目 录

[1 任务说明 1](#_Toc166249014)

[1.1 题目描述 1](#_Toc166249015)

[1.2 任务要求 2](#_Toc166249016)

[2 设计 2](#_Toc166249017)

[2.1 进程结构设计 2](#_Toc166249018)

[2.2 数据结构设计 3](#_Toc166249019)

[2.3 算法流程设计 3](#_Toc166249020)

[3 实现 5](#_Toc166249021)

[3.1 实现环境 5](#_Toc166249022)

[3.2 关键代码 5](#_Toc166249023)

[3.3 运行效果 8](#_Toc166249024)

[4 总结 11](#_Toc166249025)

[4.1 问题与解决方法 11](#_Toc166249026)

[4.2 结果分析 12](#_Toc166249027)

[5 附录 13](#_Toc166249028)

**多进程协同完成目录复制**

报告主要内容包含：

1. 题目需求分析，即任务说明，分析题目明确任务要求；

2. 程序设计，包含针对需求的进程结构设计、数据结构设计、算法流程设计；

3. 实现过程，介绍实验环境，讲述实现步骤，给出关键代码解析；

4. 运行测试效果，根据题目要求使用不同命令行参数测试运行程序，给出运行效果；

5. 附录，包含程序源代码和可执行文件。

# 任务说明

## 题目描述

在Linux系统下用C语言实现多进程合作的目录复制功能，程序支持命令行运行，包含两个命令行参数，源目录地址和目的目录地址，源目录地址默认为当前目录；

要求复制功能需要由三个进程协作执行，包含父进程A和两个子进程B、C；进程A向B传输源目录地址，进程B遍历源目录读取文件，收集文件名和文件内容传输给进程C，然后等待进程C响应；进程A同时向进程C传输目的目录地址，进程C接收到B传输的文件名和文件内容后，打开目的目录，创建需要复制的文件并写入内容，完成后将结果信息传输给B，将写入的文件大小和文件数量传输到进程A，B收取到结果信息后也返回文件大小和文件数量给父进程A；进程间需要选取合适的通信方式进行数据传输。

执行命令形式：**xcopy [-i <srcpath>] -o dstpath**，表示需要将srcpath目录下的文件复制到dstpath中；

参数含义：

[-i <srcpath>]源目录地址，可以省略，省略则默认源目录为当前目录；

-o dstpath目的目录地址，即需要复制到的目录位置，不可省略；

## 任务要求

功能需求：

1. 程序整体能够实现目录复制功能，即将源目录下的所有文件复制到目的目录；

2. 进程A功能：解析命令行参数，获取源目录地址和目的目录地址，源目录地址传输给子进程B，目的目录地址传输给子进程C，同时收取子进程传输回的文件大小和数量信息；

3. 进程B功能：遍历源目录获取所有文件的文件名和文件内容传输给兄弟进程C，同时记录文件数量和大小，传输给父进程A，同时等待收取兄弟进程C的复制结果信息；

4. 进程C功能：收取兄弟进程B传输的文件名和文件内容，打开目的目录，创建文件并写入文件内容完成复制，完成结果传输回兄弟进程B，同时记录文件数量和大小，传输给父进程A；

技术需求：

1. 在Linux系统下使用C语言编写，且只使用一个C语言源程序；

2. 程序能够通过命令行执行，支持两个参数，源目录和目的目录，其中源目录参数可以缺省；

3. 程序使用多进程协作实现，需要包含三个进程，一个父进程A和它创建的两个子进程B、C；

4. 进程间的数据传输需要选取适当的进程通信方式进行，需要每两个进程间都能够进行通信；

5. 文件目录操作需要使用Linux系统调用方式实现。

# 设计

## 进程结构设计

根据需求分析，三个进程之间需要双向通信，且通信要具有一定的数据存储量用于文件内容的传输，综合考虑选择消息队列进行进程通信，消息队列天然支持双向通信，可以进行消息随机查询，接收消息则可以根据需求进行选择读取，使用消息队列可以很好的满足实现程序需求的三个进程的互相通信和不同数据的选择性传输和接收。

