桂林航天工業學院
GUILIN UNIVERSITY OF AEROSPACE TECHNOLOGY

实 验 报 告

课程名称 操作系统（B）

开课学期 2019-2020 学年第一学期

指导教师 魏士伟

实 验 室 巡天 308

班　　级 2017 软件工程 4 班

学　　号 2017070030429

姓　　名 钟祯

成绩：_____（五级）

实验课程 评分表标准

	全勤、学习态度端正、实验认真、积极回答问题、操作过程正确，结果准确，实验报告内容规范	偶有缺勤、实验认真、回答问题较积极、操作过程正确，结果准确，实验报告内容规范	旷课2次以内、偶有迟到、实验认真、回答问题较好、操作过程基本正确，结果基本准确，实验报告内容较规范	旷课2次以上、学习态度一般、基本能回答出问题、操作过程较正确，结果基本准确，实验报告内容基本规范	经常旷课，实验过程不认真、问题回答不积极、实验报告不符合要求或未交
	优秀（90-100）	良好（80-89）	中（70-79）	及格（60-69）	不及格(<59)
实验一					
实验二					
实验三					
实验成绩总评（五级制）					

说明：1．每次实验结束，学生完成一份实验报告，课程结束后汇总,加封面装订成册存档；2．各系（部）可在以上五项栏目的基础上，可根据实验课程和实验项目的具体需要，统一设计和调整项目内容，但封面格式应统一；3．对于设计性实验，只要求说明实验的目的要求、提出可供实验的基本条件和注意事项，实验方案和步骤的设置、仪器的安排等，可由学生自己设计；4.可根据实验数量自行添加行数。打印到封面背面

桂林航天工业学院学生实验报告

课程名称	**操作系统（B）**		实验项目名称	**进程调度（4 学时）**	
开课教学单位及实验室		计算机科学与工程学院		实验日期	2019 年 11 月 15 日 2019 年 11 月 22 日
学生姓名	钟祯	学号	2017070030429	专业班级	2017 软件工程 4 班
指导教师	魏士伟		实验成绩		

一、实验目的

1.进一步理解进程的相关概念，熟悉进程状态及状态之间的转换过程。

2.了解进程调度的任务，掌握进程调度的机制和方式。

3.掌握进程调度算法的实现，并对比分析算法性能。

二、实验内容及要求

编写并调试一个单道处理系统的进程等待模拟程序。

可选择实现下列进程调度算法，对每种调度算法都要求计算并输出每个进程开始运行时刻、完成时刻、周转时间：

（1）先来先服务（FCFS）与短作业优先（SJF）；

先来先服务(FCFS)算法：按照作业/进程进入系统的先后次序进行调度，先进入系统者先调度；即启动等待时间最长的作业/进程。

短作业优先(SJF)的调度算法：从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。

（2）非抢占式与抢占式优先权调度算法

非抢占式优先权调度算法：系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。

抢占式优先权调度算法：只要系统中出现一个新的就绪进程，就进行优先权比较 。若出现优先权更高的进程，则立即停止当前执行，并将处理机分配给新到的优先权最高的进程。

建议采用静态优先权的方式，赋予各进程固定的优先级。

（3）基本时间片轮转调度算法

系统将所有的就绪进程按先来先服务的原则,排成一个队列,每次调度时,把 CPU 分配给队首进程,并令其执行一个时间片.当执行的时间片用完时,由一个计时器发出时钟中断请求,调度程序便据此信号停止该进程的执行,并将它送往就绪队列的末尾;然后,再把处理机分配给就绪队列中新的队首进程,同时也让它执行一个时间片.这样就可以保证就绪队列中的所有进程,在一给定的时间内,均能获得一时间片的处理机执行时间。

