北京邮电大学

计算机学院

内存分配实验

姓名: 吴清柳

学号: 2020211597

班级: 2020211323

指导老师: 杜晓峰

课程名称: 操作系统

December 16, 2022

Contents

1 实验题目

内存分配实验: 编写模拟程序演示内存分配, 包括: 基本内存分配, 虚拟内存分配. 提交实验报告, 并在实验报告中提出自己的见解, 思路和方法.

1.1 基本内存分配实验

模拟了对 m 个进程和 n 个内存孔使用 First fit 和 Best fit 内存分配算法进行内存分配的过程.

其中, 进程数量和每个进程的内存需求, 内存孔数量和每个内存孔大小在 config/config.txt 中读取. config.txt 可以由随本模拟程序一并实现的数据生成器 data_gen 生成, 可以调节参数为内存需求和内存孔的大小范围, 请求内存的进程数量和内存孔的数量.

实验实现在 oslab2.1_memory_management_simulation 文件夹下.

1.2 虚拟内存分配实验

本实验程序模拟了对进程的虚拟内存分配, 对单进程分配固定大小的虚拟内存, 借助 TLB 和页表实现虚拟内存和物理内存的转换. 实现了 TLB, 页表, 内存 DRAM 和二级存储 Secondary Storage, 并模拟了 TLB 的 FIFO 和 LRU 换页, 页表从二级存储调页和帧在页表中的位置分配. 使用纯请求调页的方式实现页表的建立. 实验实现在 oslab2.2_virtual_memory_management_simulation 下.

2 实验过程

2.1 基本内存分配实验

2.1.1 概述

由于只需要模拟内存分配的过程, 而不需要对内存进行具体的操作, 所以对进程和内存孔均抽象为包含大小和是否已分配或被占用两个参数的结构体. 使用动态分配的二维数组存储进程和内存孔. 在此基础上模拟 First fit 和 Best fit 算法.

程序代码结构如下.

```
 config
ã  config.txt
 makefile
 run.sh
 src
.  data_gen.c
.  main.c
.  schemas.c
.  schemas.h
.  util.c
.  util.h
```

源代码在 src 文件夹下, 其中:

- main.c 是实验的主程序源代码, 作为实验程序的入口.
- schemas.c 和 schemas.h 实现实验中的内存分配策略, 包括 First fit 和 Best fit 策略.
- util.c 和 util.h 是实验中的工具等内容, 定义了进程和内存孔数据结构, 实现了配置文件的加载等内容. 回收操作;

2.1.2 实现介绍

2.1.2.1 进程和内存孔定义和模拟

进程和内存孔使用如下结构体模拟.

```c
typedef struct Process {
        int psize;
        int pflag;
   } Process;

typedef struct Block {
        int bsize;
        int bflag;
   } Block;
```

根据配置文件 config/config.txt 加载进程队列和可用的内存孔信息;

2.1.2.2 First Fit

First Fit 算法实现在 schemas.c 中, 具体伪代码如下:

Algorithm 1 First Fit

```
Require: $(Processes, Blocks, n, m)$
 1: $in\_frag \leftarrow 0$
 2: for all $p \in Processes$ do
 3:     for all $b \in Blocks$ do
 4:         if $p.psize \leq b.bsize$ and $p.pflag = 0$ and $b.bflag = 0$ then
 5:             $b.bflag \leftarrow 1$
 6:             $p.pflag \leftarrow 1$
 7:             $in\_frag \leftarrow in\_frag + (b.bsize - p.psize)$
 8:             Break
 9:         end if
10:     end for
11: end for
```

对请求内存空间的每一个进程, 程序从第一个内存孔开始判断, 直到找到第一个大小不小于进程请求大小, 且没有被占用的内存孔, 将这个内存孔分配给这个进程, 然后处理下一个进程, 直到所有进程被处理完毕.

2.1.2.3 Best Fit

Best Fit 算法实现在 schemas.c 中, 伪代码如下.

对请求内存空间的每一个进程, 从头开始遍历每个内存孔, 对大于请求空间的内存孔, 如果没有被占用, 且使用此内存孔的内部碎片小于到目前为止的最小碎

Algorithm 2 Best Fit

```
Require: $(Processes, Blocks, n, m)$
 1: $n\_frag \leftarrow 0$
 2: $min\_frag \leftarrow INT\_MAX$
 3: $bid \leftarrow INT\_MAX$
 4: for all $p \in Processes$ do
 5:     $min\_frag \leftarrow INT\_MAX$
 6:     $bid \leftarrow INT\_MAX$
 7:     for $j \leftarrow 0, m$ do
 8:         $b \leftarrow Blocks[j]$
 9:         if $b.bsize \geq p.psize$ and $b.bflag = 0$ and $min\_frag > (b.bsize - p.psize)$ then
10:             $min\_frag \leftarrow (b.bsize - p.psize)$
11:             $bid \leftarrow j$
12:         end if
13:     end for
14:     if $min\_frag \neq INT\_MAX$ then
15:         $Blocks[bid].bflag \leftarrow 1$
16:         $p.pflag \leftarrow 1$
17:         $in\_frag \leftarrow in\_frag + Blocks[bid].bsize - p.psize$
18:     end if
19: end for
```

片, 则将使用此内存孔的内存碎片作为新的最小碎片, 此内存孔作为将要使用的内存孔. 重复这个过程, 直到所有内存孔遍历完成. 当所有进程的请求都被响应后, 内存分配结束. 与 First Fit 相比, Best Fit 造成的内部碎片较小.

2.2 虚拟内存分配实验

为了避免与基本内存分配实验重复内容过多, 虚拟内存分配实验的重点聚集在通过 TLB, 页表的协作来实现对运行在虚拟内存上的进程的虚拟内存寻址请求的响应. 当收到进程的虚拟内存寻址请求的时候, 首先查 TLB 表, TLB 表大小可调, 由宏 TLB_SIZE 控制. 实验中设置为 16. TLB 命中后, 直接取对应帧号, 配合虚拟内存地址偏移从内存数组 DRAM 中取回数据. TLB 没有命中, 则从页表进行查找. 如果找到, 从 DRAM 中取回数据. 如果页表中也没有找到, 则从二级存储 Secondary Storage 中取一帧到 DRAM 中, 取回数据, 然后更新页表 Page Table. TLB 未命中的时候, 也会更新 TLB 表项. 实验实现了 FIFO 和 LRU 两种 TLB 表项更新方法.

2.2.1 TLB 和页表实现介绍

2.2.1.1 TLB 和页表的定义 TLB 和页表使用链表实现, 且实现方式类似. 链表节点的定义如下.

```
/**
* Dual direction linked list.
*/
```

```
typedef struct LLNode {
        int page_index;
        int frame_index;
        int age;
        struct LLNode *prev;
        struct LLNode *next;
    } EntryNode;
```

其中, page_index 和 frame_index 分别是页号和帧号, age 是节点的年龄, 在 LRU 中被使用. prev 和 next 分别指向前面和后面的节点.

TLB 和页表的定义如下.

```
/**
* Define a virtual memory addressing table that can be represented as either
    a
* tlb cache or a page table.
*/
ypedef struct VMemTable {
   // Entry list WITH HEAD NODE.
   EntryNode *entryList;
   int length;
   int page_fault_count;
   int tlb_hit_count;
   int tlb_miss_count;
 VMemTable;
```

entryList 是存储 TLB 和页表表项的链表, 含有头结点. 对 TLB, length 表示 TLB 的长度, tlb_hit_count 表示 tlb 命中次数, tlb_miss_count 记录 tlb 未命中次数; 对页表, length 表示页表的长度, page_fault_count 记录 page fault 而需要查 Secoundary Storage 的次数. 由于是纯请求调页, 所以 page fault 在程序开始的时候会比较频繁. 由于内存访问的局部性原理, 后面的频率会低许多.

2.2.1.2 TLB 和页表的初始化

TLB 和页表使用如下函数进行初始化.

```
VMemTable *create_vmem_table(int length) {
    VMemTable *new_table = malloc(sizeof(VMemTable));
    if (new_table == NULL) {
        printf("Error: failed to allocate memory for Virtual Memory
            Addressing "
               "table.\n");
        exit(-1);
    }

    new_table->length = length;
    // Allocate head node.
    new_table->entryList = new_node();
    // Allocate the rest length nodes.
    struct LLNode *curr_node = new_table->entryList;
    for (int i = 0; i < length; i++) {
        EntryNode *node = new_node();
        insert_node(curr_node, node);
        curr_node = curr_node->next;
    }

    new_table->page_fault_count = 0;
    new_table->tlb_hit_count = 0;
    new_table->tlb_miss_count = 0;

    return new_table;
}
```

可以看到, TLB 和页表都是带头节点的双向链表, 且在创建的时候, 就申请了个数为 length 的表项. TLB 和页表的大小分别使用宏 TLB_SIZE 和 PAGE_TABLE_SIZE 进行控制. 由于本实验程序中内存 DRAM 的大小为 65536 字节, 而帧大小 FRAME_SIZE 为 256, 故页表大小 PAGE_TABLE_SIZE 为 256. 而 TLB_SIZE 设置为 16.

使用链表进行实现是为了方便节点的插入操作.

2.2.2 DRAM 实现介绍

内存 (DRAM) 使用二维数组实现, 大小为 $FRAME\_SIZE \times TOTAL\_FRAME\_COUNT = 256 \times 256 = 65536$. DRAM 的申请过程如下.

```
int **dram_allocate(int frame_count, int block_size) {
    int **temp;
    temp = (int **)malloc(frame_count * sizeof(int *));
    if (temp == NULL) {
        fprintf(stderr, "Error: failed to allocated space for DRAM.\n");
        exit(-1);
    }

    for (int i = 0; i < frame_count; i++) {
        temp[i] = (int *)malloc(block_size * sizeof(int));
        for (int j = 0; j < block_size; j++) {
            temp[i][j] = 0;
        }
    }

    return temp;
}
```

2.2.3 物理存储 Secondary Storage 实现介绍

使用 dd 命令生成的大小为 65536 字节的随机数据文件作为物理存储. 生成命令如下.

```
$ dd if=/dev/urandom of=secondary_storage.bin bs=65536 count=1

1+0 records in
1+0 records out
65536 bytes transferred in 0.000159 secs (412176101 bytes/sec)
```

随实验程序已经事先生成好了一个, 存放在 config/secondaty_storage.bin. 物理存储中的数据仅当页表 Page table 遇到 page fault 的时候逐帧调取到内存 DRAM 中.

