0       |
--------------------------------
|                               |
--------------------------------
|                               |
--------------------------------
|                               |
--------------------------------
```

再次用 0 号程序创建一个进程：

```
2

choose program id to create process from
0

process 1 created
Process 0 instructions: 0 4 8 12 16 12 4 20
PC: 1, PC instruction: 4
Process 1 instructions: 0 4 8 12 16 12 4 20
PC: 0, PC instruction: 0
--------------------------------
| process 0        page 0       |
--------------------------------
|                               |
--------------------------------
|                               |
--------------------------------
|                               |
--------------------------------
```

执行新的进程若干次：

```
3

which process do you want execute
1

Process 0 instructions: 0 4 8 12 16 12 4 20
PC: 1, PC instruction: 4
Process 1 instructions: 0 4 8 12 16 12 4 20
PC: 3, PC instruction: 12
--------------------------------
| process 0        page 0       |
--------------------------------
| process 1        page 0       |
--------------------------------
| process 1        page 1       |
--------------------------------
| process 1        page 2       |
--------------------------------
```

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：

```
3

which process do you want execute
0

Process 0 instructions: 0 4 8 12 16 12 4 20
PC: 2, PC instruction: 8
Process 1 instructions: 0 4 8 12 16 12 4 20
PC: 3, PC instruction: 12
--------------------------------
| process 0        page 1       |
--------------------------------
| process 1        page 0       |
--------------------------------
| process 1        page 1       |
--------------------------------
| process 1        page 2       |
--------------------------------
```

继续执行进程 0，直至进程 0 退出：

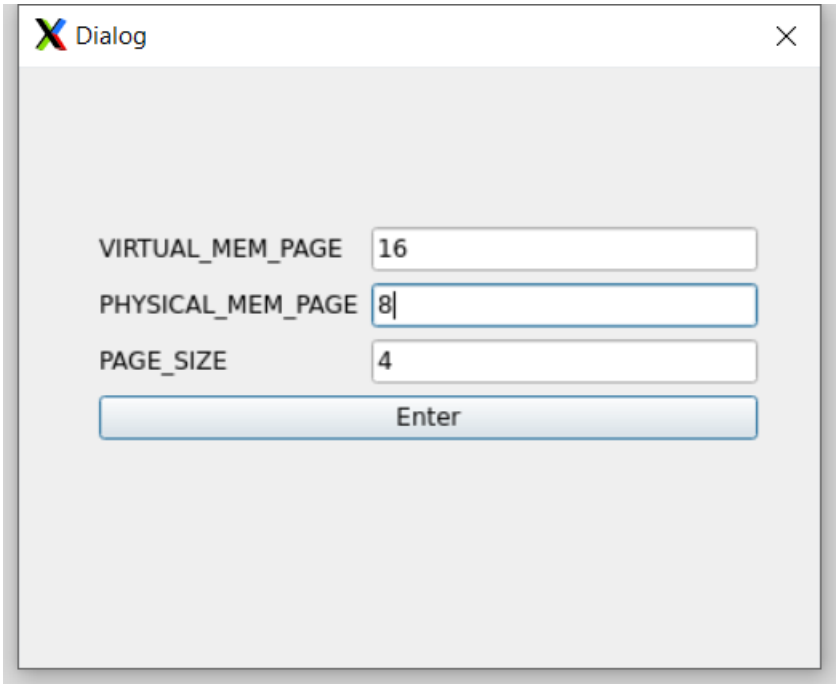
```
3