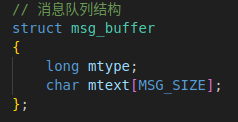
同时为了减少重复和不必要操作，由于B、C进程由A创建，它们共用A的局部变量，因此，目的目录地址和源目录地址不通过消息队列进行传输，子进程直接使用局部变量即可；同时让B、C对文件大小和文件数量的记录进行分工，子进程B负责记录文件数量，子进程C负责文件大小，再分别传回父进程A，以此减少不必要的资源消耗。

图示

描述已自动生成使用消息队列的进程协作设计图：

## 数据结构设计

消息结构体：用于消息队列结构化数据，包含两个字段，消息类型和消息内容，消息类型为长整型，消息内容为char类型数组。



## 算法流程设计

1. 声明变量：声明代码所需的变量，如进程ID、源目录和目的目录默认值、消息结构体等；

2. 声明定义消息队列：为进程B和C分别创建两个消息队列B和C，消息队列B用于进程B发送消息和其他进程向B发送消息，消息队列C用于进程C发送消息和其他进程向C发送消息；

3. 命令行参数解析：获取命令行参数并根据参数标识符解析参数，获取源目录地址和目的目录地址；

4. 检查目录正确性：判断获取到的两个目录地址是否存在，源目录不存在时提示不存在并提示重新输入，目的目录不存在时则创建目的目录；

5. 创建子进程B：创建子进程B，并编写进程B的执行逻辑；遍历源目录依次读取每个常规文件，每获取到一个文件就获取文件名和文件内容并通过消息队列C传输给进程C，遍历完成后向C发送结束信息，然后进入阻塞循环，等待进程C的响应信息，获取到成功的响应信息后结束阻塞，将记录的文件数量通过消息队列B发送给父进程A，最后正常结束进程B；

6. 创建子进程C：创建子进程C，并编写进程C的执行逻辑，设置阻塞的死循环，不断接收来自进程B的文件名和文件内容数据，将获取到的文件名和目的目录进行组合，获取绝对路径用于创建文件，然后将内容写入目标文件；当接收到B进程发送的结束消息后证明所有文件复制完成，则退出循环，通过消息队列B发送成功消息到进程B，再将记录的文件大小通过消息队列C发送到父进程A，最后正常结束进程C；

7. 主进程A：等待子进程B、C结束，并记录结束状态码，都正常退出则读取消息队列B、C中的文件总大小和总数量，并进行输出，若有子进程异常退出则提示子进程异常退出，主进程退出前需要手动删除消息队列。

图示

描述已自动生成算法流程图：

# 实现

## 实现环境

实现系统：Ubuntu 22.04.2（Windows11系统下VMware虚拟机安装版）

软件环境：VScode 1.82.2（插件：C/C++）

## 关键代码

1. 消息结构体：用于结构化消息数据；

// 消息队列结构

struct msg\_buffer

{

    long mtype;

    char mtext[MSG\_SIZE];

};

2. 消息队列创建：创建消息队列键和消息队列，并设置消息队列的访问权限：

    int msgid\_B, msgid\_C;         // 分别为进程B和进程C创建的消息队列

    key\_t key\_B = ftok(".", 'B'); // 创建进程B的消息队列键

    key\_t key\_C = ftok(".", 'C'); // 创建进程C的消息队列键

    if (key\_B == -1 || key\_C == -1)

    {

        perror("ftok");

        exit(1);

    }

    msgid\_B = msgget(key\_B, 0666 | IPC\_CREAT); // 创建进程B的消息队列

    msgid\_C = msgget(key\_C, 0666 | IPC\_CREAT); // 创建进程C的消息队列

3. 目的目录检查创建函数：对于目的目录，当目的目录不存在时需要创建，同时目的目录可能是多级目录，需要逐级检查各级目录，不存在时依次创建，并返回创建结果：

int mkdirs(char \*path)

{

    char str[512];

    strncpy(str, path, 512);

    int len = strlen(str);

    for (int i = 0; i < len; i++)

    {

        if (str[i] == '/') // 逐级检查各级目录

        {

            str[i] = '\0';

            if (access(str, 0) != 0) // 如果访问该目录返回不成功

            {

                mkdir(str, 0777); // 则创建该目录, 这里是逐级创建的.