2.2.4 具体模拟流程

首先, 程序加载 Secondary storage, 申请 DRAM, TLB table 和 Page table. 然后通过从所给文件逐条读取虚拟内存地址, 转换为物理地址并访问的形式来模拟处理进程的虚拟内存访问请求. 此程序模拟的内存地址为 16 位地址, 前 8 位为页号或帧号, 后 8 位为业内偏移. 具体读取虚拟内存地址并处理的代码如下.

```
while (fgets(address_read_buf, MAX_ADDR_LEN, address_file) != NULL) {
    virtual_addr = atoi(address_read_buf);

    // Get page number.
    page_number = get_page_number(PAGE_MASK, virtual_addr, SHIFT);

    // Get offset.
    offset_number = get_offset(OFFSET_MASK, virtual_addr);

    // Get physical address and translated value stored at that address.
    translate_address(virtual_addr, page_number, offset_number);
    translation_count++;
}
```

在函数 translate_address 中, 根据取到的页号和也偏移, 查 TLB 和页表尝试转换为对应的物理地址. 对于每个由于 page fault 而被分配到 DRAM 中的帧, 都会分配一个唯一的页号.

程序首先尝试从 TLB 中查找页号.

```
// Try to find page in TLB.
int frame_number = -1;

EntryNode *entry_node = tlbTable->entryList->next;
while (entry_node) {
    if (entry_node->page_index == page_number) {
        frame_number = entry_node->frame_index;
        tlbTable->tlb_hit_count++;
        break;
    }

    entry_node = entry_node->next;
}
```

由于 TLB 使用带头结点的链表实现, 所以使用 while 循环从第二个节点开始查找, 直到 TLB 命中或者 TLB 查找结束. 如果使用哈希表等数据结构来模拟将更加合适, 但是由于临近期末时间有限, 只能先使用链表进行实现.

如果 TLB 命中, 则根据使用的页置换策略是 FIFO 还是 LRU 来更新 TLB 表项, 并从 DRAM 中使用查表得到的帧号和帧内偏移找到得到的值.

如果 TLB 未命中, 则 frame_number 仍为-1, 增加 TLB 未命中次数并查 Page table.

```
// If page is not hit in TLB, find page in page table, and increment TLB
// miss count.
// If page is not found in page table, find it in secondary storage DRAM,
// and increment page table fault count.
if (frame_number == -1) {
    tlbTable->tlb_miss_count++;
    // printf("TLB miss count: %d\n", tlbTable->tlb_miss_count);
    // Iterative through page table to find the demanding page.
    entry_node = pageTable->entryList->next;
    while (entry_node) {
        if (entry_node->page_index == page_number) {
            frame_number = entry_node->frame_index;
            break;
        }

        entry_node = entry_node->next;
    }

    if(frame_number == -1) {
        Page table fault. Fetch frame from secondary storage,
        update page table.
    }
}
```

和 TLB 一样, 页表也使用带头结点的链表实现, 首先遍历页表查找目标页是否在页表中. 如果在, 取帧号, 用帧号和帧内偏移访问 DRAM 取回数据; 如果不在, 触发 Page fault, 从 Secondary storage 中调页. Page fault 处理过程如下.

```
// Page table fault.
if (frame_number == -1) {
    // Increment the number of page faults.
    pageTable->page_fault_count++;
    // printf("Page table miss count: %d\n",
    // pageTable->page_fault_count);

    // Read from secondary storage, and count time elapsed.
    int start = clock();
    read_from_store(page_number);
    cpu_time_elapsed += (double)(clock() - start) / CLOCKS_PER_SEC;

    secondary_storage_call_count++;

    // Set the frame_number to current next_frame index.
    frame_number = next_frame - 1;
}
```

首先 page fault 计数器自增, 然后从 secondary storage 中取目的帧并计时. next_frame 记录 DRAM 中下一个空闲帧的偏移. 取回帧后, 更新 frame_number 为分配给新取帧的帧号.

2.2.4.1 调页过程 从 secondary storage 使用页号调页的过程如下.

```
void read_from_store(int page_number) {
    // Seek to byte PAGE_READ_SIZE in secondary storage.
    // SEEL_SET lets fseek() to seek from the beginning of file.
    if (fseek(secondary_storage, page_number * PAGE_READ_SIZE, SEEK_SET) !=
        0) {
        fprintf(stderr, "Error seeking in secondary storage.\n");
    }

    // Now read PAGE_READ_SIZE bytes from secondary storage to file_read_buf.
    if (fread(file_read_buf, sizeof(signed char), PAGE_READ_SIZE,
              secondary_storage) == 0) {
```

```c
        fprintf(stderr, "Error reading secondary storage.\n");
    }

    // Load bytes into the first available fram in the physical memory 2D
        array.
    for (int i = 0; i < PAGE_READ_SIZE; i++) {
        dram[next_frame][i] = file_read_buf[i];
    }

    // Load the frame number into page table in the next page.
    EntryNode *entry_node = pageTable->entryList->next;
    for (int i = 0; i < next_page; i++) {
        entry_node = entry_node->next;
    }

    entry_node->page_index = page_number;
    entry_node->frame_index = next_frame;

    // Increment counters to track the next available frame.
    next_frame++;
    next_page++;
}
```

首先从 secondary storage 的开头处查找 page_number * PAGE_READ_SIZE 处是否存在内容. 如果存在, 读取一帧内容到 file_read_buf. 读取内容长度 PAGE_READ_SIZE 宏被设置为与帧大小相同, 均为 256, 因此是一次读取一帧. 读取成功后, 将读取缓存中的 256 个字节转移到 DRAM 中偏移 next_frame 处. next_frame 即为分配的新帧号. 这是一个全局变量, 每次页表 page fault 调页到 DRAM 的时候 next_frame 都会自增, 指示下一次出现 page fault 的时候要分配的的帧号. 调取帧到 DRAM 中后, 更新页表的 next_page 表项. 这是新增的表项, 其页号为请求虚拟地址的页号 page_number, 对应的帧号为新分配的帧号 next_frame. 完成上述过程后, next_frame 和 next_page 自增. 调页操作完成.

2.2.4.2 TLB 的 FIFO 置换

当 TLB 未命中的时候, 需要对 TLB 进行页置换. 本实验实现了 FIFO 和 LRU 两种页置换算法. 对 FIFO 策略实现如下.

```c
// TLB fifo insert method.
void tlb_fifo_insert(int page_number, int frame_number) {
    int i = 0;
    EntryNode *entry_node = tlbTable->entryList->next;
    EntryNode *prev_node = entry_node->prev;
    // Break if page already in TLB.
    for (i = 0; i < next_tlb_entry; i++) {
        if (entry_node->page_index == page_number) {
            break;
        }

        prev_node = entry_node;
        entry_node = entry_node->next;
    }

    // Page not found in TLB.
    if (i == next_tlb_entry) {
        entry_node = prev_node;
        if (next_tlb_entry == tlbTable->length) {
            // TLB table is full, replace first entry.

            // Allocate new node.
            EntryNode *new_enode = new_node();
            new_enode->page_index = page_number;
            new_enode->frame_index = frame_number;

            // Append new node to the end of TLB.
            insert_node(entry_node, new_enode);

            // Delete first node.
            delete_next_node(tlbTable->entryList);
        } else {
            // TLB table is not full, append to end of table.
            EntryNode *next_node = entry_node->next;
            next_node->page_index = page_number;
            next_node->frame_index = frame_number;
            next_node->age = 0;
        }
    } else {
        // If another frame with the same page index already in TLB.

        // Go to last page in TLB.
        for (; i < next_tlb_entry - 1; i++) {
            entry_node = entry_node->next;
        }

        // Append page to end of TLB.
        EntryNode *new_enode = new_node();
        new_enode->page_index = page_number;
        new_enode->frame_index = frame_number;
        insert_node(entry_node, new_enode);

        // Delete the node with the same page index.
        delete_next_node(prev_node);
    }

    // Increment next tlb entry.
    if (next_tlb_entry < tlbTable->length) {
        next_tlb_entry = next_tlb_entry + 1;
    }
}
```

首先遍历 TLB, 查看待插入表项页号是否已经在 TLB 表中. 如果已经在, 将原表项删除, 新表项插入表尾. 如果不存在, 判断 TLB 表项数量. 如果 TLB 已满, 则删除最早进图 TLB 的表项, 将新表项插入表尾; 如果 TLB 不满, 将表项直接插入表尾.

2.2.4.3 TLB 的 LRU 换页

实验利用表项中的 age 参数实现了 LRU 换页. 具体过程如下.

```
void tlb_lru_insert(int page_number, int frame_number) {
    Boolean free_spot_found = False;
    Boolean already_here = False;
    EntryNode *entry_node = tlbTable->entryList->next;
    EntryNode *to_replace_node = NULL;
```

```c
    // Find the index to replace and increment age for all other entries.
    for (int i = 0; i < TLB_SIZE; i++) {
        if ((entry_node->page_index) &&
            (entry_node->page_index != page_number)) {
            // If entry does not exist in TLB and is not a free spot,
                increment
            // its age.
            entry_node->age++;
        } else if (entry_node->page_index == 0) {
            // A free entry in TLB.
            if (free_spot_found == False) {
                to_replace_node = entry_node;
                free_spot_found = True;

                // Don't exit for need to increment age for remaining entries.
            }
        } else if (entry_node->page_index == page_number) {
            // Entry already in TLB, reset its age.
            if (already_here == False) {
                entry_node->age = 0;
                already_here = True;
            } else {
                // Duplicate entry with the same page number.
                fprintf(stderr,
                        "Error: Same page appeared twice in LRU TLB.\nPage "
                        "number[%d]",
                        page_number);
            }
        }

        entry_node = entry_node->next;
    }

    // Replacement.
    if (already_here == True) {
        // Already in TLB, do nothing.
        return;
    } else if (free_spot_found == True) {
        // Free entry available in TLB.
        to_replace_node->page_index = page_number;
        to_replace_node->frame_index = frame_number;
        to_replace_node->age = 0;
    } else {
        // No free entry available in TLB, replace the oldest entry.
        to_replace_node = get_oldest_entry(TLB_SIZE);
        to_replace_node->page_index = page_number;
        to_replace_node->frame_index = frame_number;
        to_replace_node->age = 0;
    }
}
```

遍历 TLB, 对页号不是待插入页号, 且非空的表项, 增加其 age; 如果找到了页号和待插入页号相同的表项, 重置其 age, 不做其他操作; 如果找到了空表项, 记录此空表项在 to_replace_node, 待更新完所有表项的 age 之后将新表项插入到此处.

