

计算机学院 操作系统 课程实验报告

实验题目： 实验 2 线程和管道通信		学号： 202300130183
日期： 2025/3/10	班级： 23 级智能班	姓名： 宋浩宇
Email： 202300130183@mail.sdu.edu.cn		
<p>实验方法介绍：</p> <p>使用 Oracle Virtual Box 运行 Ubuntu24.04 虚拟环境来编写编译相应的代码。</p>		
<p>实验过程描述：</p> <p>关于示例实验，我们先将实验指导书提供的代码复刻一遍。</p> <p>以下为复刻代码（使用 VSCode 渲染）：</p> <p>tpipe.c 文件：</p> <pre>/* * main.c : description * * function : 利用管道实现在线程间传递整数 */ #include<stdio.h> #include<unistd.h> #include<stdlib.h> #include<pthread.h> void task1(int *); void task2(int *); int pipe1[2],pipe2[2]; pthread_t thrd1, thrd2; int main(int argc, char* argv[]) { int ret; int num1, num2; if (pipe(pipe1) < 0) { perror("pipe1 not create"); exit(EXIT_FAILURE); } if (pipe(pipe2) < 0) { perror("pipe2 not create"); exit(EXIT_FAILURE); } }</pre>		

```

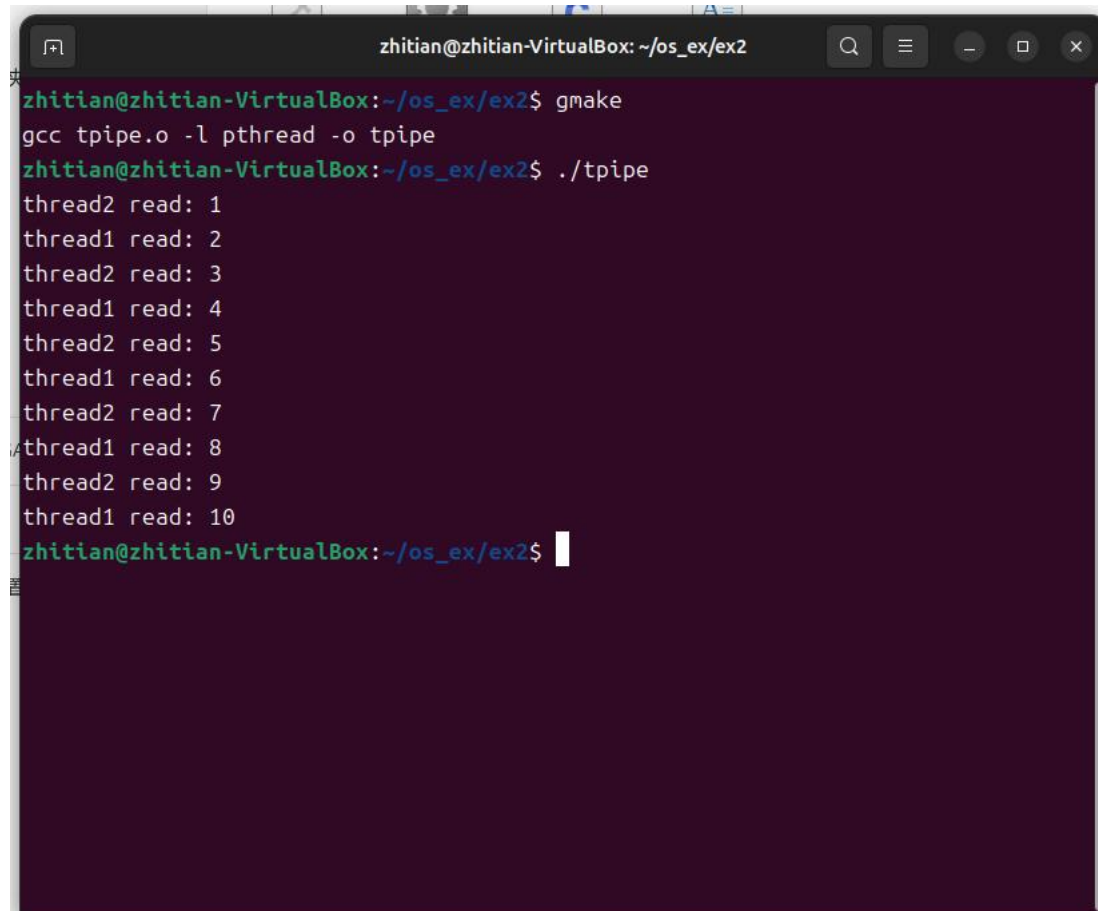
num1 = 1;
ret = pthread_create(&thrd1, NULL, (void*)task1, (void*)&num1);
if (ret)
{
    perror("pthread_create: task1");
    exit(EXIT_FAILURE);
}
num2 = 2;
ret = pthread_create(&thrd2, NULL, (void*)task2, (void*)&num2);
if (ret)
{
    perror("pthread_create: task2");
    exit(EXIT_FAILURE);
}
pthread_join(thrd2, NULL);
pthread_join(thrd1, NULL);
exit(EXIT_SUCCESS);
}
void task1(int* num)
{
    int x = 1;
    do
    {
        write(pipe1[1], &x, sizeof(int));
        read(pipe2[0], &x, sizeof(int));
        printf("thread%d read: %d\n", *num, x++);
    } while (x <= 9);
    close(pipe1[1]);
    close(pipe2[0]);
}
void task2(int* num)
{
    int x;
    do
    {
        read(pipe1[0], &x, sizeof(int));
        printf("thread2 read: %d\n", x++);
        write(pipe2[1], &x, sizeof(int));
    } while (x <= 9);
    close(pipe1[0]);
    close(pipe2[1]);
}

