Linux作业三： Linux应用程序开发-进程管理

阅读以下程序，分析程序功能，并上机调试运行，截图运行结果。

**Part 1: 进程控制类函数：fork();wait();waitpid()**

**1、fork1.c**

#include <unistd.h>

#include <stdio.h>

int main ()

{

    pid\_t fpid; //fpid表示fork函数返回的值

    int count=0;

    fpid=fork();

    if (fpid < 0)

        printf("error in fork!");

    else if (fpid == 0) {

        printf("i am the child process, my process id is %d\n",getpid());

        printf("I am childprocess\n");

        count++;

    }

    else {

        printf("i am the parent process, my process id is %d/n",getpid());

        printf("I am parent process\n");

        count++;

    }

    printf("统计结果是: %d/n",count);

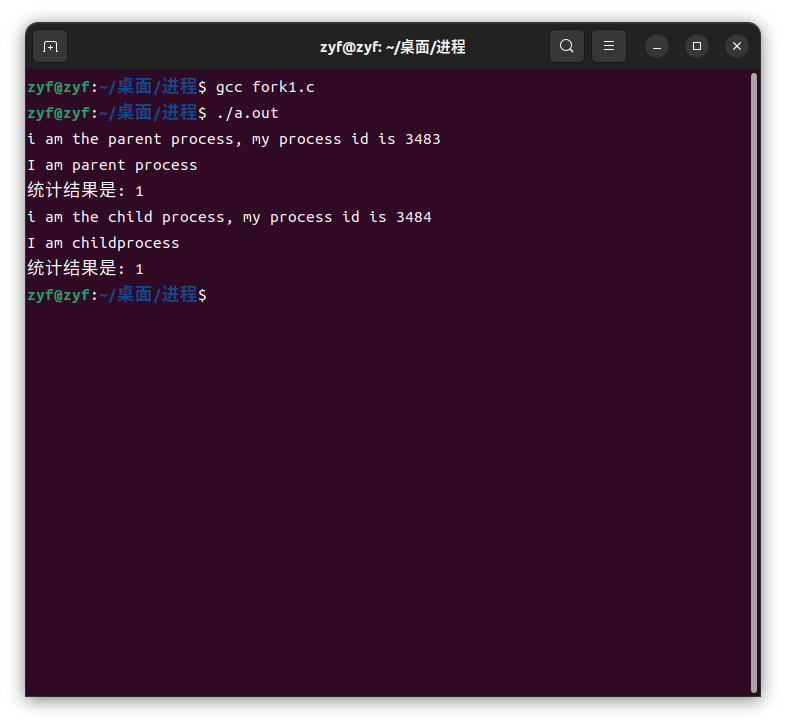
    return 0;

}

程序功能分析：

在父进程中创建子进程，创建失败则打印error in fork!，创建成功则子进程执行printf("i am the child process, my process id is %d\n",getpid());printf("I am childprocess\n"); 两句话并将子进程里面的count+1，父进程执行printf("i am the parent process, my process id is %d/n",getpid()); printf("I am parent process\n");  并将父进程里面的count+1。这段程序可以说明父子进程不共享内部变量。

运行结果图：

****

**2、fork2.c**

给出如下C程序，在linux下使用gcc编译：

#include "stdio.h"

#include "sys/types.h"

#include "unistd.h"

 int main()

{

    pid\_t pid1;

    pid\_t pid2;

    pid1 = fork();

    pid2 = fork();

    printf("pid1:%d, pid2:%d\n", pid1, pid2);

}

要求如下：

      已知从这个程序执行到这个程序的所有进程结束这个时间段内，没有其它新进程执行。

1. 请说出执行这个程序后，将一共运行几个进程。

4个

1. 如果其中一个进程的输出结果是“pid1:1001, pid2:1002”，写出其他进程的输出结果（不考虑进程执行顺序）

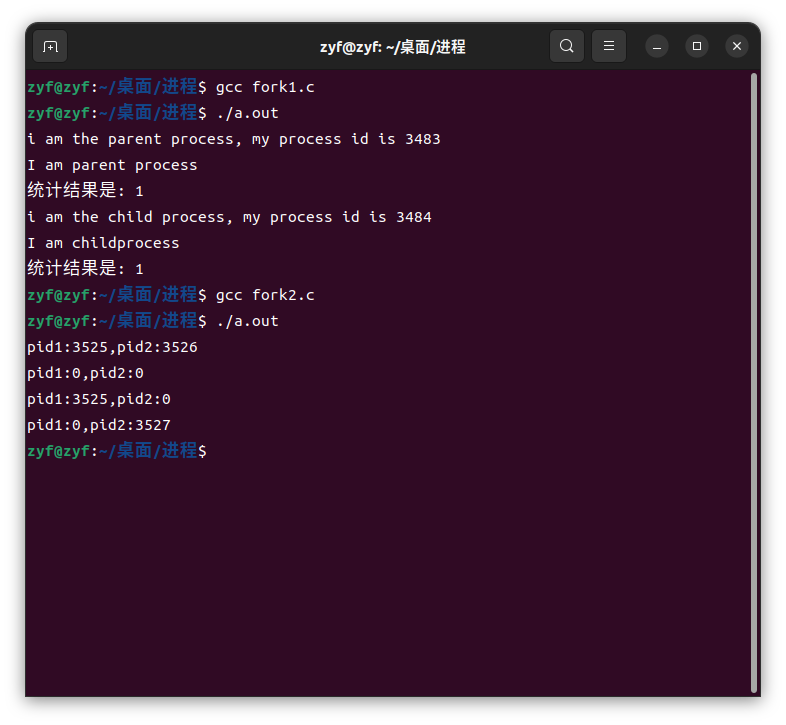
进程二“pid1:1001,pid2:0”

