



“操作系统课程设计”总结报告

学号	2022*****
姓名	***
学院	计算机与大数据学院
年级	2022 级
专业	计算机科学与技术
报告日期	2024 年 11 月 18 日
成绩	

目录

1	进程控制	1
1.1	目的	1
1.2	内容	1
1.3	数据结构	1
1.3.1	进程控制块 (PCB)	2
1.3.2	进程管理器	2
1.4	算法设计及流程图	2
1.4.1	进程控制块 (PCB) 的定义	2
1.4.2	内存块的管理	3
1.4.3	进程的创建与状态转换	4
1.4.4	内存的释放与碎片整理	4
1.4.5	系统流程图	5
1.5	运行截图	6
1.6	小结	9
2	分页式存储管理	9
2.1	目的	9
2.2	内容	9
2.3	数据结构	10
2.3.1	进程控制块 (PCB)	10
2.3.2	页表 PageTable	11
2.3.3	位示图及其基本单位 Bit	12
2.4	算法设计及流程图	12
2.4.1	FIFO 页面置换算法	13
2.4.2	LRU 页面置换算法	15
2.5	运行截图	17
2.6	小结	20
3	文件与磁盘管理	20
3.1	目的	20
3.2	内容	20
3.3	数据结构	21
3.3.1	文件管理系统及 FAT 表	21
3.3.2	文件控制块 (FCB)	21
3.4	算法设计及流程图	22

3.4.1	cd 命令	22
3.4.2	rd 命令	22
3.5	运行截图	25
3.6	小结	27
4	进程调度	27
4.1	目的	27
4.2	内容	28
4.3	数据结构	28
4.4	算法设计及流程图	29
4.4.1	FCFS 调度算法	29
4.4.2	SJF 调度算法	31
4.4.3	轮转调度算法	31
4.4.4	优先级抢占调度算法	32
4.4.5	高响应比优先调度算法	34
4.4.6	多级反馈队列调度算法	35
4.4.7	银行家算法	36
4.5	运行截图	37
4.6	小结	41
5	课程总结	41
A	代码运行环境	43
B	源代码 GitHub 仓库	43

1 进程控制

1.1 目的

本实验主要通过实现进程控制块 (PCB)，深入理解操作系统的进程管理。学生将学习进程的生命周期，掌握进程状态的转换，并通过操作 PCB 来实践内存管理，包括动态内存分配和回收。此外，实验也旨在提升学生的系统设计和问题解决技能，为更深入的操作系统学习打下基础。

1.2 内容

本实验围绕操作系统的进程控制，有以下 5 个实验任务：

1. **定义管理进程的数据结构 PCB**：包含进程 ID、进程状态（新建、就绪、执行、阻塞、完成）、所在队列指针、分配的物理内存区域（基址和界限）、上下文信息（PC 值）。
2. **创建进程**时，需要为其创建 PCB 并分配空闲内存空间，对 PCB 进行初始化，并加入就绪队列。
3. **模拟触发进程状态转换的事件**：采用键盘控制方法、或随机化方法或读文件来模拟触发进程状态切换的事件（例如系统调用，包括创建进程 (c)、结束进程 (e)、进程阻塞 (b)；中断，包括唤醒进程 (w)、时间片到 (t) 等事件），实现对应的控制程序。
4. 根据当前发生的事件执行**进程的状态切换**，并显示出当前系统中的执行队列、就绪队列和阻塞队列。
5. 选做：完成**可变分区的分配与回收**，创建进程的同时申请一块连续的内存空间，在 PCB 中设置好基址和界限，结束进程同时回收分配的内存空间。分配采用最佳适应算法，碎片大小为 2 Kb，最后回收所有进程的空间，对空间分区的合并。可以查看进程所占的空间和系统空闲空间。

1.3 数据结构

模式实现操作系统的进程控制过程，需要构建两个数据结构：进程控制块 (PCB)、进程管理器。其中，PCB 主要存放每个进程的信息，包括名称、创建时间、状态、所需内存等等；进程管理器用于管理多个进程，有就绪队列、阻塞队列，并记录正在运行的进程，负责控制进程、修改进程信息。

1.3.1 进程控制块 (PCB)

类定义：PCB 类用于管理进程信息。

```
1 class PCB:
2     def __init__(self, name: str, memory_size: int, pc:
3         int = None):
4         self.pid =
5             datetime.datetime.now().strftime("%Y%m%d%H%M%S")
6         self.name = name
7         self.next = None
8         self.memory_size = memory_size # 进程所占空间
9         self.state = '新建' # 新建, 就绪, 执行, 阻塞, 完成
10        self.pc = pc
```

1.3.2 进程管理器

类定义：ProcessManager 类负责管理所有进程和内存的分配与回收。

```
1 class ProcessManager:
2     def __init__(self):
3         self.ready_head = None # 就绪队列
4         self.blocked_head = None # 阻塞队列
5         self.finished_head = None # 结束队列
6         self.running = None # 运行进程
7         self.pc = 0 # 指令计数器
```

1.4 算法设计及流程图

1.4.1 进程控制块 (PCB) 的定义

在我们的进程管理系统中，每个进程由一个进程控制块 (PCB) 表示，该控制块存储了进程的关键信息，包括进程 ID、进程名称、进程状态（如新建、就绪、执行、阻塞、完成）、内存基址和大小、以及程序计数器 (PC 值)。这些信息为操作系统提供了必要的信息，以便有效地管理各个进程。其中，PC 值从 1 开始计数，每成功创建一个进程 PC 加 1；为了给每个进程赋予唯一的 PID，本系统利用

Python 的 *datetime* 模块，将创建时间作为每个进程的 PID，并限制了创建频率，确保 PID 唯一。

1.4.2 内存块的管理

系统中的内存通过 *MemoryBlock* 类进行管理，每个内存块包含基址、大小和状态（空闲或占用）。为了有效管理内存，我们实现了一个简单的内存分配策略，使用**最佳适应算法（Best Fit）**来分配内存块。这种方法试图找到尽可能小的足够大空闲内存块，以减少内存碎片，算法伪代码展示如下。

Algorithm 1 Best Fit Memory Allocation Algorithm

Input: *required_size*, *memory_blocks* (List of memory blocks each with properties: size, state)

Output: Base address of the allocated block or NULL

```

1: min_diff  $\leftarrow \infty$ 
2: best_fit  $\leftarrow \text{NULL}$ 
3: for each block in memory_blocks do
4:   if block.state = 'FREE' AND block.size  $\geq$  required_size then
5:     diff  $\leftarrow$  block.size - required_size
6:     if diff < min_diff then
7:       min_diff  $\leftarrow$  diff
8:       best_fit  $\leftarrow$  block
9:     end if
10:  end if
11: end for
12: if best_fit  $\neq$  NULL then
13:   if best_fit.size > required_size + MIN_FRAGMENT_SIZE then
14:     new_size  $\leftarrow$  best_fit.size - required_size
15:     best_fit.size  $\leftarrow$  required_size
16:     Split best_fit to create a new block with size new_size
17:   end if
18:   best_fit.state  $\leftarrow$  'OCCUPIED'
19:   return best_fit.base
20: else
21:   return NULL
22: end if

```

1.4.3 进程的创建与状态转换

创建进程时，系统首先检查是否已存在具有相同名称的进程。如果不存在，系统将尝试为新进程分配内存。成功分配内存后，新的 PCB 被初始化并添加到就绪队列中。这些步骤确保了进程能够被系统有效跟踪并在资源允许的情况下执行。

本系统模拟了进程状态的动态转换，包括通过用户输入事件来触发状态变化（例如，进程从执行转为阻塞或从阻塞状态唤醒）。系统中实现了几个关键的事件处理函数，如执行进程、阻塞进程、唤醒进程和结束进程，这些功能模拟了操作系统在多任务环境中的行为。

1.4.4 内存的释放与碎片整理

当一个进程终止时，它占用的内存被释放并标记为空闲。系统还尝试合并相邻的空闲内存块，以减少碎片并最大化可用内存，本算法的伪代码展示如下。

Algorithm 2 Memory Release and Fragmentation Management Algorithm

Input: *base* (Base address of the memory block to be freed)

Output: None

```

1: current  $\leftarrow$  memory_head
2: previous  $\leftarrow$  NULL
3: while current  $\neq$  NULL do
4:   if current.base = base then
5:     current.state  $\leftarrow$  'FREE'
6:     if previous  $\neq$  NULL AND previous.state = 'FREE' then
7:       previous.size  $\leftarrow$  previous.size + current.size
8:       previous.next  $\leftarrow$  current.next
9:       current  $\leftarrow$  previous
10:    end if
11:    if current.next  $\neq$  NULL AND current.next.state = 'FREE' then
12:      current.size  $\leftarrow$  current.size + current.next.size
13:      current.next  $\leftarrow$  current.next.next
14:    end if
15:    break
16:  end if
17:  previous  $\leftarrow$  current
18:  current  $\leftarrow$  current.next
19: end while