```

创建 Makefile, 内容为:

```
scr = tpipe.c
obj = tpipe.o
opt = -g -c
all: tpipe
tpipe: $(obj)
    gcc $(obj) -l pthread -o tpipe
tpipe.o: $(scr)
    gcc $(opt) $(scr)
clean:
    rm tpipe *.o
```

编译并运行，结果如下：



The screenshot shows a terminal window titled 'zhitian@zhitian-VirtualBox: ~/os_ex/ex2'. The user enters 'gmake' to compile the program, which outputs 'gcc tpipe.o -l pthread -o tpipe'. Then, the user enters './tpipe' to run the program. The output shows two threads, thread1 and thread2, each reading 10 numbers from a pipe. The numbers are interleaved: thread2 reads 1, thread1 reads 2, thread2 reads 3, thread1 reads 4, thread2 reads 5, thread1 reads 6, thread2 reads 7, thread1 reads 8, thread2 reads 9, and thread1 reads 10. The terminal window has a dark purple background and a white cursor.

```
zhitian@zhitian-VirtualBox:~/os_ex/ex2$ gmake
gcc tpipe.o -l pthread -o tpipe
zhitian@zhitian-VirtualBox:~/os_ex/ex2$ ./tpipe
thread2 read: 1
thread1 read: 2
thread2 read: 3
thread1 read: 4
thread2 read: 5
thread1 read: 6
thread2 read: 7
thread1 read: 8
thread2 read: 9
thread1 read: 10
zhitian@zhitian-VirtualBox:~/os_ex/ex2$
```

ppipe.c 文件：

```
/*
 *   Filename : ppipe.c
 *
 *   Function  : 利用管道在父子进程间传递整数
 */

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

