山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200130119 | 姓名：于斐 | | 班级：学堂计机22 |
| 实验题目： 线程和管道通信 | | | |
| 实验学时： 4 | | 实验日期： 2024年3月25日 | |
| 实验目的：  通过在Linux系统中进行线程和管道通信机制的实验，加深对pthread线程库的应用的熟悉程度。加深对线程控制和管道通信概念的理解，观察和体验多个并发线程之间通信和协作的效果。练习使用pthread线程库和无名管道进行线程通信的编程和调试技术。 | | | |
| 实验过程中遇到和解决的问题：  实验过程中在管道通信方面遇到两个问题：管道被重定向到 /dev/stdin、管道内传输数据错误。  经检查，前者是由于管道没有使用 pipe() 函数优化，从而导致管道变量内值为 0，从而定向到 /dev/stdin；后者是由于管道通信方向错误，一般情况下，管道顺序为 R/W，即，需要从 [1] 写入，从 [0] 读出。 | | | |
| 实验步骤与内容：  实验环境：Arch Linux x86\_64 with GCC 13.2.1-5  实验步骤：   1. 新建 main.c 文件，写入附录中的代码内容。   1. main函数首先通过命令行参数获取两个整数x和y，然后创建了6个管道用于线程间通信，并创建了三个线程。  2. 第一个线程fxy负责计算x和y的乘积，并将结果发送给主线程。在此过程中，它将x和y的值从管道中读取，然后创建两个子线程（fx和fy）分别计算x和y的部分乘积。  3. fx线程用递归的方式计算x的阶乘。它从管道中读取x和当前的部分乘积，如果x小于等于1，则将部分乘积发送给fxy线程，否则继续递归计算。递归结束后，将计算结果发送给fxy线程。  4. fy线程类似于fx线程，用递归的方式计算y的部分乘积。不同之处在于，fy线程会将计算的结果发送给主线程或者fxy线程，取决于参数should\_send\_to\_main的值。  5. 所有线程在计算完成后都会向管道写入结果，主线程则从相应的管道读取最终的计算结果并输出。   1. 新建 Makefile 文件，在执行 make 时做编译工作，make clean 时清除所有 .o 及无扩展名程序文件。 2. 在程序目录下执行 make clean; make && ./main，观察实验结果。   实验结果如图，执行 ./main 3 5，程序输出 14（）。 | | | |
| 实验总结：  通过本次实验，使用多线程和管道通信的机制，实现了一个计算程序来加深对这些概念的理解，练习了相关的编程和调试技术，学习了管道、线程等相关概念。  在协作方面，将任务分解为小任务并分配给多个进程/线程执行，然后将它们的结果汇总以获得最终结果，同时确保进程/线程之间的执行顺序和数据一致性。在通信方面，进程/线程之间通过消息传递或共享内存来交换数据，可以是同步等待响应的通信，也可以是无需等待的异步通信。这些特征和功能使得进程/线程能够并发执行任务、共享资源、实现模块化设计，并支持分布式计算和处理，为构建高效、可扩展的系统提供了重要基础。  多线程协作： 代码中创建了多个线程，每个线程负责执行不同的计算任务。这些线程之间相互协作，通过线程的创建、等待和汇总计算结果来实现整个任务的完成。  管道通信： 通过创建了多个管道来实现线程间的通信。管道在主线程、子线程之间传递数据，实现了数据的共享和传输。这种通信方式确保了线程之间的协作和数据同步。  通过线程的创建和等待（pthread\_create和pthread\_join函数），实现了线程之间的同步。主线程等待子线程完成计算任务后再继续执行，确保了计算结果的准确性。  管道机制是一种在UNIX和类UNIX系统中用于进程间通信的机制。它是一种单向通信管道，允许一个进程的输出连接到另一个进程的输入，从而实现进程间的数据传输。管道通常用于父子进程之间或者兄弟进程之间的通信。管道的创建包括调用pipe()系统调用，它创建了一个管道，并返回两个文件描述符，一个用于管道的读取端，另一个用于写入端。通过这两个文件描述符，可以实现对管道的读取和写入操作。在进程/线程间的协作和通信中，可以利用管道来传递数据和信息。例如，一个进程可以将数据写入管道的写入端，另一个进程则可以从管道的读取端读取这些数据。通过这种方式，进程/线程之间可以进行数据交换和协作，实现任务的并发执行和结果的汇总。在具体的编程实现中，通常需要先创建管道，然后通过文件描述符来对管道进行读取和写入操作。不同的进程/线程可以通过共享管道的文件描述符来进行通信，实现数据的传输和共享。需要注意的是，管道是单向的，因此如果需要双向通信，则需要创建两个管道来实现。 | | | |