            }

            str[i] = '/';

        }

    }

    if (len > 0 && access(str, 0) != 0) // 检查最后一级目录

    {

        mkdir(str, 0777); // 若不可访问,则创建该目录.

    }

    struct stat s;

    stat(path, &s);

    if (S\_ISDIR(s.st\_mode))

        return 0;

    return 1;

}

4. 进程B中的文件数据获取发送，如果文件大小超过消息队列消息的最大限制8192，则对其进行分片传输，即分成两个部分进行传输，基于该方法最大可传输的单个文件大小为16384字节，其中在末尾添加终止符防止文件写入不该写入的占位符：

 // 将文件名和文件内容发送到进程C

                sprintf(msg\_B.mtext, "%s", entry->d\_name);

                msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                char buffer\_part[8192];

                if (fileSize + 1 >= 8192) //单个文件大小大于8192，即大于单次能发送的最大消息，则进行分片

                {

                    sprintf(msg\_B.mtext, "%s", buffer);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                    for (int i = 8192; i < fileSize + 1; i++)

                    {

                        buffer\_part[i - 8192] = buffer[i];

                    }

                    buffer\_part[fileSize - 8192] = '\0';

                    filecount++;

                    sprintf(msg\_B.mtext, "%s", buffer\_part);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                }

                else

                {

                    // 添加空终止符

                    buffer[fileSize] = '\0';

                    filecount++;

                    sprintf(msg\_B.mtext, "%s", buffer);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                }

                free(buffer);

5. 接收文件数据并实现复制，当收取的数据大小等于8192时，则判定当前文件已被分片，则再接收一次队列消息获取文件后半部分的内容，然后再将接收的文件名和目的目录地址进行拼接得到复制文件的绝对地址，然后进行写入即可：

                struct msg\_buffer msg\_C;

                msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                if (strncmp(msg\_C.mtext, "end", 3) == 0)

                {

                    break;

                }

                char filename[1024];

                char filecontent[16384];

                // 接收文件名

                strcpy(filename, msg\_C.mtext);

                // 接收文件内容

                msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                strcpy(filecontent, msg\_C.mtext);

                while (strlen(filecontent) == 8192)

                {

                    msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                    strcat(filecontent, msg\_C.mtext);

                }

                // 将文件内容写入目标文件

                totalFileSize += strlen(filecontent);

                char dstfilename[MAX\_PATH\_LENGTH];

                snprintf(dstfilename, sizeof(dstfilename), "%s/%s", dstpath, filename);

                int fd = open(dstfilename, O\_WRONLY | O\_CREAT, 0666);

                if (fd == -1)

                {

                    perror("打开目标文件失败");

                    exit(1);

                }

                write(fd, filecontent, strlen(filecontent));

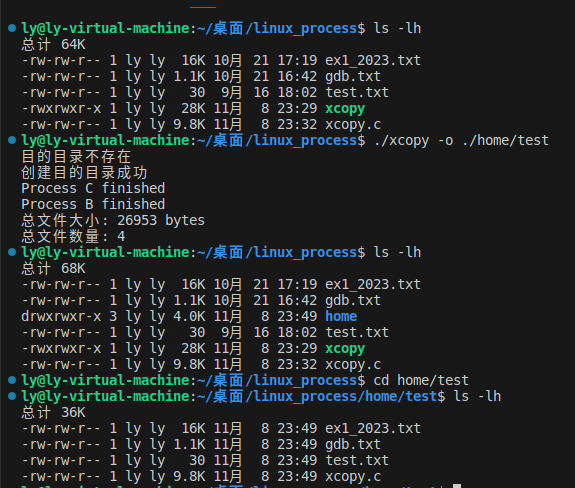
                close(fd);