```
int main(int argc, char *argv[])
{
    int pid;
    int pipe1[2];
    int pipe2[2];
    int x;
    if (pipe(pipe1) < 0)
    {
        perror("pipe not create");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipe2) < 0)
    {
        perror("pipe not create");
        exit(EXIT_FAILURE);
    }
    if ((pid = fork())<0)
    {
        perror("process not create error");
        exit(EXIT_FAILURE);
    }
    else
    {
        if (pid == 0)
        {
            close(pipe1[1]);
            close(pipe2[0]);
            do
            {
                read(pipe1[0], &x, sizeof(int));
                printf("child %d read %d\n", getpid(), x);
                write(pipe2[1], &x, sizeof(int));
            } while (x <= 9);
            close(pipe1[0]);
            close(pipe2[1]);
            exit(EXIT_FAILURE);
        }
        else
        {
            close(pipe1[0]);
            close(pipe2[1]);
            do
            {

```

```

        write(pipe1[1], &x, sizeof(int));
        read(pipe2[0], &x, sizeof(int));
        printf("parent %d read %d\n", getpid(), x++);
    } while (x <= 9);
    close(pipe1[1]);
    close(pipe2[0]);
}
}
return EXIT_SUCCESS;
}

```

创建 Makefile, 内容如下:

```

srcs = ppipe.c
objs = ppipe.o
opts = -g -c
all: ppipe
ppipe: $(objs)
    gcc $(objs) -o ppipe
ppipe.o:$(srcs)
    gcc $(opts) $(srcs)
clean:
    rm ppipe *.o

```

编译并运行, 结果如下:

```
zhitian@zhitian-VirtualBox: ~/os_ex/ex2/ttpipe
zhitian@zhitian-VirtualBox:~/os_ex/ex2/ttpipe$ gmake
gcc -g -c ppipe.c
gcc ppipe.o -o ppipe
zhitian@zhitian-VirtualBox:~/os_ex/ex2/ttpipe$ ./ppipe
child 6675 read 0
parent 6674 read 0
child 6675 read 1
parent 6674 read 1
child 6675 read 2
parent 6674 read 2
child 6675 read 3
parent 6674 read 3
child 6675 read 4
parent 6674 read 4
child 6675 read 5
parent 6674 read 5
child 6675 read 6
parent 6674 read 6
child 6675 read 7
parent 6674 read 7
child 6675 read 8
parent 6674 read 8
child 6675 read 9
parent 6674 read 9
child 6675 read 9
zhitian@zhitian-VirtualBox:~/os_ex/ex2/ttpipe$
```

以上为示例实验的结果。

关于独立实验，我们首先需要写出实现题目要求功能的代码：

sfex.c 文件内容：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void fx(int*);
void fy(int*);
void fxy(int*);
int calculate_fx(int num);
int calculate_fy(int num);
int pipe_xy_to_x[2], pipe_x_to_xy[2], pipe_y_to_xy[2], pipe_xy_to_y[2];
pthread_t thrd_xy, thrd_x, thrd_y;
int main()
{
    int num1 = 1, num2 = 2, num3 = 3;
```

```

int x, y;
x = 1;
y = 1;
printf("x = ");
scanf("%d", &x);
printf("y = ");
scanf("%d", &y);
int for_thrd_xy[3] = {num1, x, y};
int for_thrd_x[2] = { num2, x };
int for_thrd_y[2] = { num3, y };
if (pipe(pipe_x_to_xy) < 0)
{
    perror("pipe_x_to_xy create failed");
}
if (pipe(pipe_y_to_xy) < 0)
{
    perror("pipe_y_to_xy create failed");
}
if (pipe(pipe_xy_to_x) < 0)
{
    perror("pipe_xy_to_x create failed");
}
if (pipe(pipe_xy_to_y) < 0)
{
    perror("pipe_xy_to_y create failed");
}
if (pthread_create(&thrd_xy, NULL, (void*)fxy, (void*)for_thrd_xy))
{
    perror("pthread_create thrd_xy failed");
}
if (pthread_create(&thrd_x, NULL, (void*)fx, (void*)&for_thrd_x))
{
    perror("pthread_create thrd_x failed");
}
if (pthread_create(&thrd_y, NULL, (void*)fy, (void*)&for_thrd_y))
{
    perror("pthread_create thrd_y failed");
}
pthread_join(thrd_xy, NULL);
pthread_join(thrd_x, NULL);
pthread_join(thrd_y, NULL);
return EXIT_SUCCESS;
}

void fxy(int* num)