```

1.4.5 系统流程图

系统执行总流程图如下图所示。

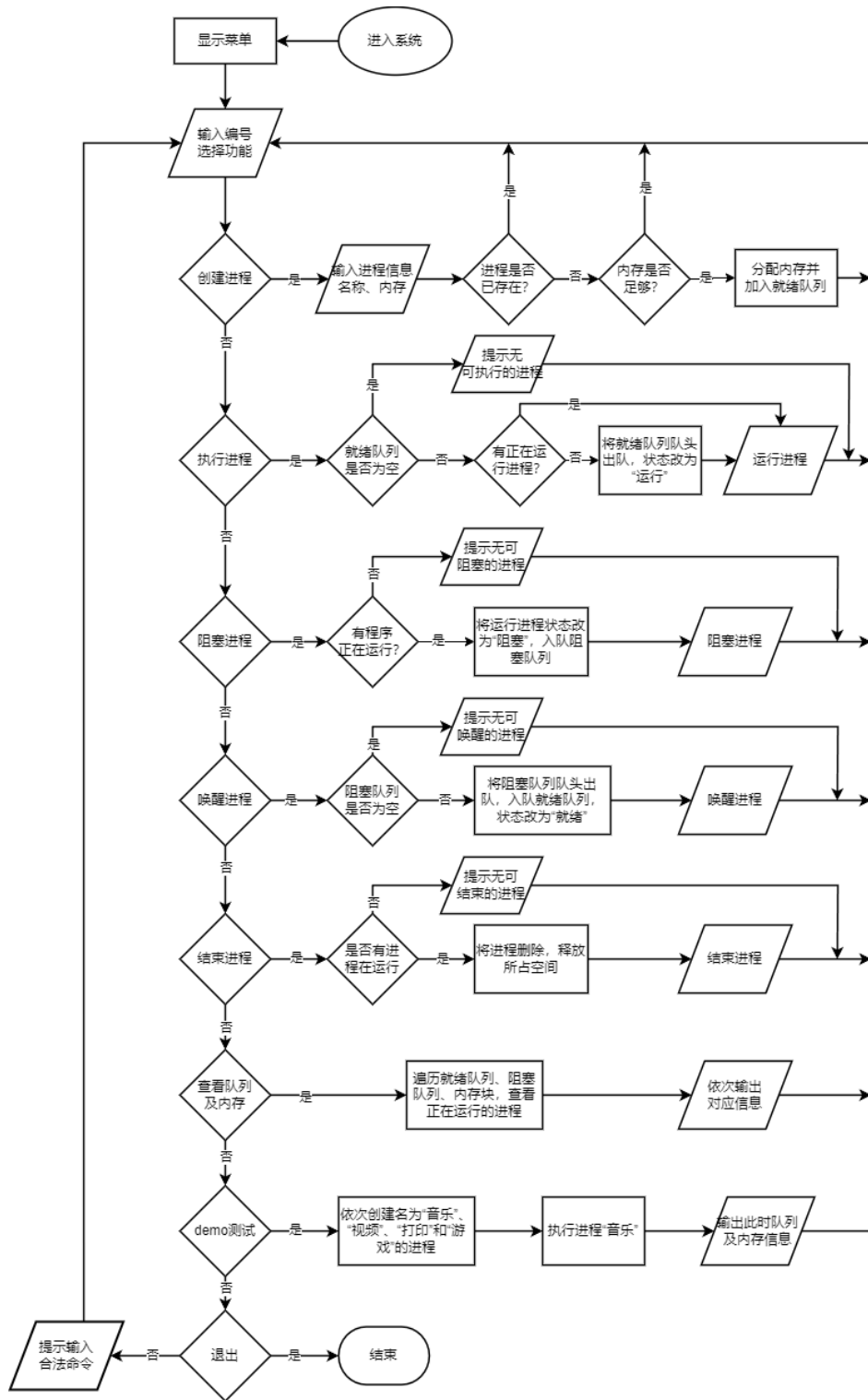


图 1-1: 进程管理系统流程图

1.5 运行截图



图 1-2: 进程管理系统的菜单

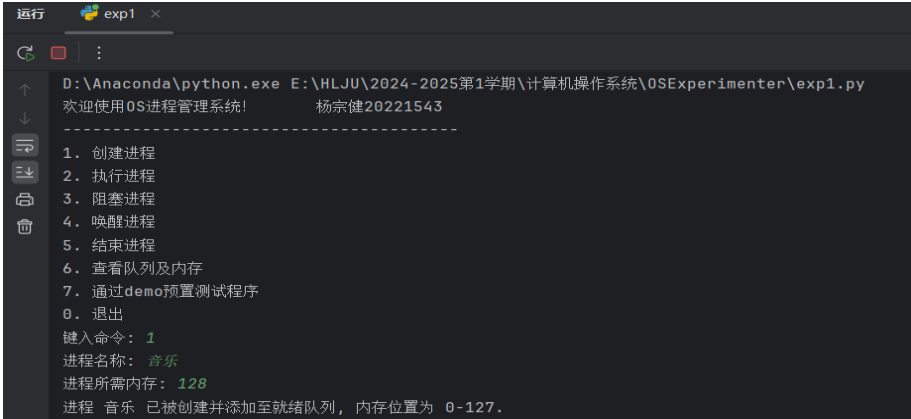


图 1-3: 创建一个名为“音乐”的进程，分配 128 字节空间



图 1-4: 执行当前就绪队列中第一个进程，将其状态改为“运行”

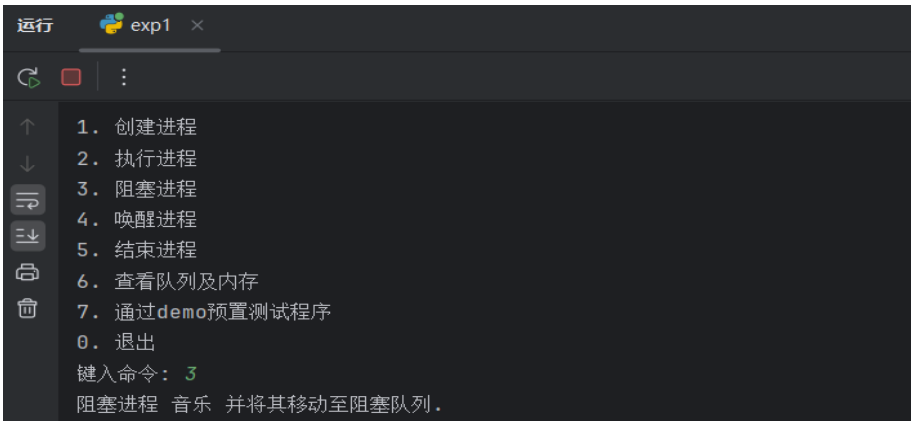


图 1-5: 将正在运行的进程“音乐”阻塞，放入阻塞队列



图 1-6: 将在阻塞队列中的进程“音乐”唤醒，放入就绪队列中等待执行



图 1-7: 将正在运行的进程“音乐”结束



图 1-8: 为方便测试编写的预置 demo。依次建进程音乐、视频、打印、游戏，并运行第一个创建的进程“音乐”

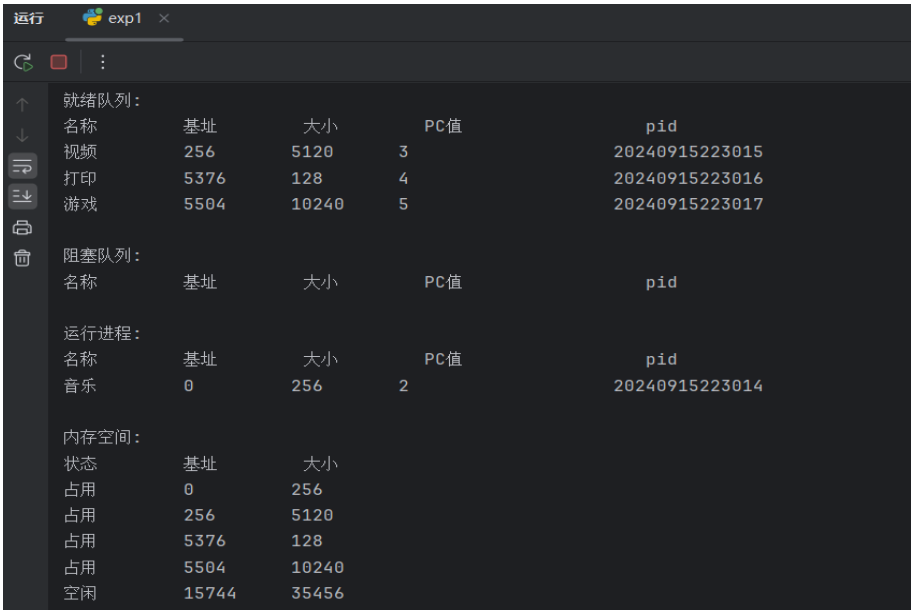


图 1-9: 展示各个状态队列中进程的信息（包括基址、占用内存、pid、PC 值）和内存分配情况

1.6 小结

本实验成功地实现了进程控制块 (PCB) 的设计与操作, 深入探讨了操作系统中的进程管理机制。通过定义和操作 PCB, 我不仅理解了进程的生命周期, 还具体体验了进程状态的转换, 包括进程的创建、执行、阻塞、唤醒和终止等关键过程。

在实验中, 通过动态地分配和回收内存, 我亲自实践了内存管理的基本技术。特别是在选做任务中, 使用最佳适应算法进行内存分配, 成功地模拟了内存管理的高级功能, 如内存分区的合并和碎片管理, 这不仅增强了我对理论知识的理解, 也提升了处理实际操作系统问题的实践能力。

此外, 通过模拟进程状态切换的不同触发事件, 如系统调用和中断, 使我能够观察和分析进程在各种操作系统管理策略下的行为。这种实践经验对于我未来在系统设计和问题解决中的应用是极其宝贵的。

通过本实验, 我不仅加深了对操作系统核心概念的理解, 还熟悉了操作系统中进程管理的实际应用, 为今后深入学习操作系统的其他高级主题打下了坚实的基础。实验结果表明, 所有实验任务都已成功完成, 达到了实验的预期目的。

2 分页式存储管理

2.1 目的

1. 熟练掌握分页式管理基本原理, 并在实验过程中体现内存空间的分配与回收、地址转换过程。
2. 掌握利用“位示图”管理内存与置换空间的分配与回收。
3. 掌握基本的位运算。
4. 掌握请求分页式存储管理基本原理, 并在实验过程中体现内存与置换空间的分配与回收、地址转换以及缺页处理过程。

2.2 内容

在实验 1 基础上实现分页式存储管理内存分配和地址转换过程。进一步实现请求分页式存储管理过程, 包括内存和置换空间管理、地址转换以及缺页处理, 能够体现 FIFO 和 LRU 算法思想。具体内容如下:

1. **位示图构建**: 建立一个位示图数据结构, 用来模拟内存的使用情况。
2. **PCB 扩展**: 在实验 1 基础上扩充 PCB, 增加进程大小和页表字段。
3. **地址转换**: 输入当前执行进程的逻辑地址, 并将其转换为对应的物理地址。

4. **改进内存释放**：进程退出时，根据其页表内容将位示图中对应位置的“1”回填为“0”。
5. **页表扩展**：扩充页表，使其支持请求和置换功能，增加存在位等信息。
6. **实现页面置换**：在地址转换过程中遇到缺页时，分别采用 FIFO 或 LRU 置换算法进行页面置换。

2.3 数据结构

为了符合实验要求,本文在实验 1 的基础上添加了页表 (Pagetable)、位示图基本单位 (Bit) 两个数据结构,对于每个进程 PCB 都添加了内存所占块数和表示内存使用情况的页表。此外,为了实现 LRU 算法,本文设计了一种特殊的栈结构,兼顾集合和循环队列的特性;更换了内存表达形式,实验 2 也删除了实验 1 定义的用于表现进程内存使用情况的数据结构 MemoryBlock。

2.3.1 进程控制块 (PCB)

相比实验 1，由于实验 2 要求采用位示图和页表来表示内存占用情况，故每个进程单位 PCB 都需要添加表示该进程内存使用情况的属性 *page_table*。更改的 PCB 定义见下面的代码块：

```

1 class PCB:
2     def __init__(self, name: str, memory_size: int, pc:
        int = None):
3         self.pid =
            datetime.datetime.now().strftime("%Y%m%d%H%M%S")
4         self.name = name
5         self.next = None
6         self.memory_size = memory_size # 进程所占空间
7         self.state = '新建' # 新建，就绪，执行，阻塞，完成
8         self.pc = pc
9         self.block_num = ceil(self.memory_size /
            BLOCK_SIZE) # 进程所占内存块数
10        self.page_table = [PageTable(no) for no in
            range(self.block_num)] # 定义并初始化页表

```

2.3.2 页表 PageTable

在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，以保证能够正确运行，即能在内存中找到每个页面所对应的物理块。页表是存储块的基本单位，每个进程按照所占存储块数分配页表。在进程地址空间内的所有页，依次在页表中有一页表项，其中记录了相应页在内存中对应的物理块号，如图2-1所示。

