

**《操作系统》课程实习报告**

**学 号：** 20201003943

**班 级：** 194201

**姓 名：** 戴子权

**任课老师：** 康晓军

**成 绩：**

**中国地质大学（武汉）**

**2022年 5 月 28**

## 一、磁盘调度

## 1.1实验内容与目的

实验目的：

1、对磁盘调度的相关知识作进一步的了解，明确磁盘调度的原理。

2、加深理解磁盘调度的主要任务。

3、通过编程，掌握磁盘调度的主要算法。

实验内容：

1、对于如下给定的一组磁盘访问进行调度：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 请求服务到达 | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| 访问的磁道号 | 30 | 50 | 100 | 180 | 20 | 90 | 150 | 70 | 80 | 10 | 160 | 120 | 40 | 110 |

2、要求分别采用先来先服务、最短寻道优先以及电梯调度算法进行调度。

3、要求给出每种算法中磁盘访问的顺序，计算出平均移动道数。

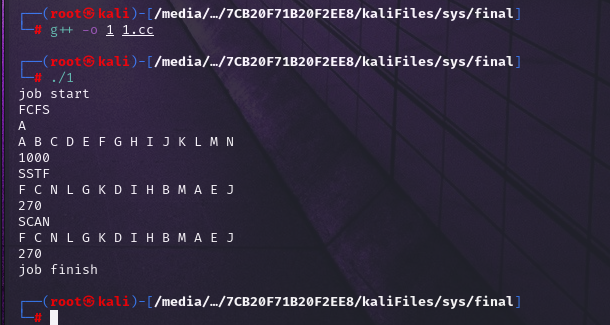
4、假定当前读写头在90号，电梯调度算法向磁道号增加的方向移动。

## 1.2实验步骤

该程序将共包含FCFS、SSTF、SCAN三个函数。

1. 首先在程序开头定义node结构体和findisk类用于构建模拟磁盘中的磁道，本程序将使用链表的数据结构来模拟磁道，在结构体中建立lst、nxt、alst、anxt指针方便模拟读写头在磁道中的移动过程，其中lst以及nxt指针用于在物理磁道号间进行移动，alst以及anxt指针用于在请求服务到达顺序号间进行移动，head以及ahead分别为物理磁道号以及访问顺序号的头指针。
2. 开始FCFS算法。该算法及先请求先服务算法，及根据请求服务的顺序进行访问，故直接输出请求顺序序列，首先从90号移动到A所在的30号，取90号到30号的绝对值作为当前移动道数，然后从A移动到B，记录下移动道数，以此类推直到移动到N，计算出移动的总道数，以及直接输出请求顺序序列，得到最终结果。
3. 开始SSTF算法。该算法为最短寻道优先算法，及从当前的90号开始，下一个访问的磁道是距离90号最近的磁道，及80号，查询到请求80号磁道的服务是I，因此输出I，然后计算90号到80号的绝对值作为当前移动的磁道数，由于开头部分设计的链表是按照磁道号从小到大递增的顺序，因此求距离当前80号磁道最近的磁道只需要通过指针nxt向后移动一位，当指针移动到队列的末尾时指针返回到90号磁道，然后向前移动，直到移动到磁道的头部，即head指针处。最终能够得到访问磁道号以及移动的总道数，得出最终结果。

4. 开始SCAN算法，该算法为扫描算法，根据当前读写头所在的位置是90号，对其进行访问的服务是F，因此从F开始，计算其到下一请求以及上一请求的距离，取较小值，然后顺着相同方向（递增或递减）进行扫描，当扫描到最外层或者最里层时读写头回到90号位置，然后向反方向进行扫描，直到扫描到最里层或最外层停止扫描，依次记录扫描过的磁道号以及计算扫描相邻两磁道号所移动的距离，最后相加整合得到最终结果。