如果表项既不已经存在于 TLB 中, 且也没有空表项, 则使用 get_oldest_entry() 函数找到 TLB 中的最老表项, 并将次表项替换为待插入表项, 重置其 age.

3 实验结果

3.1 基本内存分配实验

3.1.1 运行环境

程序在以下环境下构建和测试.

操作系统 macOS 13.1 22C5044e arm64;

c 编译器 gcc (Homebrew GCC 12.2.0) 12.2.0;

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

调试工具 Visual Studio Code Version: 1.74.0 (Universal), lldb-1400.0.38.13

3.1.2 程序运行方式

程序使用 make 进行构建和测试. 读者优先程序和写者优先程序都会被构建在 build/目录下.

对于程序的构建, 执行如下命令.

```
make
```

对于程序的执行, 可以使用如下命令:

```
build/main
```

建议使用测试的方式运行模拟程序, 使用以下命令:

```
make test FROM=0 TO=1000 NUM_BLOCKS=1000 NUM_PROCESSES=100
```

在这种情况下, 将会先使用数据生成器 data_gen 根据后面的参数生成配置文件到 config/config.txt, 然后再执行程序. 在参数中, FROM 和 TO 指定进程请求内存大小和内存孔大小的下界和上界, NUM_BLOCKS 指定内存孔的数量, NUM_PROCESSES 指定请求内存的进程的数量.

之后程序会读取 config/config.txt 配置文件, 加载进程数量, 每个进程请求内存大小, 内存孔数量, 每个内存孔大小. 进入后可以看到菜单如下, 输入对应标号可以进入对应界面.

```
MEMORY MANAGEMENT ALLOCATION SIMULATION
Options:
0. Customize configuration
1. First Fit
2. Best Fit
q. Exit
```

- 输入 0 可以重新设置进程数量, 每个进程请求内存大小等内容. 设置仅对本次执行有效.
- 输入 1 将模拟 First Fit 分配.
- 输入 2 将模拟 Best Fit 内存分配.
- 输入 q 将退出程序.

3.1.3 程序日志格式

程序将随内存分配将分配结果输出到 console. 格式为

```
process[<pid>] in block[<bid>] <psize>/<bsize>
```

其中, pid 为进程 id, bid 为内存孔 id, psize 为进程申请空间大小, bsize 为内存孔大小.

如果进程没有合适的孔可以分配, 则输出日志

```
process[<pid>] is not assigned.
```

在内存分配结束后, 将输出内部碎片和外部碎片统计信息, 例如

```
------FRAGMENTATION STATISTICS------

Total internal fragmentation: 10910
Total external fragmentation: 7283
```

表示总的内部碎片大小为 10910 字节, 外部碎片大小为 7283 字节.

3.1.4 运行结果

```
# Build
make test FROM=0 TO=1000 NUM_BLOCKS=1000 NUM_PROCESSES=1000

# Run First fit simulation.
1

# Run Best fit simulation.
2

# Exit.
q
```

设定进程申请内存大小和内存孔大小范围在 0 到 1000 的情况下,

- 10000 进程, 10000 个内存孔 First fit 的内部和外部碎片统计如图 1.
- 10000 进程, 10000 个内存孔 Best fit 的内部和外部碎片统计如图 2.
- 5000 进程, 10000 个内存孔 First fit 的内部和外部碎片统计如图 3.
- 5000 进程, 10000 个内存孔 Best fit 的内部和外部碎片统计如图 4.

