计算机操作系统实验报告

学号：102201307 姓名：邱聪祝

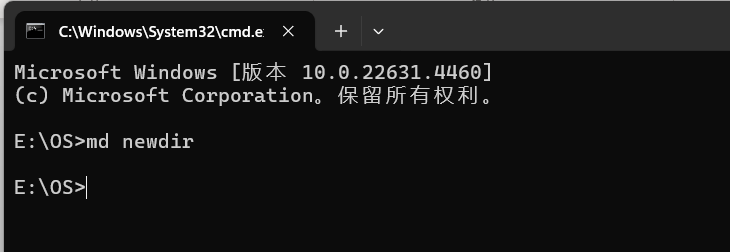
第一部分 Windows操作系统实验

1. Windows基本字符命令的使用

文件系统是操作系统的重要组成部分，用于存储、管理和组织文件及目录。Windows操作系统通过基本字符命令为用户提供了直接操作文件系统的接口，这些命令可以高效地完成文件与目录的管理、磁盘操作以及文件属性的设置。而除了文件系统相关的命令外，Windows的基本字符命令还包含了系统配置、网络管理、进程和服务管理等多种其他的接口，但这里就仅围绕文件系统相关的命令接口进行展开。

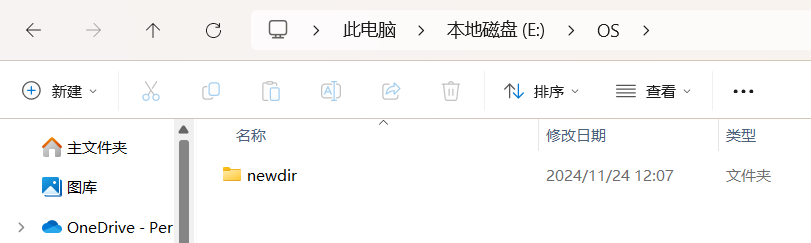
1. md命令

md命令的功能为在当前目录下创建一个子目录，比如在图1中，想要在E:\OS目录下新建一个子目录名为newdir，则可以使用md newdir命令来进行创建。



**图1**

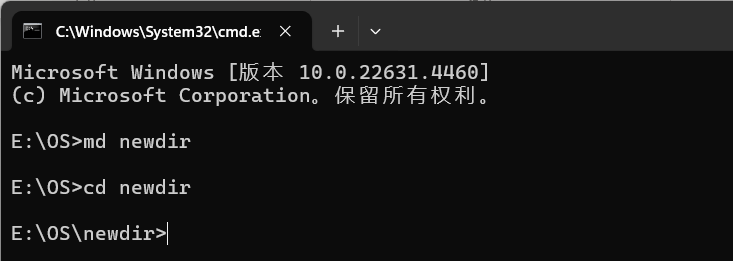
创建完成之后，打开E盘的OS文件夹，如图2所示，可以看到成功的创建了一个名为newdir的子目录。



**图2**

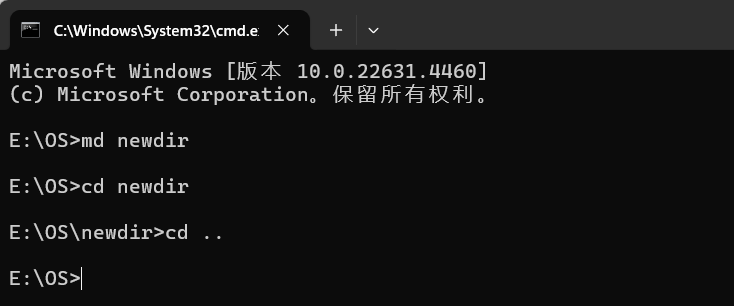
1. cd命令

cd命令的功能为改变或显示当前工作目录，一般可以用来进入到当前目录的子目录中，如图3所示，使用cd newdir将会使终端进入新创建的newdir子目录。



**图3**

如果需要返回到当前目录的父级目录，则可以使用cd ..命令来进行实现。如图4所示。



**图4**

1. dir命令

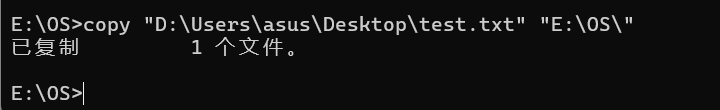
dir命令的功能是查看指定目录的文件及子目录，直接使用dir命令会默认查看当前目录下的内容，而如图5所示使用dir C:\Users则可以查看C:\Users目录下的内容。