## 二、熟悉linux文件系统调用接口

## 2.1 实验内容与目的

使用文件系统调用编写一个文件工具 filetools，使其具有以下功能：

1.创建新文件

2.写文件

3.读文件

4.修改文件权限

5.查看当前文件权限

0.退出

提示用户输入功能号，并根据用户输入的功能选择相应的功能。

文件按可变记录文件组织，具体记录内容自行设计。

## 2.2 实验步骤

该程序主要分为6个部分，分别为打开/新建文件（open函数实现），读文件（read）函数实现，写文件（write函数实现），查询文件权限（execv），修改文件权限（chmod）以及lseek调用（lseek函数实现）。下文将一一介绍。

1. 打开/新建文件。该功能由系统函数int open(const char \*pathname, int oflag, ... /\* mode\_t mode \*/);实现，该函数有三个参数，第一个参数const char \*pathname是要打开或创建的文件的名字；第二个参数int oflag用来说明打开文件的方式，分为3个选项，O\_RDONLY（只读打开）,O\_WRONLY（只写打开）,O\_RDWR（读写打开）；第三个参数是可选参数，有O\_APPEND,O\_CREAT,O\_EXCL,O\_TRUNC,O\_NONBLOCK以及O\_SYNC选项，其中：

O\_APPEND 每次写时都加到文件的尾端。

O\_CREAT 若此文件不存在则创建它。使用此选择项时，需同时说明第三个参数mode，用其说明该新文件的存取许可权位。

O\_EXCL 如果同时指定了O\_CREAT，而文件已经存在，则出错。这可测试一个文件是否存在。

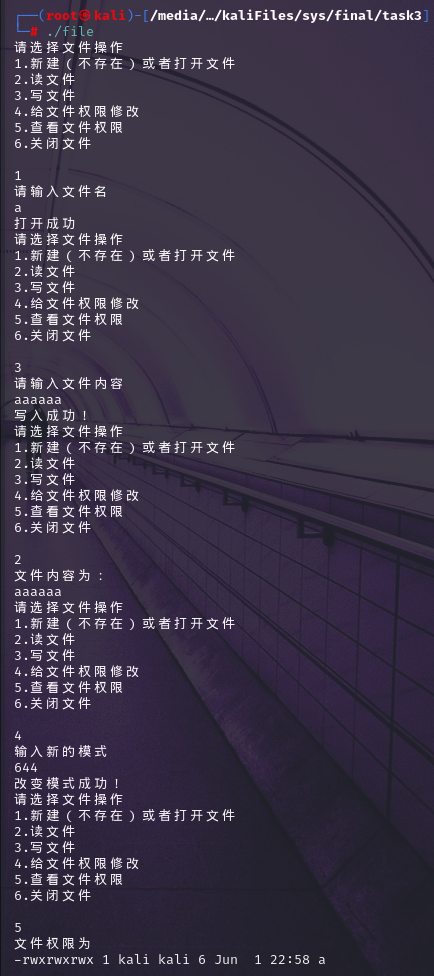
O\_TRUNC 如果此文件存在，而且为只读或只写成功打开，则将其长度截短为0。

O\_NONBLOCK 如果pathname指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的I /

O操作设置非阻塞方式。

O\_SYNC 使每次write都等到物理I / O操作完成。

1. 读/写文件。该功能由包含于头文件unistd.h中的函数ssize\_t read(int fd, void \*buff, size\_t nbytes) ;以及ssize\_t write(int fd, const void \*buf, size\_t nbytes) ;完成。默认情况下在打开文件时写操作都从文件当前位移量处开始，若在打开文件时使用O\_APPEND选项指定开始位置，则在每次写操作之前都会从当前位置结尾处开始，在成果读写一次后，该文件位移量增加实际写的字节数。
2. 查文件权限。该功能由函数char\* pargv[4]={"ls","-l","file1",NULL}实现，通过函数execv("/bin/ls",pargv);显示文件权限，查询文件权限只需要将参数pargv改为待操作的文件名，然后执行execv(“/bin/ls”,pargv);函数即可。
3. 修改文件权限。该功能通过chmod命令实现。Linux下有三类文件权限，分别是：owner,group,others,每类都占权限位的三位，每一类有三种状态，分别为r=4，w=2，x=1，其中r为可读，w为可写，x为可以执行。例如：-rw-r-r（644）为owner有读写权限，group和others都是只有可读权限，在改变模式时需要用到chmod，实例为：chmod(filename, new\_mode)，其中filename为文件名，new\_mode为新的权限模式。