which process do you want execute
0

Process 1 instructions: 0 4 8 12 16 12 4 20
PC: 3, PC instruction: 12
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
```

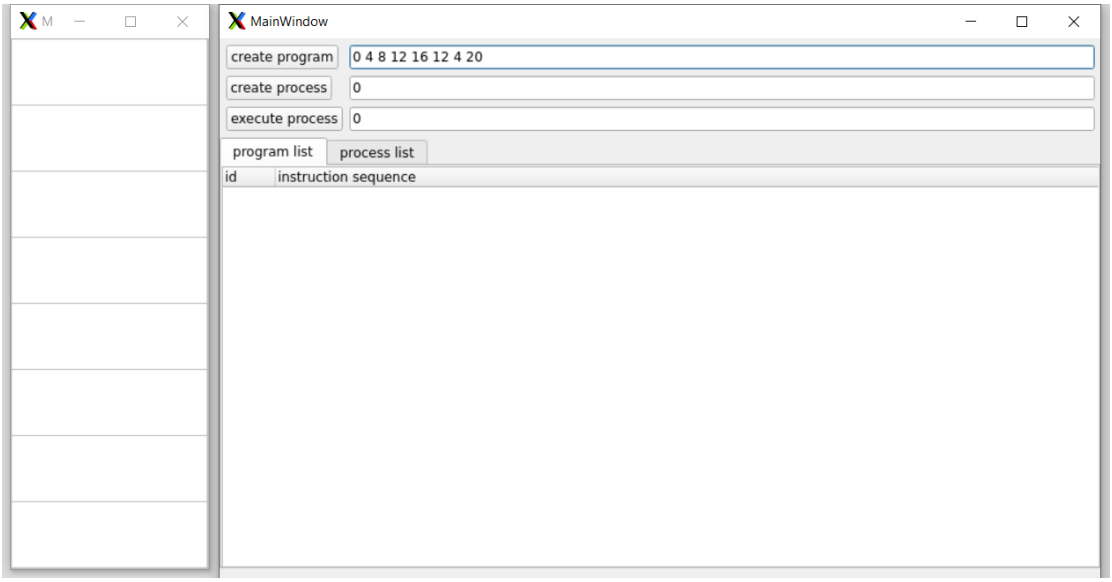
继续执行进程 1，直至进程 1 退出：

```
which process do you want execute
1

Process 1 instructions: 0 4 8 12 16 12 4 20
PC: 7, PC instruction: 20
--------------------------------
| process 1        page 3      |
--------------------------------
| process 1        page 4      |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------