**图5**

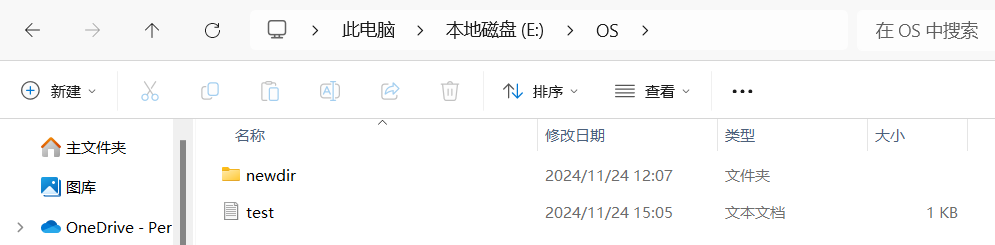
1. copy命令

copy命令的功能是进行文件的复制，其基本的语法为copy [<源文件路径>] [<目标路径>]，如图6所示，执行copy "D:\Users\asus\Desktop\test.txt" "E:\OS\"命令，可以将电脑桌面上的test.txt文件复制到E盘的OS文件夹中。



**图6**

而在OS文件夹中，也可以验证test.txt文件被成功的复制过来了，如图7所示。



**图7**

1. type命令

type命令的功能是查看文本文件的内容，它可以以简单直观的方式将文件内容输出到命令行窗口，但是不支持编辑，仅用于快速地查看。如图8所示，使用type test.txt命令可以将刚才复制进来的test.txt文件的内容展示在终端里面。



**图8**

除了上面列出的5个基本的命令以外，FAT文件系统的文件使用命令还有不少，并且对于这5个命令，它们在使用时也具有更多不同的高级用法，这里就不再进行过多的展示了。

1. 并发程序设计
2. 习题第11题，重复消费PC问题

（1）题目

有3个进程R、D、S共享一个缓冲区buf。进程R每次从输入设备上读取一组数据并存入缓冲区buf中。进程D把缓冲区中的数据取出并在屏幕上显示，同时，进程S把缓冲区buf中的数据取出并保存到磁盘上。规定：R每次读取的数据都要在屏幕上显示，同时也要保存到磁盘上。请用信号量机制实现R、D、S这3个进程的并发执行。

（2）解题思路

首先在本题中，每个进程都有两个主要的步骤，需要先进行列出。在R进程中的两个主要的步骤为：读取一组数据，将数据存入缓冲区；D进程中的为：将buf中的数取出，将数据在屏幕上显示；S进程中的为：将buf中的数据取出，将数据保存到磁盘上。

在弄清楚所有的主要操作之后，就需要分析同步和互斥的情况。首先，D和S进程中取数据的操作应该在R进程中存入数据之后，这里分别需要一个不同的信号量full1和full2来表示缓冲区中是有数据的，在R中存入缓冲区后使用v操作将信号量+1，D进程和S进程的开始处使用p操作来将信号量-1。需要注意这里不能仅使用一个信号量，因为这样的话在R进程执行完一次之后，D进程和S进程就只有其中一个能进行了，故需要两个信号量。而在R进程读取数据后，存入缓冲区前，必须保证D进程和S进程都已经将数据从buf中取出，所以这里也是分别需要一个不同的信号量empty1和empty2来表示缓冲区是否为空，即新数据是否可以直接进行覆盖。

在处理完了同步信号量之后，还需检查是否需要额外的互斥信号量来避免错误的产生。这里R与D进程，R与S进程在读写缓冲区时需要互斥，但是同步信号量已经规定了其访问的先后顺序不同出现同时访问的情况，所以不需要额外的互斥信号量；D和S进程在读取缓冲区时也不需要进行互斥操作，所以本题中不需要额外的互斥信号量。

（3）并发程序设计

Semaphore empty1 = 1, empty2 = 1, full1 = 0, full2 = 0;

R(){

读取一组数据;

p(empty1);

p(empty2);

存入缓冲区buf;

v(full1);

v(full2);

}

D(){

p(full1);

将buf中的数据取出;

v(empty1);

在屏幕上显示数据;

}

S(){

p(full2);

将buf中的数据取出;

v(empty2);

将数据保存到磁盘上;