三、实验源代码及说明

实验环境: Clion

编程语言：C

调度算法：

① FCFS 算法

② SJF 算法

设计思路：

① FCFS 算法按照进程的到达时间将进程进行排序

② SJF 算法按照进程的到达时间将进程进行排序后，从第 2 个进程开始按服务时间排序

③ 两个算法所要计算的开始时间、完成时间、周转时间、带权周转时间方法相同

源代码：

开始时间 = 上一个进程的完成时间（第一个进程为到达时间） = 上一个进程的到达时间 + 上一个进程的服务时间

完成时间 = 开始时间 + 服务时间

周转时间 = 完成时间 - 到达时间

带权周转时间 = 周转时间 / 服务时间

```c
#include <stdio.h>
#include <stdlib.h>

#define value 100

typedef struct process {
    char PID;           // 进程 ID
    float arrivalTime;  // 到达时间
    float serviceTime;  // 服务时间
    float startTime;    // 开始时间
    float finishTime;   // 完成时间
    float aroundTime;   // 周转时间
    float weightAroundTime; // 带权周转时间
    struct process *link;   // 结构体指针
} Algorithm;

Algorithm *p, *q, *head = NULL;

struct process array[value];

/*
 * 按到达时间将 进程 进行排序
 *
 * processArray[] 存放排序后的进程
 * processNumber 进程数量
 * type :
 *      0 : FCFS
 *      1 : SJF
 *
 * */
struct process *sortArrivalTime(struct process processArray[], int processNumber, int type) {
    struct process temp;
    for (int i = 0; i < processNumber; i++) {
        for (int j = i + 1; j < processNumber; j++) {
            if (processArray[i].arrivalTime > processArray[j].arrivalTime) {
                temp = processArray[i];
                processArray[i] = processArray[j];
                processArray[j] = temp;
            }
        }
    }

    // 进入 SJF 排序阶段
    if (type == 1) {
        for (int m = 0; m < processNumber - 1; m++) {
            if (m == 0)
                processArray[m].finishTime = processArray[m].arrivalTime + processArray[m].serviceTime;
            else {
                if (processArray[m - 1].finishTime >= processArray[m].arrivalTime) {
                    processArray[m].startTime = processArray[m - 1].finishTime;
                } else {
```

```
                processArray[m].startTime = processArray[m].arrivalTime;
            }
            processArray[m].finishTime = processArray[m].startTime + processArray[m].serviceTime;
        }

        int count = 0;
        for (int n = m + 1; n <= processNumber - 1; n++) {
            if (processArray[n].arrivalTime <= processArray[m].finishTime)
                count++;
        }

        //按服务时间排序
        float min = processArray[m + 1].serviceTime;
        int next = m + 1;
        for (int k = m + 1; k < m + count; k++) {
            if (processArray[k + 1].serviceTime < min) {
                min = processArray[k + 1].serviceTime;
                next = k + 1;
            }

        }
        temp = processArray[m + 1];
        processArray[m + 1] = processArray[next];
        processArray[next] = temp;
    }
  }
  return processArray;
}

// 打印日志
void publicPrint(struct process processArray[], int processNumber, int type) {
  int i;
  if (type == 0) {
    printf("进程 ID 到达时间 服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
    for (i = 0; i < processNumber; i++) {
      printf("  %c     %.3f   %.3f   %.3f   %.3f    %.3f    %.3f \n",
             processArray[i].PID,
             processArray[i].arrivalTime,
             processArray[i].serviceTime,
             processArray[i].startTime,
             processArray[i].finishTime,
             processArray[i].aroundTime,
             processArray[i].weightAroundTime);
    }
  } else if (type == 1) {
    printf("进程 ID 服务时间 到达时间 开始时间 完成时间 周转时间 带权周转时间\n");
    for (i = 0; i < processNumber; i++) {
      printf("  %c     %.3f   %.3f   %.3f   %.3f    %.3f    %.3f \n",
             processArray[i].PID,
             processArray[i].serviceTime,
             processArray[i].arrivalTime,
             processArray[i].startTime,
             processArray[i].finishTime,
```

```
                processArray[i].aroundTime,
                processArray[i].weightAroundTime);
        }
    }
}

// 各个时间的相关计算
void publicAlgorithm(struct process processArray[], int processNumber, int type) {
    int i;

    // 接收变量
    float receiveTime;

    // 开始时间 = 上一个进程的完成时间（第一个进程为到达时间） = 上一个进程的到达时间 + 上一个进程的服务时间
    // 完成时间 = 开始时间 + 服务时间
    // 周转时间 = 完成时间 - 到达时间
    // 带权周转时间 = 周转时间 / 服务时间

    for (i = 0; i < processNumber; i++) {
        if (i == 0) {
            processArray[i].startTime = processArray[i].arrivalTime;
        }
        if (i != 0) {
            receiveTime = processArray[i - 1].startTime + processArray[i - 1].serviceTime;
            if (processArray[i].arrivalTime > receiveTime) {
                processArray[i].startTime = processArray[i].arrivalTime;
            } else {
                processArray[i].startTime = receiveTime;
            }
        }
        processArray[i].finishTime = processArray[i].startTime + processArray[i].serviceTime;
        processArray[i].aroundTime = processArray[i].finishTime - processArray[i].arrivalTime;
        processArray[i].weightAroundTime = processArray[i].aroundTime / processArray[i].serviceTime;
    }
    printf("-----------------------------------\n");
    printf("publicAlgorithm: \n");
    publicPrint(processArray, processNumber, type);
}

// 选择
void checkAlgorithm(struct process processArray[], int processNumber) {
    int type;
    printf("\n");
    printf("----------------------\n");
    printf("| 0 ---- FCFS |\n");
    printf("| 1 ---- SJF |\n");
    printf("----------------------\n");
    printf("请选择调度算法（其余指令 >>> 退出系统）: ");
    scanf("%d", &type);
    switch (type) {
        case 0:
            publicAlgorithm(sortArrivalTime(processArray, processNumber, 0), processNumber, 0);
            checkAlgorithm(processArray, processNumber);
```

```
            break;
        case 1:
            publicAlgorithm(sortArrivalTime(processArray, processNumber, 1), processNumber, 1);
            checkAlgorithm(processArray, processNumber);
            break;
        default:
            break;
    }
}

// 主函数
int main() {
    int processNumber;
    int i;
    printf("请输入进程数（processNumber）: ");
    scanf("%d", &processNumber);
    for (i = 0; i < processNumber; i++) {
        printf("-----------------------------------\n");
        printf("第%d 个进程的 ID（PID）:", i + 1);
        scanf("%s", &array[i].PID);
        printf("第%d 个进程到达时间（arriveTime）:", i + 1);
        scanf("%f", &array[i].arrivalTime);
        printf("第%d 个进程服务时间（serviceTime）:", i + 1);
        scanf("%f", &array[i].serviceTime);
    }
    checkAlgorithm(array, processNumber);
    return 0;
}
```