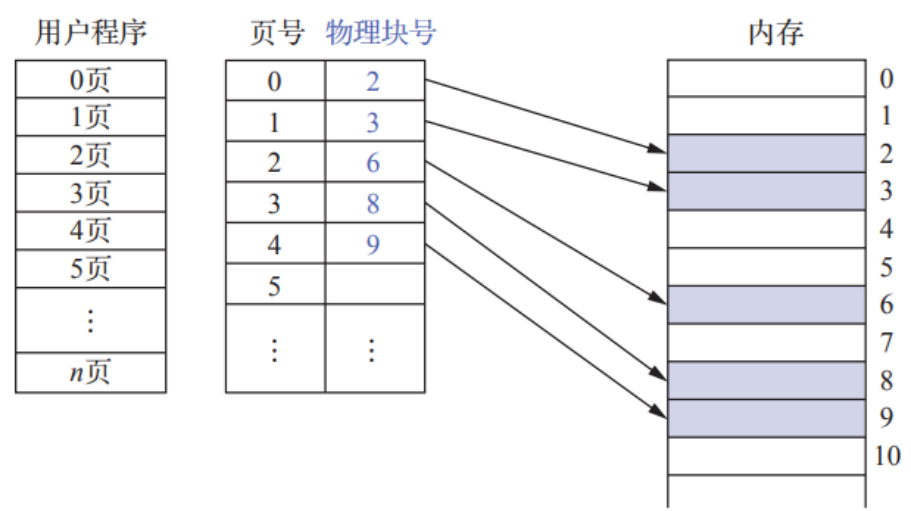


图 2-1: 页表的作用

根据实验要求，分配给每个进程的内存块数至多为 3，所以进程占用的外存块采用 *address* 属性记录；状态位 *state* 表示该页表是否被分配空间（是为 1，否为 0）；访问字段 *visit_time*、修改位 *dirty* 在本实验中没有实际作用。

```
1 class PageTable:
2     def __init__(self, no: int, p=0, m=False):
3         self.no = no # 页号
4         self.block = None # 物理块号
5         self.state = p # 状态位
6         self.visit_time = 0 # 访问字段
7         self.dirty = m # 修改位
8         self.address = None #
          外存地址，若该页不在内存中，则为置换区位置(块号)
```

2.3.3 位示图及其基本单位 Bit

位示图是指利用二进制的一位来表示磁盘中一个盘块的使用情况。当其值为“0”时，表示对应的盘块空闲；当其值为“1”时，表示对应的盘块已被分配。磁盘块上的所有盘块都有一个二进制位与之对应，这样，由所有盘块所对应的位构成的一个集合称为位示图。通常可用 $m \times n$ 个位来构成位示图，并使 $m \times n$ 等于磁盘块的总块数，如图2-2所示。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	0
2	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1
3	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0
...																
16																

图 2-2: 位示图

本实验中，内存位示图由 8 个字节组成，下面介绍的数据结构 Bit 实际上是“字节”。Bit 只有一个属性，就是该字节的值，用二进制表示。字节的每一位代表一个存储块，其值表示该块的使用情况，其值的改变采用位运算的方式进行，如代码块中的 use 和 free 函数。

```

1 class Bit:
2     def __init__(self):
3         self.val = random.randint(0, 255) # 随机数初始化
4
5     def free(self, idx: int) -> None:
6         """释放内存，将1改为0"""
7         self.val &= ~(1 << idx)
8
9     def use(self, idx: int) -> None:
10        """使用内存，将0改为1"""
11        self.val |= 1 << idx

```

2.4 算法设计及流程图

虽然引入了位示图管理空间，但是空间的分配、释放等流程的算法不算复杂，故本文不对此过多介绍。下面主要介绍本实验的侧重点：**置换算法**。

2.4.1 FIFO 页面置换算法

先进先出 (first in first out, FIFO) 页面置换算法是最早出现的页面置换算法。该算法总是会淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只要把进程已调入内存的页面按先后次序链接成一个队列，并设置一个指针（称为替换指针），使它总是指向最老的页面。但该算法与进程实际运行的规律不相适应，因为在进程中，有些页面（如含有全局变量、常用函数、例程等的页面）会经常被访问，而 FIFO 页面置换算法并不能保证这些页面不被淘汰。算法伪代码和流程图如下所示：

Algorithm 3 FIFO Page Replacement Algorithm

Input: *page_table* (List of pages with properties: block, address), *addr_seq* (Sequence of logical addresses), *memory_size* (Number of pages that can be in memory simultaneously)

Output: Page fault rate, number of page swaps

```

1: Initialize an empty queue deque with maximum size memory_size
2: page_faults  $\leftarrow$  0, hits  $\leftarrow$  0
3: for each addr in addr_seq do
4:   page_no  $\leftarrow$  addr  $\div$  PAGE_SIZE
5:   offset  $\leftarrow$  addr mod PAGE_SIZE
6:   if page_no exists in deque then
7:     hits  $\leftarrow$  hits + 1
8:   else
9:     page_faults  $\leftarrow$  page_faults + 1
10:    if deque is full then
11:      Remove the oldest page from deque
12:    end if
13:    Add page_no to deque
14:    Perform page swap between memory and storage
15:  end if
16: end for
17: fault_rate  $\leftarrow$   $\frac{\text{page\_faults}}{\text{page\_faults} + \text{hits}}$ 
18: return fault_rate, page_faults

```

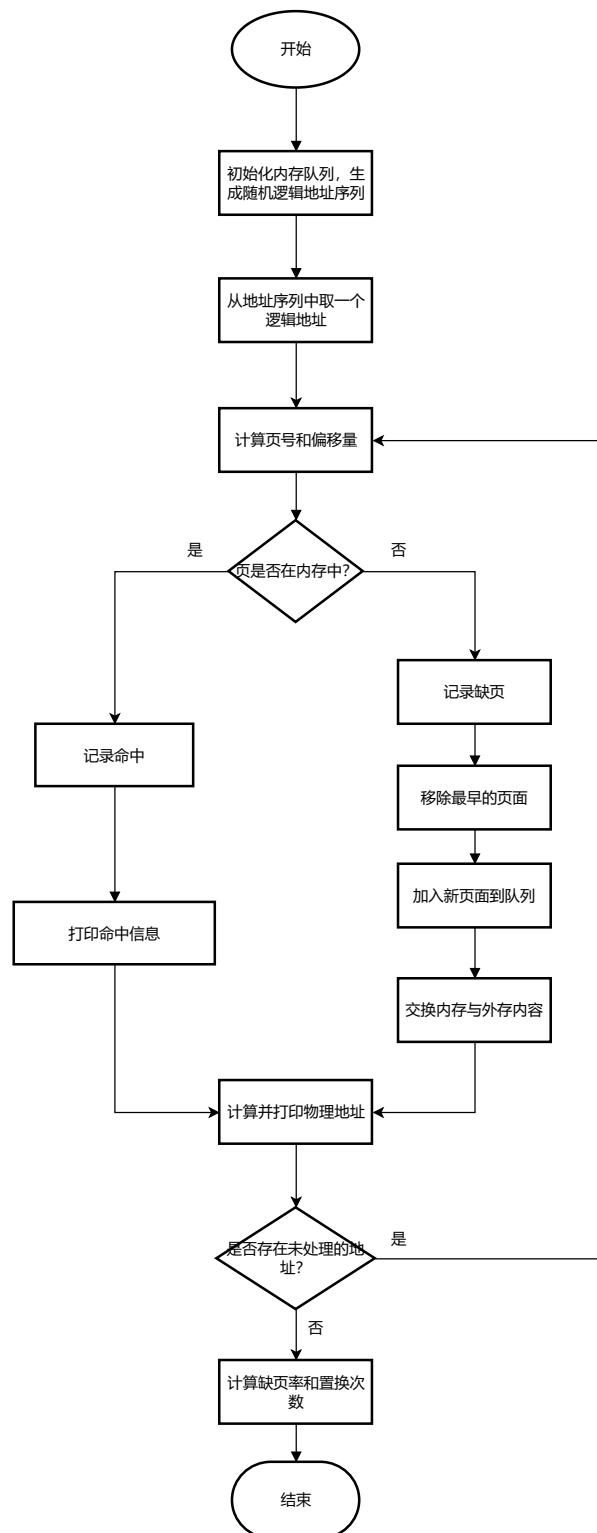


图 2-3: FIFO 算法流程图

2.4.2 LRU 页面置换算法

LRU (Least Recently Used, 最近最少使用) 算法是一种常见的页面置换算法, 它的核心思想是: 在页面访问的过程中, 总是替换掉最久没有被访问的页面。当需要加载新页面时, LRU 会选择那些最久没有使用的页面进行替换。使用一个缓存来跟踪页面的访问顺序, 每当一个页面被访问时, 它就被标记为最近使用。为了实现这一点, LRU 通常使用链表或哈希表结合双向链表来追踪页面的使用顺序, 或者借助栈来维护页面的访问顺序。其算法伪代码和流程图如下所示:

Algorithm 4 LRU Page Replacement Algorithm

Input: *page_table* (List of pages with properties: block, address), *addr_seq* (Sequence of logical addresses), *memory_size* (Number of pages that can be in memory simultaneously)

Output: Page fault rate, number of page swaps

```

1: Initialize an empty list cache to store pages in memory
2: page_faults  $\leftarrow$  0, hits  $\leftarrow$  0
3: for each addr in addr_seq do
4:   page_no  $\leftarrow$  addr  $\div$  PAGE_SIZE
5:   offset  $\leftarrow$  addr mod PAGE_SIZE
6:   if page_no exists in cache then
7:     hits  $\leftarrow$  hits + 1
8:     Move page_no to the front of cache (most recently used)
9:   else
10:    page_faults  $\leftarrow$  page_faults + 1
11:    if cache is full then
12:      Remove the least recently used page from cache
13:    end if
14:    Add page_no to the front of cache
15:    Perform page swap between memory and storage
16:  end if
17: end for
18: fault_rate  $\leftarrow$   $\frac{\text{page\_faults}}{\text{page\_faults} + \text{hits}}$ 
19: return fault_rate, page_faults