which process do you want execute
1

--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
|                              |
--------------------------------
```

7.2 GUI 版本

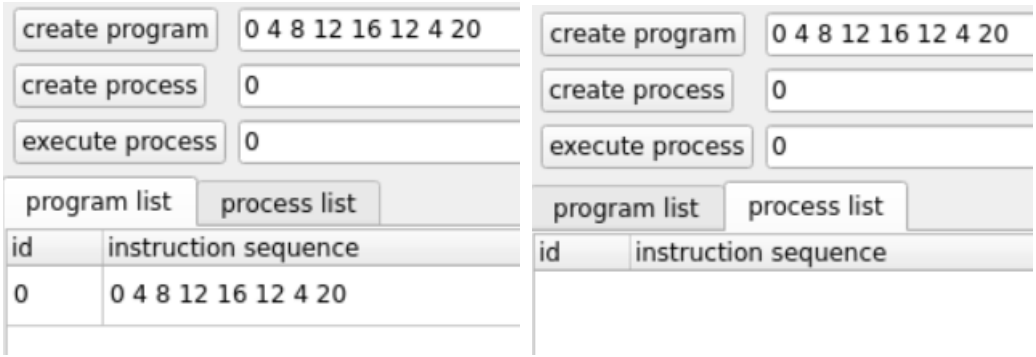
初始化：



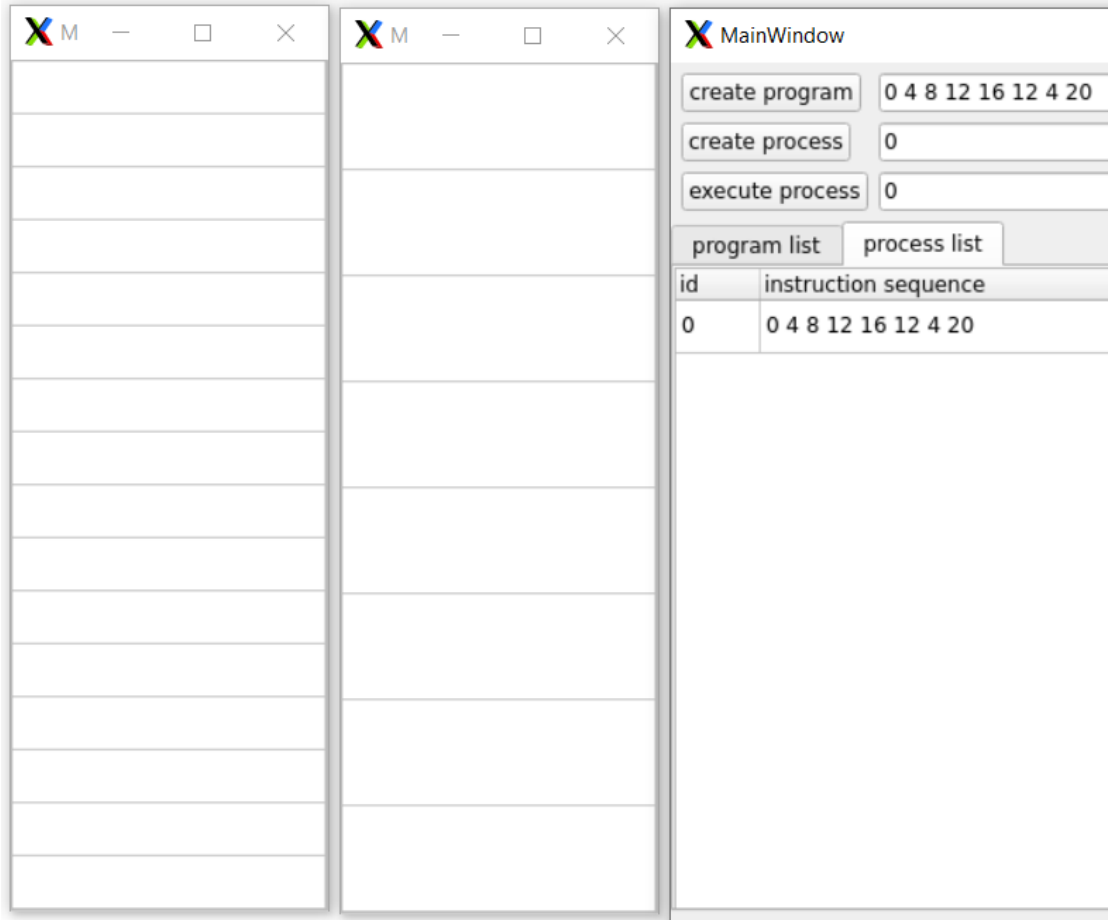


左侧的方格即为物理内存的可视化，可以看到总数为 8 块。

创建程序：

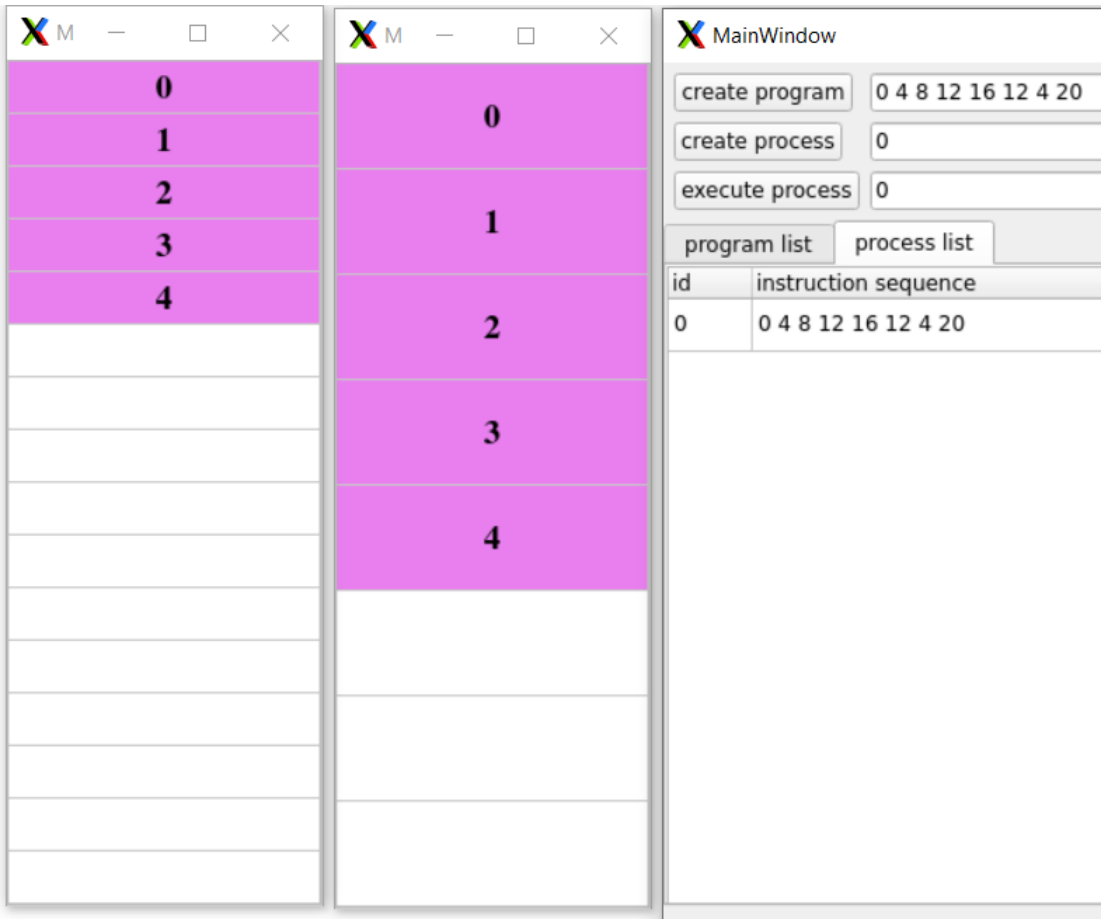