进程三“pid1:0,pid2:1003”

进程四“pid1:0,pid2:0”

题述进程为执行main函数的进程即两次fork均为父进程，进程二在第一个fork时为父进程在第二个fork时为子进程，进程三在第一次fork时为子进程在第二次fork时为父进程，进程四在第一次和第二次fork时均为子进程。

执行代码结果：



**3、fork.c**

请问下面的程序一共输出多少个“hello”？

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

   int i;

   for(i=0; i<3; i++){

      fork();

      printf("hello\n");

   }

    return 0;

}

程序分析：一共14个hello

父进程执行0、1、2共三次循环产生3个hello，也产生三个子进程。

子进程1执行0、1、2共三次循环产生3个hello，也产生两个孙子进程。

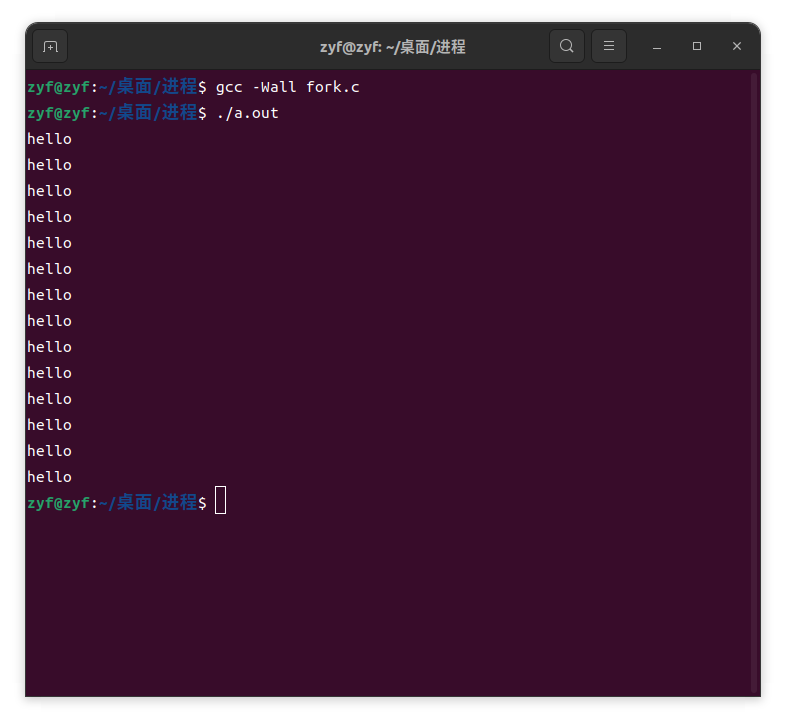
孙子进程1执行1、2两次循环产生2个hello，也产生一个重孙进程，执行一次循环产生1个hello。

孙子进程2执行2一次循环产生1个hello。

子进程2执行1、2两次循环产生2个hello，也产生一个孙子进程，执行一次循环产生1个hello。

子进程3执行一次循环产生1个hello。

运行结果图：



**4、wait.c**

/\* wait2.c \*/

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

int main()

{

int status;

pid\_t pc,pr;

pc = fork();

if (pc < 0)

printf("error ocurred!\n");

else if (pc == 0)

{

printf("This is child process with pid of %d.\n",getpid());

exit(3);

}

else

{

pr = wait(&status);

if (WIFEXITED(status))                                                                      
{

printf("the child process %d exit normally.\n",pr);

printf("the return code is %d.\n",WEXITSTATUS(status));

}

else

printf("the child process %d exit abnormally.\n",pr);

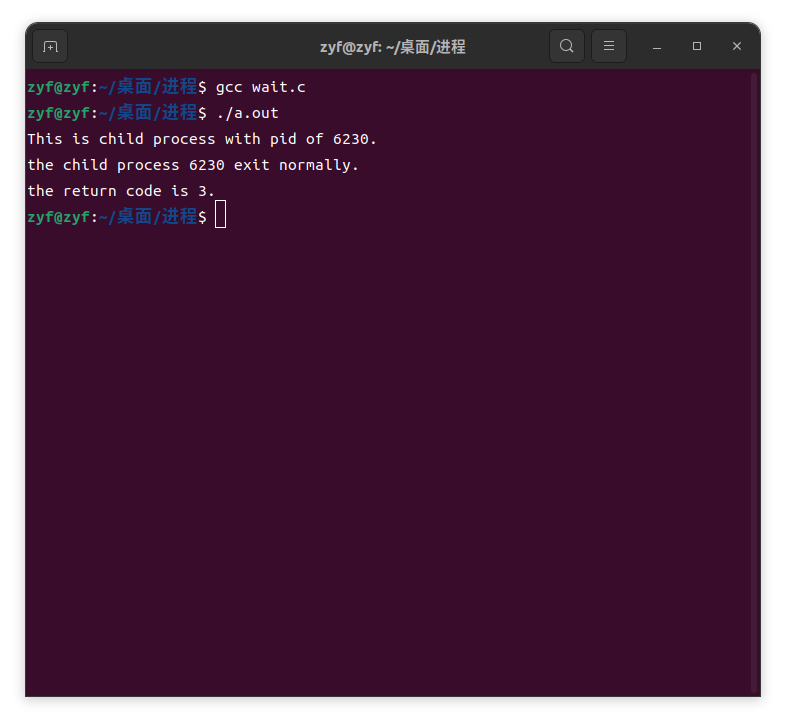
}

}

程序分析：

该段代码的主要功能为子进程执行打印函数打印自己的pid值并退出，父进程等待子进程退出后开始执行自己的代码，得到子进程退出时返回的status值，并打印子进程的进程号和子进程的返回值。