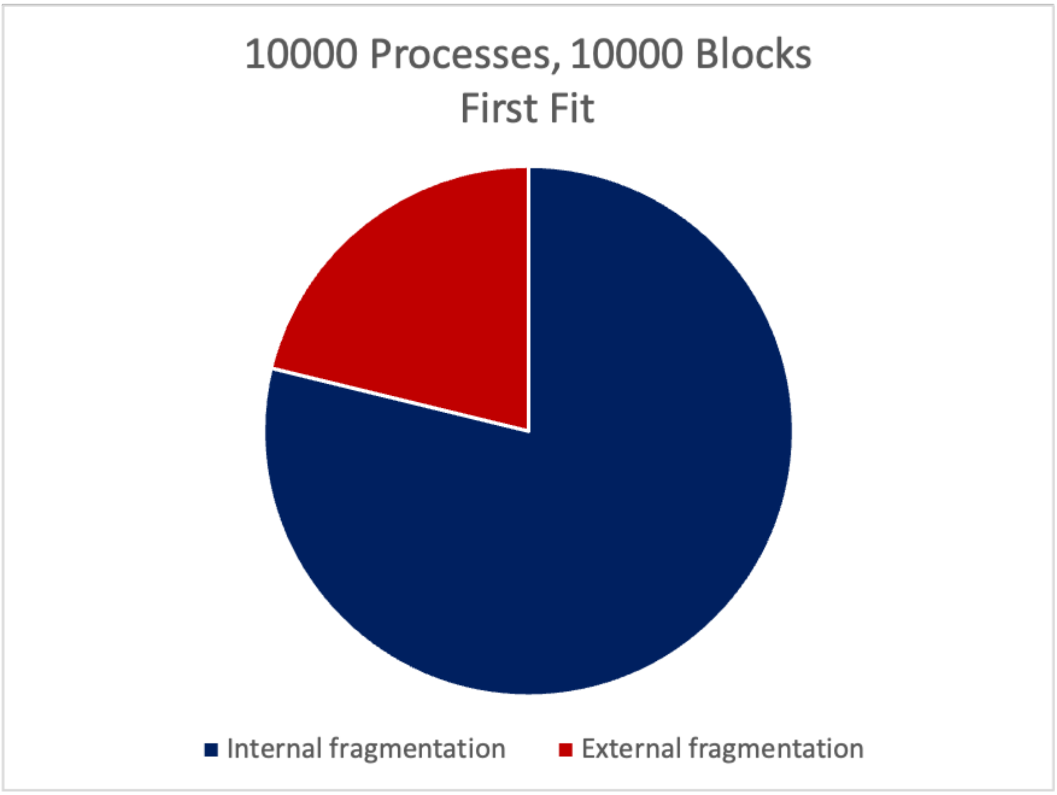

Figure 1: 10000 进程,10000 内存孔 First Fit

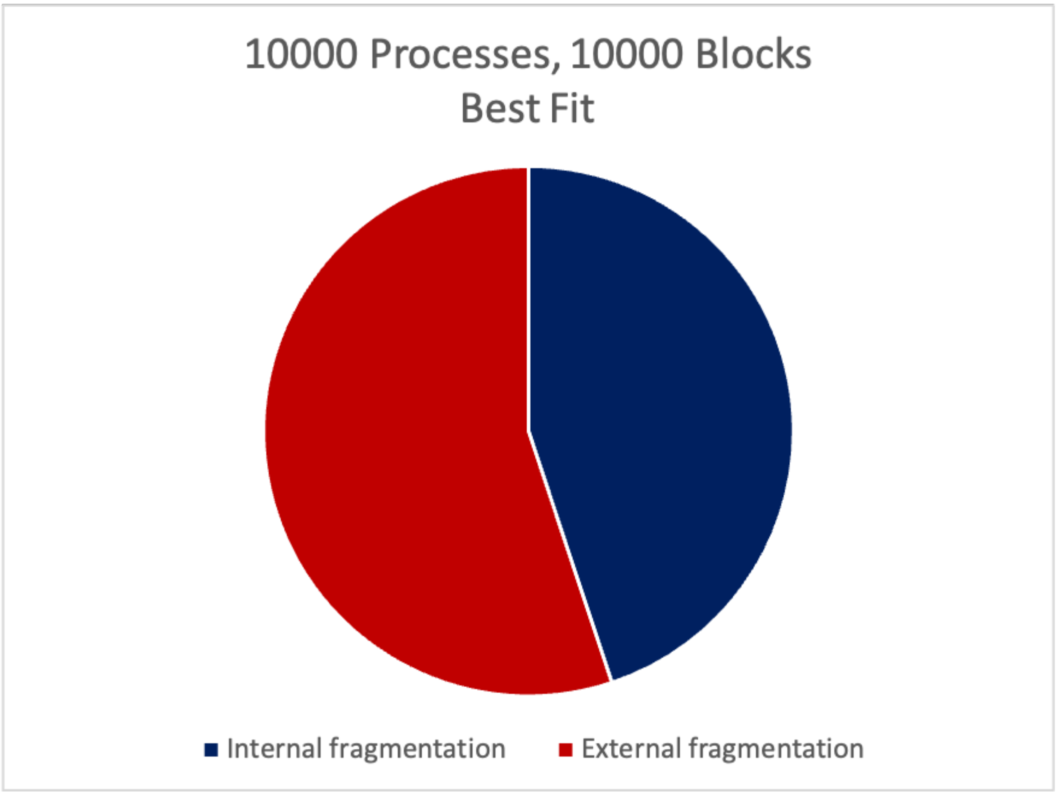

Figure 2: 10000 进程,10000 内存孔 Best Fit

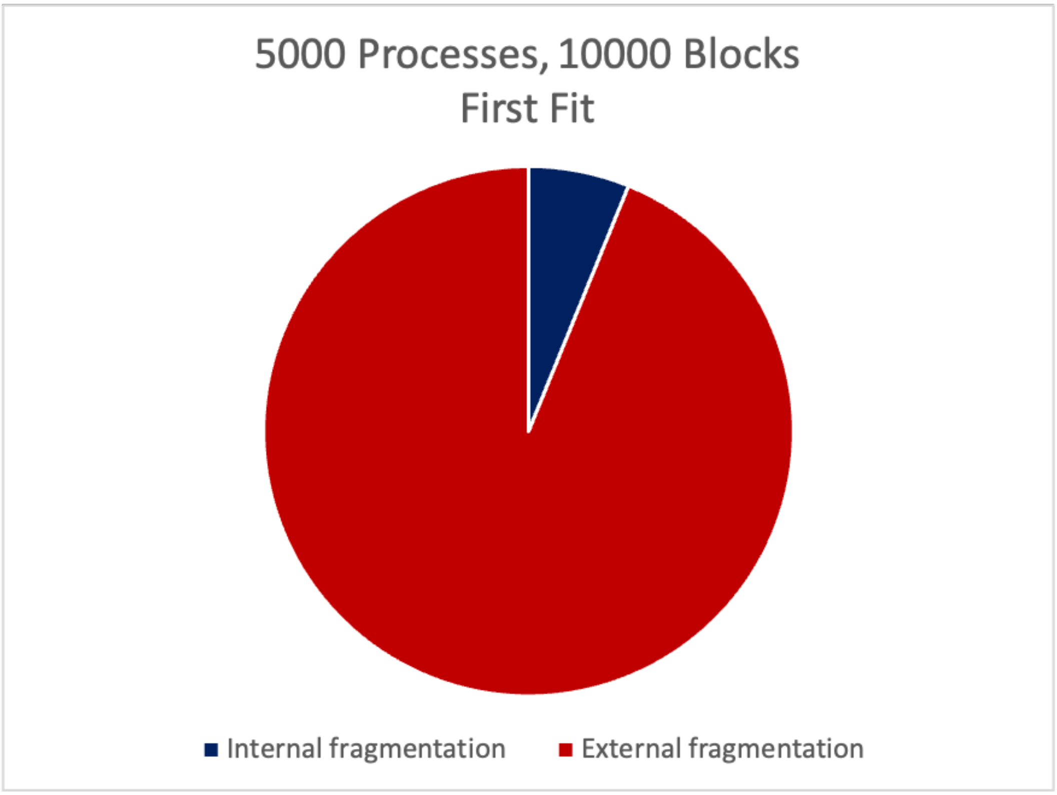

Figure 3: 5000 进程,10000 内存孔 First Fit

- 10000 进程, 5000 个内存孔 Best fit 的内部和外部碎片统计如图 5.

对比进程数和内存孔数均为 10000 的情况下的 First Fit 和 Best Fit, 可以发现在进程的总大小和所有内存孔的总大小相同的情况下, First Fit 的 Internal Fragmentation 相比 Best Fit 方法的开销大得多.

而在进程数远少于内存孔的 5000 进程 10000 内存孔的情况下, Best Fit 能几乎完全的消除内部碎片, 而 First Fit 仍有不小数量的内部碎片存在.

当进程数量远大于内存孔数量的时候, 即便是 First Fit 也能几乎占用所有的内存孔.

3.2 虚拟内存分配实验

3.2.1 运行环境

程序在以下环境下构建和测试.

操作系统 macOS 13.1 22C5044e arm64;

c 编译器 gcc (Homebrew GCC 12.2.0) 12.2.0;

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

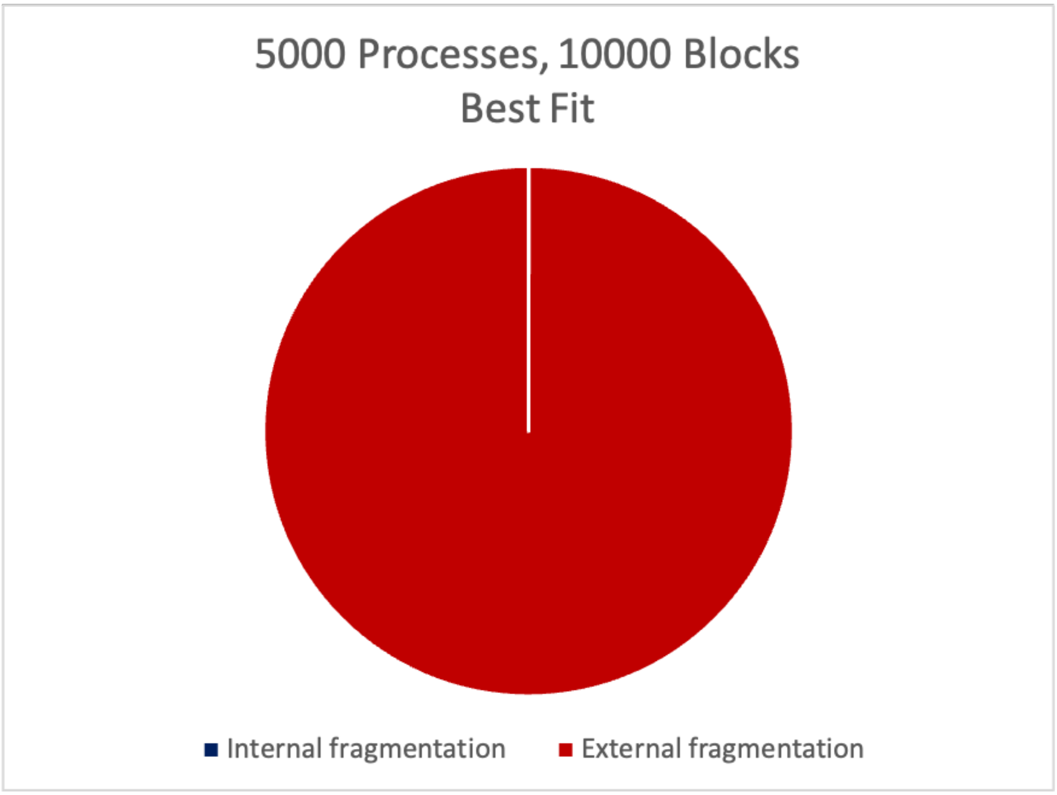

Figure 4: 5000 进程,10000 内存孔 Best Fit

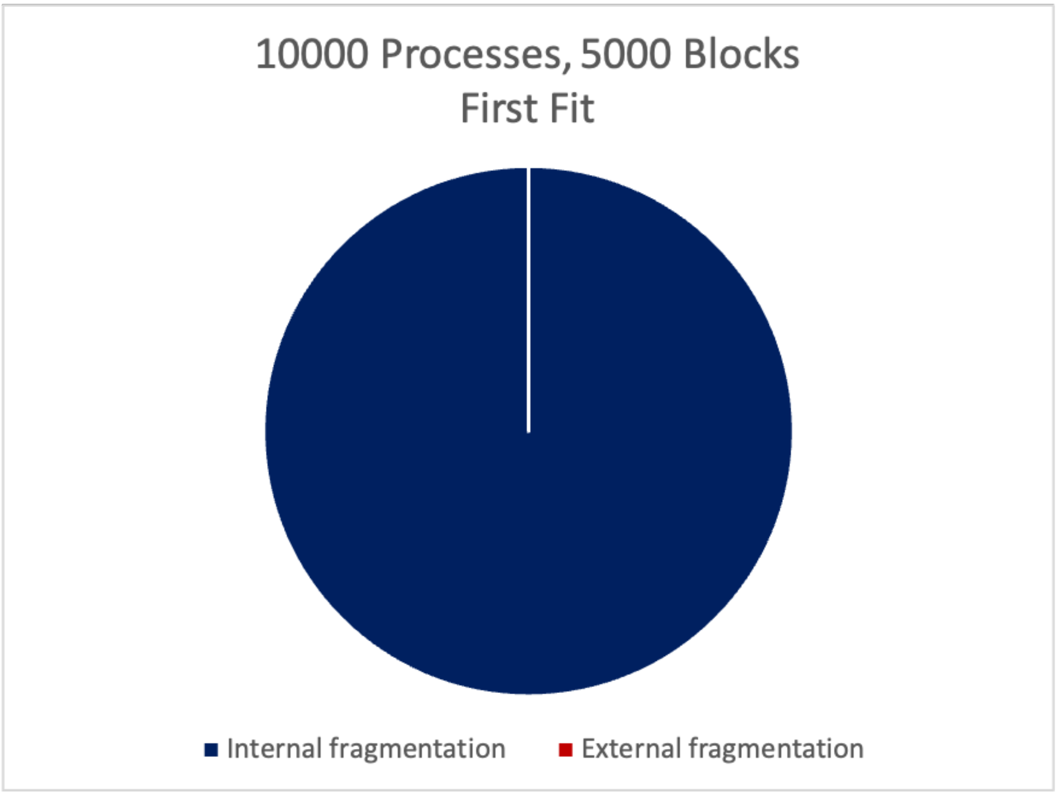

Figure 5: 10000 进程,5000 内存孔 First Fit

文本编辑器 nvim v0.8.1

调试工具 Visual Studio Code Version: 1.74.0 (Universal), lldb-1400.0.38.13

3.2.2 程序运行方式

程序使用 make 进行构建和测试. 读者优先程序和写者优先程序都会被构建在 build/目录下.

对于程序的构建, 执行如下命令.

```
make
```

对于程序的执行, 可以使用如下命令:

```
Usage: build/main <input file> <verbose[1/0]> <replacement method[1: FIFO,
    2: LRU]>
```

其中, <input file> 为待访问的虚拟内存地址队列, verbose 控制输出日志是详细 (1) 还是简略 (0); replacement method 控制 TLB 的置换方式是 FIFO 还是 LRU.

程序的测试使用本实验同时实现的数据生成器 data_gen, 根据**局部性原理**生成符合实际情况的待访问虚拟内存地址队列, 生成的数据将保存在 config/input.txt 中. 可使用如下命令.

```
make test AMOUNT=100000 UPPER_BORDER=256 VERBOSE=0
```

执行 make test 的时候, 首先会使用 data_gen 生成 AMOUNT 个待访问虚拟地址, 这些虚拟地址连续最多有 UPPER_BORDER 个在同一个帧中. VERBOSE=1 可以输出更详细的日志, 即虚拟地址, 转换到的物理地址, 以及内存中物理地址对应存储的数据.

3.2.3 程序日志格式

程序的日志有 3 种等级.