四、实验结果及分析

先来先服务 FCFS 算法与短作业优先 SJF 算法优劣对比：

优点：

① FCFS：对所有作业/进程公平，算法简单稳定

② SJF：灵活性高，相同时限下能处理更多的任务

缺点：

① FCFS：不够灵活，对紧急进程的优先处理权限不够，在相同时限下处理任务数量可能更少。

② SJF：不够稳定，算法更复杂

数据键入（初始化）：

```
CLionCode
D:\CLionCode\cmake-build-debug\CLionCode.exe
请输入进程数（processNumber）：4
-----------------------------------
第1个进程的ID（PID）:P1
第1个进程到达时间（arriveTime）:0
第1个进程服务时间（serviceTime）:8
-----------------------------------
--
第2个进程的ID（PID）:P2
第2个进程到达时间（arriveTime）:1
第2个进程服务时间（serviceTime）:4
-----------------------------------
--
第3个进程的ID（PID）:P3
第3个进程到达时间（arriveTime）:2
第3个进程服务时间（serviceTime）:9
-----------------------------------
--
第4个进程的ID（PID）:P4
第4个进程到达时间（arriveTime）:3
第4个进程服务时间（serviceTime）:5
```

FCFS 算法：

```
----------------------
〔    0 ---- FCFS    〕
〔    1 ---- SJF     〕
----------------------
请选择调度算法（其余指令 >>> 退出系统）：0
-----------------------------------
publicAlgorithm:
进程ID 到达时间 服务时间 开始时间 完成时间 周转时间 带权周转时间
  P     0.000   8.000   0.000   8.000    8.000    1.000
  P     1.000   4.000   8.000   12.000    11.000    2.750
  P     2.000   9.000   12.000   21.000    19.000    2.111
  P     3.000   5.000   21.000   26.000    23.000    4.600
```

SJF 算法：

```
----------------------
〔    0 ---- FCFS    〕
〔    1 ---- SJF     〕
----------------------
请选择调度算法（其余指令 >>> 退出系统）：1
-----------------------------------
publicAlgorithm:
进程ID 服务时间 到达时间 开始时间 完成时间 周转时间 带权周转时间
  P     8.000   0.000   0.000   8.000    8.000    1.000
  P     4.000   1.000   8.000   12.000    11.000    2.750
  P     5.000   3.000   12.000   17.000    14.000    2.800
  P     9.000   2.000   17.000   26.000    24.000    2.667
```

桂林航天工业学院学生实验报告

课程名称	**操作系统（B）**		实验项目名称	**银行家算法（4 学时）**	
开课教学单位及实验室		计算机科学与工程学院		实验日期	2019 年 11 月 29 日 2019 年 12 月 06 日
学生姓名	钟祯	学号	2017070030429	专业班级	2017 软件工程 4 班
指导教师	魏士伟		实验成绩		

一、实验目的

1.掌握有关资源申请、死锁等相关概念，进一步理解死锁产生的原因和必要条件。

2.掌握利用银行家算法避免死锁的方法，能结合具体应用分析系统状态。

二、实验内容及要求

本实验要求通过编写和调试一个系统动态分配资源的简单模拟程序，观察死锁产生的条件，并采用适当的算法，有效地防止和避免死锁的发生。具体要求如下：

（1）模拟一个银行家算法，判断是否处于安全状态；

（2）初始化时让系统拥有一定的资源；

（3）用键盘输入的方式申请资源；

（4）如果预分配后，系统处于安全状态，则修改系统的资源分配情况，判断其安全序列；

（5）如果预分配后，系统处于不安全状态，则提示不能满足请求；

三、实验源代码及说明

实验环境: CLion

编程语言：C

设计思路：

先对用户提出的请求进行合法性检查，即检查请求的是不大于需要的，是否不大于可利用的。若请求合法，则进行试分配。最后对试分配后的状态调用安全性检查算法进行安全性检查。若安全，则分配，否则，不分配，恢复原来状态，拒绝申请。