代码运行结果：

****

**5、waitpid.c**

/\* waitpid.c \*/

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

int main()

{

pid\_t pc, pr;

pc = fork();

if (pc < 0)

printf("Error occured on forking.\n");

else if (pc == 0)

{

sleep(10);

exit(0);  
}

else

do

{

pr = waitpid(pc, NULL, WNOHANG);

if (pr == 0)

{

printf("No child exited\n");

sleep(1);

}

}

while (pr == 0);

if (pr == pc)

printf("successfully get child %d\n", pr);

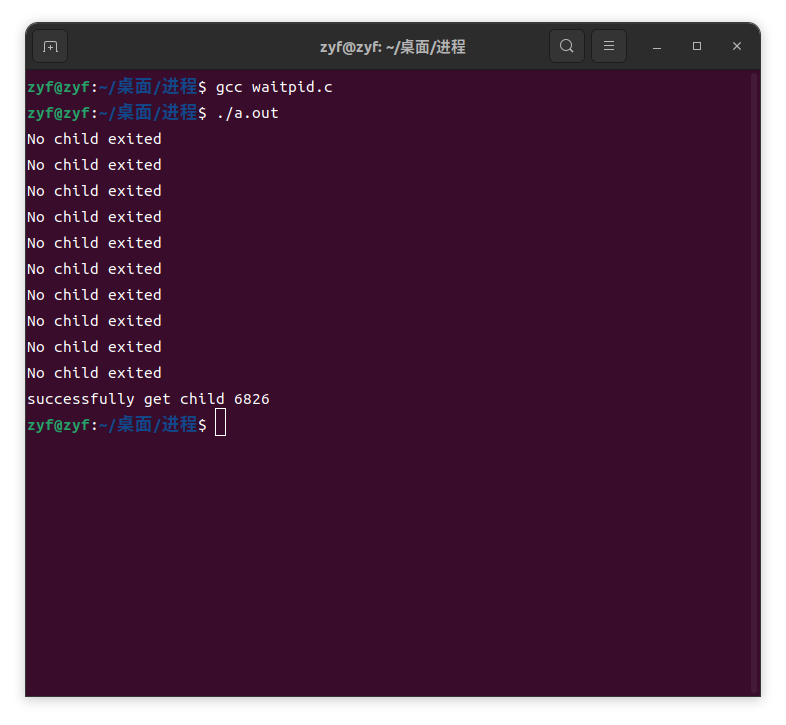
else

printf("some error occured\n");

}

程序分析：该段程序共有两个进程，父进程和子进程。子进程执行sleep函数，睡眠10s后退出，父进程等待子进程退出，并在等待过程中打印No child exited，睡眠1s，直到子进程退出时，父进程继续往下执行，打印接下来的代码。

运行结果截图：



**Part 2:** 信号

**6、Signal.c**

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

#include<sys/wait.h>

main()

{pid\_t pid;

int status;

if(!(pid=fork()))

{printf("Hi I am child process!\n");

sleep(100);

return;

}

else

{printf("send signal to child process (%d)\n",pid);

sleep(1);

kill(pid ,SIGKILL);

wait(&status);

if(WIFSIGNALED(status))

printf("chile process receive signal %d",WTERMSIG(status));

}

}

这段程序的主要功能如下：

在进程开始时，程序通过 fork() 函数创建了一个子进程，并返回子进程的进程 ID 存入 pid 变量中。

然后，通过判断 pid 是否为 0，来区分当前进程是父进程还是子进程，在本例中是父进程。

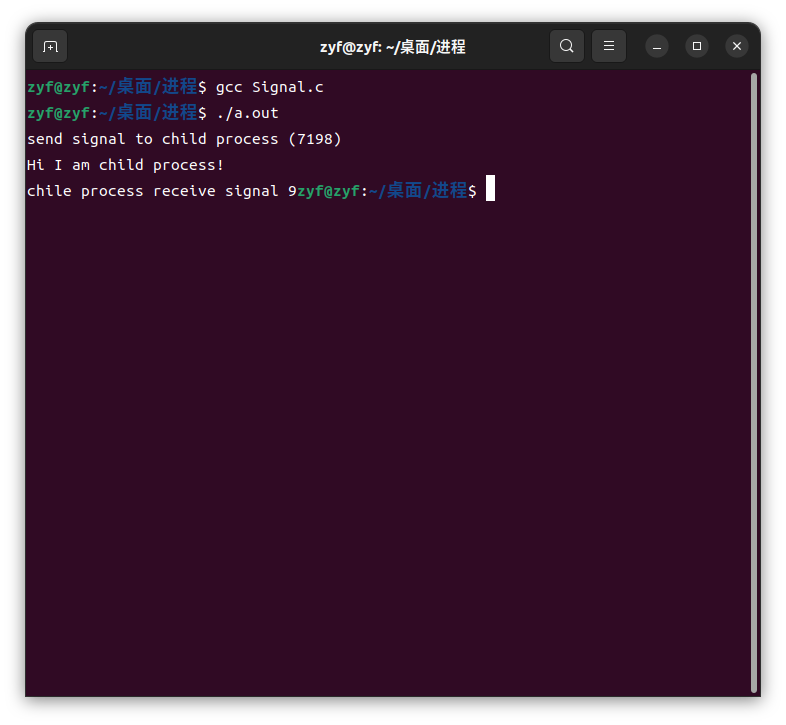
父进程输出一条信息，并通过 kill() 函数发送 SIGKILL 信号给子进程。该信号将会立即终止子进程的执行。