```

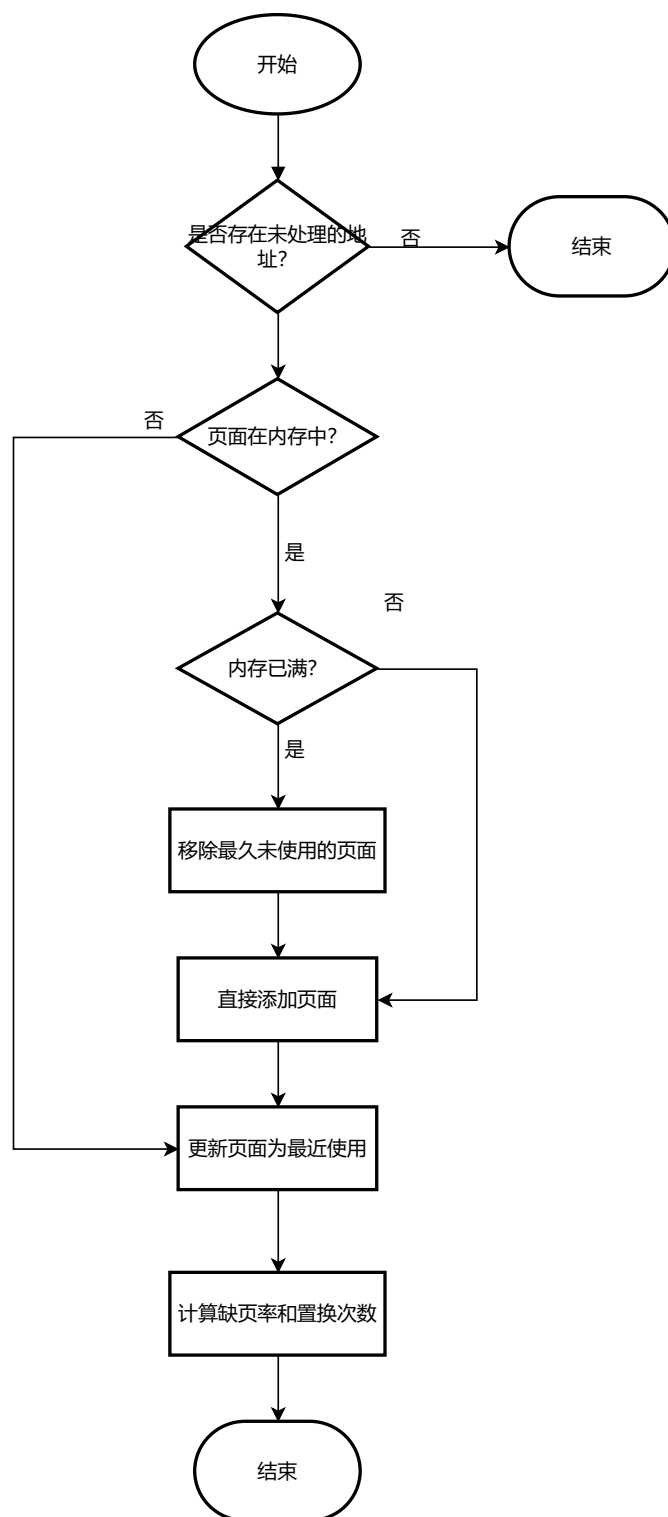


图 2-4: LRU 算法流程图

2.5 运行截图

计算机的初始内存使用情况如下所示（位示图）：

第0字节	00111001
第1字节	00001100
第2字节	10001100
第3字节	01111101
第4字节	01110010
第5字节	01000111
第6字节	00110100
第7字节	00101100

图 2-5: 初始状态下，经过随机生成的位示图

内存空间（位示图）：

第0字节	11111111
第1字节	11111111
第2字节	10001101
第3字节	01111101
第4字节	01110010
第5字节	01000111
第6字节	00110100
第7字节	00101100

图 2-6: 创建多个进程后，位示图的变化情况

模拟执行 FIFO 算法后程序的输出如下所示。

Listing 1: FIFO 算法运行结果

```
1 选择要执行的算法：FIFO[a]或LRU[b]a
2 自动随机生成范围为[0，
   4096]、长度为10的进程逻辑地址序列：[759，3112，792，
   2940，2817]
3
4 逻辑地址：759
5 逻辑地址759对应的页号为0，页内偏移地址为759
6 0号页已存在于内存，无需置换
7 逻辑地址759对应的物理地址为：15095
8
9 逻辑地址：3112
10 逻辑地址3112对应的页号为3，页内偏移地址为40
11 3号页不存在于内存，外存块号为12，需置换...
12     利用FIFO算法选中内存队列0号页，该页内存块号为14，
       修改位为False，
13     内存14号块内容写入置换区12号块，
14     置换区12内容写入内存14号块--置换完毕！
15 逻辑地址3112对应的物理地址为：14376
16
17 逻辑地址：792
18 逻辑地址792对应的页号为0，页内偏移地址为792
19 0号页不存在于内存，外存块号为12，需置换...
20     利用FIFO算法选中内存队列0号页，该页内存块号为11，
       修改位为False，
21     内存11号块内容写入置换区12号块，
22     置换区12内容写入内存11号块--置换完毕！
23 逻辑地址792对应的物理地址为：12056
24
25 逻辑地址：2817
26 逻辑地址2817对应的页号为2，页内偏移地址为769
27 2号页已存在于内存，无需置换
28 逻辑地址2817对应的物理地址为：11009
29 缺页率为40.0%，交换次数为2
```

模拟执行 LRU 算法后程序的输出如下所示。

Listing 2: LRU 算法运行结果

```
1 选择要执行的算法：FIFO[a]或LRU[b]b
2 自动随机生成范围为[0,
   4096]、长度为10的进程逻辑地址序列：[2276, 1273, 1763,
   2757, 837]
3
4 逻辑地址：2276
5 逻辑地址2276对应的页号为 2，页内偏移地址为 228
6 2 号页已存在于内存，无需置换，将该页放在栈顶。
7 逻辑地址2276对应的物理地址为：1252
8
9 逻辑地址：1273
10 逻辑地址1273对应的页号为 1，页内偏移地址为 249
11 1 号页已存在于内存，无需置换，将该页放在栈顶。
12 逻辑地址1273对应的物理地址为：5369
13
14 逻辑地址：1763
15 逻辑地址1763对应的页号为 1，页内偏移地址为 739
16 1 号页已存在于内存，无需置换，将该页放在栈顶。
17 逻辑地址1763对应的物理地址为：5859
18
19 逻辑地址：2757
20 逻辑地址2757对应的页号为 2，页内偏移地址为 709
21 2 号页已存在于内存，无需置换，将该页放在栈顶。
22 逻辑地址2757对应的物理地址为：1733
23
24 逻辑地址：837
25 逻辑地址837对应的页号为 0，页内偏移地址为 837
26 0 号页已存在于内存，无需置换，将该页放在栈顶。
27 逻辑地址837对应的物理地址为：6981
28 缺页率为0.0%，交换次数为0
```

2.6 小结

本次实验通过实现分页式存储管理，帮助我深入理解了内存空间的分配与回收、地址转换过程，以及分页存储管理中的位运算操作。在实验中，我实现了基于位示图的内存和置换空间管理，掌握了如何有效地分配和回收内存空间。通过对内存的请求分页式管理，我不仅实践了内存分配和地址转换的原理，还能够模拟缺页处理过程。特别是在实现 FIFO 和 LRU 算法时，我体验到了两者在不同情况下的优缺点，并进一步掌握了页面置换算法的实现技巧。这次实验不仅加深了我对操作系统内存管理机制的理解，也提高了我在实际编程中应用操作系统原理的能力。

3 文件与磁盘管理

3.1 目的

1. 熟练掌握树型目录结构的数据结构设计、存储以及遍历等管理过程。
2. 掌握交互式命令的设计与使用。
3. 了解磁盘文件的存取。

3.2 内容

本实验围绕操作系统的文件管理，有以下 5 个实验任务：

1. **文件系统实现：**利用内存或外存存储结构实现文件系统的树型目录结构，并通过交互式命令完成文件与目录管理。
2. **命令支持：**提供如下命令（大小写均可识别）：MD（创建空目录）、CD（切换当前目录）、RD（删除空目录）、MK（创建文件）、DEL（删除文件）和 DIR（列出目录信息）。
3. **DIR 命令格式：**DIR 命令的显示格式应符合规定要求，仅在出错时显示提示信息。
4. **CD 命令提示：**CD 命令应显示当前目录位置，如“\test\”，其它命令在正常执行时无需提示信息，出错时才提供反馈。
5. **目录结构格式：**目录结构必须使用 FCB 或 i-node 格式，目录项至少包含文件或目录名称、类型（文件或目录）、创建日期、文件大小等基本信息。

3.3 数据结构

3.3.1 文件管理系统及 FAT 表

FATFileSystem 是一个基于外存虚盘的文件管理系统，具有多个属性：

- **fat**: 表示 FAT 表，包含 8 个块，每个块代表一个文件或文件夹（即 FCB）。
- **disk_file**: 表示模拟磁盘空间的虚拟磁盘文件。在本实验中，它是一个实际存在的 .bin 文件（virtual_disk.bin），而不是仅存在于内存中的临时数据结构。该虚拟磁盘文件存储着 FAT 表及 FCB 信息。
- **init_disk**: 该函数的作用是打开或创建 virtual_disk.bin 文件，从而初始化虚拟磁盘文件。
- **current_directory**: 表示当前文件系统所在位置对应的 FAT 块号。0 号块是根目录（root），因此在初始化文件系统时，current_directory 的值为 0，并会随着目录位置的变化而更新。
- **create_time**: 表示根目录的创建时间，使用 datetime 包获取当前时刻。

```
1 class FATFileSystem:
2     def __init__(self):
3         self.fat = [EMPTY_BLOCK] * BLOCK_COUNT #初始化FAT表
4         self.disk_file = open(DISK_FILE, "wb+")
5         self.init_disk()
6         self.current_directory = 0 # 根目录起始块号
7         self.create_time =
            datetime.now().strftime("%Y/%m/%d %H:%M")
```

3.3.2 文件控制块 (FCB)

文件控制块 (File Control Block) 是文件系统中用于存储文件元数据的数据结构，每个文件或目录对应一个 FCB。本实验中，FCB 结构采用 Python 语言的 struct.pack 函数进行打包，共 32 字节。通过设置字符串 "8sIHHH14s" 来实现各项数据内容以指定格式打包。

- **name**: 文件或目录的名称，长度为 8 字节。通过 name.encode() 将名称转换为字节形式存储。
- **size**: 文件的大小，使用 4 字节存储，表示文件的字节数。
- **first_block**: 文件的起始数据块号，使用 2 字节存储，指向文件内容在磁盘上的起始位置。

- **file_type**: 文件类型，使用 2 字节存储。可以区分文件或目录（1 代表文件，2 代表目录等）。
- **parent_block**: 父目录的块号，使用 2 字节存储。它指向包含当前文件或目录的父目录的 FCB。
- **create_time**: 文件或目录的创建时间，使用 14 字节存储。存储格式为“年月日时分秒”（例如：“20241117123045”）。

```
1 def create_fcb(name, size, file_type, first_block,
2     parent_block):
3     datetime_str =
4         datetime.now().strftime("%Y%m%d%H%M%S")
5     fcb_data = struct.pack("8sIH14s", name.encode(),
6         size, first_block, file_type, parent_block,
7         datetime_str.encode())
8     return fcb_data
```