源代码：

```
#include<stdio.h>

#define processNumber 5     // 5 个进程
#define resourceNumber 3    // 3 类资源

int Available[resourceNumber];                // 可利用资源向量
int Max[processNumber][resourceNumber];       // 最大需求矩阵
int Allocation[processNumber][resourceNumber];  // 分配矩阵
int Need[processNumber][resourceNumber];      // 需求矩阵

void InitializeData();

void ShowData(int line);

void CalcMaxMatrix();

int Equals(int a[resourceNumber], int b[resourceNumber]);

int CheckSafe();

int CheckFinish(int Finish[resourceNumber]);

void Add(int *a, int b[resourceNumber]);

void Minus(int *a, int b[resourceNumber]);

int Request(int P, int Request[resourceNumber]);

void RequestShowMsg(int P, int R[resourceNumber]);

int main() {
    InitializeData();
    printf("==================初始数据如下==================\n");
    ShowData(0);
    CheckSafe();

    //进程 P1 申请资源{1,0,2}
    int apply[resourceNumber] = {1, 0, 2};
    RequestShowMsg(1, apply);

    //进程 P4 申请资源{1,0,2}
    int apply2[resourceNumber] = {3, 3, 0};
    RequestShowMsg(4, apply2);

    //进程 P0 申请资源{0,2,0}
    int apply3[resourceNumber] = {0, 2, 0};
    RequestShowMsg(0, apply3);

    return 0;
}
```

```
// 手动初始化数据,资源分配表
void InitializeData() {
    Allocation[0][0] = 0, Allocation[0][1] = 1, Allocation[0][2] = 0;
    Allocation[1][0] = 2, Allocation[1][1] = 0, Allocation[1][2] = 0;
    Allocation[2][0] = 3, Allocation[2][1] = 0, Allocation[2][2] = 2;
    Allocation[3][0] = 2, Allocation[3][1] = 1, Allocation[3][2] = 1;
    Allocation[4][0] = 0, Allocation[4][1] = 0, Allocation[4][2] = 2;

    Need[0][0] = 7, Need[0][1] = 4, Need[0][2] = 3;
    Need[1][0] = 1, Need[1][1] = 2, Need[1][2] = 2;
    Need[2][0] = 6, Need[2][1] = 0, Need[2][2] = 0;
    Need[3][0] = 0, Need[3][1] = 1, Need[3][2] = 1;
    Need[4][0] = 4, Need[4][1] = 3, Need[4][2] = 1;

    Available[0] = 3, Available[1] = 3, Available[2] = 2;

    CalcMaxMatrix();
}

// 进程资源请求 P:进程 i,r 申请资源数{1,1,1} 返回 1 成功 0 失败
int Request(int P, int Request[resourceNumber]) {
    printf("进程 P%d 申请资源%d %d %d:\n", P, Request[0], Request[1], Request[2]);

    if (!Equals(Request, Need[P])) {
        printf("进程 P%d,Request:%d %d %d > Need:%d %d %d 申请失败,所需资源数超过宣布最大值！\n",
               P, Request[0], Request[1], Request[2], Need[P][0], Need[P][1], Need[P][2]);
        return 0;
    }

    if (!Equals(Request, Available)) {
        printf("进程 P%d,Request:%d %d %d > Available:%d %d %d 申请失败,尚无足够资源，该进程需要等待！\n",
               P, Request[0], Request[1], Request[2], Available[0], Available[1], Available[2]);
        return 0;
    }

    printf("进程 P%d,Request:%d %d %d <= Need:%d %d %d\n",
           P, Request[0], Request[1], Request[2], Need[P][0], Need[P][1], Need[P][2]);
    printf("进程 P%d,Request:%d %d %d <= Available:%d %d %d \n",
           P, Request[0], Request[1], Request[2], Available[0], Available[1], Available[2]);

    Minus(Available, Request);  // Available -= Request
    Add(Allocation[P], Request);// Allocation += Request
    Minus(Need[P], Request);    // Need -= Request

    int safeState = CheckSafe();
    if (safeState) {
        return safeState;       // 分配后处于安全状态 分配成功
    }

    //分配后处于不安全状态 分配失败，本次分配作废，回复原来的资源分配状态
    Add(Available, Request);        //Available += Request
    Minus(Allocation[P], Request);  //Allocation -= Request
```

```
    Add(Need[P], Request);          //Need += Request
    return safeState;
}

//带命令提示符提示的请求
void RequestShowMsg(int P, int R[resourceNumber]) {
    // 拟定模拟：进程 P 申请资源 Request{1,0,2}
    printf("\n 模拟分配资源：P%d 申请资源 %d %d %d\n=====================\n", P, R[0], R[1], R[2]);
    int State = Request(P, R);
    if (State) {
        printf("本次资源分配成功！\n");
        ShowData(0);
    } else {
        printf("本次资源分配失败！进程 P%d 需要等待\n", P);
    }
}

//安全性检测，判断当前是否处于安全状态
int CheckSafe() {
    printf("开始安全性检查:\n");
    int Finish[processNumber] = {0};
    int Work[resourceNumber] = {0};
    Add(Work, Available);
    for (int i = 0; i < processNumber; i++) {
        if (Finish[i])continue;
        if (!Equals(Need[i], Work))continue;
        Add(Work, Allocation[i]);   //Work += Allocation;
        Finish[i] = 1;              //Finish[i]=True;
        printf("P%d 进程,Work=%d %d %d,Finish=true,安全状态\n", i, Work[0], Work[1], Work[2]);
        i = -1;
    }
    if (CheckFinish(Finish)) {
        printf("安全状态检查完毕:【Finish 全为 true,系统处于安全状态】\n");
        return 1;
    }
    printf("安全状态检查完毕:【Finish 存在 False,系统处于不安全状态】\n");
    return 0;
}

// 检查标志所有都为 True，是返回 1 不是返回 0
int CheckFinish(int Finish[resourceNumber]) {
    for (int i = 0; i < resourceNumber; i++) {
        if (Finish[i] == 0) return 0;
    }
    return 1;
}

// 打印数据
void ShowData(int line) {
    printf("PID    Max    Allocation Need   Available\n");
    for (int i = 0; i < processNumber; i++) {
        printf("p%d:\t", i);
        for (int j = 0; j < resourceNumber; j++) {
```

```
            printf("%d ", Max[i][j]);
        }
        printf("\t  ");
        for (int j = 0; j < resourceNumber; j++) {
            printf("%d ", Allocation[i][j]);
        }
        printf("\t");
        for (int j = 0; j < resourceNumber; j++) {
            printf("%d ", Need[i][j]);
        }

        if (line == i) {
            printf("\t  ");
            for (int j = 0; j < resourceNumber; j++) {
                printf("%d ", Available[j]);
            }
        }

        printf("\n");
    }

}

// 计算 Max
void CalcMaxMatrix() {
    for (int i = 0; i < processNumber; i++) {
        for (int j = 0; j < resourceNumber; j++) {
            Max[i][j] = Need[i][j] + Allocation[i][j];
        }
    }
}

void Add(int *a, int b[resourceNumber]) {
    for (int i = 0; i < resourceNumber; i++) {
        a[i] = a[i] + b[i];
    }
}

void Minus(int *a, int b[resourceNumber]) {
    for (int i = 0; i < resourceNumber; i++) {
        a[i] = a[i] - b[i];
    }
}

//资源比较   a<=b 返回 1     a>b 返回 0
int Equals(int a[resourceNumber], int b[resourceNumber]) {
    for (int i = 0; i < resourceNumber; i++) {
        if (a[i] > b[i]) return 0;
    }
    return 1;
}
```