父进程通过 wait() 函数等待子进程结束，并通过 WIFSIGNALED() 函数判断子进程是否在接收到信号而结束。如果子进程因为接收到信号而结束，则程序输出该信号的编号。

子进程也会输出一条信息，但是由于使用了 sleep(100) 暂停了进程的执行，因此在父进程先结束之前不会发生任何其他操作。

综上所述，该程序的主要功能是通过信号机制，实现对子进程的强制终止，并检查其是否因信号而结束。

程序运行图：



**Part 3:** 信号量

运行下列两个程序，分析实验结果

7、semphore1.c

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/sem.h>

union semun

{

    int val;

    struct semid\_ds \*buf;

    unsigned short \*arry;

};

static int sem\_id = 0;

static int set\_semvalue();

static void del\_semvalue();

static int semaphore\_p();

static int semaphore\_v();

int main(int argc, char \*argv[])

{

    char message = 'X';

    int i = 0;

    //创建信号量

    sem\_id = semget((key\_t)1234, 1, 0666 | IPC\_CREAT);

    if(argc > 1)

    {

        //程序第一次被调用，初始化信号量

        if(!set\_semvalue())

        {

            fprintf(stderr, "Failed to initialize semaphore\n");

            exit(EXIT\_FAILURE);

        }

        //设置要输出到屏幕中的信息，即其参数的第一个字符

        message = argv[1][0];

        sleep(2);

    }

    for(i = 0; i < 10; ++i)

    {

        //进入临界区

        if(!semaphore\_p())

            exit(EXIT\_FAILURE);

        //向屏幕中输出数据

        printf("%c", message);

        //清理缓冲区，然后休眠随机时间

        fflush(stdout);

        sleep(rand() % 3);

        //离开临界区前再一次向屏幕输出数据

        printf("%c", message);

        fflush(stdout);

        //离开临界区，休眠随机时间后继续循环

        if(!semaphore\_v())

            exit(EXIT\_FAILURE);

        sleep(rand() % 2);

    }

    sleep(10);

    printf("\n%d - finished\n", getpid());

    if(argc > 1)

    {

        //如果程序是第一次被调用，则在退出前删除信号量

        sleep(3);

        del\_semvalue();

    }

    exit(EXIT\_SUCCESS);

}

static int set\_semvalue()

{

    //用于初始化信号量，在使用信号量前必须这样做

    union semun sem\_union;

    sem\_union.val = 1;

    if(semctl(sem\_id, 0, SETVAL, sem\_union) == -1)

        return 0;

    return 1;

}

static void del\_semvalue()

{

    //删除信号量

    union semun sem\_union;

    if(semctl(sem\_id, 0, IPC\_RMID, sem\_union) == -1)

        fprintf(stderr, "Failed to delete semaphore\n");

}

static int semaphore\_p()

{

    //对信号量做减1操作，即等待P（sv）

    struct sembuf sem\_b;

    sem\_b.sem\_num = 0;

    sem\_b.sem\_op = -1;//P()

    sem\_b.sem\_flg = SEM\_UNDO;

    if(semop(sem\_id, &sem\_b, 1) == -1)

    {

        fprintf(stderr, "semaphore\_p failed\n");

        return 0;

    }

    return 1;

}

static int semaphore\_v()

{

    //这是一个释放操作，它使信号量变为可用，即发送信号V（sv）

    struct sembuf sem\_b;

    sem\_b.sem\_num = 0;

    sem\_b.sem\_op = 1;//V()

    sem\_b.sem\_flg = SEM\_UNDO;

    if(semop(sem\_id, &sem\_b, 1) == -1)

    {

        fprintf(stderr, "semaphore\_v failed\n");

        return 0;

    }

    return 1;

}

程序主要功能如下：

创建一个名为1234的 IPC 信号量;

如果程序第一次调用 (即有带参数)，则初始化信号量，并指定需要输出到屏幕中的信息；

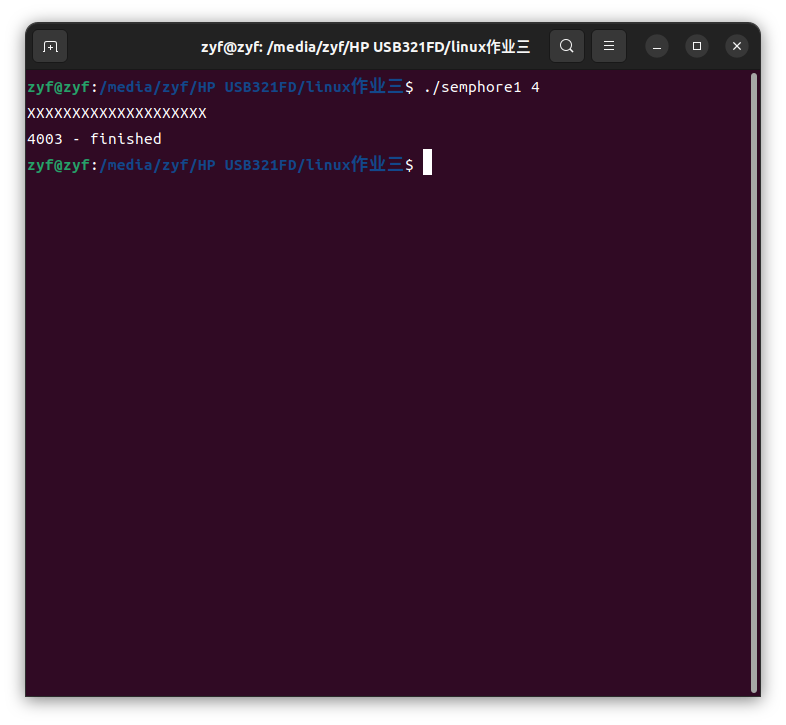
使用循环对进程进行多次操作，当一个进程进入共享资源临界区时使用 semaphore\_p() 对信号量做减1操作。