3.4 算法设计及流程图

本实验的侧重点在于实现 FCB 数据结构的定义和查询等操作，故下面将对实验中涉及到的两个复杂命令（cd 和 rd）的实现算法进行介绍。

3.4.1 cd 命令

cd 命令的算法思想是通过判断目标名称来决定切换路径的方式。如果目标名称为“..”，表示切换到上一级目录，此时需要检查当前是否位于根目录，如果是则直接提示已在根目录并结束操作；否则，通过读取当前目录的块数据，解析其中的文件控制块（FCB）以获取父目录的块号，随后切换到父目录。如果目标名称是具体的子目录名称，则在当前目录的数据块中逐个解析 FCB，查找与目标名称匹配且类型为目录的块号；若找到，则切换到目标目录块号，否则提示目标目录未找到。这种设计结合了递归式的目录结构和 FCB 的解析操作，确保能够高效、准确地完成目录切换功能。流程图如图3-1所示。

3.4.2 rd 命令

rd 命令的核心思想是：首先通过目录名查找其对应的块号，如果未找到，则说明目录不存在，直接返回；如果找到，则检查该目录是否为空。如果目录非空，输出提示信息并结束操作；如果目录为空，释放与该目录相关的 FAT 表记录，同时从其父目录中移除对应的 FCB 信息。通过读取父目录块数据并遍历其中的 FCB，

找到目标目录后清空其对应的 FCB 记录，最后将修改后的父目录数据写回。整个过程在确认目录为空的前提下，递归地释放块和更新父目录，最终完成删除操作并输出结果。流程图如图3-2所示。

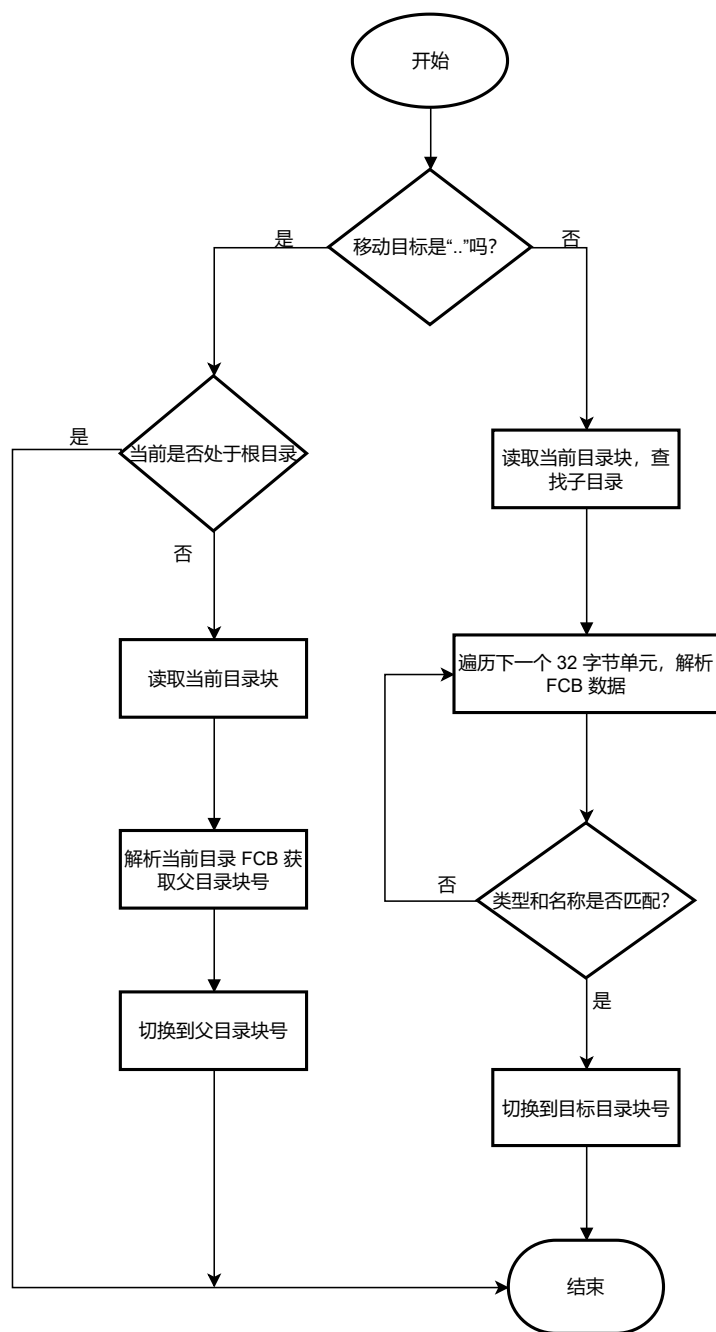
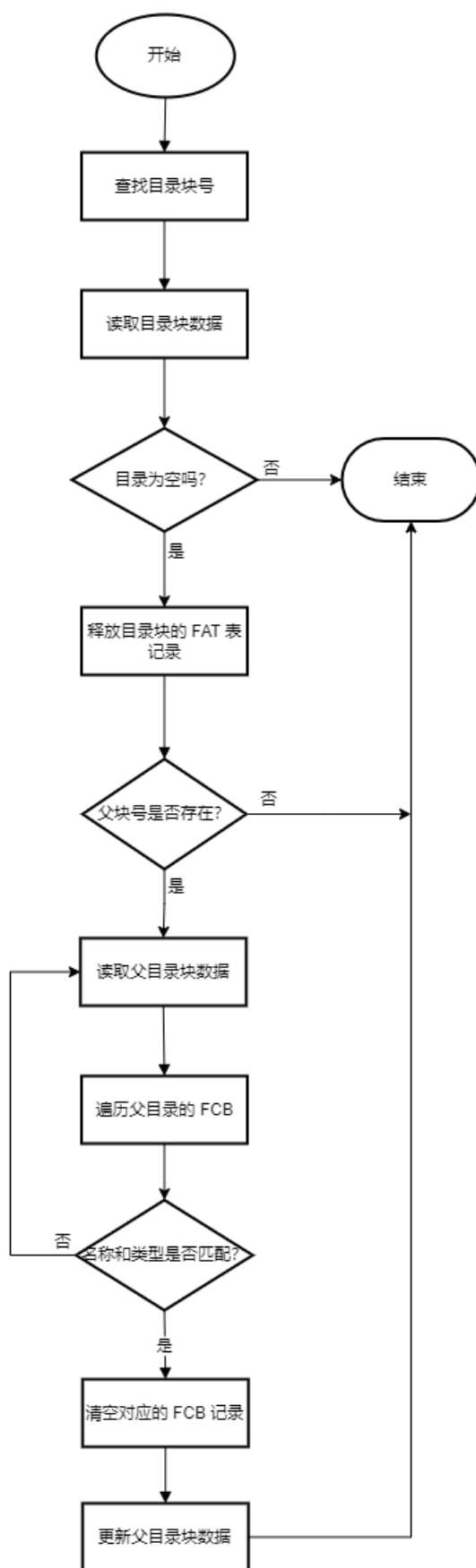
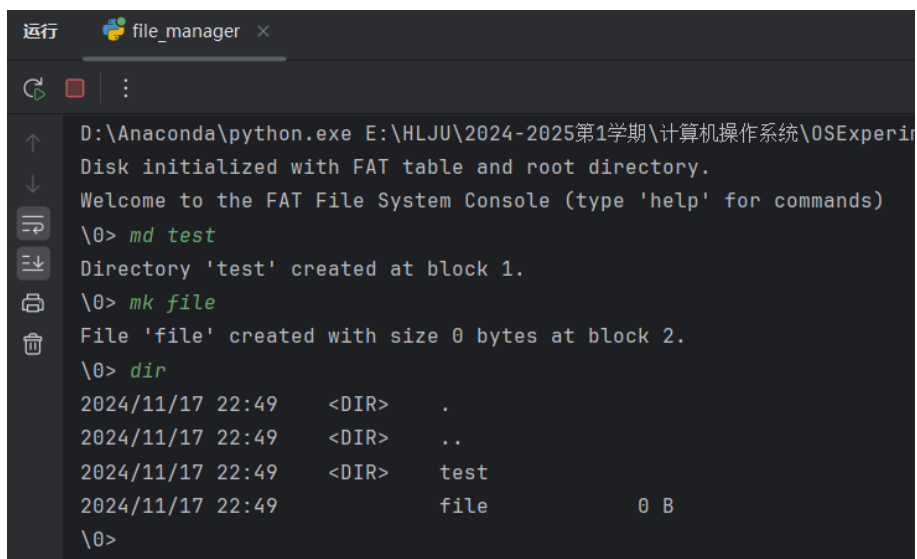


图 3-1: `cd` 命令流程图

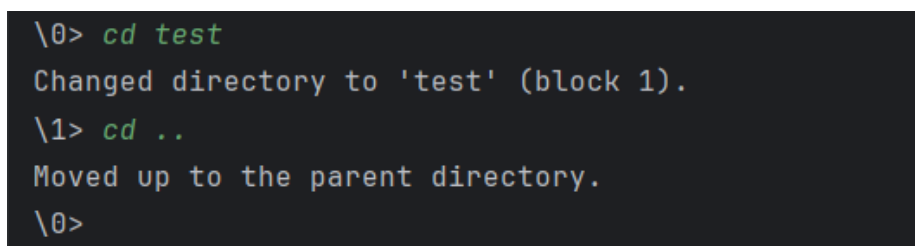
图 3-2: `rd` 命令流程图

3.5 运行截图



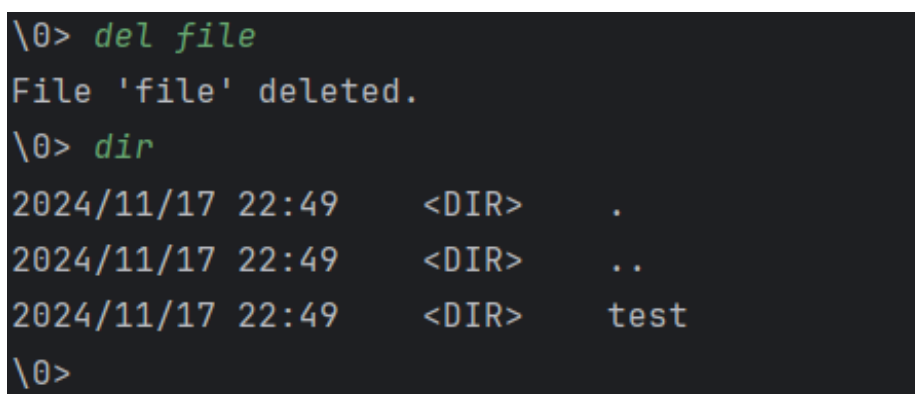
```
运行 file_manager x
D:\Anaconda\python.exe E:\HLJU\2024-2025第1学期\计算机操作系统\0SEExperin
Disk initialized with FAT table and root directory.
Welcome to the FAT File System Console (type 'help' for commands)
\0> md test
Directory 'test' created at block 1.
\0> mk file
File 'file' created with size 0 bytes at block 2.
\0> dir
2024/11/17 22:49 <DIR> .
2024/11/17 22:49 <DIR> ..
2024/11/17 22:49 <DIR> test
2024/11/17 22:49 file 0 B
\0>
```