创建进程：



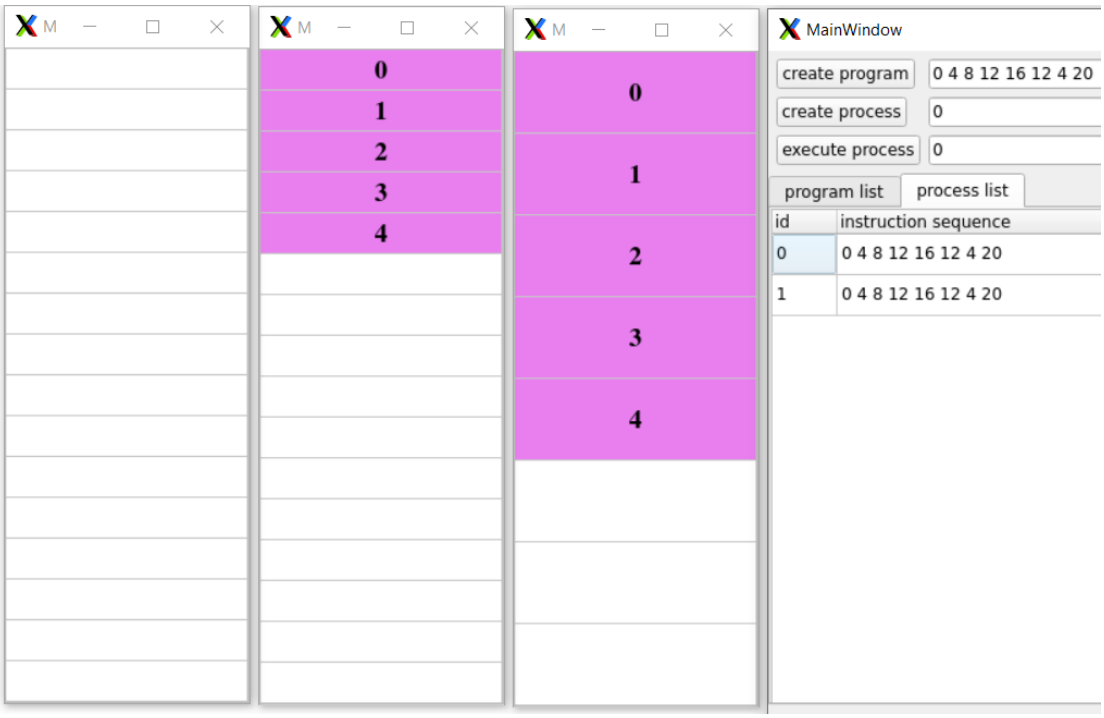
最左侧的方格为 0 号进程看到的虚拟内存的可视化，可以看到总数为 16 个页面。

执行进程 0 若干次：

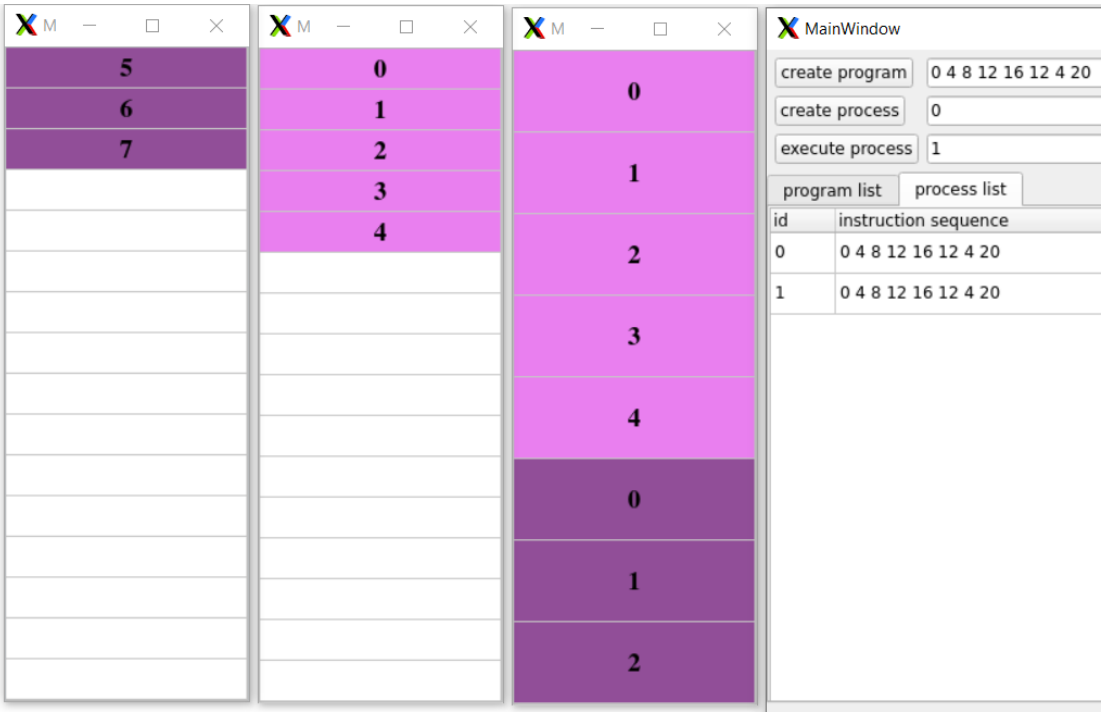