}

（4）bacc源代码

在bacc的源代码中，有一些地方和并发程序设计中的结果有所不同。首先是信号量的命名，为了避免名称冲突的错误，在代码中将empty1, empty2, full1, full2分别命名为了s1, s2, s3, s4；其次，这里将buf缓冲区定义为了一个int变量，每个进程中暂存数据的地方使用局部变量x, y1, y2来替代；最后，由于在bacc进行模拟的时候，难以做到将数据保存到磁盘上的操作，所以这里将该信息打印在屏幕上以作替代。但需要额外注意的是，将信息打印在屏幕上就意味着打印的临界资源可能发生共用，所以需要在打印操作前后多加一个mutex互斥信号量，来保证其中一个进程在打印时另一个进程并发进行打印。下面为bacc的具体源代码。

int buf = 0;

semaphore s1 = 1, s2 = 1, s3 = 0, s4 = 0, mutex = 1;

void R()

{

int x = 0;

cout << "input x = ";

cin >> x;

cout << endl;

p(s1);

p(s2);

buf = x;

v(s3);

v(s4);

}

void D()

{

int y1 = 0;

p(s3);

y1 = buf;

v(s1);

p(mutex);

cout << "D().y1 = " << y1 << " has been displayed on the screen" << endl;

v(mutex);

}

void S()

{

int y2 = 0;

p(s4);

y2 = buf;

v(s2);

p(mutex);

cout << "S().y2 = " << y2 <<" has been saved to the disk"<< endl;

v(mutex);

}

main()

{

cobegin{

R();

D();

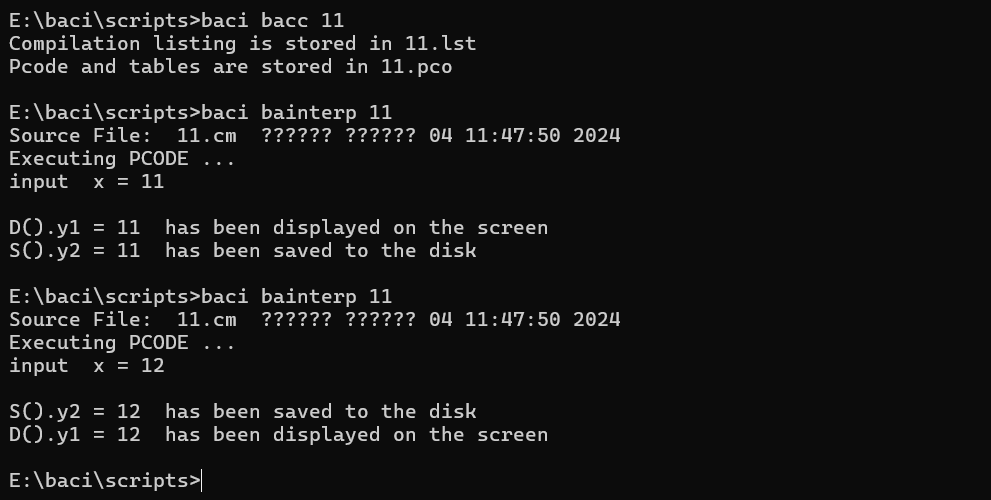
S();

}

}

（5）运行结果

将源代码文件命名为11.cm，并使用baci bacc 11进行编译，得到11.lst和11.pco文件，之后执行baci bainterp 11命令来运行程序。其中，进程R先执行，在屏幕上打印出input x =，提示用户输入一个数字，此时由于信号量s3和s4的原因D和S进程处于阻塞状态。当输入11并回车之后，D和S进程并发执行，打印出了将数据11显示再屏幕上，以及保存到磁盘中的提示信息。当再次运行该程序时，可以发现S和D进程打印信息的顺序发生了变换，体现出了并发执行的随机性。具体运行的结果如图9所示。



**图9**

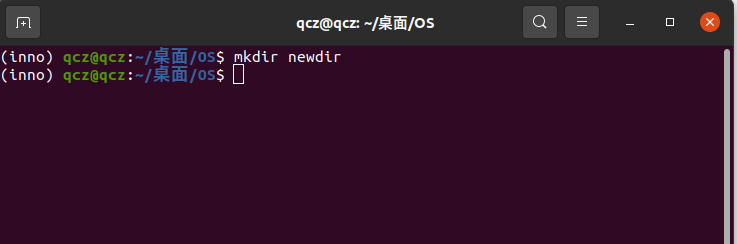
第二部分 Linux操作系统消息队列通信及其应用

1. 基本shell命令使用

在Linux环境中，系统的命令接口称为Shell命令，程序接口为系统调用。Shell命令不仅可以完成文件和目录管理、用户权限操作等基本任务，还支持复杂的脚本编程，极大地提升了操作效率。相较于第一部分中Windows的字符命令，Shell命令更注重简洁性与扩展性，具备更强的组合能力。在这部分中，我们将选取5个Linux操作系统中的基本Shell命令，并与对应的Windows基本字符命令进行对比，分析它们在操作逻辑和功能设计上的异同。