四、实验结果及分析

数据初始化：

```
CLionCode
D:\CLionCode\cmake-build-debug\CLionCode.exe
==================初始数据如下==================
PID     Max     Allocation      Need    Available
p0:     7 5 3     0 1 0         7 4 3     3 3 2
p1:     3 2 2     2 0 0         1 2 2
p2:     9 0 2     3 0 2         6 0 0
p3:     2 2 2     2 1 1         0 1 1
p4:     4 3 3     0 0 2         4 3 1
```

安全性检测：

```
开始安全性检查:
P1进程,Work=5 3 2,Finish=true,安全状态
P3进程,Work=7 4 3,Finish=true,安全状态
P0进程,Work=7 5 3,Finish=true,安全状态
P2进程,Work=10 5 5,Finish=true,安全状态
P4进程,Work=10 5 7,Finish=true,安全状态
安全状态检查完毕:【Finish全为true,系统处于安全状态】
```

模拟资源分配①：

```
模拟分配资源：P4申请资源 3 3 0
================================================
进程P4申请资源3 3 0:
进程P4,Request:3 3 0 > Available:2 3 0 申请失败,尚无足够资源，该进程需要等待!
本次资源分配失败！进程P4需要等待
```

模拟资源分配②：

```
模拟分配资源：P1申请资源 1 0 2
====================================================
进程P1申请资源1 0 2:
进程P1,Request:1 0 2 <= Need:1 2 2
进程P1,Request:1 0 2 <= Available:3 3 2
开始安全性检查:
P1进程,Work=5 3 2,Finish=true,安全状态
P3进程,Work=7 4 3,Finish=true,安全状态
P0进程,Work=7 5 3,Finish=true,安全状态
P2进程,Work=10 5 5,Finish=true,安全状态
P4进程,Work=10 5 7,Finish=true,安全状态
安全状态检查完毕:【Finish全为true,系统处于安全状态】
本次资源分配成功!
PID     Max     Allocation      Need    Available
p0:     7 5 3     0 1 0         7 4 3     2 3 0
p1:     3 2 2     3 0 2         0 2 0
p2:     9 0 2     3 0 2         6 0 0
p3:     2 2 2     2 1 1         0 1 1
p4:     4 3 3     0 0 2         4 3 1
```

模拟资源分配③：

```
模拟分配资源：P0申请资源 0 2 0
====================================================
进程P0申请资源0 2 0:
进程P0,Request:0 2 0 <= Need:7 4 3
进程P0,Request:0 2 0 <= Available:2 3 0
开始安全性检查:
安全状态检查完毕:【Finish存在False,系统处于不安全状态】
本次资源分配失败！进程P0需要等待
```

分析：

银行家算法就是一个分配资源的过程，使分配的序列不会产生死锁。此算法的中心思想是：按该法分配资源时，每次分配后总存在着一个进程，如果让它单独运行下去，必然可以获得它所需要的全部资源，也就是说，它能结束，而它结束后可以归还这类资源以满足其他申请者的需要。

桂林航天工业学院学生实验报告

课程名称	**操作系统（B）**		实验项目名称	**存储器管理（2 学时）**	
开课教学单位及实验室	计算机科学与工程学院			实验日期	2019 年 12 月 06 日
学生姓名	钟祯	学号	2017070030429	专业班级	2017 软件工程 4 班
指导教师	魏士伟		实验成绩		