图 3-3: 在根目录下创建文件和文件夹后使用 `dir` 命令



```
\0> cd test
Changed directory to 'test' (block 1).
\1> cd ..
Moved up to the parent directory.
\0>
```

图 3-4: 进入子目录然后返回父目录



```
\0> del file
File 'file' deleted.
\0> dir
2024/11/17 22:49 <DIR> .
2024/11/17 22:49 <DIR> ..
2024/11/17 22:49 <DIR> test
\0>
```

图 3-5: 使用 `del` 命令删除文件

```
\0> rd test
Directory 'test' deleted.
\0> cd test
Directory 'test' not found.
\0> dir
2024/11/17 22:49    <DIR>      .
2024/11/17 22:49    <DIR>      ..
```

图 3-6: 使用 rd 命令删除空文件夹

```
运行 file_manager x
D:\Anaconda\python.exe E:\HLJU\2024-2025第1学期\计算机操作系统\OSExperiment1
Disk initialized with FAT table and root directory.
Welcome to the FAT File System Console (type 'help' for commands)
\0> md test
Directory 'test' created at block 1.
\0> cd test
Changed directory to 'test' (block 1).
\1> mk file
File 'file' created with size 0 bytes at block 2.
\1> cd ..
Moved up to the parent directory.
\0> rd test
Directory 'test' is not empty.
\0>
```

图 3-7: 使用 rd 命令删除非空文件夹

file_manager.py virtual_disk.bin x

</

图 3-8: 文件系统依赖的虚盘文件 virtual_disk.bin 内容

```
\0> info
00000000 FFFF FFFF FFFF 00 00 00 00 00
00000000 74 65 73 74 00 00 00 00 01 00 00 01 00 02 00 test.....
00000010 00 00 32 30 32 34 31 31 31 37 32 33 31 33 33 34 ..20241117231334
00000020 66 69 6C 65 00 00 00 00 00 00 00 00 02 00 01 00 file.....
00000030 00 00 32 30 32 34 31 31 31 37 32 33 31 33 33 39 ..20241117231339
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

图 3-9: 使用命令 info 查看当前目录对应的虚盘信息

3.6 小结

通过本次实验，我熟练掌握了树型目录结构的数据结构设计 with 实现，包括目录结构的存储、遍历和管理。在实验过程中，采用了基于内存或外存的存储结构，实现了文件系统的基础功能，并设计了交互式命令用于文件与目录的管理。通过编写和测试 MD、CD、RD、MK、DEL 和 DIR 等命令，深入理解了文件系统的操作逻辑和实际应用。

实验过程中，成功实现了符合要求的 DIR 命令格式，并在 CD 命令中准确提示当前目录路径。同时，对于错误处理逻辑进行了优化，确保用户操作的直观性和系统的健壮性。此外，结合 FCB 数据结构设计目录项，包含文件或目录名称、类型、创建日期、大小等基本信息，进一步强化了对文件控制块和文件元信息的理解。

总体而言，本次实验不仅巩固了文件与磁盘管理的理论知识，还提升了实际编程能力，为更复杂的文件系统开发奠定了坚实基础。

4 进程调度

4.1 目的

- 1. 掌握进程调度算法的实现与应用：学习并实现先来先服务（FCFS）、短作业优先（SJF）和时间片轮转（RR）等常见调度算法，掌握其原理和应用。
- 2. 计算进程的周转时间与带权周转时间：在模拟进程调度的过程中，计算每个

进程的周转时间、带权周转时间，及其对系统整体性能的影响。

3. **分析调度算法的性能：**通过实验计算各类调度算法下的平均周转时间和平均带权周转时间，分析不同算法的优劣和适用场景。
4. **理解并实现银行家算法：**探索并实现一个独立的避免死锁的银行家算法，用于资源分配的安全性检测和避免死锁的发生。

4.2 内容

1. **实现进程调度算法：**在前面实验的基础上，实现以下调度算法：
 - 先来先服务 (FCFS)
 - 短作业优先 (SJF)
 - 时间片轮转 (RR)
2. **进程调度模拟：**根据当前所设定的调度算法，连续调度所有进程，并计算每个进程的：
 - 周转时间
 - 带权周转时间
 - 所有进程的平均周转时间
 - 所有进程的平均带权周转时间
3. 选做：**实现银行家算法**，用于避免死锁的资源分配安全性检测。

4.3 数据结构

本实验聚焦于进程的调度算法，故实现实验仅对 PCB 内容作了特殊修改：

- **name:** 进程的名称，用于标识进程。
- **priority:** 进程的优先级，用于在调度算法中决定进程的调度顺序。默认为 0。
- **arrival_time:** 进程到达时间，表示进程进入系统的时刻。
- **servicing_time:** 进程的服务时间，表示进程需要的总运行时间。
- **running_time:** 进程已运行的时间，初始值为 0，在进程运行时更新。
- **finished_time:** 进程的结束时间，表示进程完成所有任务的时刻。初始值为 None，表示尚未结束。
- **max:** 进程的最大资源需求列表，用于记录进程执行期间可能需要的最大资源数量（如内存、CPU 时间等）。
- **allocation:** 进程当前已经分配的资源数量列表，表示系统当前分配给该进程的资源。

- **need:** 进程还需的资源数量列表, 通过最大需求减去当前分配的资源来计算, 即 $need = max - allocation$ 。

```

1 class PCB:
2     def __init__(self, name: str, arrival_time: int,
3         servicing_time: int,
4         priority: int = 0, max_r: list[int] =
5             None, alloc: list[int] = None):
6         self.name = name
7         self.priority = priority # 优先级
8         self.arrival_time = arrival_time # 到达时间
9         self.servicing_time = servicing_time # 服务时间
10        self.running_time = 0 # 已运行时间
11        self.finished_time = None # 结束时间
12        self.max = max_r or [] # 最大资源需求
13        self.allocation = alloc # 分配资源
14        self.need = [max_r - alloc for max_r, alloc in
15            zip(self.max, self.allocation)] # 还需资源

```

4.4 算法设计及流程图

本节对实验 4 涉及到的 6 个进程调度算法以及避免死锁的银行家算法及其安全性算法进行介绍, 提供所有算法的伪代码和部分流程复杂算法的流程图。

4.4.1 FCFS 调度算法

FCFS (First Come First Serve, 先来先服务) 是一种简单的调度算法, 其核心思想是按照进程的到达顺序执行, 不考虑进程的执行时间。该算法假设进程在到达时就可以进入队列, 且执行顺序仅由它们到达的时间决定。在调度过程中, 系统会先处理到达的第一个进程, 直到它完成, 然后再处理下一个进程, 依此类推。FCFS 的优点是简单易实现, 但它也存在缺点, 主要是可能导致“长进程阻塞短进程”的情况, 即如果一个长进程先到达, 后续的短进程可能会因为等待较长时间而导致较大的周转时间。算法的伪代码和流程图如下所示:

Algorithm 5 FCFS Scheduling Algorithm**Input:** Process list with arrival times and service times**Output:** Average turnaround time, average weighted turnaround time

- 1: **Initialize:** $t \leftarrow 0$, $T_{turnaround} \leftarrow 0$, $T_{weighted} \leftarrow 0$
- 2: Sort processes by arrival time
- 3: **for** each process in process list **do**
- 4: Calculate start time and finish time
- 5: Calculate $T_{turnaround}$ and $T_{weighted}$
- 6: Update current time to finish time
- 7: Add values to total $T_{turnaround}$ and $T_{weighted}$
- 8: **end for**
- 9: $avg_{turnaround} \leftarrow T_{turnaround} / \text{number of processes}$
- 10: $avg_{weighted} \leftarrow T_{weighted} / \text{number of processes}$
- 11: **return** $avg_{turnaround}, avg_{weighted}$

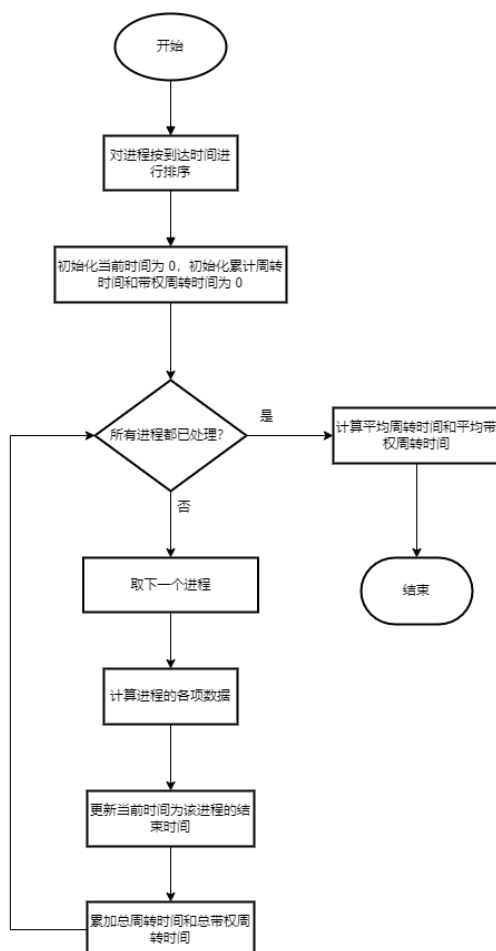


图 4-1: FCFS 算法流程图

4.4.2 SJF 调度算法

SJF (Shortest Job First, 短作业优先) 调度算法的基本思路是按照进程的服务时间来排序, 优先选择服务时间最短的进程进行执行, 直至所有进程完成。这个算法具有非抢占性, 意味着一旦一个进程开始执行, 它会一直执行到完成, 不会被中断。其优点在于能够显著降低平均周转时间。然而, 它也可能导致“饥饿”现象, 特别是在短作业过多的情况下。

在 SJF 算法中, 首先会按进程到达时间排序, 然后在每个时刻选择服务时间最短的进程执行。每个进程完成后, 都会计算其周转时间和带权周转时间。最终, 算法可以求得整个进程队列的平均周转时间和平均带权周转时间。算法伪代码和流程图如下所示:

Algorithm 6 SJF Scheduling Algorithm

Input: Process list with arrival times and service times

Output: Average turnaround time, average weighted turnaround time