## 运行效果

1. 命令： **./xcopy -o ./home/test**

运行效果：

由图可知，目的目录创建成功，除了可执行文件外，当前目录下的所有文件均已成功复制到指定位置，且主进程成功获取输出文件总大小和总数量。



图形用户界面, 应用程序

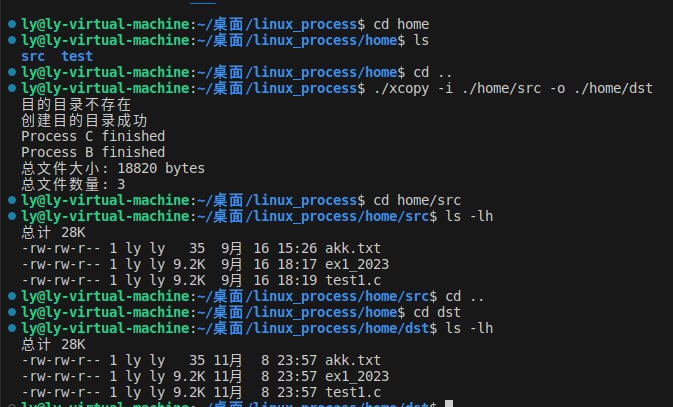
描述已自动生成

图形用户界面, 应用程序, Word

描述已自动生成

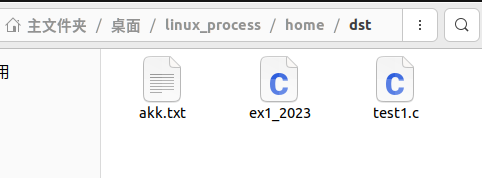
2. 命令：./**xcopy -i ./home/src -o ./home/dst**

运行效果：



图形用户界面, 应用程序

描述已自动生成



3. 命令：**./xcopy -i src -o ./home/dst2**

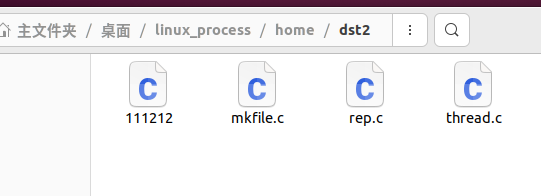
运行效果：

文本

中度可信度描述已自动生成

图形用户界面, 应用程序

描述已自动生成

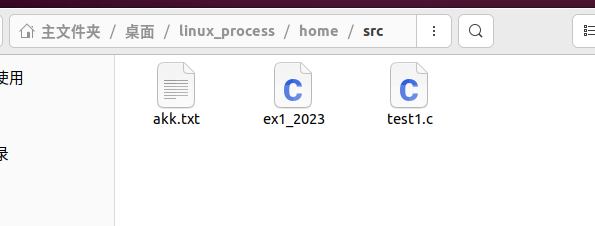


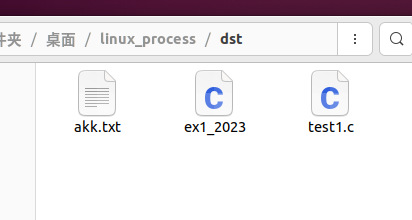
4. 命令：**./xcopy -i ./home/src -o dst**

运行效果：

图片包含 文本

描述已自动生成





4. 其他错误命令：

文本

描述已自动生成

# 总结

## 问题与解决方法

问题1：如果只使用单个消息队列，由于进程并行，会导致消息读取顺序混乱；

解决方案：由于三个进程都需要数据传输，B、C进程还需要相互的读写收发，消息队列中的消息是顺序放入顺序读取的，消息的顺序难以控制，单个消息队列会导致无法正确读取本该当前进程消费的消息；解决方案是发别为B、C进程创建两个消息队列，分别用于它们各自的消息发送和接收，能有效避免消息的错误读取。

问题3：当前目录复制时，由于当前目录下的可执行文件正在运行，不应该复制，且复制后也会产生无法使用的二进制文件；

解决方案：为了避免读取和运行产生冲突，在复制时进行文件过滤，即只复制常规文件，过滤可执行文件和子目录；具体代码：if (S\_ISREG(fileStat.st\_mode) && (fileStat.st\_mode & S\_IXUSR) == 0)，即在B进程中读取文件之前，获取文件状态信息，然后通过该判断语句，检查文件为常规文件并且没有执行权限，只有符合条件的文件才会被打开读取再进行复制，从而过滤可执行文件和子目录等不符合复制要求的文件。