一、实验目的

1.熟悉存储管理策略，掌握分页存储管理的过程、原理和虚拟存储的实现方式。

2.掌握虚拟存储管理的页面淘汰算法，并结合具体应用分析算法性能。

二、实验内容及要求

设计一个请求页式存储管理方案，编写模拟程序实现具体过程，并计算缺页率(所有内存开始都是空的，凡第一次用到的页面都产生一次缺页中断)。

要求实现下列页面置换算法:

（1）先进先出算法(FIFO) ：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。

（2）最近最久未使用算法(LRU) ：淘汰最近最久未被使用的页面。

程序中用户可选择置换算法，先输入所有页面号，为系统分配物理块，依次按照 FIFO 或 LRU 算法进行置换。

三、实验源代码及说明

实验环境: CLion

编程语言：C

设计思路：

① 先进先出置换算法（FIFO）：当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。因为最早调入主存的页面不再被使用的可能性最大。

② 最近最久未使用（LRU）算法：利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也

不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。

源代码：

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

#define MEMORY_BLOCKS 20    // 内存物理总块数
#define PROGRAM_PAGES 50    // 程序分页数上限
#define PAGE_USE_LENGTH 100 // 页面走向长度最大值
#define FIFO 1              // 先进先出置换
#define LRU 2               // 最久未用置换

int memoryBlocks = 0;       // 分配内存物理块数
int programPages = 0;       // 程序分页数

int memoryAllocation[MEMORY_BLOCKS];    // 内存分配情况

/*
 * 访问字段
 * */
struct Visit {
    int loadTime;   // 装载时间
    int frequency;  // 使用频率的顺序，0 为最高
};

/*
 * 页表
 * */
struct pageTable {
    int pageID;         // 页号
    int blockID;        // 物理块号
    bool state;         // 状态位
    bool modify;        // 修改位
    struct Visit visit; // 访问字段
} Page[PROGRAM_PAGES];

int createRandom(int i, int j);

void init(int m, int p);

void pageAdjust(int p);

bool aSingleVisit(int p, int f);

int replacePage(int f);

void newLine(int length);
```

```
void showData();

void simulation(int l, int f);

void menu();

/*
 * 生成随机数
 * */
int createRandom(int i, int j) {
    int randomNumber;
    randomNumber = rand() % (j - i + 1) + i;
    return randomNumber;
}

/*
 * 初始化
 * */
void init(int m, int p) {
    memoryBlocks = m;
    programPages = p;
    for (int i = 0; i < memoryBlocks; i++) {
        memoryAllocation[i] = -1;
    }
    for (int i = 0; i < programPages; i++) {
        // 初始化赋予默认值
        Page[i].pageID = i;
        Page[i].blockID = -1;
        Page[i].state = false;
        Page[i].modify = false;
        Page[i].visit.loadTime = 0;
        Page[i].visit.frequency = -1;
    }
    printf("\n 初始化完成\n");
}

/*
 * 内存中其他页的调整
 * */
void pageAdjust(int p) {
    int i;
    for (i = 0; i < programPages; i++) {
        if (Page[i].state == true) {    // 内存中的页
            Page[i].visit.loadTime += 1;
            if (Page[p].visit.frequency == -1) {    // 该页不在内存
                Page[i].visit.frequency += 1;
            } else {    // 该页在内存
                if (Page[i].visit.frequency < Page[p].visit.frequency) {
                    Page[i].visit.frequency += 1;
                }
            }
        }
```

```
    }
}

/*
 * 单次访问
 * p 为要访问的页
 * f 为置换算法
 * */
bool aSingleVisit(int p, int f) {
    int i;
    bool flag = false;    // 访问成功的标记
    if (Page[p].state == false) {              // 该页未在内存中
        for (i = 0; i < memoryBlocks; i++) {
            if (memoryAllocation[i] == -1) {    // 有内存块可供分配
                flag = true;
                pageAdjust(p);    // 调整当前顺序
                memoryAllocation[i] = p;
                Page[p].blockID = i;
                Page[p].state = true;
                Page[p].visit.loadTime += 1;
                Page[p].visit.frequency = 0;
                break;
            }
        }
    } else {    // 该页已在内存中
        flag = true;
        pageAdjust(p);
        Page[p].visit.frequency = 0;
    }
    if (flag == false) {    // 需要置换
        replacePage(f);
    }
    return flag;
}

/*
 * 页面置换算法
 * */
int replacePage(int f) {
    int i;
    int swapOut = -1;   // 被换出的页号
    int free = -1;      // 空出的物理块号
    if (f == FIFO) {
        for (i = 0; i < memoryBlocks; i++) {
            if (Page[memoryAllocation[i]].visit.loadTime > swapOut) {
                swapOut = Page[memoryAllocation[i]].visit.loadTime;
                free = i;
            }
        }
        swapOut = memoryAllocation[free];
    } else if (f == LRU) {
        for (i = 0; i < memoryBlocks; i++) {
            if (Page[memoryAllocation[i]].visit.frequency == memoryBlocks - 1) {
```