1. mkdir命令

mkdir命令的功能为创建一个子目录，使用的时候直接在当前目录下执行mkdir，或者在后面加上一个路径比如mkdir /usr/myData，在指定的路径下创建一个子目录。该命令的功能类似于Windows基本字符命令中的md命令，需要注意的是Linux操作系统中的文件路径是以/为分隔符，而Windows是以\为分隔符。如图10所示在桌面中的OS文件夹下创建了一个名为newdir的子目录。

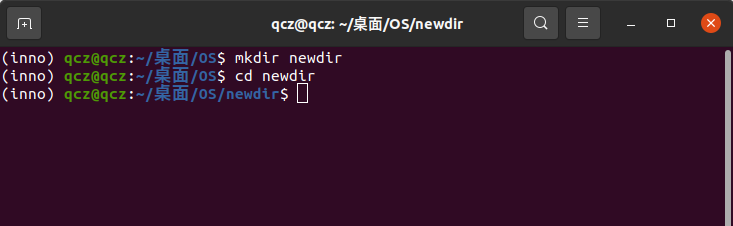
图10

打开桌面上的OS文件夹，如图11所示，newdir被创建，可以验证命令mkdir newdir成功执行。

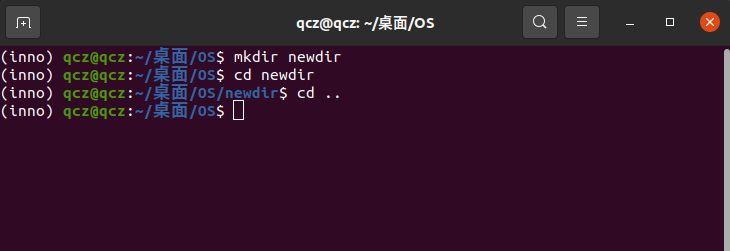
图11

1. cd命令

cd命令的功能为改变当前的工作目录，使用时需要在cd后加上工作目录的绝对或相对路径，若进入当前目录子目录则加上子目录名称。该命令和Windows基本字符命令中的cd命令是几乎一样的，这里同样采用进入新创建的newdir来进行验证，执行结果如图12所示。

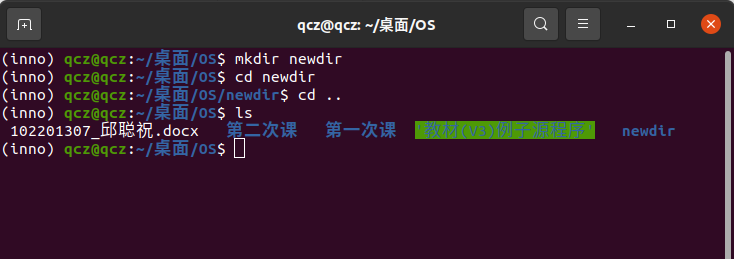
图12

如果要返回上一级的目录，则同样可以使用cd ..来进行返回，执行结果如图13所示。

图13

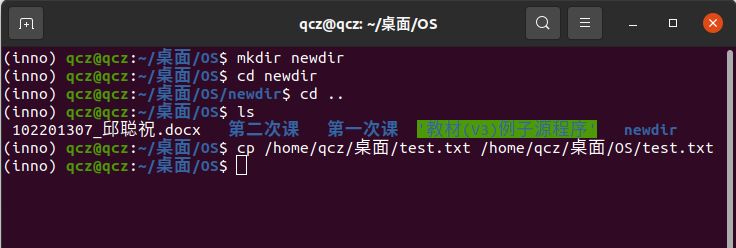
1. ls命令

ls命令的功能为列出当前工作目录下的文件名及子目录名，直接使用ls命令会默认查看当前目录下的内容。该命令和Windows基本字符命令中的dir命令较为类似，具体的执行结果如图14所示。