问题3：当目的目录是多级目录，且该多级目录不存在时的处理；

解决方案：当目的目录不存在时，正常的处理逻辑是创建目录再进行复制，但是多级目录不存在时，只使用单个的mkdir函数无法满足要求，需要对多级目录进行逐级创建，因此单独编写int mkdirs(char \*path)函数用于处理多级目录；其逻辑是通过“/”符分隔每级目录，依次检查每级目录是否可访问，不可访问的目录则需要进行创建，同时最后一级目录进行单独判断，无法访问也进行创建，创建完成后再次尝试访问，访问成功证明创建成功，返回状态码1，否则创建失败，返回状态码0。

问题4：Linux系统原生的消息队列具有消息大小限制，单次传输消息不得超过8192 bytes，当单次需要发送的文件内容大小超过8192 bytes时的处理；

解决方案：对超过最大限制的文件内容，实行分片传输，即在进程B中读取文件时，如果判断到文件大小超过了8192 bytes，则发送时将文件分片，先发送前8192 bytes部分，截取剩下部分再进行后半段发送，同时在进程C中，如果接收到大小为8192 bytes的文件内容，则认为当前文件已被分片传输，则进入循环，等待后半部分的发送，当收取到的消息大小小于8192 bytes时退出循环；这样就能正常传输大小超过消息队列最大限制的文件，当然单个文件仍然设置了最大限制，防止过长响应。

问题5：复制后的文件末尾出现一些不应存在的二进制占位符或乱码；

解决方案：考虑到是文件内容获取到char类型数组中时数组大小大于文件大小，导致自动填入了占位符，后续复制没有进行处理，解决方案是在申请数组大小时使用动态申请，使用char \*buffer = (char \*)malloc(fileSize + 1)申请比文件大小大1的数组存储文件内容，读取完成之后在多申请的一位中添加空结束符buffer[fileSize] = '\0'，这样传输的文件读取能够完整读取，并且在复制写入时遇到空结束符时会自动停止，防止占位符写入。

## 结果分析

编写的程序整体上没有很大的缺陷和问题，能够正常解析命令行参数，对源目录和目的目录进行检查，能够成功创建两个子进程协作实现文件复制，且进程之间能够通过消息队列进行数据传输和通信，实现了设计的逻辑，满足了基本需求；

当然代码也存在一定的局限性和改进点，由于消息队列最大数据量的限制，单次传输的最大文件大小进行了限制，同时程序整体基本都直接在主函数中创建进程实现功能，没有过多的封装函数，但是为了增强代码可读性和可维护性，应该对部分可单独封装的功能进行函数封装，同时代码的测试量还不足，没有进一步测试子进程中间错误的情况，以及其他文件无法打开无法写入等情况的测试，在这些情况下代码应该也需要能够正常退出或给出相应的错误提示，仍具有很大的改进空间。

总体不存在较大的功能缺陷和巨大错误，但需要进一步优化结构和实现方法，总体基本满足题目要求。

# 附录

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <dirent.h>

#include <fcntl.h>

#define MAX\_PATH\_LENGTH 1024

#define MSG\_SIZE 8192

// 消息队列结构

struct msg\_buffer

{

    long mtype;

    char mtext[MSG\_SIZE];

};

int mkdirs(char \*path)

{

    char str[512];

    strncpy(str, path, 512);

    int len = strlen(str);

    for (int i = 0; i < len; i++)

    {

        if (str[i] == '/') // 逐级检查各级目录

        {

            str[i] = '\0';

            if (access(str, 0) != 0) // 如果访问该目录返回不成功

            {

                mkdir(str, 0777); // 则创建该目录, 这里是逐级创建的.

            }

            str[i] = '/';

        }

    }

    if (len > 0 && access(str, 0) != 0) // 检查最后一级目录

    {

        mkdir(str, 0777); // 若不可访问,则创建该目录.

    }

    struct stat s;

    stat(path, &s);

    if (S\_ISDIR(s.st\_mode))

        return 0;

    return 1;

}

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

{

    int msgid\_B, msgid\_C;         // 分别为进程B和进程C创建的消息队列

    key\_t key\_B = ftok(".", 'B'); // 创建进程B的消息队列键

    key\_t key\_C = ftok(".", 'C'); // 创建进程C的消息队列键

    if (key\_B == -1 || key\_C == -1)

    {

        perror("ftok");

        exit(1);