虚拟内存页上的数字为该页在内存中的页框号，物理内存页框上的数字为该页框对应的虚拟页号（用颜色来表示是哪个进程的页，这里为进程 0 随机分配的颜色为浅紫色）。

创建一个新的进程：

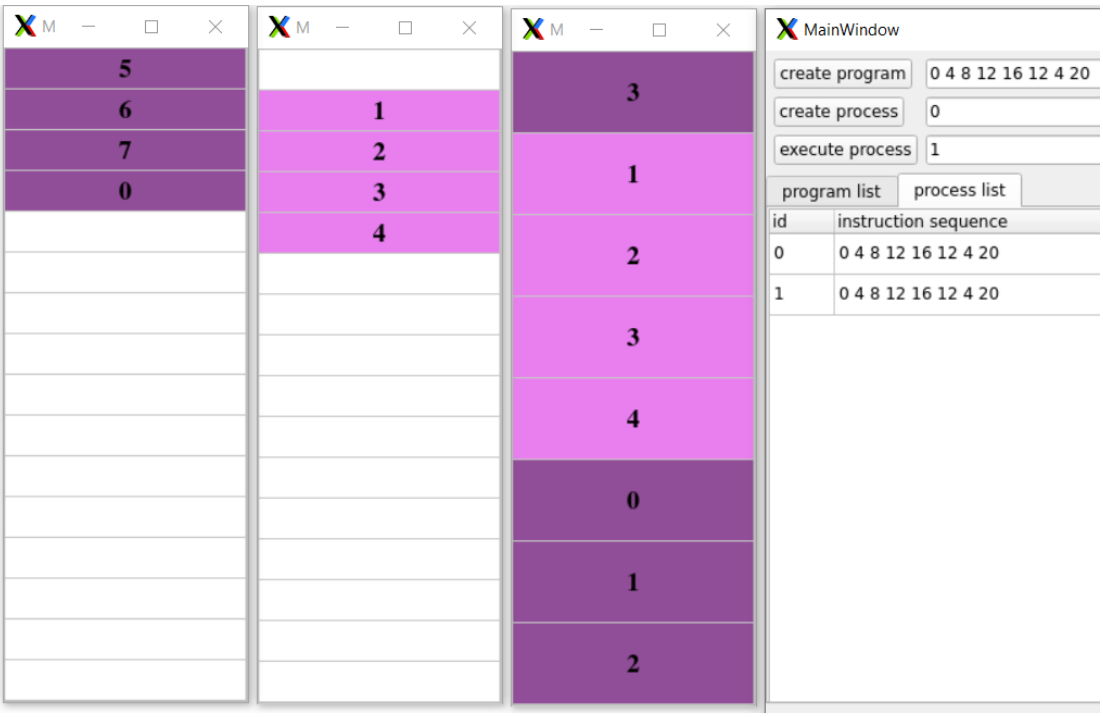


执行新进程（1 号进程）若干次：

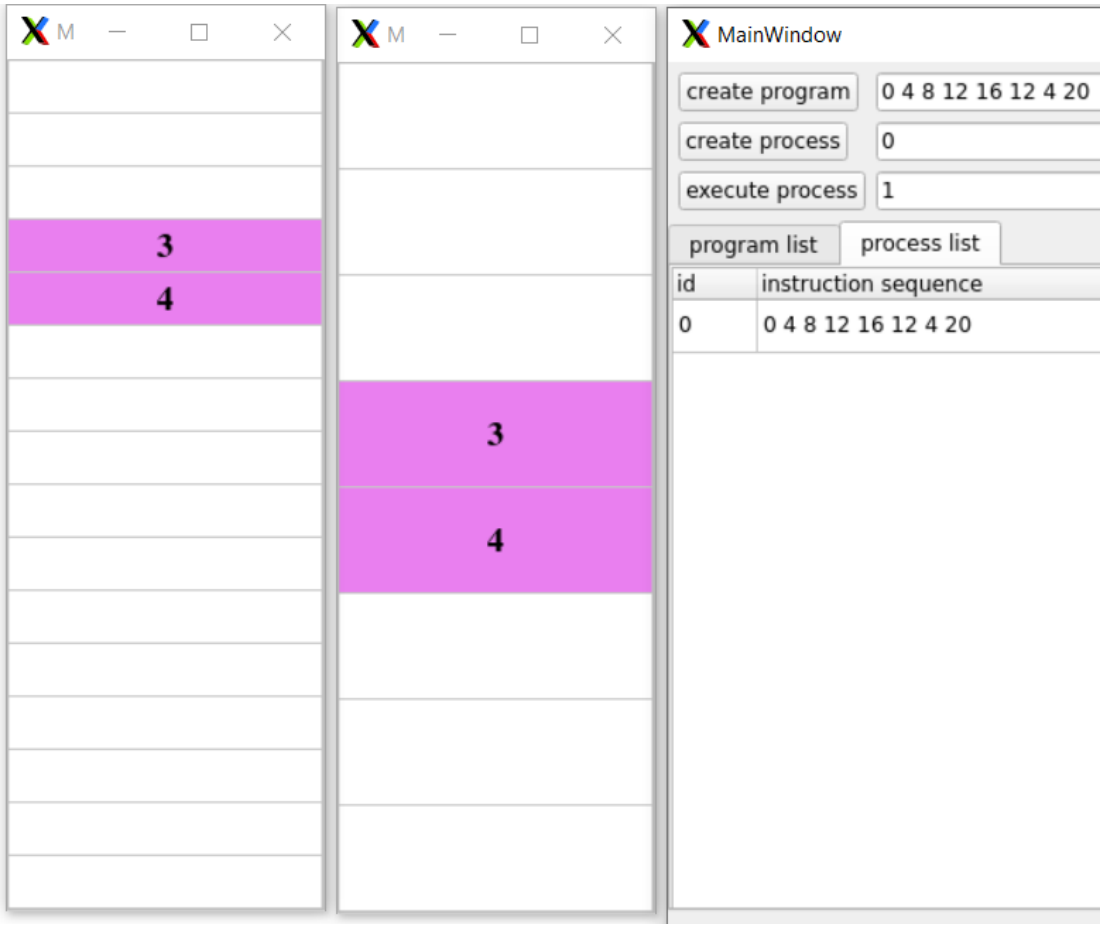