一级是最详细的日志, 包括虚拟地址, 物理地址和内存中物理地址对应存储的数据等信息, 当 VERBOSE=1 的时候输出此等级 log. 截取示例如下.

```
VIRTUAL MEMORY SIMULATION

Logical pages: 256
Page size: 256 bytes.
Page table size: 256
TLB size: 16 entries.
Physical frames: 256
Frame size: 256 bytes.
Physical memory size: 65536 bytes.
Virtual address[0x0003] Physical address[0x0003] Value[0000]
Virtual address[0x0103] Physical address[0x0003] Value[0120]
Virtual address[0x0203] Physical address[0x0103] Value[0070]
Virtual address[0x0303] Physical address[0x0203] Value[0016]
Virtual address[0x0403] Physical address[0x0303] Value[0095]
Virtual address[0x0503] Physical address[0x0403] Value[0090]
Virtual address[0x0007] Physical address[0x0007] Value[0017]
Virtual address[0x0107] Physical address[0x0007] Value[0017]
Virtual address[0x0207] Physical address[0x0107] Value[0002]
Virtual address[0x0307] Physical address[0x0207] Value[0097]
Virtual address[0x0407] Physical address[0x0307] Value[-122]
Virtual address[0x0507] Physical address[0x0407] Value[-107]
Virtual address[0x0003] Physical address[0x0003] Value[0120]
Virtual address[0x0103] Physical address[0x0003] Value[0120]
Virtual address[0x0203] Physical address[0x0103] Value[0070]
Virtual address[0x0303] Physical address[0x0203] Value[0016]
Virtual address[0x0403] Physical address[0x0303] Value[0095]
Virtual address[0x0503] Physical address[0x0403] Value[0090]
Virtual address[0x0007] Physical address[0x0007] Value[0017]
Virtual address[0x0107] Physical address[0x0007] Value[0017]
Virtual address[0x0207] Physical address[0x0107] Value[0002]
Virtual address[0x0307] Physical address[0x0207] Value[0097]
Virtual address[0x0407] Physical address[0x0307] Value[-122]
Virtual address[0x0507] Physical address[0x0407] Value[-107]
```

二级是总结性日志, 包括本次运行使用的 TLB 替换算法名称, 访问的虚拟内存地址数量, Page fault 次数, Page fault 比例, TLB 命中次数, TLB 命中率. 截取部分日志示例如下.

```
------VIRTUAL MEMORY STATISTICS------

Replacement method: FIFO
Number of addresses translated: 24
Page faults: 5
Page fault rate: 20.833%
TLB hits: 19
TLB hit rate: 79.167%
Average thme retrieving data from secondary storage: 4.600 ms

-------------------------------------
```

第三级日志很简略, 每次程序运行会追加一行到日志文件 logs/stat.log 的结尾, 一行共三个整数, 分别为访问的虚拟地址个数, Page fault 次数和 TLB 命中次数, 示例如下.

```
24 5 19
```

表示进行了 24 次虚拟内存访问, 有 5 次 page fault, 19 次 TLB 命中.

3.2.4 运行结果

在 TLB 表大小为 32 的情况下, 分别采用 FIFO 和 LRU 的 TLB 替换策略, 设置不同的连续同帧虚拟地址访问请求上界 UPPER_BORDER 进行测试, 具体运行命令如下.

```
# 连续同帧虚拟地址上界32768
make test UPPER_BORDER=32768

# 连续同帧虚拟地址上界16384
make test UPPER_BORDER=16384

# 连续同帧虚拟地址上界4096
make test UPPER_BORDER=4096
```

```
# 连续同帧虚拟地址上界1024
make test UPPER_BORDER=1024

# 连续同帧虚拟地址上界256
make test UPPER_BORDER=256

# 连续同帧虚拟地址上界64
make test UPPER_BORDER=64

# 连续同帧虚拟地址上界16
make test UPPER_BORDER=16

# 连续同帧虚拟地址上界4
make test UPPER_BORDER=4

# 连续同帧虚拟地址上界2
make test UPPER_BORDER=2

# 连续同帧虚拟地址上界1
make test UPPER_BORDER=1
```

得运行日志和 Page fault, TLB hit 统计如下.

- 统计信息日志为 logs/test_stat.log.
- Page fault 折线图如图 6.
- TLB 命中折线图如图 7.

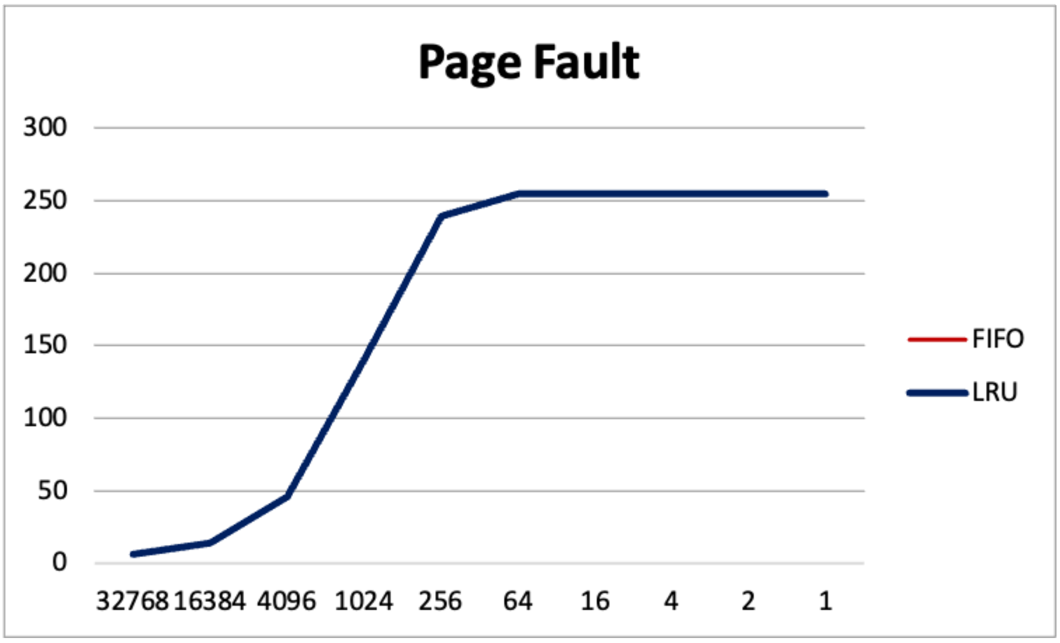

Figure 6: Page Fault 统计图

可以看到, 当帧内连续虚拟地址非常大的时候, 由于一帧内只有 256 个编址, 因此 Page fault 次数很低; 随着帧内连续访问虚拟地址数量下降, Page Fault 的次数迅速增加, 当将 secondary storage 中全部 256 个帧调入 DRAM 后, Page fault 不再增加.

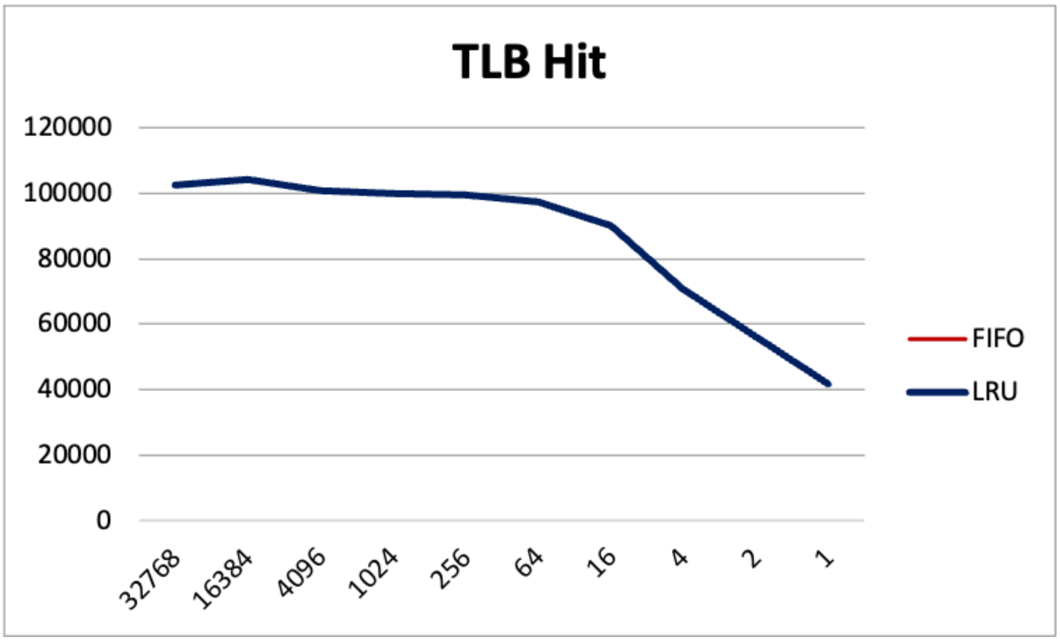

Figure 7: TLB 命中统计图

当帧内连续虚拟地址非常大的时候, TLB 命中率很高; 随着帧内连续访问虚拟地址数量下降, TLB 命中次数也逐渐下降.

此外, 还可以看到 FIFO 和 LRU 的 TLB 替换策略在 Page fault 和 TLB 命中率上表现类似, 猜测可能是生成的数据不能很好的利用 LRU 的特性, 使得 FIFO 已经可以有足够好的表现.

4 实验总结

通过这次实验, 我实践并测试了基本内存分配和虚拟内存分配管理, 操作系统的内存分配和管理有了更加直观和亲身的认识. 通过基本内存分配, 比较了 First Fit 和 Best Fit 内存分配方法的异同优劣; 通过虚拟内存分配管理实验, 亲身体验了 TLB, 页表和内存, 以及二级存储的协作处理虚拟地址的过程. 通过虚拟内存转换, 有利于更加有效的使用物理内存, 且对程序屏蔽了物理内存的细节, 也更加方便了程序的编写工作.

在构建和管理项目的工具方面, 使用了 make 工具, 方便的实现了对程序的构建和测试; 此外, 对于文档, 也使用了 make 工具作为管理, 很大程度上方便了实验报告修改后的再次编译. 在实验报告编写方面, 练习了 latex 的使用, 对 latex 语法和使用更加熟悉.

但是, 由于时间限制, 没有来得及添加更多内容, 以及进行更多测试. 例如, 在虚拟内存分配中, 没有来得及实现多个进程的虚拟内存空间. 但是即便如此, 也已经有了丰富的收获.

总的来说, 这次实验在内存分配和管理之内及之外都学到了许多知识, 收获颇丰.