图14

1. cp命令

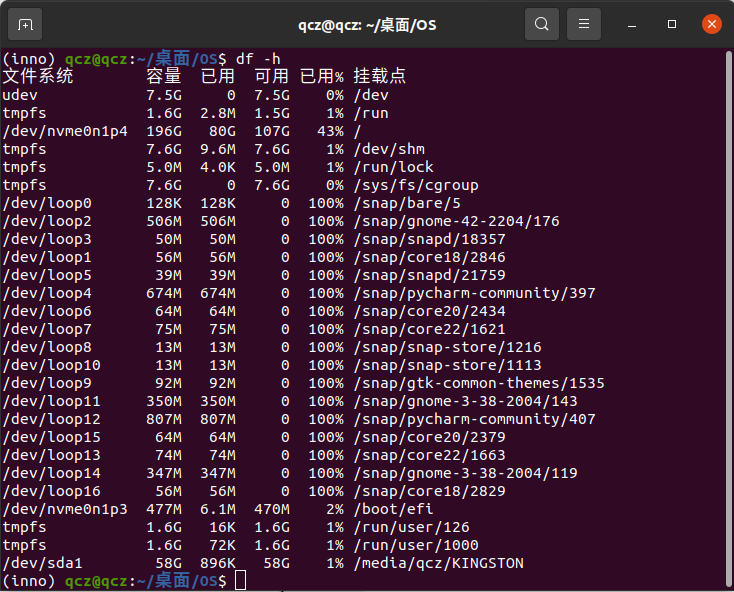
cp命令的功能为复制文件，所对应的windows基本字符命令为copy。他们的具体用法也较为相似，cp的用法为cp [<源文件路径>] [<目标路径>]，例如cp /home/qcz/桌面/test.txt /home/qcz/桌面/OS/test.txt，可以将桌面的test.txt文件复制到OS文件夹中，命令执行结果如图15所示。

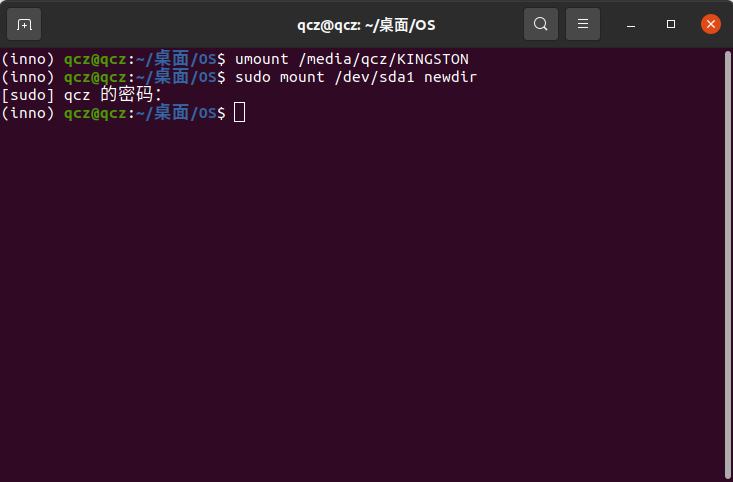
图15

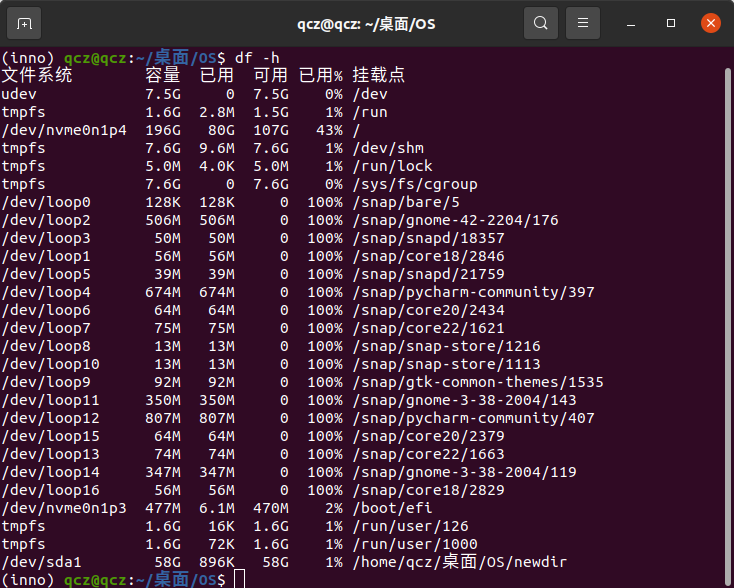
打开OS文件夹，可以从图16看出test.txt确实被复制了过来，验证了cp命令的成功执行。