```

1: Initialize:  $t \leftarrow 0$ ,  $T_{turnaround} \leftarrow 0$ ,  $T_{weighted} \leftarrow 0$ 
2: Sort processes by arrival time
3: while process list or ready queue is not empty do
4:   Move all processes from process list that have arrived to ready queue
5:   if ready queue is not empty then
6:     Sort ready queue by service time
7:     Select process with shortest service time
8:     Calculate start time, finish time,  $T_{turnaround}$ , and  $T_{weighted}$ 
9:     Update current time to finish time
10:    Add values to total turnaround and weighted turnaround times
11:   else
12:     Increment current time
13:   end if
14: end while
15:  $avg_{turnaround} \leftarrow T_{turnaround} / \text{number of processes}$ 
16:  $avg_{weighted} \leftarrow T_{weighted} / \text{number of processes}$ 
17: return  $avg_{turnaround}$ ,  $avg_{weighted}$ 

```

4.4.3 轮转调度算法

轮转 (Round Robin, RR) 调度算法是一种简单且常用的进程调度算法, 尤其适用于时间共享系统。它的核心思想是将每个进程分配一个固定长度的时间片

(time quantum), 每个进程在执行时会占用 CPU 一个时间片, 若该进程在时间片内完成, 则继续执行下一个进程; 如果时间片结束而进程仍未完成, 则将其放回就绪队列末尾, 等待下一轮调度。

RR 算法的执行过程是就将就绪队列中的进程按到达顺序依次调度。每个进程在一个时间片内被调度执行, 若进程没有完成, 它会被重新加入就绪队列并等待下次调度。若进程完成, 则从队列中移除, 并计算其周转时间和带权周转时间。这样循环直到所有进程完成。该算法公平性较好, 因为每个进程都能获得相同的 CPU 时间, 但其缺点是当时间片较大时, 可能无法充分发挥系统的响应能力, 且对于较短的进程会有一定的时间浪费。

Algorithm 7 RR Scheduling Algorithm

Input: Process list with arrival times, service times, and time quantum

Output: Average turnaround time, average weighted turnaround time

```

1: Initialize:  $t \leftarrow 0$ ,  $T_{turnaround} \leftarrow 0$ ,  $T_{weighted} \leftarrow 0$ 
2: Sort processes by arrival time
3: while there are processes to execute do
4:   Add processes to ready queue that have arrived by current time
5:   if ready queue is not empty then
6:     Dequeue process, execute for time quantum or until completion
7:     if process completes then
8:       Calculate  $T_{turnaround}$  and  $T_{weighted}$ 
9:       Update total  $T_{turnaround}$ 
10:    else
11:      Add process back to ready queue
12:    end if
13:  else
14:    Increment current time
15:  end if
16: end while
17:  $avg_{turnaround} \leftarrow T_{turnaround} / \text{number of processes}$ 
18:  $avg_{weighted} \leftarrow T_{weighted} / \text{number of processes}$ 
19: return  $avg_{turnaround}$ ,  $avg_{weighted}$ 

```

4.4.4 优先级抢占调度算法

优先级抢占调度算法 (Priority Scheduling, Preemptive) 根据进程的优先级来决定调度顺序。每个进程都有一个优先级, 优先级高的进程会被优先执行。如果有

新的高优先级进程到达，正在执行的进程将被中断，新的高优先级进程将立即执行。该算法是一种抢占式调度算法，可以确保高优先级的任务得到及时处理，避免低优先级进程被长时间忽略。

然而，优先级抢占调度也可能导致低优先级进程长期无法执行，产生“饥饿”现象。为了缓解这种情况，通常会对进程的优先级进行动态调整。该算法的执行过程中，每个进程的周转时间和带权周转时间会被计算，并最终得到平均周转时间和平均带权周转时间。

Algorithm 8 Preemptive Priority Scheduling Algorithm (PS)

Input: Process list with arrival times, service times, and priorities

Output: Average turnaround time, average weighted turnaround time

```

1: Sort processes by arrival time
2: Initialize:  $t \leftarrow 0$ ,  $T_{turnaround} \leftarrow 0$ ,  $T_{weighted} \leftarrow 0$ 
3: while process list or ready queue is not empty do
4:   for each process that has arrived at current time do
5:     Add process to ready queue
6:   end for
7:   if ready queue is not empty then
8:     Sort ready queue by priority and arrival time
9:     Select highest priority process
10:    Execute for 1 time unit, update current time
11:    if process completes then
12:      Calculate  $T_{turnaround}$  and  $T_{weighted}$ 
13:      Update total turnaround and weighted turnaround times
14:      Remove completed process from queue
15:    end if
16:  else
17:    Increment current time
18:  end if
19: end while
20:  $avg_{turnaround} \leftarrow T_{turnaround} / \text{number of processes}$ 
21:  $avg_{weighted} \leftarrow T_{weighted} / \text{number of processes}$ 
22: return  $avg_{turnaround}$ ,  $avg_{weighted}$ 

```

4.4.5 高响应比优先调度算法

高响应比优先 (Highest Response Ratio Next, HRRN) 调度算法是优先级调度算法的一个特例。在执行过程中, HRRN 首先会计算每个进程的响应比, 并按照响应比的大小进行排序, 优先选择响应比最高的进程进行执行。通过平衡等待时间和服务时间, HRRN 算法能够提高系统的整体效率, 并有效避免进程长期处于等待状态。响应比的数学表达式如下:

$$\text{响应比} = \frac{\text{等待时间} + \text{服务时间}}{\text{服务时间}}$$

Algorithm 9 Highest Response Ratio Next Scheduling Algorithm (HRRN)

Input: Process list with arrival times and service times

Output: Average turnaround time, average weighted turnaround time

```

1: Sort processes by arrival time
2: Initialize:  $t \leftarrow 0$ ,  $T_{\text{turnaround}} \leftarrow 0$ ,  $T_{\text{weighted}} \leftarrow 0$ 
3: while process list or ready queue is not empty do
4:   for each process that has arrived at current time do
5:     Add process to ready queue
6:   end for
7:   if ready queue is not empty then
8:     for each process in ready queue do
9:       Calculate waiting time and response ratio
10:    end for
11:    Sort ready queue by response ratio in descending order
12:    Select process with highest response ratio
13:    Execute selected process, update current time
14:    if process completes then
15:      Calculate and update  $T_{\text{turnaround}}$  and  $T_{\text{weighted}}$ 
16:      Remove completed process from queue
17:    end if
18:  else
19:    Increment current time
20:  end if
21: end while
22:  $\text{avg}_{\text{turnaround}} \leftarrow T_{\text{turnaround}} / \text{number of processes}$ 
23:  $\text{avg}_{\text{weighted}} \leftarrow T_{\text{weighted}} / \text{number of processes}$ 
24: return  $\text{avg}_{\text{turnaround}}, \text{avg}_{\text{weighted}}$ 

```

4.4.6 多级反馈队列调度算法

多级反馈队列调度算法 (Multilevel Feedback Queue Scheduling, MLFQ) 是一种灵活且高效的调度算法, 通过将进程分配到多个优先级队列中进行管理, 根据其运行行为动态调整优先级, 达到兼顾系统响应速度和吞吐量的目的。

在 MLFQ 算法中, 系统维护一组队列, 每个队列都有不同的优先级和时间片 (如图4-2所示)。初始时, 所有进程被放置在最高优先级队列中, 并按照时间片轮转执行。如果某个进程未能在当前队列时间片内完成, 它将被降级到下一级队列。低优先级队列通常拥有更大的时间片, 但调度频率较低。相反, 若某进程在运行中表现出较高的交互性 (如频繁阻塞), 它可以被提升至更高优先级队列, 从而减少等待时间。

Algorithm 10 Multilevel Feedback Queue Scheduling Algorithm (MLFQ)

Input: Process list with arrival times, service times; time slices for each queue

Output: Average turnaround time, average weighted turnaround time

```

1: Initialize: Create  $N$  queues,  $t \leftarrow 0$ ,  $T_{turnaround} \leftarrow 0$ ,  $T_{weighted} \leftarrow 0$ 
2: Sort processes by arrival time
3: while process list or queues are not empty do
4:   Move arrived processes to the highest priority queue
5:   for each queue from high to low priority do
6:     if queue not empty then
7:       Execute process for time slice or until completion
8:       if process completes then
9:         Update  $T_{turnaround}$ ,  $T_{weighted}$  and move process to completed list
10:      else
11:        Move process to next queue or back to same queue
12:      end if
13:    Break
14:  end if
15: end for
16: If no processes ready,  $t \leftarrow t + 1$ 
17: end while
18:  $avg_{turnaround} \leftarrow T_{turnaround} / \text{number of processes}$ 
19:  $avg_{weighted} \leftarrow T_{weighted} / \text{number of processes}$ 
20: return  $avg_{turnaround}$ ,  $avg_{weighted}$ 

```

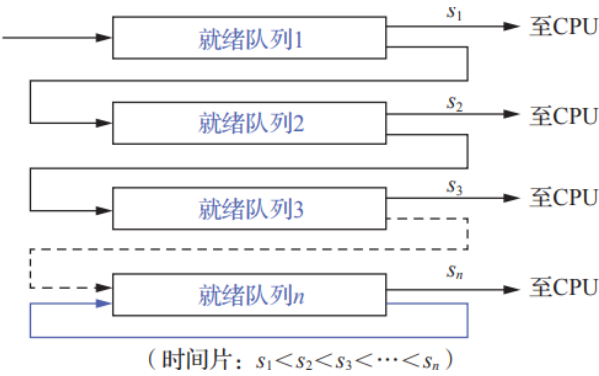


图 4-2: 多级反馈队列调度算法

4.4.7 银行家算法

银行家算法 (Banker's Algorithm) 是一种用于实现系统资源分配的安全性检测算法，常用于避免死锁问题。它的基本思想是，在为进程分配资源时，系统会模拟分配操作并检测分配后的状态是否安全。只有在保证系统处于安全状态时，资源分配请求才会被批准，否则会被拒绝。

银行家算法需要维护以下关键信息：

- **最大需求矩阵 (Max)**：每个进程对各资源类型的最大需求。
- **分配矩阵 (Allocation)**：每个进程当前已分配的资源量。
- **需求矩阵 (Need)**：每个进程还需要的资源量，由公式 $Need = Max - Allocation$ 计算得到。
- **可用资源向量 (Available)**：系统当前可用的资源数量。

Algorithm 11 Safety Algorithm

Input: Available resources $Work$, processes' $Need$ and $Allocation$

Output: Whether the system is in a safe state, and the safe sequence if it exists

