操作系统实验报告

学 生 姓 名　　苏靖博
学　　号　　20105050110
班　　级　　计实验20

实验名称	页面置换算法	实验序号	2
实验日期	Oct. 13 2022	实验人	苏靖博

一、实验目的和要求

1. 请求页式存储管理是一种常用的虚拟存储管理技术。

2. 本实验目的是通过请求页式存储管理中页面置换算法的模拟设计，了解虚拟存储技术的特点，掌握请求页式存储管理的页面置换算法。

二、相关背景知识

- Page Fault happens when a process accesses a memory page that is mapped into a virtual address space but not loaded in physical memory.
- The actual physical memory is much smaller than virtual memory.
- The target of replacement algorithms is to reduce the number of page fault.
- Bélády's anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

三、实验内容

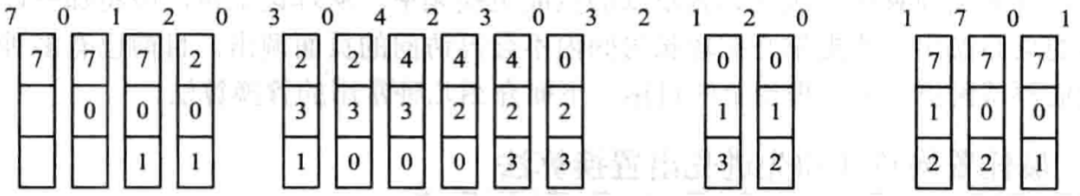
1. 通过随机数产生一个指令序列，共320条指令。指令的地址按下述原则生成：
   a) 50%的指令是顺序执行的；
   b) 25%的指令是均匀分布在前地址部分；
   c) 25%的指令是均匀分布在后地址部分；

   具体的实施方法是：
   1. 在[0, 319]的指令地址之间随机选取一起点m；
   2. 顺序执行一条指令，即执行地址为m+1的指令；
   3. 在前地址[0, m+1]中随机选取一条指令并执行，该指令的地址为m'；
   4. 顺序执行一条指令，其地址为m'+1；
   5. 在后地址[m'+2, 319]中随机选取一条指令并执行；
   6. 重复上述步骤1~5，直到执行320次指令。

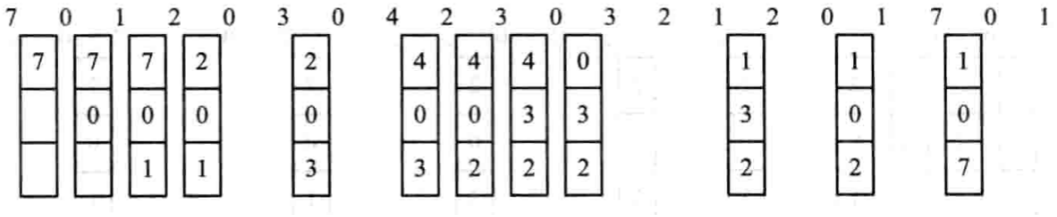
2. 将指令序列变换成页地址流，设
   1. 页面大小为1K；
   2. 用户内存容量为4页到32页；
   3. 用户虚存容量为32K。

   在用户虚存中，按每K存放10条指令排列虚存地址，即320条指令在虚存中存放的方式为：
   第0条至第9条指令为第0页（对应虚存地址为[0，9]）；
   第10条至第19条指令为第1页（对应虚存地址为[10，19]）；
   ……
   第310条至第319条指令为第31页（对应虚存地址为[310，319]）；
   按以上方式, 用户指令可以组成32页。

3. 计算并输出下述各种算法在不同内存容量下的命中率
   1. 先进先出页面淘汰算法（FIFO）



2. 最近最久未使用页面淘汰法（LRU）



命中率 = 1 - 页面失效次数/页地址流长度

在本实验中，页地址流长度为320，页面失效次数为每次访问相应指令时，该指令对应的页不在内存的次数.