5 算法源代码

5.1 基本内存分配管理

5.1.1 程序入口

```c
#include "schemas.h"
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    char ch = 'i'; // Initialize option variable.
    int num_processes = 0, num_blocks = 0;
    Process *p = NULL;
    Block *b = NULL;

    load_config(&p, &b, &num_processes, &num_blocks);

    do {
        print_menu();

        ch = get_valid_option();

        switch (ch) {
        case '0':
            config(&p, &b, &num_processes, &num_blocks);
            assert(p != NULL);
            assert(b != NULL);
            break;

        case '1':
            re_init(p, b, num_processes, num_blocks);
            printf("\nFirst fit.\n");
            first_fit(p, b, num_processes, num_blocks);
            break;

        case '2':
            re_init(p, b, num_processes, num_blocks);
            printf("\nBest fit.\n");
            best_fit(p, b, num_processes, num_blocks);
            break;

        case 'q':
            break;

        default:
            printf("\nInvalid option.");
        }
    } while (valid_option(ch));

    // Free allocated heap memory space.
    if (p != NULL) {
        free(p);
    }
    if (b != NULL) {
        free(b);
    }
    return 0;
}
```

5.1.2 内存分配方案

```
#ifndef SCHEMAS_H
#define SCHEMAS_H
#include "util.h"

void fragmentation_statistics(int in_frag, Block *b, int m);
void re_init(Process *p, Block *b, int n, int m);
void first_fit(Process *p, Block *b, int n, int m);
void best_fit(Process *p, Block *b, int n, int m);
#endif // !SCHEMAS_H
```

```
#include "schemas.h"
#include <assert.h>
#include <limits.h>
#include <stdio.h>

void re_init(Process *p, Block *b, int n, int m) {
    assert(p != NULL);
    assert(b != NULL);
    for (int i = 0; i < n; i++) {
        p[i].pflag = 0;
    }

    for (int i = 0; i < m; i++) {
        b[i].bflag = 0;
    }
}

void fragmentation_statistics(int in_frag, Block *b, int m) {
    int ex_frag = 0;
    printf("\n\n ------FRAGMENTATION STATISTICS------");
    printf("\n\n Total internal fragmentation: %d", in_frag);
    for (int j = 0; j < m; j++) {
        if (b[j].bflag == 0) {
            ex_frag += b[j].bsize;
        }
    }

    printf("\n Total external fragmentation: %d", ex_frag);
}

void first_fit(Process *p, Block *b, int n, int m) {
    int in_frag = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (p[i].psize <= b[j].bsize && b[j].bflag == 0 &&
                p[i].pflag == 0) {
                b[j].bflag = p[i].pflag = 1;
                in_frag += b[j].bsize - p[i].psize;
                printf("\n process[%d] in block[%d] %d/%d", i, j, p[i].psize,
                        b[j].bsize);

                break;
            }
        }

        if (p[i].pflag == 0) {
            printf("\n process[%d] is not assigned.", i);
        }
    }

    fragmentation_statistics(in_frag, b, m);
}

void best_fit(Process *p, Block *b, int n, int m) {
    int in_frag = 0;
    int min_frag = INT_MAX;
    int bid = INT_MAX;
    for (int i = 0; i < n; i++) {
        min_frag = INT_MAX;
        bid = INT_MAX;
        for (int j = 0; j < m; j++) {
            int tmp_in_frag = b[j].bsize - p[i].psize;
            if (tmp_in_frag >= 0 && b[j].bflag == 0 && min_frag >
                tmp_in_frag) {
                min_frag = tmp_in_frag;
                bid = j;
            }
        }

        if (min_frag != INT_MAX) {
            b[bid].bflag = 1;
            p[i].pflag = 1;
            in_frag += b[bid].bsize - p[i].psize;
            printf("\n process[%d] in block[%d] %d/%d", i, bid, p[i].psize,
                   b[bid].bsize);
        }

        if (p[i].pflag == 0) {
            printf("\n process[%d] is not assigned.", i);
        }
    }

    fragmentation_statistics(in_frag, b, m);
}
```

5.1.3 Utils

```
#ifndef UTIL_H
#define UTIL_H

#include <stdint.h>
#ifndef BOOLEAN
#define TRUE 1
#define FALSE 0
#endif
```

```c
typedef struct Process {
    int psize;
    int pflag;
} Process;

typedef struct Block {
    int bsize;
    int bflag;
} Block;

typedef uint8_t Boolean;

void print_menu();
Boolean config(Process **p, Block **b, int *n, int *m);
Boolean load_config(Process **p, Block **b, int *num_processes,
                   int *num_blocks);
Boolean valid_option(char ch);

char get_valid_option();
#endif // !UTIL_H
```

```c
#include "util.h"
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void print_menu() {
    printf("\n\nMEMORY MANAGEMENT ALLOCATION SIMULATION");
    printf("\nOptions:");
    printf("\n0. Customize configuration");
    printf("\n1. First Fit");
    printf("\n2. Best Fit");
    printf("\nq. Exit\n");
}

Boolean config(Process **p, Block **b, int *n, int *m) {

    // Free allocated heap memory space.
    if (*p != NULL) {
        free(*p);
    }
    if (*b != NULL) {
        free(*b);
    }

    printf("\nEnter number of blocks: ");
    scanf("%d", m);
    *b = (Block *)malloc((*m) * sizeof(Block));
    if (b == NULL) {
        printf("\nError: failed to allocate memory for blocks.\n");
        return FALSE;
    }
    for (int i = 0; i < *m; i++) {
        printf("Enter size of block[%d]: ", i);
        scanf("%d", &(*b)[i].bsize);
    }
    printf("\nEnter number of processes: ");
    scanf("%d", n);
    *p = (Process *)malloc((*n) * sizeof(Process));
    if (p == NULL) {
        printf("\nError: failed to allocate memory for processes.\n");
        return FALSE;
    }
    for (int i = 0; i < *n; i++) {
        printf("Enter size of process[%d]: ", i);
        scanf("%d", &(*p)[i].psize);
    }

    return TRUE;
}

Boolean load_config(Process **p, Block **b, int *n, int *m) {
    char filename[] = "config/config.txt";
    FILE *config_file = fopen(filename, "r");
    if (config_file == NULL) {
        printf("\nError: failed to load config from file.");
        return FALSE;
    }
    printf("\nLoad config from %s.", filename);

    // Read config from file.
    fscanf(config_file, "%d", m);
    *b = (Block *)malloc(*m * sizeof(Block));
    if (b == NULL) {
        printf("\nError: failed to allocate memory for blocks.\n");
        return FALSE;
    }
    for (int i = 0; i < *m; i++) {
        fscanf(config_file, "%d", &(*b)[i].bsize);
    }

    fscanf(config_file, "%d", n);
    *p = (Process *)malloc(*n * sizeof(Process));
    if (p == NULL) {
        printf("\nError: failed to allocate memory for processes.\n");
        return FALSE;
    }
    for (int i = 0; i < *n; i++) {
        fscanf(config_file, "%d", &(*p)[i].psize);
    }

    return TRUE;
}

Boolean valid_option(char ch) {
    switch (ch) {
    case '0':
    case '1':
    case '2':
    case '3':
    case 'i':
        return TRUE;
    case 'q':
    default:
        return FALSE;
    }
}

char get_valid_option() {
    char ch;
    ch = getchar();
    while (ch == '\n' || ch == ' ') {
        ch = getchar();
    }
    return ch;
}
```

5.1.4 数据生成器

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv) {
    if (argc < 3) {
        printf("Not enough arguments give.\n");
        printf("Usage: data_gen <start> <end> <number of blocks> <number of "
               "processes>.\n");
        printf("Default write to config/config.txt.\n");
    }

    char filename[] = "config/config.txt";
    FILE *fout = fopen(filename, "w");

    int low = atoi(argv[1]);
    int high = atoi(argv[2]);
    int num_blockes = atoi(argv[3]);
    int num_processes = atoi(argv[4]);

    srand(time(NULL));

    fprintf(fout, "%d\n", num_blockes);
    for (int i = 0; i < num_blockes; i++) {
        int rnd_num = (rand() % (high - low + 1) + low);
        fprintf(fout, "%d ", rnd_num);
    }

    fprintf(fout, "\n%d\n", num_processes);
    for (int i = 0; i < num_processes; i++) {
        int rnd_num = (rand() % (high - low + 1) + low);
        fprintf(fout, "%d ", rnd_num);
    }
    putchar('\n');

    return 0;
}
```

5.2 虚拟内存分配管理

5.2.1 程序入口

```c
#include "vmem.h"
#include "vmem_sim.h"
#include <stdio.h>
#include <stdlib.h>

char address_read_buf[MAX_ADDR_LEN];

int main(int argc, char *argv[]) {

    /**
     * vmemTable is a generic struct defined to implement any number of
         caches
     * for logical address translation.
     */
    tlbTable = create_vmem_table(TLB_SIZE);
    pageTable = create_vmem_table(PAGE_TABLE_SIZE);
    dram = dram_allocate(TOTAL_FRAME_COUNT, FRAME_SIZE);

    if (argc != 4) {
        fprintf(stderr, "Usage: build/main <input file> <verbose[1/0]> "
                        "<replacement method[1: FIFO, 2: LRU]>\n");
        return -1;
    }

    // Open secondary storage file.
    printf("Load secondary storage file %s\n", secondary_storage_name);
    secondary_storage = fopen(secondary_storage_name, "rb");
    if (secondary_storage == NULL) {
        fprintf(stderr, "Error opening %s\n", secondary_storage_name);
        return -1;
    }

    // Open the file containing logical addresses.
    printf("Load address file %s\n", argv[1]);
    address_file = fopen(argv[1], "r");
    display_option = atoi(argv[2]);
    replace_method = atoi(argv[3]);

    if (address_file == NULL) {
        fprintf(stderr,
                "Error opening %s. Expecting input file containing logical "
                "addresses.\n",
                argv[1]);

        return -1;
    }

    printf("\nVIRTUAL MEMORY SIMULATION\n");
    printf("\nLogical pages: %d\n", PAGE_TABLE_SIZE);
    printf("Page size: %d bytes.\n", PAGE_READ_SIZE);
    printf("Page table size: %d\n", PAGE_TABLE_SIZE);
    printf("TLB size: %d entries.\n", TLB_SIZE);
    printf("Physical frames: %d\n", TOTAL_FRAME_COUNT);
    printf("Frame size: %d bytes.\n", FRAME_SIZE);
    printf("Physical memory size: %d bytes.\n", PAGE_READ_SIZE * FRAME_SIZE);

    // do {
    // printf("\nDisplay all physical addresses? [y/n]: ");
    // if ((display_option = getchar()) == '\n' || display_option == ' ') {
    // display_option = getchar();
    // }
    // } while (display_option != 'n' && display_option != 'y');

    // do {
    // printf("Chose TLB replacement method: \n");
    // printf("1: FIFO\n");
    // printf("2: LRU\n");
    // if ((replace_method = getchar()) == '\n' || replace_method == ' ') {
    // replace_method = getchar();
    // }
    //
    // } while (replace_method != '1' && replace_method != '2');