```

```

{
    int x, y, id;
    id = num[0];
    x = num[1];
    y = num[2];
    write(pipe_xy_to_x[1], &x, sizeof(int));
    write(pipe_xy_to_y[1], &y, sizeof(int));
    int ans_x, ans_y;
    read(pipe_x_to_xy[0], &ans_x, sizeof(int));
    read(pipe_y_to_xy[0], &ans_y, sizeof(int));
    printf("thread %d calculate fxy(%d,%d) = %d\n", id, x, y, ans_x + ans_y);
    close(pipe_x_to_xy[0]);
    close(pipe_x_to_xy[1]);
    close(pipe_y_to_xy[0]);
    close(pipe_y_to_xy[1]);
    close(pipe_xy_to_x[0]);
    close(pipe_xy_to_x[1]);
    close(pipe_xy_to_y[0]);
    close(pipe_xy_to_y[1]);
}

void fx(int* num)
{
    int x, id;
    id = num[0];
    // x = num[1];
    // read(pipe_xy_to_x[0], &x, sizeof(int));
    if (read(pipe_xy_to_x[0], &x, sizeof(int)) == -1)
    {
        perror("fx read failed");
        exit(EXIT_FAILURE);
    }
    printf("x=%d\n", x);
    int ans = calculate_fx(x);
    printf("thread %d calculate fx(%d) = %d\n", id, x, ans);
    write(pipe_x_to_xy[1], &ans, sizeof(int));
}

void fy(int* num)
{
    int y, id;
    id = num[0];
    // y = num[1];
    if (read(pipe_xy_to_y[0], &y, sizeof(int)) == -1)
    {
        perror("fy read failed");
    }
}

```



```

        exit(EXIT_FAILURE);
    }
    printf("y=%d\n", y);
    int ans = calculate_fy(y);
    printf("thread %d calculate fy(%d) = %d\n",id,y,ans);
    write(pipe_y_to_xy[1], &ans, sizeof(int));
}
int calculate_fx(int num)
{
    if (num == 1)
    {
        return 1;
    }
    if (num > 1)
    {
        return calculate_fx(num - 1) * num;
    }
}
int calculate_fy(int num)
{
    if (num == 1 || num == 2)
    {
        return 1;
    }
    if (num > 2)
    {
        return calculate_fy(num - 1) + calculate_fy(num - 2);
    }
}

```

创建 Makefile, 内容如下:

```

srcs = sfex.c
objs = sfex.o
opts = -g -c
all: sfex
sfex: $(objs)
    gcc $(objs) -o sfex
sfex.o:$(srcs)
    gcc $(opts) $(srcs)
clean:
    rm sfex *.o

```

编译并运行, 结果如下:

```
zhitian@zhitian-VirtualBox: ~/os_ex/ex2/sfex
zhitian@zhitian-VirtualBox:~/os_ex/ex2/sfex$ gmake
gcc sfex.o -o sfex
zhitian@zhitian-VirtualBox:~/os_ex/ex2/sfex$ ./sfex
x = 8
y = 6
x=8
thread 2 calculate fx(8) = 40320
y=6
thread 3 calculate fy(6) = 8
thread 1 calculate fxy(8,6) = 40328
zhitian@zhitian-VirtualBox:~/os_ex/ex2/sfex$ ./sfex
x = 5
y = 1
x=5
thread 2 calculate fx(5) = 120
y=1
thread 3 calculate fy(1) = 1
thread 1 calculate fxy(5,1) = 121
zhitian@zhitian-VirtualBox:~/os_ex/ex2/sfex$
```

结论分析:

1. 根据示例实验程序和独立实验程序观察和记录的调试和运行的信息, 说明它们反映出操作系统教材中讲解的进/线程协作和进/线程通信概念的哪些特征和功能?