附录：程序关键源代码

#include "main.h"

int pipes[6][2];

// pipes[0]: main thread => thread 1

// pipes[1]: main thread <= thread 1

// pipes[2]: thread 1 => thread 2

// pipes[3]: thread 1 <= thread 2

// pipes[4]: thread 1 => thread 3

// pipes[5]: thread 1 <= thread 3

pthread\_t threads[3];

int fxy(); int fx(); int fy();

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

    // x, y are passed into the program through args

    if (argc != 3) {

        fprintf(stderr, "Usage: %s <integer x> <integer y>.\n", argv[0]);

        exit(EXIT\_FAILURE);

    }

    int x = atoi(argv[1]), y = atoi(argv[2]); assert(x > 0 && y > 0);

    for (int i = 0; i < 6; ++i)

        if (pipe(pipes[i]) < 0) {

            perror("Error while creating pipes"); exit(EXIT\_FAILURE);

        }

    int status[3];

    int result = pthread\_create(threads + 0, NULL, (void \*) fxy, (void \*) status + 0);

    if (result) { perror("Error while creating thread 1"); exit(EXIT\_FAILURE); }

    result = pthread\_create(threads + 1, NULL, (void \*) fx, (void \*) status + 1);

    if (result) { perror("Error while creating thread 2"); exit(EXIT\_FAILURE); }

    result = pthread\_create(threads + 2, NULL, (void \*) fy, (void \*) status + 2);

    if (result) { perror("Error while creating thread 3"); exit(EXIT\_FAILURE); }

    write(pipes[0][1], &x, sizeof(int));

    write(pipes[0][1], &y, sizeof(int));

    pthread\_join(threads[0], NULL);

    int ans = 0; read(pipes[1][0], &ans, sizeof(int));

    printf("%d\n", ans); exit(EXIT\_SUCCESS);

}

int fxy() {

    int x = 0, y = 0, ans = 0;

    read(pipes[0][0], &x, sizeof(int));

    read(pipes[0][0], &y, sizeof(int));

    int init\_ans = 1;

    write(pipes[2][1], &x, sizeof(int));

    write(pipes[2][1], &init\_ans, sizeof(int));

    int should\_send\_to\_main = 1;

    write(pipes[4][1], &y, sizeof(int));

    write(pipes[4][1], &should\_send\_to\_main, sizeof(int));

    int result = 0; pthread\_join(threads[1], NULL);

    read(pipes[3][0], &result, sizeof(int)); ans += result;

    result = 0; pthread\_join(threads[2], NULL);

    read(pipes[5][0], &result, sizeof(int)); ans += result;

    write(pipes[1][1], &ans, sizeof(int));

    return EXIT\_SUCCESS;

}

int fx() {

    int x, ans;

    read(pipes[2][0], &x, sizeof(int));

    read(pipes[2][0], &ans, sizeof(int));

    if (x <= 1) {

        write(pipes[3][1], &ans, sizeof(int)); return EXIT\_SUCCESS;

    }

    ans \*= x; --x; write(pipes[2][1], &x, sizeof(int));

    write(pipes[2][1], &ans, sizeof(int)); fx();

    return EXIT\_SUCCESS;

}

int fy() {

    int y, ans = 0, should\_send\_to\_main; int offset = 0;

    read(pipes[4][0], &y, sizeof(int));

    read(pipes[4][0], &should\_send\_to\_main, sizeof(int));

    if (y <= 2) {

        if (should\_send\_to\_main) write(pipes[5][1], &y, sizeof(int));

        else write(pipes[4][1], &y, sizeof(int));

        return EXIT\_SUCCESS;

    }

    y--; write(pipes[4][1], &y, sizeof(int));

    write(pipes[4][1], &offset, sizeof(int));

    fy(); read(pipes[4][0], &offset, sizeof(int));

    ans += offset; offset = 0;

    y--; write(pipes[4][1], &y, sizeof(int));

    write(pipes[4][1], &offset, sizeof(int));

    fy(); read(pipes[4][0], &offset, sizeof(int));

    ans += offset; offset = 0;

    if (should\_send\_to\_main) write(pipes[5][1], &ans, sizeof(int));

    else write(pipes[4][1], &ans, sizeof(int));

    return EXIT\_SUCCESS;

}