    // virtual address.
    int virtual_addr;
    int page_number;
    int offset_number;

    // Read through the input file and print virtual addresses.
    while (fgets(address_read_buf, MAX_ADDR_LEN, address_file) != NULL) {
        virtual_addr = atoi(address_read_buf);

        // Get page number.
        page_number = get_page_number(PAGE_MASK, virtual_addr, SHIFT);

        // Get offset.
        offset_number = get_offset(OFFSET_MASK, virtual_addr);

        // Get physical address and translated value stored at that address.
        translate_address(virtual_addr, page_number, offset_number);
        translation_count++;
    }

    // Set stdout replacement method name.
    algo_name = (replace_method == FIFO) ? "FIFO" : "LRU";

    printf("\n\n------VIRTUAL MEMORY STATISTICS------\n\n");
    printf("Replacement method: %s\n", algo_name);
    printf("Number of addresses translated: %d\n", translation_count);
    double pf_rate = (double)pageTable->page_fault_count / translation_count;
    double tlb_rate = (double)tlbTable->tlb_hit_count / translation_count;

    printf("Page faults: %d\n", pageTable->page_fault_count);
    printf("Page fault rate: %.3f%%\n", pf_rate * 100);
    printf("TLB hits: %d\n", tlbTable->tlb_hit_count);
    printf("TLB hit rate: %.3f%%\n", tlb_rate * 100);
    printf("Average thme retrieving data from secondary storage: %.3f ms\n",
           get_avg_time_in_secondary_storage());
    printf("\n-------------------------------------\n\n");

    // Append statistics to file.
```

```
    FILE *stat_file = fopen("logs/stat.log", "a");
    fprintf(stat_file, "%d %d %d\n", translation_count,
           pageTable->page_fault_count, tlbTable->tlb_hit_count);

    // Close input file and secondary storage file.
    fclose(address_file);
    fclose(secondary_storage);
    fclose(stat_file);

    // Free memory allocated on heap.
    free_vmem_table(&tlbTable);
    free_vmem_table(&pageTable);
    free_DRAM(&dram, TOTAL_FRAME_COUNT);

    return 0;
}
```

5.2.2 模拟虚拟内存分配管理过程

```
#ifndef VMEM_SIM_H
#define VMEM_SIM_H

#include "vmem.h"
#include <stdio.h>
#include <time.h>
#define FRAME_SIZE 256 // Size of each frame
#define TOTAL_FRAME_COUNT 256 // Total number of frames in physical memory
#define PAGE_MASK 0xFF00 // Mask to get page number
#define OFFSET_MASK 0xFF // Mask to get offset
#define SHIFT 8 // Bitmask shift amount
#define TLB_SIZE 16 // size of TLB
#define PAGE_TABLE_SIZE 256 // size of page table
#define MAX_ADDR_LEN \
    10 // The number of characters to read for each line from
       // input file
#define PAGE_READ_SIZE 256 // Number of bytes to read

typedef enum { FIFO = 1, LRU = 2 } ReplacementMethod;
typedef enum { YES = 1, NO = 0 } DisplayOption;

typedef enum { True = 1, False = 0 } Boolean;

extern VMemTable *tlbTable;
extern VMemTable *pageTable;
extern int **dram;

extern ReplacementMethod replace_method;
extern DisplayOption display_option;

extern FILE *address_file;
extern char secondary_storage_name[];
extern FILE *secondary_storage;

extern int translation_count;

extern char *algo_name;
```

```
/**
 * Translate virtual memory address into physical memory address, retrieve
    the
 * translated value stored.
 *
 * Parameters:
 * - replace_method: Replacement method used for TLB replacement.
 * 1 is FIFO, and 2 is LRU.
 */
void translate_address(int virtual_addr, int page_number, int offset_number);

void read_from_store(int page_number);

void tlb_fifo_insert(int page_number, int frame_number);

void tlb_lru_insert(int page_number, int frame_number);

EntryNode *get_oldest_entry(int tlb_size);

double get_avg_time_in_secondary_storage();

#endif // !VMEM_SIM_H
```

```
#include "vmem_sim.h"
#include "vmem.h"
#include <stdio.h>
#include <time.h>

// Generating length of time for a function.
clock_t start, end;
double cpu_time_elapsed;
int secondary_storage_call_count = 0;

int **dram; // Physical memory.

int next_tlb_entry = 0; // Next available index of TLB entry.
int next_page = 0; // Next available page in page table.
int next_frame = 0; // Next available frame TLB or page table.

ReplacementMethod replace_method; // Menu stdin.
DisplayOption display_option; // Menu stdin.
                               //

// Input file and backup.
FILE *address_file;
char secondary_storage_name[] = "config/secondary_storage.bin";
FILE *secondary_storage;

// The translated value of the byte(signed char) in memory.
signed char translated_value;

VMemTable *tlbTable; // The TLB struct.
VMemTable *pageTable; // The Page Table.
int translation_count = 0;
char *algo_name;

// The buffer containing reads from backing store.
```

```c
signed char file_read_buf[PAGE_READ_SIZE];

void translate_address(int virtual_addr, int page_number, int offset_number)
    {
    // Try to find page in TLB.
    int frame_number = -1;

    EntryNode *entry_node = tlbTable->entryList->next;
    while (entry_node) {
        if (entry_node->page_index == page_number) {
            frame_number = entry_node->frame_index;
            tlbTable->tlb_hit_count++;
            break;
        }

        entry_node = entry_node->next;
    }

    // If page is not hit in TLB, find page in page table, and increment TLB
    // miss count.
    // If page is not found in page table, find it in secondary storage DRAM,
    // and increment page table fault count.
    if (frame_number == -1) {
        tlbTable->tlb_miss_count++;
        // printf("TLB miss count: %d\n", tlbTable->tlb_miss_count);

        // Iterative through page table to find the demanding page.
        entry_node = pageTable->entryList->next;
        while (entry_node) {
            if (entry_node->page_index == page_number) {
                frame_number = entry_node->frame_index;
                break;
            }

            entry_node = entry_node->next;
        }

        // Page table fault.
        if (frame_number == -1) {
            // Increment the number of page faults.
            pageTable->page_fault_count++;
            // printf("Page table miss count: %d\n",
            // pageTable->page_fault_count);

            // Read from secondary storage, and count time elapsed.
            int start = clock();
            read_from_store(page_number);
            cpu_time_elapsed += (double)(clock() - start) / CLOCKS_PER_SEC;

            secondary_storage_call_count++;

            // Set the frame_number to current next_frame index.
            frame_number = next_frame - 1;
        }
    }

    // Page replacement.
    if (replace_method == FIFO) {
        // FIFO replacement method to insert the page number and frame
        // number into TLB.
        tlb_fifo_insert(page_number, frame_number);
    } else {
        tlb_lru_insert(page_number, frame_number);
    }

    // Get translated value with frame_number and offset_number.
    translated_value = dram[frame_number][offset_number];

    // // Debug.
    // printf("\nFrame Number[0x%04x]\tOffset[0x%04x]\n", frame_number,
    // offset_number);

    if (display_option == YES) {
        // Print the virtual address, physical address and translated value
            of
        // the signed char.
        printf("Virtual address[0x%04x]\t\tPhysical "
               "address[0x%04x]\tValue[%04d]\n",
               virtual_addr, (frame_number << SHIFT) | offset_number,
               translated_value);
    }
}

void read_from_store(int page_number) {
    // Seek to byte PAGE_READ_SIZE in secondary storage.
    // SEEL_SET lets fseek() to seek from the beginning of file.
    if (fseek(secondary_storage, page_number * PAGE_READ_SIZE, SEEK_SET) !=
        0) {
        fprintf(stderr, "Error seeking in secondary storage.\n");
    }

    // Now read PAGE_READ_SIZE bytes from secondary storage to file_read_buf.
    if (fread(file_read_buf, sizeof(signed char), PAGE_READ_SIZE,
              secondary_storage) == 0) {
        fprintf(stderr, "Error reading secondary storage.\n");
    }

    // Load bytes into the first available fram in the physical memory 2D
        array.
    for (int i = 0; i < PAGE_READ_SIZE; i++) {
        dram[next_frame][i] = file_read_buf[i];
    }

    // Load the frame number into page table in the next page.
    EntryNode *entry_node = pageTable->entryList->next;
    for (int i = 0; i < next_page; i++) {
        entry_node = entry_node->next;
    }

    entry_node->page_index = page_number;
    entry_node->frame_index = next_frame;

    // Increment counters to track the next available frame.
    next_frame++;
    next_page++;
}
```

```c
// TLB fifo insert method.
void tlb_fifo_insert(int page_number, int frame_number) {
    int i = 0;
    EntryNode *entry_node = tlbTable->entryList->next;
    EntryNode *prev_node = entry_node->prev;
    // Break if page already in TLB.
    for (i = 0; i < next_tlb_entry; i++) {
        if (entry_node->page_index == page_number) {
            break;
        }

        prev_node = entry_node;
        entry_node = entry_node->next;
    }

    // Page not found in TLB.
    if (i == next_tlb_entry) {
        entry_node = prev_node;
        if (next_tlb_entry == tlbTable->length) {
            // TLB table is full, replace first entry.

            // Allocate new node.
            EntryNode *new_enode = new_node();
            new_enode->page_index = page_number;
            new_enode->frame_index = frame_number;

            // Append new node to the end of TLB.
            insert_node(entry_node, new_enode);

            // Delete first node.
            delete_next_node(tlbTable->entryList);
        } else {
            // TLB table is not full, append to end of table.
            EntryNode *next_node = entry_node->next;
            next_node->page_index = page_number;
            next_node->frame_index = frame_number;
            next_node->age = 0;
        }
    } else {
        // If another frame with the same page index already in TLB.