图16

1. mount / umount命令

mount命令的功能是装载文件系统，而umout命令的功能是卸载文件系统，这两条命令经常被应用于插入U盘后的挂载操作上。挂载是操作系统加载文件系统的过程，通过挂载，系统能够解析设备上的文件系统结构，并将文件和目录映射到挂载点，方便用户访问。如果不进行挂载的话，用户就无法以文件和目录的形式来访问数据。mount的一般使用方法为mount <设备文件> <挂载点>，umount则为umount <挂载点>。如图17所示为使用命令





1. 消息队列通信的4个系统调用的简介

消息队列是 Linux 操作系统中重要的进程间通信机制之一，用于在不同进程之间传递结构化的数据。通过消息队列，进程可以实现异步通信，发送方无需等待接收方的响应，消息将暂存于内核中的队列中直至被读取。该部分内容为针对消息队列的创建、发送、接收和删除这4个基本操作的实践，学习其具体的工作原理和使用方法。

1. msgget

功能：创建或获取一个消息队列。

格式：int msgget(key\_t key, int msgflg);

参数：key是消息队列的逻辑标识符，用户通过它来查找或创建消息队列，该值可以被直接指定，也可以用 ftok 函数来生成。msgflg 为消息队列的权限和行为控制标志，如果消息队列不存在且设置了 IPC\_CREAT，则会创建一个新的队列。

返回值：成功时返回该消息队列的标识符msqid，该值为内核分配的整数，用于唯一标识此队列。用于后续的消息发送、接收和管理。失败时则返回-1。

程序中的使用：在课上演示的程序中，该系统调用的使用为qid = msgget(key, 0777 | IPC\_CREAT);其中key手动设置为了85，使用了权限 0777，允许所有用户对该队列进行读写。并且在服务端指定了 IPC\_CREAT，负责初始化一个新的消息队列；而在客户端没有指定 IPC\_CREAT，需要通过同一个 key 与服务端共享同一个消息队列。

2. msgsnd

功能：向指定的消息队列发送消息。

格式：int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

参数：msqid为消息队列的标识符，通过 msgget 获得。msgp为指向要发送的消息的指针，消息的结构体通常由long mtype;代表消息类型，以及消息正文和其他内容组成。msgsz为消息正文的长度。msgflg控制发送行为，为0时是阻塞模式，当消息队列满时进程会进入阻塞状态；为IPC\_NOWAIT时是非阻塞模式，队列满时会立即返回错误。

返回值：若发送成功则返回0。若发送失败则返回-1，并设置 errno。

程序中的使用：在课上演示的程序中，该系统调用的使用为msgsnd(qid, &m, mlen, 0);将消息 m 发送到标识为qid的消息队列中。在服务端中，消息m中会设置 mtype 为客户端的 pid，确保消息是专门为该客户端发送的；而在客户端中会设置 mtype为1，确保服务端能够接收此消息。最后用0设置阻塞模式，当消息队列满了的时候进程进入阻塞状态。

3. msgrcv

功能：从指定的消息队列中接收消息。

格式：ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);

参数：msgqid为消息队列的标识符，通过 msgget 获得。msgp指向存储接收消息的缓冲区，是与发送消息相同的结构体类型。msgsz为接收消息正文的大小。msgtyp为消息类型，0为接收队列中最早的消息；正整数为接收指定类型的消息；负整数为接收小于等于 |msgtyp| 的最小类型消息。msgflg控制发送行为，与前面的相同，为0时是阻塞模式；为IPC\_NOWAIT时是非阻塞模式。

返回值：若接受成功则返回接收的消息正文大小。若接收失败则返回 -1，并设置 errno。

程序中的使用：在课上演示的程序中，该系统调用的使用在服务端为msgrcv(qid, &m, mlen, 1, 0);而在客户端则为msgrcv(qid, &m, mlen, pid, 0);这里的参数与msgsnd类似，唯一不同的在于msgtyp，由于客户端发送给服务端的消息中mtype设置为了1，所以这里需要接收类型为1的消息；而服务端发送给客户端的消息中mtype设置为了pid，也就是客户端进程的pid，所以客户端需要接收的消息类型为自己的pid。这样的设置能够保证客户端与服务端在多对一通信中，准确的接收到只属于自己的消息。