这里为进程 1 随机分配的颜色为深紫色。

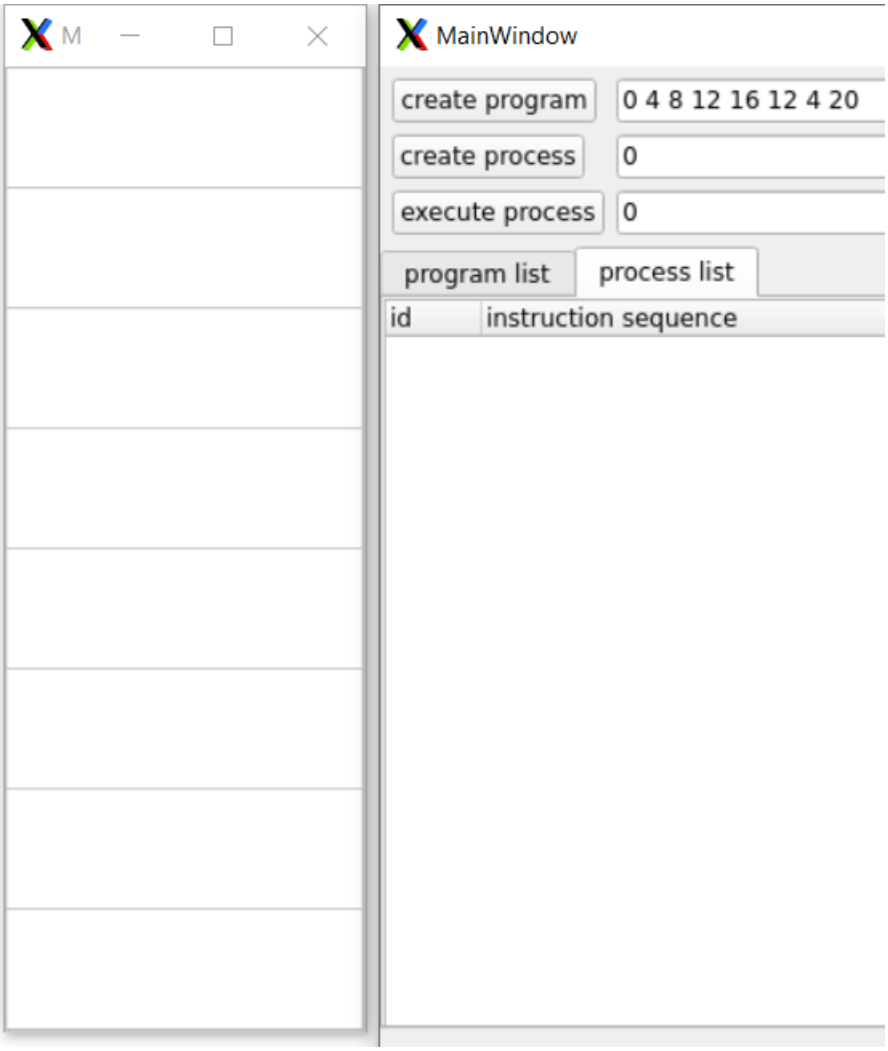
再次执行进程 1，将发生页面换入换出：



继续执行进程 1，直至退出：



继续执行进程 0，直至退出：



8. 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