实验链接🔗 https://github.com/Sue217/NCUT/tree/main/OS/lab2

四、关键数据结构与函数的说明

- **ins.h**

```
// instruction initialization
#define __TOTAL__ 320
#define __1K__ 10

int ins[__TOTAL__];

void __init__() {
  for (int i = 0; i < __TOTAL__; i++) {
    ins[i] = i;
  }
}
```

- **fifo.h**

```
// FIFO
int fifo(int value, int ram) {
  const int max_size = ram * __1K__;
  for (int i = 0; i < max_size; i++) {
    if (value == user_vm[i]) {
      // hit
      return 0;
    }
  }
  // didn't hit
  size += (size < max_size) * 10;
   for (int i = 0, j = (value / 10) * 10; i < 10; i++, j++,
   idx++) {
    user_vm[idx] = ins[j];
```

```
  }
  idx %= max_size;
  return 1;
}
```

- **lru.h**

```
// LRU
int lru(int value, int ram) {
  const int max_size = ram * __1K__;
  list_t target = is_hit(lst, value);
  if (target != NULL) {
    // hit
    replace(lst, target);
    return 0;
  }
  // didn't hit
  int a[__1K__];
  for (int i = 0, j = (value / 10) * 10; i < 10; i++, j++) {
    a[i] = ins[j];
  }
  if (len >= max_size) {
    // page fault happens
    pop(lst);
    append(lst, a);
  } else {
    // list is not full
    append(lst, a);
  }
  return 1;
}
```

- **list.h**

```
// data structure
typedef struct list {
  struct list* next;
  struct list* prev;
  int page[__1K__];
} *list_t;

list_t create(); // create and return a new list’s head

void __clear__(list_t head); // clear the list starting with
   `head`

int is_empty(list_t head); // check if the list is empty

void append(list_t head, int a[]); // add a page to the back
   of the list starting with `head`

void pop(list_t head); // delete a page from the top of the
   list starting with `head`


void replace(list_t head, list_t x); // move the `x` page to
```

```
    the back of the list starting with `head`

list_t is_hit(list_t head, int value); // check if the value
    is hit in the list starting with `head`
```

五、编译与执行过程截图

USE **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out fifo`** to test the First In First Out page replacement algorithm ONLY

OR **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out lru`** to test the Least Recently Used page replacement algorithm ONLY

OR **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out`** can test both algorithms together with the same series of random instructions.

result:

User Mem	FIFO	LRU
4	0.5219	0.5281
5	0.5594	0.5594
6	0.6062	0.6031
7	0.5844	0.5781
8	0.5906	0.5938
9	0.6156	0.6062
10	0.6312	0.6375
11	0.6344	0.6344
12	0.6469	0.6344
13	0.6562	0.6844
14	0.7063	0.7031
15	0.7156	0.7125
16	0.7188	0.7344
17	0.7469	0.7531
18	0.7719	0.7812
19	0.7594	0.7594
20	0.7531	0.7562
21	0.7781	0.7875
22	0.8063	0.8031
23	0.8375	0.8125
24	0.8125	0.8438
25	0.8063	0.8219
26	0.8438	0.8469
27	0.8719	0.8594
28	0.8625	0.8719
29	0.8844	0.8875
30	0.8906	0.8844
31	0.9000	0.8938
32	0.9000	0.9000

六、实验结果与分析

1. 随着页面数增加，两种置换算法的命中率也随之增加，且两种算法命中率在多次实验下平均情况基本相同.

2. 当页面最终达到32时, 二者命中率都基本等于90%. 说明32页缺页大概率下都会出现.

3. 当页数为4(min)时, 两种置换算法命中率并没有很低而是均维持在 50% 以上

4. 页面数量增多反而命中率会下降反应出页面置换算法的抖动现象，其名称为: Bélády's anomaly: 指增加页框数量会导致给定内存访问模式的页面错误数量增加的现象.

七、调试时遇到的问题及解决方法

在进行实验之前，并没有完全理解操作系统的缺页置换机制，单纯的以为当指令没有命中时才进行置换处理. 因此实验所的结果命中率都异常的高, 平均达到96%+.

设想如果操作系统如此设计，那么缺页导致的上下文切换会非常频繁，最终会使操作系统的效率大大降低.

因此对程序进行改进，使操作系统每次发生缺页时会将该句代码所在的整个页面(%)存入虚拟内存, 顺利完成实验.

八、调试后的程序源代码

🔗 https://github.com/Sue217/NCUT/tree/main/OS/lab2

九、实验体会

1. 本次实验纠正了我在学习相关知识时错误的理解, 纠正后顺利完成了实验.

2. 本次实验对常见错误处理和代码调试更加熟练. 能够更加熟练使用断点和输出方法诊断问题的所在并加以修改.

3. 理解解了常见的页面置换算法、发生时间及处理机制，还了解了置换算法抖动的神奇现象, 收获颇丰.