进程获得信号量后向屏幕输出数据，并在恰当的位置睡眠随机时间，再次向屏幕输出数据，在临界区内的输出结果可保证互斥；

离开临界区之前通过 semaphore\_v() 对信号量做加1操作释放许可，以便其他等待的进程可以继续操作共享资源；

第1~5步操作重复10次，循环结束后等待并输出进程 ID 并删除所创建的信号量。

程序运行结果：



8、nosemphore2.c

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

    char message = 'X';

    int i = 0;

    if(argc > 1)

        message = argv[1][0];

    for(i = 0; i < 10; ++i)

    {

        printf("%c", message);

        fflush(stdout);

        sleep(rand() % 3);

        printf("%c", message);

        fflush(stdout);

        sleep(rand() % 2);

    }

    sleep(10);

    printf("\n%d - finished\n", getpid());

    exit(EXIT\_SUCCESS);

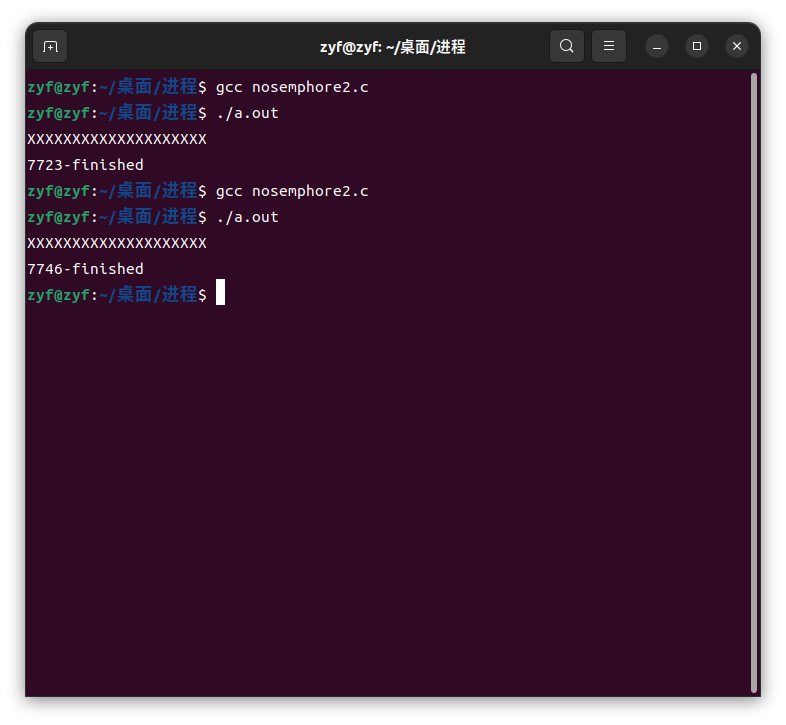
}

本段程序主要功能：

向终端输出指定字符，并在合适的位置睡眠一段随机时长后再次输出同一字符。程序会重复执行两次输出操作，在输出之间等待不同的随机时间。

程序可以接受命令行传入一个字符（如果存在），否则使用默认字符 'X'。循环执行10次输出操作，最后等待10秒钟以确保没有任何子进程未完成；在完成所有操作后，进程通过 EXIT\_SUCCESS 来正常退出。

本段程序运行图：



**Part 3:管道通信**

9、pipe.c

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{

int d1[2];

int d2[2];

int d3[2];

int r,j,k;

char buff[200];

printf("please input a string:");

scanf("%s",buff);

r=pipe(d1);

if(r==-1)

{

printf("chuangjianguandaoshibai 1\n");

exit(1);

}

r=pipe(d2);

if(r=-1)

{

printf("chuangjianguandaoshibai 2\n");

exit(1);

}

r=pipe(d3);

if(r==-1)

{

printf("chuangjianguandaoshibai 3\n");

exit(1);

}

r=fork();

if(r)

{

close(d1[1]);

read(d1[0],buff,sizeof(buff));

if(strlen(buff)%2==1)

{

j=fork();

if(j)

{

close(d2[1]);

read(d2[0],buff,sizeof(buff));

printf("p3 pipe2 odd length string: %s\n",buff);

close(d2[0]);

exit(0);

}

else

{

close(d2[0]);

write(d2[1],buff,strlen(buff));

printf("P2 finishes writing to pipe2.\n");

close(d2[1]);

exit(0);

}

}

else

{

k=fork();

if(k)

{

close(d3[1]);

read(d3[0],buff,sizeof(buff));

printf("P4 pipe3 even length string:%s\n",buff);

close(d3[0]);

exit(0);

}

else

{

close(d3[0]);

write(d3[1],buff,strlen(buff));

printf("P2 finishes writing to pipe3.\n");

close(d3[1]);

exit(0);

}

}

}

else

{

close(d1[0]);

write(d1[1],buff,strlen(buff));

close(d1[1]);

exit(0);