    }

    msgid\_B = msgget(key\_B, 0666 | IPC\_CREAT); // 创建进程B的消息队列

    msgid\_C = msgget(key\_C, 0666 | IPC\_CREAT); // 创建进程C的消息队列

    if (msgid\_B == -1 || msgid\_C == -1)

    {

        perror("msgget");

        exit(1);

    }

    pid\_t pidB, pidC;

    char \*srcpath = ".";

    char \*dstpath = NULL;

    struct msg\_buffer msg\_B, msg\_C;

    msg\_B.mtype = 1;

    msg\_C.mtype = 1;

    // 解析命令行参数

    if (argc < 3)

    {

        printf("Usage: xcopy [-i <srcpath>] -o <dstpath>\n");

        exit(1);

    }

    for (int i = 1; i < argc; i++)

    {

        if (strcmp(argv[i], "-i") == 0)

        {

            if (i + 1 < argc)

            {

                srcpath = argv[i + 1];

                i++;

            }

        }

        else if (strcmp(argv[i], "-o") == 0)

        {

            if (i + 1 < argc)

            {

                dstpath = argv[i + 1];

                i++;

            }

        }

        else

        {

            printf("Usage: xcopy [-i <srcpath>] -o <dstpath>\n");

            exit(1);

        }

    }

    // 检查源目录是否存在

    struct stat st;

    if (stat(srcpath, &st) != 0)

    {

        printf("源目录不存在，请重新输入\n");

        // 删除消息队列

        msgctl(msgid\_B, IPC\_RMID, NULL);

        msgctl(msgid\_C, IPC\_RMID, NULL);

        return 0;

    }

    //目的目录检查

    if (dstpath == NULL)

    {

        printf("目的目录不能为空，请重新输入\n");

        // 删除消息队列

        msgctl(msgid\_B, IPC\_RMID, NULL);

        msgctl(msgid\_C, IPC\_RMID, NULL);

        return 0;

    }

    if (stat(dstpath, &st) == 0)

    {

        if (S\_ISDIR(st.st\_mode))

        {

            printf("目的目录存在\n");

        }

        else

        {

            printf("指定路径存在，但不是一个目录\n");

        }

    }

    else

    {

        printf("目的目录不存在\n");

        // 目录不存在，逐级创建

        int ret = mkdirs(dstpath);

        if (ret == 0)

        {

            printf("创建目的目录成功\n");

        }

        else

        {

            printf("创建目的目录失败\n");

        }

    }

    // 创建进程B

    pidB = fork();

    if (pidB == 0)

    {

        // 子进程B

        DIR \*dir;

        struct dirent \*entry;

        dir = opendir(srcpath);

        if (dir == NULL)

        {

            perror("打开源目录失败");

            exit(1);

        }

        char filename[1024];

        int filecount = 0; // 初始化文件数量为0

        while ((entry = readdir(dir)) != NULL)

        {

            snprintf(filename, sizeof(filename), "%s/%s", srcpath, entry->d\_name);

            int fd = open(filename, O\_RDONLY);

            if (fd == -1)

            {

                perror("打开文件失败");

                exit(1);

            }

            struct stat fileStat;

            if (fstat(fd, &fileStat) == -1)

            {

                perror("获取文件大小失败");

                close(fd);

                exit(1);

            }

            if (S\_ISREG(fileStat.st\_mode) && (fileStat.st\_mode & S\_IXUSR) == 0)

            {

                off\_t fileSize = fileStat.st\_size;

                if (fileSize + 1 >= 16384)

                {

                    perror("单个文件过大，无法复制");

                    close(fd);

                    exit(1);

                }

                char \*buffer = (char \*)malloc(fileSize + 1);

                // char buffer[MSG\_SIZE];

                if (!buffer)

                {

                    perror("分配内存失败");

                    close(fd);

                    exit(1);

                }

                ssize\_t bytesRead = read(fd, buffer, fileSize);

                if (bytesRead == -1)

                {

                    perror("获取文件内容失败");

                    free(buffer);

                    close(fd);

                    exit(1);

                }

                // 将文件名和文件内容发送到进程C

                sprintf(msg\_B.mtext, "%s", entry->d\_name);

                msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                char buffer\_part[8192];

                if (fileSize + 1 >= 8192)