```
                free = i;
                break;
            }
        }
        swapOut = memoryAllocation[free];
    }
    // 收尾操作
    memoryAllocation[free] = -1;
    Page[swapOut].blockID = -1;
    Page[swapOut].state = false;
    Page[swapOut].visit.loadTime = 0;
    Page[swapOut].visit.frequency = -1;
    return swapOut;
}

/*
 * 换行（页表显示用）
 * */
void newLine(int length) {
    int i;
    printf("\n+--------");
    for (i = 0; i < length; i++)
        printf("-----");
    printf("+\n");
}

/*
 * 页表显示
 * */
void showData() {
    if (memoryBlocks != 0 && programPages != 0) {
        printf("\n            内存表");
        newLine(memoryBlocks);
        printf("|  块号  |");
        for (int i = 0; i < memoryBlocks; i++) {
            printf("%3d |", i);
        }
        newLine(memoryBlocks);
        printf("|  页号  |");
        for (int i = 0; i < memoryBlocks; i++) {
            printf("%3d |", memoryAllocation[i]);
        }
        newLine(memoryBlocks);
        printf("\n            页表");
        newLine(programPages);
        printf("|  页号  |");
        for (int i = 0; i < programPages; i++) {
            printf("%3d |", i);
        }
        newLine(programPages);
        printf("|物理块号|");
        for (int i = 0; i < programPages; i++) {
            printf("%3d |", Page[i].blockID);
```

```
        }
        newLine(programPages);
        printf("| 状态位 |");
        for (int i = 0; i < programPages; i++) {
            if (Page[i].state == true)
                printf("  y |");
            else
                printf("  n |");
        }
        newLine(programPages);
        printf("| 修改位 |");
        for (int i = 0; i < programPages; i++) {
            if (Page[i].modify == true)
                printf("  y |");
            else
                printf("  n |");
        }
        newLine(programPages);
        printf("|装载时间|");
        for (int i = 0; i < programPages; i++) {
            printf("%3d |", Page[i].visit.loadTime);
        }
        newLine(programPages);
        printf("|频率顺序|");
        for (int i = 0; i < programPages; i++) {
            printf("%3d |", Page[i].visit.frequency);
        }
        newLine(programPages);
    } else {
        printf("\n 尚未初始化，请先初始化数据\n");
    }
}

/*
 * 模拟操作
 * */
void simulation(int l, int f) {
    init(memoryBlocks, programPages);
    int lose = 0;
    int length[PAGE_USE_LENGTH];
    for (int i = 0; i < l; i++) {
        int a;
        a = createRandom(0, programPages - 1);
        length[i] = a;
        bool flag;
        flag = aSingleVisit(a, f);
        if (flag == false) {
            lose += 1;
            aSingleVisit(a, f);
        }
    }
    float m;
    float Lose = lose;
```

```
    float L = l;
    m = Lose / L;
    printf("\n随机访问序列为(页号) ");
    for (int i = 0; i < l; i++) {
        printf("%3d", length[i]);
    }
    printf("\n\n缺页率为%.2f%%\n", m * 100);
}

/*
 * 菜单
 * */
void menu() {
    printf("\n                              页面置换模拟\n");
    printf("\n初始化(init) 显示页表信息(show) 页面置换算法(FIFO)或(LRU) 清屏(clear)\n");
    char code[20];
    while (1) {
        printf("\n");
        scanf("%s", code);
        if (_stricmp(code, "init") == 0) {          // 初始化
            memoryBlocks = 0;
            programPages = 0;
            int a, b;
            printf("\n请输入进程的分页数(最大%d)  ", PROGRAM_PAGES);
            scanf("%d", &a);
            while (a > PROGRAM_PAGES || a < 1) {
                printf("\n输入数值非法,请重新输入", PROGRAM_PAGES);
                scanf("%d", &a);
            }
            printf("\n请输入为该进程分配的物理块数(最大%d)  ", MEMORY_BLOCKS);
            scanf("%d", &b);
            while (b > MEMORY_BLOCKS || b < 1) {
                printf("\n输入数值非法,请重新输入", MEMORY_BLOCKS);
                scanf("%d", &b);
            }
            init(b, a);
        } else if (_stricmp(code, "show") == 0) {   // 显示数据
            showData();
        } else if (_stricmp(code, "FIFO") == 0) {   // FIFO
            int l;
            printf("\n请输入页面走向长度  ");
            scanf("%d", &l);
            simulation(l, FIFO);
        } else if (_stricmp(code, "LRU") == 0) {    // LRU
            int l;
            printf("\n请输入页面走向长度  ");
            scanf("%d", &l);
            simulation(l, LRU);
        } else if (_stricmp(code, "clear") == 0) {  // 清屏
            system("cls");
            printf("\n                              页面置换模拟\n");
            printf("\n初始化(init) 显示页表信息(show) 页面置换算法(FIFO)或(LRU) 清屏(clear)\n");
        } else printf("命令无效, 请重新输入\n");
```

```
    }
}
int main() {
    srand(time(NULL));
    menu();
    getchar();
    return 0;
}
```

四、实验结果及分析

初始化：