}

}

本段程序主要功能如下：

用户输入一个字符串，并将其存储在缓冲区中；

创建了三个有名管道(d1, d2, d3)用于父进程和它的子进程之间进行通信和数据传输；

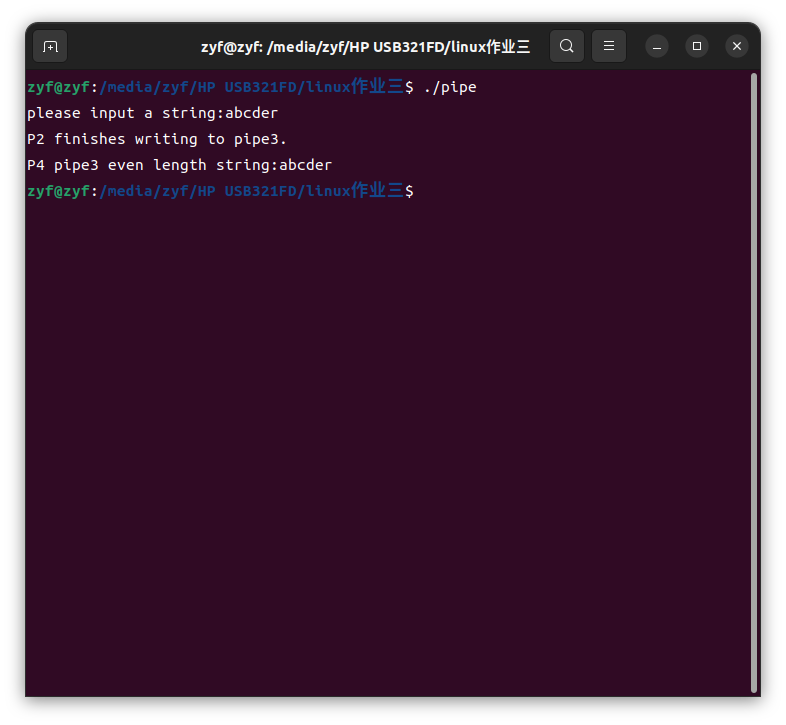
执行fork()创建两个子进程，其中一个子进程(p3)，用于处理从d1读取的字符串长度为奇数的情况；另一个子进程(P4)，负责处理长度为偶数的情况。

在奇数长度的情况下，p3 子进程被启动将原始字符串写入d2管道。父进程监听d2读取到来的数据，一旦数据到达，就打印出来并关闭管道；

在偶数长度的情况下，p4子进程被启动将原始字符串写入d3管道。父进程监听d3读取到来的数据，一旦数据到达，就打印出来并关闭管道；

程序输出“P2 finishes writing to pipe2”（奇数情况）和“P2 finishes writing to pipe3”（偶数情况），表示 P2 子进程执行结束并成功写入管道。

程序运行结果：



**Part 4:IPC 消息队列通信**

10、msg\_Client.c

// 客户端程序msg\_client.c

# include <sys/types.h>

# include <sys/ipc.h>

# include <sys/msg.h>

# include <stdio.h>

# include <unistd.h>

# define MSGKEY 75

struct msgform

{

long mtype;

char mtext[256];

};

Int main()

{

struct msgform msg;

int msgqid,pid,\*pint;

msgqid=msgget(MSGKEY,0777);

pid=getpid();

printf(“client:pid=%d\n”,pid);

pint=(int\*)msg.mtext;

\*pint=pid;

msg.mtype=1;

msgsnd(msgqid,&msg,sizeof(int),0);

msgrcv(msgqid,&msg,256,pid,0);

printf(“client:receive from pid%d\n”,\*pint);

}

本段程序功能：

程序首先定义了表示消息结构体 msgform，其中 mtype 变量表示消息类型，mtext 数组存储消息数据；

执行 msgget() 从系统中获取/创建一个 IPC 消息队列，并将消息队列ID值保存在变量 msgqid 中；

获得当前进程的 PID 值，并将其存储在 msg.mtext 数组中；

设置待发送消息的类型为 1，然后调用 msgsnd() 将消息放入到消息队列中；

在等待接收其他进程通过消息队列发送的消息时，调用 msgrcv() 函数阻塞程序的运行状态，并根据 pid 参数指定接收到特定进程发出的消息；

一旦有进程使用该消息队列向本程序发送消息，就会触发 msgrcv() 函数读取，同时打印出从中获取到的消息内容。

该程序的主要流程是：客户端在初始化后与预处理服务建立连接，通过共享内存进行数据传输并做出相应的响应。

msg \_Server.c

/\* 服务端程序msg\_server.c\*/

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#define MSGKEY 75

struct msgform

{

    long mtype;

    char mtext[256];

};

int msgqid;

void cleanup()

{

    msgctl(msgqid, IPC\_RMID, 0); /\*删除队列\*/

    exit(0);

}

int main()

{

    struct msgform msg;

    int pid, \*pint, i;

    // extern cleanup();

    for (i = 0; i < 23; i++)

        signal(i, cleanup);

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

    printf("server : pid = % d\n", getpid());

    for (;;)

    {

        msgrcv(msgqid, &msg, 256, 1, 0);

        pint = (int \*)msg.mtext;

        pid = \*pint;

        printf("server: receive from pid %d\n", pid);

        msg.mtype = pid; /\*将接收的客户进程的pid为消息类型\*/

        \*pint = getpid();

        msgsnd(msgqid, &msg, sizeof(int), 0);