```
1:  $Finish[i] \leftarrow false, SafeSequence \leftarrow []$ 
2: while there exists an  $i$  such that  $Finish[i] = false$  and  $Need[i] \leq Work$  do
3:    $Work \leftarrow Work + Allocation[i], Finish[i] \leftarrow true$ 
4:   Append  $P_i$  to  $SafeSequence$ 
5: end while
6: if all  $Finish[i] = true$  then
7:   return  $true, SafeSequence$ 
8: else
9:   return  $false, \emptyset$ 
10: end if
```

Algorithm 12 Banker’s Algorithm for Resource Request

Input: Process P , resource request R

Output: Whether the system remains in a safe state

1: if all $R[i] \leq P.Need[i]$ and $R[i] \leq Available[i]$ for all i then

2: Tentatively allocate resources:

3: $Available[i] \leftarrow Available[i] - R[i]$

4: $P.Allocation[i] \leftarrow P.Allocation[i] + R[i]$

5: $P.Need[i] \leftarrow P.Need[i] - R[i]$

6: Perform safety check:

7: if system is in a safe state then

8: Approve the request and return true

9: else

10: Rollback:

11: $Available[i] \leftarrow Available[i] + R[i]$

12: $P.Allocation[i] \leftarrow P.Allocation[i] - R[i]$

13: $P.Need[i] \leftarrow P.Need[i] + R[i]$

14: Reject the request and return false

15: end if

16: else

17: Reject the request as invalid

18: end if

4.5 运行截图

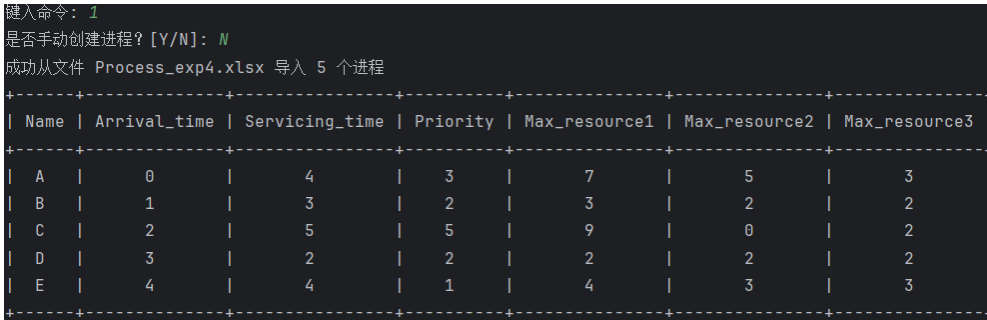


图 4-3: 通过导入.xlsx 文件自动创建进程，快捷录入进程信息

键入命令: 2

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
A	0	4	4	4	1.0000
B	1	3	7	6	2.0000
C	2	5	12	10	2.0000
D	3	2	14	11	5.5000
E	4	4	18	14	3.5000

平均周转时间: 9.0000
平均带权周转时间: 2.8000

图 4-4: 执行 FCFS 算法

键入命令: 3

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
A	0	4	4	4	1.0000
D	3	2	6	3	1.5000
B	1	3	9	8	2.6667
E	4	4	13	9	2.2500
C	2	5	18	16	3.2000

平均周转时间: 8.0000
平均带权周转时间: 2.1233

图 4-5: 执行 SJF 算法

键入命令: 4

请输入轮转长度: 1

进程执行顺序: ABCDEABCDEABCEACEC

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
D	3	2	9	6	3.0000
B	1	3	12	11	3.6667
A	0	4	15	15	3.7500
E	4	4	17	13	3.2500
C	2	5	18	16	3.2000

平均周转时间: 12.2000
平均带权周转时间: 3.3733

图 4-6: 执行轮转长度为 1 的 RR 算法

键入命令: 5

进程执行顺序: ABBBEEEEDDAAACCC

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
B	1	3	4	3	1.0000
E	4	4	8	4	1.0000
D	3	2	10	7	3.5000
A	0	4	13	13	3.2500
C	2	5	18	16	3.2000

平均周转时间: 8.6000

平均带权周转时间: 2.3900

图 4-7: 执行 PSP 算法

键入命令: 6

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
A	0	4	4	4	1.0000
B	1	3	7	6	2.0000
D	3	2	9	6	3.0000
C	2	5	14	12	2.4000
E	4	4	18	14	3.5000

平均周转时间: 8.4000

平均带权周转时间: 2.3800

图 4-8: 执行 HRRN 算法

键入命令: 7

进程名	到达时间	服务时间	完成时间	周转时间	带权周转时间
B	1	3	9	8	2.6667
D	3	2	12	9	4.5000
A	0	4	15	15	3.7500
C	2	5	17	15	3.0000
E	4	4	18	14	3.5000

平均周转时间: 12.2000

平均带权周转时间: 3.4833

图 4-9: 执行 MLFQ 算法

模拟执行银行家算法后程序的输出如下所示：

Listing 3: 银行家算法运行结果

```
1  请输入执行算法的进程名称: B
2  请输入请求资源: 1 0 2
3  系统处于安全状态。安全序列为: ['B', 'D', 'A', 'C', 'E']
4  各轮次状态:
5  进程: B
6  Work: [2, 3, 0]
7  Need: [0, 2, 0]
8  Allocation: [3, 0, 2]
9  Work+Allocation: [5, 3, 2]
10 Finish: [True, False, False, False, False]
11 进程: D
12 Work: [5, 3, 2]
13 Need: [0, 1, 1]
14 Allocation: [2, 1, 1]
15 Work+Allocation: [7, 4, 3]
16 Finish: [True, True, False, False, False]
17 进程: A
18 Work: [7, 4, 3]
19 Need: [7, 4, 3]
20 Allocation: [0, 1, 0]
21 Work+Allocation: [7, 5, 3]
22 Finish: [True, True, True, False, False]
23 进程: C
24 Work: [7, 5, 3]
25 Need: [6, 0, 0]
26 Allocation: [3, 0, 2]
27 Work+Allocation: [10, 5, 5]
28 Finish: [True, True, True, True, False]
29 进程: E
30 Work: [10, 5, 5]
31 Need: [4, 3, 1]
32 Allocation: [0, 0, 2]
33 Work+Allocation: [10, 5, 7]
34 Finish: [True, True, True, True, True]
35 请求成功: 可以找到一个安全序列: ['B', 'D', 'A', 'C', 'E']
```

4.6 小结

本次实验旨在实现并模拟常见的进程调度算法，包括先来先服务（FCFS）、短作业优先（SJF）和时间片轮转（RR）算法。在实现过程中，我们不仅需要根据不同算法调度所有进程，还需计算每个进程的周转时间和带权周转时间，并进一步得出所有进程的平均周转时间和平均带权周转时间。通过这些计算，我们可以对各调度算法的性能进行分析，评估其效率。

在实验过程中，除了基本的调度算法实现外，还要求在调度过程中输出各进程状态队列的变化情况，包括进程的已执行时间和还需服务时间（特别是在时间片轮转算法中）。通过这些输出，能够更直观地观察调度过程中的资源分配与进程执行情况。此外，本次实验还探索了银行家算法的实现，作为避免死锁的一种解决方案。银行家算法通过确保资源分配安全，避免了系统在运行过程中可能出现的死锁问题。

总的来说，实验加深了对进程调度算法原理的理解，并锻炼了在实际场景中进行资源管理与调度的能力。

5 课程总结

在“计算机操作系统”课程的综合设计中，我全面地涉及了进程控制、分页式存储管理、文件与磁盘管理以及进程调度四大核心模块。每个模块都涉及了操作系统内部重要的基本原理和技术细节。在这些实验的过程中，我不仅掌握了操作系统的基本理论知识，而且通过动手实现相关功能，进一步加深了对操作系统工作机制的理解。

进程控制模块的实验让我深刻理解了操作系统是如何管理进程的生命周期，如何进行进程的调度与状态切换。在这一部分的实验中，我模拟了进程的创建和撤销，并掌握了进程的状态转移过程（如从就绪到运行、从运行到阻塞等）。在实际编码过程中，我还实现了简单的内存空间管理功能，通过内存分配与释放机制来确保系统资源的合理利用。此外，我还学习了进程调度策略的应用，这为我理解操作系统如何高效调度多个进程提供了重要的理论基础。

分页式存储管理模块使我掌握了内存管理的关键技术，特别是如何通过分页机制来优化内存的使用。在这部分实验中，我不仅熟悉了内存空间分配与回收的基本过程，还深入理解了地址转换的具体实现方法。为了实现请求分页式存储管理，我实现了内存和置换空间的管理，并通过算法如 FIFO 和 LRU 来处理缺页异常。在过程中，我还掌握了位运算的应用，尤其是如何通过位示图来有效管理内存和置换空间。这一模块让我对操作系统的内存管理机制有了全面的认识，也让我理解了内存管理中算法优化的重要性。

文件与磁盘管理模块让我对操作系统如何管理文件系统有了更深刻的认识。我设计并实现了树型目录结构，并实现了通过交互式命令来管理文件与目录的功能（如创建目录、切换目录、删除目录、创建文件等）。在此过程中，我学习了如何使用内存或外存存储结构来构建文件系统，并掌握了目录项的设计（包括文件或目录名称、类型、创建日期、大小等信息）。通过实现文件系统的管理命令，我不仅掌握了文件系统的基本管理操作，还加深了对磁盘文件存取过程的理解。目录结构的设计让我进一步明白了如何高效地存储与访问文件，并且提高了我对文件系统设计的兴趣。

进程调度模块的实验让我深入理解了操作系统如何根据不同的调度算法来管理多个进程。在这一部分，我实现了几种经典的进程调度算法：先来先服务（FCFS）、短作业优先（SJF）和时间片轮转（RR）。通过模拟这些算法的调度过程，我能够计算每个进程的周转时间与带权周转时间，并能够分析不同调度算法对系统性能的影响。对于时间片轮转算法，我还计算了每个进程的已执行时间与剩余服务时间，并模拟了进程在不同调度方式下的状态变化。此外，我还对进程调度中的优先级调度进行了扩展，尝试结合银行家算法来避免死锁问题。通过这一系列的实验，我更加深入地理解了操作系统中进程调度的原理与实际应用，并掌握了如何根据不同的调度策略来提高系统的整体性能。

通过本次综合设计实验，我对操作系统的各个模块有了更加系统的了解，尤其是在实际编码实现过程中，我深入理解了操作系统中的核心原理和技术。在编程与实验的过程中，我不仅提升了自己的编程技能，还锻炼了我的问题分析与解决能力。每个模块的实验都有其独特的挑战，解决这些问题的过程中，我学习了如何在实际应用中结合理论与技术。特别是在分页式存储管理和进程调度算法的实现中，我加深了对操作系统资源管理和调度机制的理解，也增强了自己在解决复杂问题时的思维能力。

此外，本次实验还让我更加明确了操作系统设计中的细节和优化思路。例如，内存管理不仅仅是关于内存分配，更关乎如何高效地进行资源调度；而进程调度中的算法选择直接影响到系统的响应时间和整体效率。在未来的学习中，我将继续深化这些理论的理解，并尝试应用到更复杂的操作系统设计与开发中。

总体而言，这次课程设计让我充分理解了操作系统设计中的复杂性，也让我意识到操作系统各个模块的紧密配合是构建高效系统的关键。我将在未来的学习中，继续扩展对操作系统各模块的知识，尤其是在高并发和分布式系统中的应用，进一步提升自己的操作系统设计能力。

A 代码运行环境

项目	内容
操作系统	Windows 11
编程语言	Python 3.9
编程环境	PyCharm 2024.1.4 (Professional Edition)
硬件环境	11th Gen Intel(R) Core(TM) i7-1195G7
机带 RAM	40.0GB
软件包依赖	pandas==2.2.2, tabulate==0.9.0, numpy==1.23.0

B 源代码 GitHub 仓库

个人 GitHub 地址: <https://github.com/Saury997>
源代码仓库: <https://github.com/Saury997/OS-Assignments>
实验报告 L^AT_EX 模板(待更新):https://github.com/Saury997/OS-Assignments/blob/main/Latex_template.tex