```
CLionCode
D:\CLionCode\cmake-build-debug\CLionCode.exe

                                页面置换模拟

初始化(init) 显示页表信息(show) 页面置换算法(FIFO)或(LRU) 清屏(clear)

init

请输入进程的分页数(最大50)20

请输入为该进程分配的物理块数(最大20)10

初始化完成
```

```
show

            内存表
+-----------------------------------------------------+
| 块号  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+-----------------------------------------------------+
| 页号  | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
+-----------------------------------------------------+

            页表
+-----------------------------------------------------------------------------------------------------------+
| 页号  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
+-----------------------------------------------------------------------------------------------------------+
|物理块号| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
+-----------------------------------------------------------------------------------------------------------+
| 状态位 | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n |
+-----------------------------------------------------------------------------------------------------------+
| 修改位 | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n |
+-----------------------------------------------------------------------------------------------------------+
|装载时间| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
+-----------------------------------------------------------------------------------------------------------+
|频率顺序| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
+-----------------------------------------------------------------------------------------------------------+
```

FIFO：

```
FIFO

请输入页面走向长度20

初始化完成

随机访问序列为(页号)   7  3 14 10 16 13  2  6 15 14  0  0 11 16 13  6 12  3 15 15

缺页率为15.00%
```

```
show

            内存表
+-----------------------------------------------------+
|  块号  | 0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
+-----------------------------------------------------+
|  页号  | 11 | 12 |  3 | 10 | 16 | 13 |  2 |  6 | 15 |  0 |
+-----------------------------------------------------+

            页表
+-----------------------------------------------------------------------------------------------------+
|  页号  | 0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
+-----------------------------------------------------------------------------------------------------+
|物理块号|  9 | -1 |  6 |  2 | -1 | -1 |  7 | -1 | -1 | -1 |  3 |  0 |  1 |  5 | -1 |  8 |  4 | -1 | -1 | -1 |
+-----------------------------------------------------------------------------------------------------+
| 状态位 |  y |  n |  y |  y |  n |  n |  y |  n |  n |  n |  y |  y |  y |  y |  n |  y |  y |  n |  n |  n |
+-----------------------------------------------------------------------------------------------------+
| 修改位 |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |  n |
+-----------------------------------------------------------------------------------------------------+
|装载时间| 10 |  0 | 14 |  3 |  0 |  0 | 13 |  0 |  0 |  0 | 17 |  8 |  4 | 15 |  0 | 12 | 16 |  0 |  0 |  0 |
+-----------------------------------------------------------------------------------------------------+
|频率顺序|  7 | -1 |  9 |  1 | -1 | -1 |  3 | -1 | -1 | -1 | 10 |  6 |  2 |  4 | -1 |  0 |  5 | -1 | -1 | -1 |
+-----------------------------------------------------------------------------------------------------+
```

LRU：

```
LRU

请输入页面走向长度18

初始化完成

随机访问序列为(页号)  18 18  0  5 15  8 11 13 16 18 17  9 19 19 17  1  1  7

缺页率为16.67%
```

```
show

            内存表
+-----------------------------------------------------+
| 块号  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+-----------------------------------------------------+
| 页号  |18 |19 | 1 | 7 | 8 |11 |13 |16 |17 | 9 |
+-----------------------------------------------------+

            页表
+-----------------------------------------------------------------------------------------------+
| 页号  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |15 |16 |17 |18 |19 |
+-----------------------------------------------------------------------------------------------+
|物理块号| -1 | 2 | -1 | -1 | -1 | -1 | -1 | 3 | 4 | 9 | -1 | 5 | -1 | 6 | -1 | -1 | 7 | 8 | 0 | 1 |
+-----------------------------------------------------------------------------------------------+
| 状态位 | n | y | n | n | n | n | n | y | y | y | n | y | n | y | n | n | y | y | y | y |
+-----------------------------------------------------------------------------------------------+
| 修改位 | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n | n |
+-----------------------------------------------------------------------------------------------+
|装载时间| 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 13 | 7 | 0 | 12 | 0 | 11 | 0 | 0 | 10 | 8 | 18 | 6 |
+-----------------------------------------------------------------------------------------------+
|频率顺序| -1 | 1 | -1 | -1 | -1 | -1 | -1 | 0 | 9 | 4 | -1 | 8 | -1 | 7 | -1 | -1 | 6 | 2 | 5 | 3 |
+-----------------------------------------------------------------------------------------------+
```

FIFO，LRU 这 2 种置换算法的优劣对比：

优点：

① FIFO 页面置换算法实现简单，要求的硬件支持较少。

② LRU 页面置换算法利用“最近的过去”代替“最近的将来”，以此模拟 Optimal 算法，是实际应用中缺页率最低的算法。

缺点：

① FIFO 算法所依据的条件是各个页面调入内存的时间，而页面调入内存的先后并不能反映页面的使用情况。

② LRU 算法是根据各页以前的使用情况，来代替各页面将来的使用情况，进而判断要替换出去的页面，而页面过去和将来的走向之间并无必然的联系；其实际应用时要求较多的硬件支持，因而多采用近似算法。