    }

}

本段程序功能：

定义了表示消息结构体 msgform，其中的 mtype 变量表示消息类型，mtext 数组用于存储消息数据；

首先通过 signal() 函数为函数退出设置回调函数 cleanup()，以确保服务器程序正常退出时删除 IPC 通道队列；

执行 msgget() 从系统中获取/创建一个 IPC 消息队列，并将其 ID 值保存在变量 msgqid 中；

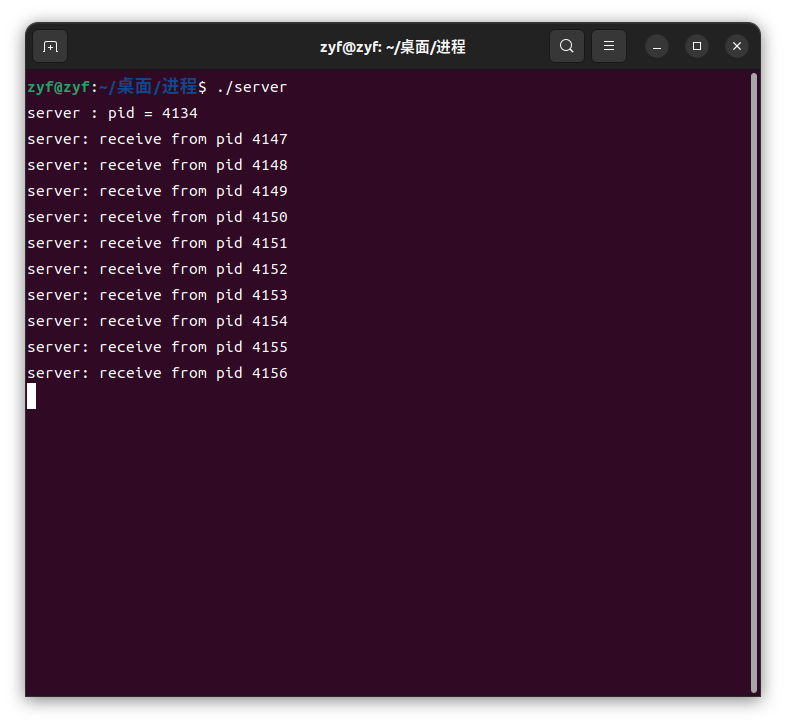
不断执行 msgrcv() 函数阻塞服务器程序运行状态，等待其他进程使用该消息队列向本程序发送消息；

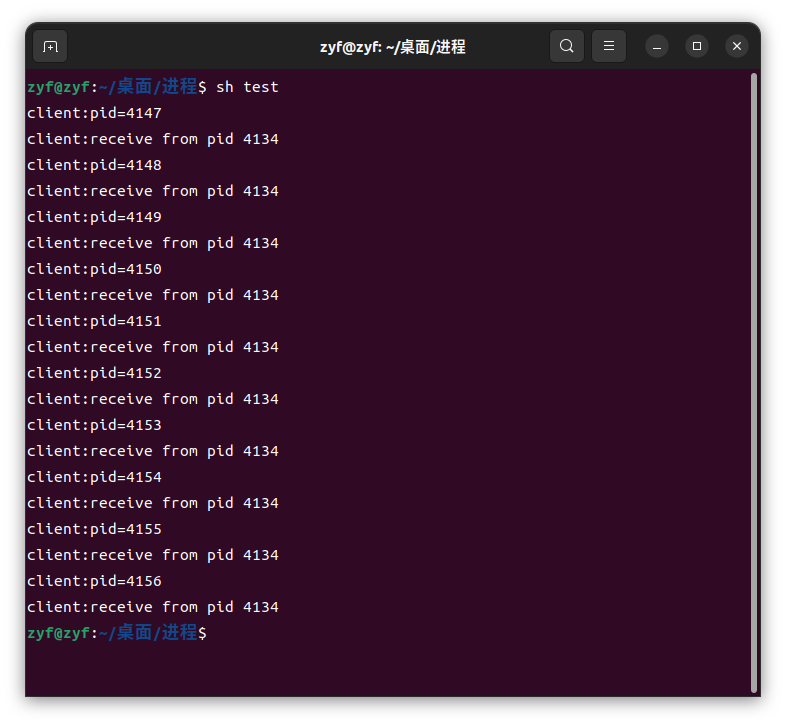
每当有其他进程通过消息队列向此服务器程序发送消息时，通过 msg.type=pint; 将原始客户进程的 PID 设为当前接收到消息的消息类型，同时将本服务进程 PID 存储在 msg.mtext[] 中；

最后通过 msgsnd() 向消息队列发送处理完成的消息。

该程序的主流程是：服务端在初始化后，通过 IPC 消息队列与其他客户端建立连接并利用共享内存进行数据传输。网络客户只需要向该服务进程发送一个请求，服务进程就会根据请求生成相应的响应并返回至客户端实现网络通信。

程序运行结果图：





**Part 5:** 共享存储通信

11、shm.c

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int main(void)

{

    int x, shmid;

    int \*shmptr;

    if((shmid=shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT|0666)) < 0)

        printf("shmget error"), exit(1);

    if((shmptr=(int \*)shmat(shmid, 0, 0)) == (int \*)-1)

        printf("shmat error"), exit(1);

    printf("Input a initial value for \*shmptr: ");

    scanf("%d", shmptr);

    while((x=fork())==-1);

    if(x==0)

    {

        printf("When child runs, \*shmptr=%d\n", \*shmptr);

        printf("Input a value in child: ");

        scanf("%d", shmptr);

        printf("\*shmptr=%d\n", \*shmptr);