        // Go to last page in TLB.
        for (; i < next_tlb_entry - 1; i++) {
            entry_node = entry_node->next;
        }

        // Append page to end of TLB.
        EntryNode *new_enode = new_node();
        new_enode->page_index = page_number;
        new_enode->frame_index = frame_number;
        insert_node(entry_node, new_enode);

        // Delete the node with the same page index.
        delete_next_node(prev_node);
    }

    // Increment next tlb entry.
    if (next_tlb_entry < tlbTable->length) {
        next_tlb_entry = next_tlb_entry + 1;
    }
}

void tlb_lru_insert(int page_number, int frame_number) {
    Boolean free_spot_found = False;
    Boolean already_here = False;
    EntryNode *entry_node = tlbTable->entryList->next;
    EntryNode *to_replace_node = NULL;

    // Find the index to replace and increment age for all other entries.
    for (int i = 0; i < TLB_SIZE; i++) {
        if ((entry_node->page_index) &&
            (entry_node->page_index != page_number)) {
            // If entry does not exist in TLB and is not a free spot,
                increment
            // its age.
            entry_node->age++;
        } else if (entry_node->page_index == 0) {
            // A free entry in TLB.
            if (free_spot_found == False) {
                to_replace_node = entry_node;
                free_spot_found = True;

                // Don't exit for need to increment age for remaining entries.
            }
        } else if (entry_node->page_index == page_number) {
            // Entry already in TLB, reset its age.
            if (already_here == False) {
                entry_node->age = 0;
                already_here = True;
            } else {
                // Duplicate entry with the same page number.
                fprintf(stderr,
                        "Error: Same page appeared twice in LRU TLB.\nPage "
                        "number[%d]",
                        page_number);
            }
        }

        entry_node = entry_node->next;
    }

    // Replacement.
    if (already_here == True) {
        // Already in TLB, do nothing.
        return;
    } else if (free_spot_found == True) {
        // Free entry available in TLB.
        to_replace_node->page_index = page_number;
        to_replace_node->frame_index = frame_number;
        to_replace_node->age = 0;
    } else {
        // No free entry available in TLB, replace the oldest entry.
        to_replace_node = get_oldest_entry(TLB_SIZE);
        to_replace_node->page_index = page_number;
        to_replace_node->frame_index = frame_number;
```

```c
        to_replace_node->age = 0;
    }
}

EntryNode *get_oldest_entry(int tlb_size) {
    EntryNode *entry_node = tlbTable->entryList;
    int max = entry_node->age;
    EntryNode *max_age_node = NULL;

    // Iterate through TLB to find max age node.
    for (int i = 0; i < tlb_size; i++) {
        entry_node = entry_node->next;

        if (entry_node->age > max) {
            max = entry_node->age;
            max_age_node = entry_node;
        }
    }

    if (max_age_node == NULL) {
        fprintf(stderr, "Error getting oldest entry in TLB to replace.\n");
    }

    // Return max age node.
    return max_age_node;
}

double get_avg_time_in_secondary_storage() {
    double tmp = (double)cpu_time_elapsed / secondary_storage_call_count;
    return tmp * 1e6;
}
```

5.2.3 TLB 和页表的定义和实现等

```c
#ifndef VMEM_H
#define VMEM_H

/**
 * Dual direction linked list.
 */
typedef struct LLNode {
    int page_index;
    int frame_index;
    int age;
    struct LLNode *prev;
    struct LLNode *next;
} EntryNode;

/**
 * Allocate new linked list node.
 */
void *new_node();

void free_list(struct LLNode *list);

void insert_node(struct LLNode *curr_node, struct LLNode *new_node);
void delete_next_node(struct LLNode *node);

/**
 * Get length of linked list.
 */
int list_len(struct LLNode *node);

/**
 * Define a virtual memory addressing table that can be represented as
    either a
 * tlb cache or a page table.
 */
typedef struct VMemTable {
    // Entry list WITH HEAD NODE.
    EntryNode *entryList;
    int length;
    int page_fault_count;
    int tlb_hit_count;
    int tlb_miss_count;
} VMemTable;

/**
 * Create a new Virtual Memory Table for Logical Address Referencing.
 * Can represent TLB or Page Table Cache.
 */
VMemTable *create_vmem_table(int length);

/**
 * Print contents of the VMemTable.
 */
void display_table(VMemTable **table);

/**
 * Free memory that allocated on heap.
 */
void free_vmem_table(VMemTable **table);

/**
 * Accept an int double pointer for creating simulated memory space.
 */
int **dram_allocate(int frame_count, int block_size);

/**
 * Free DRAM memory after use.
 */
void free_DRAM(int ***dbl_pptr_arr, int frame_count);

/**
 * 32-bit masking function to extract page number.
 */
int get_page_number(int mask, int value, int shift);

/**
 * 32-bit masking function to extract page offset.
 */
int get_offset(int mask, int value);
```

```
#endif // !VMEM_H
```

```
#include "vmem.h"
#include <stdio.h>
#include <stdlib.h>

/**
 * Linked list implementation
 */

void *new_node() {
    struct LLNode *node = malloc(sizeof(struct LLNode));
    if (node == NULL) {
        fprintf(stderr, "Failed to allocate new linked list node.\n");
        exit(-1);
    }
    node->next = NULL;
    node->prev = NULL;
    node->page_index = 0;
    node->frame_index = 0;
    node->age = 0;

    return node;
}

/**
 * Delete given node's next node.
 */
void delete_next_node(struct LLNode *node) {
    struct LLNode *next_node = node->next;
    // Return if node has no next node.
    if (!next_node) {
        return;
    }

    // Next node is the last node.
    if (!(next_node->next)) {
        free(next_node);
        node->next = NULL;
    } else {
        // Next node is not the last node.
        struct LLNode *new_next_node = next_node->next;
        free(next_node);
        node->next = new_next_node;
        new_next_node->prev = node;
    }
}

void free_list(struct LLNode *list) {
    struct LLNode *node = list;
    struct LLNode *prev_node = NULL;

    while (!node) {
        prev_node = node;
        node = node->next;
        free(prev_node);
    }
}
```

```c
/**
 * Insert new_node after curr_node. If curr_node is linked to next node,
    insert
 * new_node between two nodes.
 */
void insert_node(struct LLNode *curr_node, struct LLNode *new_node) {
    if (curr_node->next) {
        // Has hext node.
        struct LLNode *next_node = curr_node->next;
        new_node->prev = curr_node;
        new_node->next = next_node;

        curr_node->next = new_node;
        next_node->prev = new_node;
    } else {
        // No next node.
        curr_node->next = new_node;
        new_node->prev = curr_node;
    }
}

int list_len(struct LLNode *node) {
    int len = 0;
    while (node) {
        len++;
        node = node->next;
    }

    return len;
}

/**
 * Create new virtual memory addressing table.
 */
VMemTable *create_vmem_table(int length) {
    VMemTable *new_table = malloc(sizeof(VMemTable));
    if (new_table == NULL) {
        printf("Error: failed to allocate memory for Virtual Memory
            Addressing "
               "table.\n");
        exit(-1);
    }

    new_table->length = length;
    // Allocate head node.
    new_table->entryList = new_node();
    // Allocate the rest length nodes.
    struct LLNode *curr_node = new_table->entryList;
    for (int i = 0; i < length; i++) {
        EntryNode *node = new_node();
        insert_node(curr_node, node);
        curr_node = curr_node->next;
    }

    new_table->page_fault_count = 0;
    new_table->tlb_hit_count = 0;
    new_table->tlb_miss_count = 0;
    return new_table;
}

/**
 * Free the memory that virtual memory allocation table occupied.
 */
void free_vmem_table(VMemTable **table) {
    free_list((*table)->entryList);

    free(*table);
}

/**
 * Print contents of the VMemTable.
 */
void display_table(VMemTable **table) {
    EntryNode *entryNode = (*table)->entryList;
    printf("------VIRTUAL MEMORY TABLE BEGIN------");
    for (int i = 0; i < (*table)->length; i++) {
        entryNode = entryNode->next;
        if (!entryNode) {
            break;
        }

        printf("INDEX[%4d]: PAGE NUMBER[%4d] FRAME NUMBER[%4d]\n", i,
               entryNode->page_index, entryNode->frame_index);
        entryNode = entryNode->next;
    }
    printf("------VIRTUAL MEMORY TABLE END------");
}

/**
 * Accept an int double pointer for creating simulated memory space.
 */
int **dram_allocate(int frame_count, int block_size) {
    int **temp;
    temp = (int **)malloc(frame_count * sizeof(int *));
    if (temp == NULL) {
        fprintf(stderr, "Error: failed to allocated space for DRAM.\n");
        exit(-1);
    }

    for (int i = 0; i < frame_count; i++) {
        temp[i] = (int *)malloc(block_size * sizeof(int));
        for (int j = 0; j < block_size; j++) {
            temp[i][j] = 0;
        }
    }

    return temp;
}

/**
 * Free DRAM memory after use.
 */
void free_DRAM(int ***dbl_ptr_arr, int frame_count) {
    for (int i = 0; i < frame_count; i++) {
        if ((*dbl_ptr_arr)[i] != NULL) {
            free((*dbl_ptr_arr)[i]);
        }
    }

    free(*dbl_ptr_arr);
}

/**
 * 32-bit masking function to extract page number.
 */
int get_page_number(int mask, int value, int shift) {
    return ((value & mask) >> shift);
}

/**
 * 32-bit masking function to extract page offset.
 */
int get_offset(int mask, int value) { return value & mask; }
```

5.2.4 数据生成器

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Not enough arguments give.\n");
        printf("Usage: data_gen <number of virtual addresses to access> <up "
               "border of number of consistent accesses in the same
                   frame>.\n");
        printf("Default write to config/input.txt.\n");

        exit(-1);
    }

    char filename[] = "config/input.txt";
    FILE *fout = fopen(filename, "w");

    srand(time(NULL));

    int num_vaddr = atoi(argv[1]);
    int in_frame_border = atoi(argv[2]);

    int frame, offset;
    int cnt = num_vaddr;
    int in_frame_cnt;

    int fate_addr = 0x6677;

    // Simulate locality of reference.
    while (cnt >= 0) {

        // Frame index.
```

```
            frame = rand() % (1 << 8);
            // Counter of addresses in this frame.
            in_frame_cnt = rand() % (in_frame_border + 1);
            printf("%4d virtual addresses to access in frame[0x%04x]\n",
                   in_frame_cnt, (frame & 0xFF));
            while (in_frame_cnt >= 0) {
                if ((rand() % 100) < 20) {
                    fprintf(fout, "%d\n", fate_addr);
                } else {
                    offset = rand() % (1 << 8);
                    fprintf(fout, "%d\n", ((frame & 0xFF) << 8) | (offset &
                        0xFF));

                    in_frame_cnt--;
                    cnt--;
                }
            }
        }

        FILE *stat_file = fopen("logs/stat.log", "a");
        fprintf(stat_file, "%d\n", in_frame_border);

        fclose(fout);
        fclose(stat_file);

        return 0;
    }
```

6 附录

代码仓库: OSLab