lseek调用。当前文件位移量是一个非负整数，用来度量从文件开始处计算的字节数，对文件进行的各种操作一般都是从当前文件位移量处开始，按照系统默认设定，每当打开一个文件时除非指定O\_APPEND选项，否则位移量默认设置为0。具体调用方式如下：off\_t lseek(int fd, off\_t offset, int whence);其中参数offset与whence的值有关，如果whence时SEEK\_SET，则将当前文件位移量设置为距离文件开始处offset个字节，如果whence时SEEK ——CUR,则将当前文件的位移量设置为其当前值增加offset，offset可以为正数或者负数，如果whence是SEEK\_END，则将当前文件的位移量设置为文件长度加offset，offset也是可以为正值或者负值。

**三、进程管理**

**3.1 实验内容与目的**

实验目的：

1.理解进程的概念，明确进程和程序的区别。

2.理解并发执行的实质。

3.掌握进程的同步、撤销等进程控制方法。

实验内容：

父进程使用系统调用pipe()建立一个管道，然后使用系统调用fork()创建两个子进程：子进程1和子进程2

子进程1每隔1秒通过管道向子进程2发送数据：I send message x times.（x初值为1，以后发送一次后做加一操作）子进程2从管道读出信息，并显示在屏幕上

父进程用系统调用signal()来捕捉来自键盘的中断信号SIGINT（即按Ctrl+C键,）；当捕捉到中断信号后，父进程用系统调用kill()向两个子进程发出信号，子进程捕捉到信号后分别输出如下信息后终止： Child Process 1 is killed by Parent!

Child Process 2 is killed by Parent!

父进程等待两个子进程终止后，释放管道并输出如下的信息后终止   
 Parent Process is Killed!

**3.2 实验步骤**

1.首先使用pipe创建管道，然后设置信号（SIGINT）并创建两个子进程1和2

2.子进程1的操作：

1.屏蔽SIGINT信号

2.关闭读管道

3.设置信号SIGABRT，用于接受父进程的通知

4.开始使用循环语句向管道写入数据“I send message x times”, x=1++;

3.子进程2的操作：

1.屏蔽SIGINT信号

2.关闭写管道

3.设置信号SIGABRT，用于接受父进程的通知

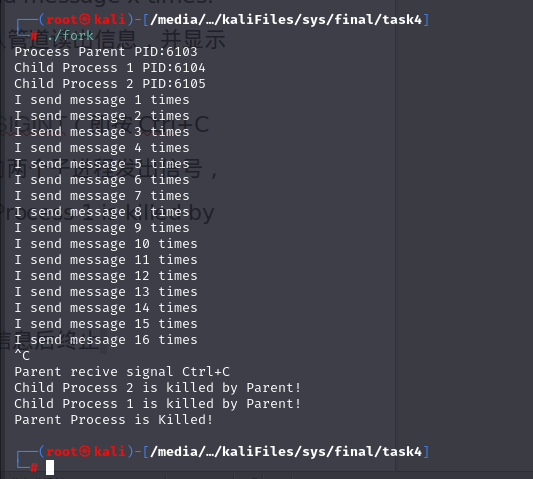
4.从管道循环中读取数据并输出

6.父进程：

1.阻塞并等待两个子进程结束

2.输出”Parent Process is killed”