4. msgctl

功能：对消息队列进行控制，如获取状态、修改属性、删除队列。

格式：int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

参数：msqid为消息队列的标识符，通过 msgget 获得。cmd为控制命令，为IPC\_STAT时获取队列状态，存储到 buf 中；为IPC\_SET时设置队列属性，比如权限，从 buf 获取；为IPC\_RMID时删除消息队列。buf为指向消息队列状态结构体的指针，用于存储或设置状态信息，如果要删除消息队列的话此项置为0就好了。

返回值：若操作成功则返回0。若操作失败则返回 -1，并设置 errno。

程序中的使用：在课上演示的程序中，该系统调用的使用为msgctl(qid, IPC\_RMID, 0);且仅出现在服务端。其中cmd被指定为了IPC\_RMID，buf设置为了0，用于服务端在通信结束后删除标识符为qid的消息队列，释放系统资源。

通过上面四个基本的系统调用，就可以完整地实现 Linux 下的消息队列通信了。

通过 msgget、msgsnd、msgrcv 和 msgctl 四个核心系统调用，就可以完整地实现消息队列的创建、消息的发送与接收以及队列的管理等功能了。它允许不同进程通过内核维护的队列以结构化数据的形式进行交互，适用于多进程环境中的数据共享与任务协作。在实际应用当中，合理设置消息类型与权限还能够提升通信的安全性与效率，是开发复杂 IPC 程序的重要工具之一。

1. 消息队列通信的CS结构的应用

消息队列作为一种高效的进程间通信机制，在实现客户端-服务器架构中具有广泛的应用价值。在这一部分中，我们需要基于教材中关于四则运算应用的源码进行修改和补充，构建出一个全新的使用消息队列通信的 CS 应用。通过在本地打开多个进程，来模拟客户端发送请求、服务器处理并返回响应的过程，并深入理解消息队列的工作原理和实际使用中的方法与技巧。

1. 应用介绍及功能

本程序的主要功能是通过消息队列实现客户端与服务端的交互，完成 n 阶矩阵行列式的计算。客户端负责接收用户输入，包括矩阵的阶数和具体元素，将这些数据封装为消息发送至服务端；服务端接收消息后，解析矩阵数据并通过递归算法计算行列式的值，然后将结果返回给客户端。程序的亮点在于支持动态输入矩阵阶数，而不是局限于单一的矩阵阶数，这样能够使程序更加灵活，适用不同的场景。同时，服务端也能够识别非法请求，比如阶数超出范围，并且会返回详细的错误信息。客户端会根据服务端返回的状态标识正确显示计算结果或提示错误。此外，程序也具备了多客户端支持的功能，服务端按照请求顺序依次处理，能够确保通信的准确性，不会将计算结果发送给错误的进程。

2. 客户端主要流程及代码

// head.h 头文件

#define MSGKEY 1234 // 定义消息队列的唯一键值，用于标识消息队列

#define MAX\_N 10 // 定义支持的最大矩阵阶数

// 定义消息数据结构

struct msgform {

long mtype; // 消息类型，决定消息的接收方

int source\_pid; // 消息发送者的进程ID，用于识别客户端

int n; // 矩阵的阶数

double matrix[MAX\_N][MAX\_N]; // 用于存储矩阵数据

double result; // 存储行列式的计算结果

char result\_msg[128]; // 存储计算状态信息，标记成功或错误

};

// 定义消息正文的大小，排除消息类型字段的大小

int msgsize = sizeof(struct msgform) – sizeof(long);

// client.c 客户端文件

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#include "head.h"

int main() {

struct msgform msg; // 定义消息结构体实例

int pid; // 当前客户端的进程ID

int msgqid; // 消息队列标识符

// 获取消息队列ID，与服务器共享消息队列

msgqid = msgget(MSGKEY, 0777);

if (msgqid < 0) { // 如果失败，打印错误信息并退出

perror("msgget");

return -1;

}

pid = getpid(); // 获取当前客户端进程的PID

while (1) { // 循环接收用户输入并与服务器交互

printf("Enter the matrix dimension (n): ");

scanf("%d", &msg.n); // 输入矩阵的阶数

// 检查阶数是否有效，必须在 1 到 MAX\_N 之间

if (msg.n <= 0 || msg.n > MAX\_N) {

printf("Error: Matrix dimension must be between 1 and %d.\n", MAX\_N);

continue;