答: 进/线程协作有依赖性、同步性、共享资源、优先级、可中断性的特征, 有保证数据一致性、提高资源利用率、实现并发执行、模块化设计的功能。进/线程通信具有同步性、可靠性、隔离性的特征, 具有数据传输、数据共享、事件通知、协同处理、服务请求的功能。

2. 在真实的操作系统中它是怎样实现和反映出教材中进/线程通信概念的。你对于进/线程协作和进/线程通信的概念和实现有哪些新的理解和认识?

答: 在真实的操作系统中, 主要通过共享内存、消息队列、管道、信号来实现进/线程通信的。这些实现方式具有不同的性能和特性, 共享内存速度最快但缺乏安全性, 并且因为操作系统的调度顺序的问题共享内存并不同步。消息队列、管道和信号都是由操作系统来维护, 不同的操作系统的实现方式不同, 但因为是间接的通信, 中间涉及一些复制操作, 性能不如共享内存, 但是灵活性更高。套接字使用了网络协议来实现, 甚至可以做到不同机器上的进程之间的通信。

3. 管道机制的机理是什么?

答: 管道本质上是由操作系统维护的一块缓冲区, 以 linux 系统举例, 在使用 pipe() 时, 存

放在 `fd[0]` 和 `fd[1]` 中的信息是两个文件描述符，前者用于读取数据，后者用于写入数据，它们将进程和管道连接起来，在管道中，数据的传输遵循先进先出的原则，对于写操作来说，如果管道缓冲区未满，写入进程/线程就会将数据写入缓冲区，如果满了，写入进程/线程会被阻塞，直到有其他进程/线程从缓冲区中读出数据释放出足够的空间。如果文件描述符被关闭，则写入进程会收到 `SIGPIPE` 信号。读操作也是类似的，如果缓冲区中有数据，则会读入，如果缓冲区为空，则进行读操作的进程/线程也会被阻塞，直到其他进程/线程向管道写入数据。如果读入的文件描述符被关闭，则读入进程/线程会收到 `SIGPIPE` 信号，如果此时读入文件描述符没有被关闭，写入文件描述符被关闭，则会读取到 `EOF`。当管道的写入端和读出端都被关闭时，操作系统就会释放这一块缓冲区占用的资源。管道有匿名管道和命名管道，匿名管道只能用于父子进程/线程或兄弟进程/线程之间的通信，而命名管道可以用于任意线程/进程之间的通信。命名管道特殊的地方在于他在文件系统中有一个对应的文件节点，任何进程都可以通过打开这个文件节点来建立与管道之间的连接。另外，也可以通过设置 `read` 和 `write` 函数的参数来避免阻塞。

4. 怎样利用管道完成进/线程间的协作和通信？

答：使用 `int fd[2];` 来存储管道的写入端和读取端，使用 `pipe(fd);` 来建立管道。在不同的线程/进程里，通过 `read(fd[0], &x, sizeof(x));` 和 `write(fd[1], &x, sizeof(x));` 来进行数据的读取和写入。写入时，管道中会写入从地址 `&x` 开始，`sizeof(x)` 个字节大小的数据，读取时，管道中会读出 `sizeof(x)` 个字节大小的数据，写入地址 `&x` 起始的对应字节大小的空间。

结论：

线程/进程之间具有协作的功能，线程/线程给予了一个程序同步执行的能力，提高了进程执行的并行性能，并且使用这个机制可以将一个任务拆解成多个子任务，提高程序的模块化和可维护性，也可以通过这个机制充分地利用 CPU 的性能，另外也可以满足一些特殊的应用场景下的需求，比如说对于一些高性能的并行计算。