3.关闭管道

****

1. **请求分页系统中的置换算法**

## 实验内容与目的

实验目的：

1. 了解虚拟存储技术的特点；
2. 掌握请求分页系统的页面置换算法。

实验内容：

1.通过如下方法产生一指令序列，共 320 条指令。

* 1. 在[1，32k-2]的指令地址之间随机选取一起点M,访问 M；
  2. 顺序访问M+1；
  3. 在[0，M-1]中随机选取M1，访问 M1；
  4. 顺序访问M1+1；
  5. 在[M1+2，32k-2]中随机选取M2，访问 M2；
  6. 顺序访问M2+1；
  7. 重复 A—F，直到执行 320 次指令。

1. 指令序列变换成页地址流设：（1）页面大小为 1K；

（2）用户虚存容量为 32K。

1. 计算并输出下述各种算法在不同内存页块(页块个数范围:8-32)下的命中率。
   1. 先进先出（FIFO）页面置换算法
   2. 最近最久未使用（LRU）页面置换算法
   3. 最佳（Optimal）页面置换算法

## 实验步骤

1. 该函数将涉及到三个算法，FIFO算法，LRU算法，Optimal算法，三种算法对应三个函数，每个函数都是double f(int a)的格式，参数中的参数为当前内存块的大小，返回值是命中率。
2. 首先设置全局变量用于保存指令个数以及指令数组，通过make\_instruct()函数生成随机数填充指令数组，用链表来组织内存。在该函数中instruct表示指令号。结构体部分代码如下：

srand((unsigned int)time(NULL));

int rand(int min, int max)

{

return rand() % (max - min + 1) + min;

}

1. 产生指令序列。需要连续获得多个随机数，该函数再main函数中生成一个种子，然后不断调用函数。Rand函数代码如下：

srand((unsigned int)time(NULL));

int rand(int min, int max)

{

return rand() % (max - min + 1) + min;

}

为构造指令序列，需要一个循环不断调用以上的rand函数，具体过程如下：

void make\_instruct() {

int n = 0;

while (n < instruct\_num) {

int M = rand(1,30);

instruct\_arr[n++] = M;

instruct\_arr[n++] = M + 1;

int M1 = rand(0, M - 1);

instruct\_arr[n++] = M1;

instruct\_arr[n++] = M1 + 1;

int M2 = rand(M1 + 2, 30);

instruct\_arr[n++] = M2;

instruct\_arr[n++] = M2 + 1;

}

}

1. 讨论先进先出算法。首先新建头节点head，头节点的instrcut域表示当前内存中实际被占用的部分，head的next域指向内存中第一个被占用的块。再定义一个指向内存尾部的指针end。接下来说明先进先出算法的具体实现过程，算法首先遍历所有指令，并且每次判断内存是否有空闲空间，若没有空闲空间则需要淘汰当前队列中的一个指令，并将当前指令插入到合适位置，若有空闲空间则将指令直接加入链表。然后判断该指令是否再内存当中，若在则将hit值置为1，且不做以上操作，若不在内存中则将failur\_times值加一，并进行上述操作，判断内存是否还有空闲空间。在该算法中每次找到最先进入内存的指令，每次将进入的指令插入到链表表尾，链表表头后面的指令总是最后进入内存的，故每次当内存空间不足时删除链表表头的指令并将新的指令加入到链表表尾，并且每次更新end指针的位置。
2. 讨论最久未使用算法。该算法与先进先出算法的不同在于LRU算法具有变量clock，初始值为999，每循环一次clock值减一。若当前指令在内存中则将其实践刷新，置为当前的时钟值。每次淘汰的过程中需要遍历内存序列找到time值最大的，并将其置换为新的节点，新加入的节点的time值为当前时钟clock。
3. 讨论最佳算法。该算法过程较为简单，运行起来较为单一，手段直接，但代价是算法效率太低。下面介绍具体过程。每次遍历内存队列中的指令时，对于每个指令重新遍历全部指令集，找到将来这条指令还需多久会被再次执行，若该条指令在最长时间内未被使用到则淘汰该条指令用于储存新的指令。