}

printf("Enter the %dx%d matrix (row by row):\n", msg.n, msg.n);

for (int i = 0; i < msg.n; i++) { // 循环逐行输入矩阵元素

for (int j = 0; j < msg.n; j++) {

printf("matrix[%d][%d]: ", i, j);

scanf("%lf", &msg.matrix[i][j]); // 存储到矩阵字段中

}

}

msg.source\_pid = pid; // 设置消息的来源PID

msg.mtype = 1; // 消息类型为1，表示发送给服务器

// 发送消息到服务器

msgsnd(msgqid, &msg, msgsize, 0);

// 等待服务器返回结果，消息类型为客户端的PID

msgrcv(msgqid, &msg, msgsize, pid, 0);

// 根据返回的状态信息，打印结果或错误信息

if (msg.result\_msg[0] == '1') {

printf("Determinant of the matrix is: %.2f\n", msg.result);

} else {

printf("Error: %s\n", msg.result\_msg);

}

// 提示用户是否继续计算

char cont;

printf("Continue? (y/n): ");

getchar(); // 消耗多余的换行符

cont = getchar(); // 获取用户输入

if (cont == 'n' || cont == 'N') { // 如果用户选择不继续，退出循环

break;

}

}

return 0;

}

3. 服务端主要流程及代码

// server.c 服务端源文件

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#include "head.h"

// 消息队列标识符，用于创建和管理消息队列

int msgqid;

// 清理函数，当接收到中断信号时调用

void cleanup() {

msgctl(msgqid, IPC\_RMID, 0); // 删除消息队列

printf("\nServer: Message queue deleted.\n");

exit(0); // 退出程序

}

// 递归计算 n 阶矩阵的行列式，每次调用时要做的事情

// 仅为对当前矩阵的第一行每个元素进行展开，并构造子矩阵

// 在累加第一行元素 \* 符号 \* 子矩阵行列式的同时，进行递归调用

double determinant(double matrix[MAX\_N][MAX\_N], int n) {

if (n == 1) {

return matrix[0][0]; // 1阶矩阵的行列式是唯一元素

}

double det = 0.0; // 初始化行列式结果

double submatrix[MAX\_N][MAX\_N]; // 用于存储子矩阵

for (int x = 0; x < n; x++) { // 对第一行展开

int subi = 0; // 子矩阵的行索引

for (int i = 1; i < n; i++) { // 遍历原矩阵的剩余行

int subj = 0; // 子矩阵的列索引

for (int j = 0; j < n; j++) { // 遍历原矩阵的列

if (j == x) continue; // 跳过当前列

submatrix[subi][subj++] = matrix[i][j]; // 构造子矩阵

}

subi++;

}

double sign = (x % 2 == 0) ? 1 : -1; // 根据列索引计算符号

det += sign \* matrix[0][x] \* determinant(submatrix, n - 1); // 递归计算余子式

}

return det;

}

int main() {

struct msgform msg; // 定义消息结构体实例

// 捕捉中断信号（如 Ctrl+C），以清理消息队列

signal(SIGINT, cleanup);

// 创建或获取消息队列

msgqid = msgget(MSGKEY, 0777 | IPC\_CREAT);

if (msgqid < 0) { // 如果创建失败，打印错误信息并退出

perror("msgget");

return -1;

}

printf("Server is ready. PID=%d, msgqid=%d\n", getpid(), msgqid);

while (1) { // 循环处理客户端请求

// 等待接收消息类型为 1 的消息

msgrcv(msgqid, &msg, msgsize, 1, 0);

printf("Server: Received %dx%d matrix from PID=%d\n", msg.n, msg.n, msg.source\_pid);

// 检查矩阵阶数是否超过最大限制

if (msg.n > MAX\_N) {

snprintf(msg.result\_msg, sizeof(msg.result\_msg), "Error: Matrix dimension exceeds MAX\_N (%d).", MAX\_N);

msg.result\_msg[0] = '0'; // 标记错误状态

} else {

msg.result = determinant(msg.matrix, msg.n); // 计算行列式

msg.result\_msg[0] = '1'; // 标记成功状态

}

// 设置返回消息类型为客户端的PID

msg.mtype = msg.source\_pid;

// 发送计算结果回客户端

msgsnd(msgqid, &msg, msgsize, 0);

printf("Server: Sent determinant %.2f to PID=%d\n", msg.result, msg.source\_pid);

}

return 0;

}

4. 运行结果展示