                {

                    sprintf(msg\_B.mtext, "%s", buffer);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                    for (int i = 8192; i < fileSize + 1; i++)

                    {

                        buffer\_part[i - 8192] = buffer[i];

                    }

                    buffer\_part[fileSize - 8192] = '\0';

                    filecount++;

                    sprintf(msg\_B.mtext, "%s", buffer\_part);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                }

                else

                {

                    // 添加空终止符

                    buffer[fileSize] = '\0';

                    filecount++;

                    sprintf(msg\_B.mtext, "%s", buffer);

                    msgsnd(msgid\_C, &msg\_B, sizeof(msg\_B.mtext), 0);

                }

                free(buffer);

            }

            close(fd);

        }

        strcpy(msg\_B.mtext, "end");

        msgsnd(msgid\_C, &msg\_B, strlen(msg\_B.mtext) + 1, 0);

        while (1)

        {

            struct msg\_buffer msg\_B;

            msgrcv(msgid\_B, &msg\_B, sizeof(msg\_B.mtext), 1, 0);

            if (strncmp(msg\_B.mtext, "Success", 7) == 0)

            {

                printf("Process C finished\n");

                break;

            }

        }

        // 将文件数量发送到进程A

        sprintf(msg\_B.mtext, "%d", filecount);

        msgsnd(msgid\_B, &msg\_B, sizeof(msg\_B.mtext), 0);

        printf("Process B finished\n");

        exit(0);

    }

    else if (pidB > 0)

    {

        // 父进程A

        pidC = fork();

        if (pidC == 0)

        {

            // 子进程C

            int totalFileSize = 0;

            while (1)

            {

                struct msg\_buffer msg\_C;

                msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                if (strncmp(msg\_C.mtext, "end", 3) == 0)

                {

                    break;

                }

                char filename[1024];

                char filecontent[16384];

                // 接收文件名

                strcpy(filename, msg\_C.mtext);

                // 接收文件内容

                msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                strcpy(filecontent, msg\_C.mtext);

                while (strlen(filecontent) == 8192)

                {

                    msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                    strcat(filecontent, msg\_C.mtext);

                }

                // 将文件内容写入目标文件

                totalFileSize += strlen(filecontent);

                char dstfilename[MAX\_PATH\_LENGTH];

                snprintf(dstfilename, sizeof(dstfilename), "%s/%s", dstpath, filename);

                int fd = open(dstfilename, O\_WRONLY | O\_CREAT, 0666);

                if (fd == -1)

                {

                    perror("打开目标文件失败");

                    exit(1);

                }

                write(fd, filecontent, strlen(filecontent));

                close(fd);

            }

            // 成功时发送成功消息给进程B

            strcpy(msg\_C.mtext, "Success");

            msgsnd(msgid\_B, &msg\_C, strlen(msg\_C.mtext) + 1, 0);

            // 发送文件总大小到进程A

            sprintf(msg\_C.mtext, "%d", totalFileSize);

            msgsnd(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 0);

            exit(0);

        }

        else if (pidC > 0)

        {

            // 父进程A

            int statusB, statusC;

            waitpid(pidB, &statusB, 0);

            waitpid(pidC, &statusC, 0);

            if (statusB == 0 && statusC == 0)

            {

                // 接收文件总大小

                msgrcv(msgid\_C, &msg\_C, sizeof(msg\_C.mtext), 1, 0);

                int totalFileSize = atoi(msg\_C.mtext);

                printf("总文件大小: %d bytes\n", totalFileSize);

                // 接收文件数量

                msgrcv(msgid\_B, &msg\_B, sizeof(msg\_B.mtext), 1, 0);

                int totalFileCount = atoi(msg\_B.mtext);

                printf("总文件数量: %d\n", totalFileCount);

            }

            else

            {

                printf("子进程异常退出\n");

                // 删除消息队列

                msgctl(msgid\_B, IPC\_RMID, NULL);

                msgctl(msgid\_C, IPC\_RMID, NULL);

                return 0;

            }

        }

    }

    else

    {

        perror("进程创建出错");

        exit(1);

    }

    // 删除消息队列

    msgctl(msgid\_B, IPC\_RMID, NULL);

    msgctl(msgid\_C, IPC\_RMID, NULL);

    return 0;

}