    }

    else

    {

        wait();

        printf("After child runs, in parent, \*shmptr=%d\n", \*shmptr);

    if ( shmctl(shmid, IPC\_RMID, 0) < 0 )

        printf("shmctl error"), exit(1);

    }

        return 0;

}

该段程序功能：

调用 shmget() 函数创建一个新的共享内存区域，并将其 ID 值保存在 shmid 变量中；

将分配的共享内存联接到当前进程中，获取其地址指针并保存在 shmptr 变量中；

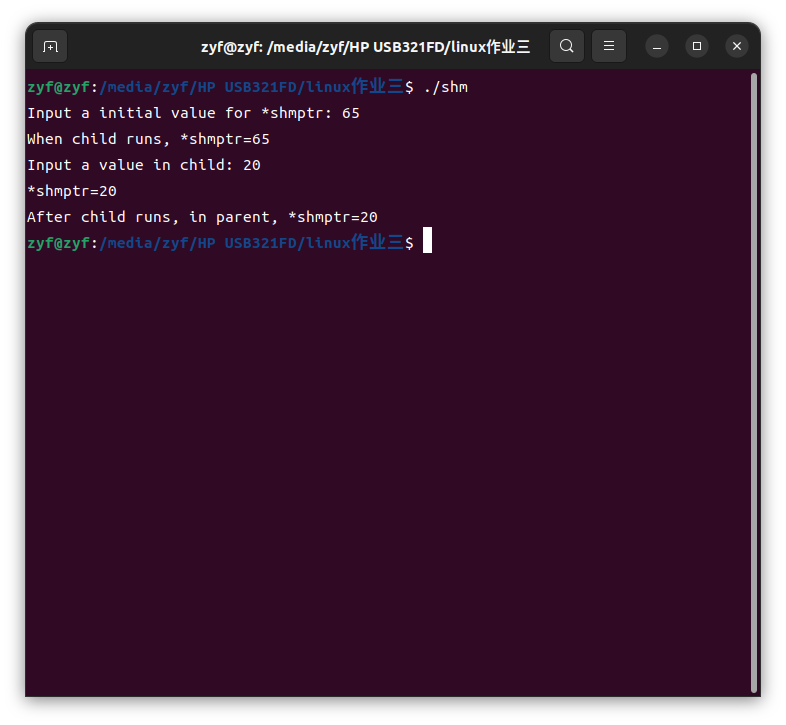
通过标准输入流读取用户输入的一个整数值，保存在 \*shmptr 这个整型变量在共享内存中；

执行一个子进程，根据不同的进程 ID 进行不同的处理操作；

在子进程中，首先打印出共享内存上的数据内容，可以看到此时的共享内存数据与父进程中输入的值相同；然后从标准输入流中获取一个整数值，更新共享内存中的值，再次打印更新后的数据内容；

在父进程中，等待子进程运行结束后，打印共享内存的数据内容，可以看到子进程已经将共享内存上的数据完成了更改。最后执行 shmctl() 函数删除共享内存。

运行结果图：



**Part 6:** 线程

12、thread.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

int num = 100;

// myfunc 线程

void \*myfunc(void \*arg)

{

printf("child pthread id = %ld\n", pthread\_self());

for (int i = 0; i < 5; i++)

{

printf("child pthread i = %d\n", i);

if (i == 2)

{

num = 666; // 验证不同线程可以利用全局变量通信

// pthread\_exit(NULL); // 不携带数据的退出

pthread\_exit(&num); // 携带数据的退出

}

}

return 0;

}

int main()

{

int ret;

int i = 0;

pthread\_t pthid;

ret = pthread\_create(&pthid, NULL, myfunc, NULL); // 线程创建

if (ret != 0) // 创建失败判断

{

printf("error number is %d\n", ret);

printf("%s\n", strerror(ret));

}

printf("parent pthread id = %ld\n", pthread\_self());

void \*ptr = NULL;

ptr = malloc(32); // 如果不动态申请内存会发生段错误

void \*tmp = ptr; // 用 tmp 指向申请的内存来操作内存，以防改变 ptr 的指向导致 free 时产生段错误

pthread\_join(pthid, &tmp); // 第二个参数是 void \* 类型的二级指针，可以用来获取 exit 参数携带的数据

printf("num = %d\n", \*(int \*)tmp);

free(ptr); // 释放掉申请的内存

ptr = NULL; // 指针指向 NULL 以防后续误操作

while (i < 5)

{

i++;

printf("parent pthread i = %d\n", i);

}

sleep(2);

return 0;

}

该段程序功能：

1.主程序中定义了一个全局变量 num，线程函数 myfunc() 可以通过修改该全局变量的值来实现线程通信；

2.主程序中创建了一个新的线程 pthid，通过调用 pthread\_create() 函数并将 要执行的线程函数 myfunc() 和相应的参数传递给它进行线程创建；

3.在主程序中和子线程中分别打印出当前线程 ID，可以看到不同线程具有不同的线程 ID；

4.在子线程中使用循环输出一些东西，并在第三次循环时更新了全局变量 num的值并退出线程（同时传递了一个携带数据的退出参数），以验证多个线程之间可以利用全局变量进行通信；

5.在主程序中使用 pthread\_join() 函数等待子线程结束，在这里我们传入了一个指向指针的指针变量 tmp 来接收子线程运行结束时的返回值，通过强制类型转换把二级指针 tmp 转换为整型指针来接收退出参数数据；

6.最后释放动态申请的内存，在释放后将指针置为空以防再次操作；

7.通过在父进程和子进程中都使用循环打印内容来展示多线程环境下不同线程之间的运行互不影响。

程序运行结果图：

