北京郵電大學

Beijing University of Posts and Telecommunications

实 验 报 告

课 程 名 称	操作系统
题 目	实验二 进程控制
姓 名	沈原灏
学 号	2021211108
班 级	2021211306
指 导 教 师	李文生

2023 年 10 月

目录

一、实验概述

1.1 实验内容

Collatz 猜想：任意写出一个正整数 N，并且按照以下的规律进行变换：

- 如果是个奇数，则下一步变成 3N + 1；
- 如果是个偶数，则下一步变成 N/2。

无论 N 是怎样的一个数字，最终都会变成 1。

1.1.1 实验一

采用系统调用 fork()，编写一个 C 程序，以便在子进程中生成这个序列。

要求：

（1）从命令行提供启动数字。

（2）由子进程输出数字序列。

（3）父进程等子进程结束后再退出。

1.1.2 实验二

以共享内存技术编程实现 Collatz 猜想。要求：

在父子进程之间建立一个共享内存对象，允许子进程将序列内容写入共享内存对象，当子进程完成时，父进程输出序列。

父进程包括如下步骤：

- 建立共享内存对象（shm_open(), ftruncate(), mmap()）
- 建立子进程并等待终止
- 输出共享内存的内容
- 删除共享内存对象。

1.1.3 实验三：普通管道通信

设计一个程序，通过普通管道进行通信，让一个进程发送一个字符串消息给第二个进程，第二个进程收到此消息后，变更字母的大小写，然后再发送给第一个进程。比如，第一个进程发消息：“I am Here”，第二个进程收到后，将它改变为：“i AM hERE”之后，再发给第一个进程。

提示：

（1）需要创建子进程，父子进程之间通过普通管道进行通信。

（2）需要建立两个普通管道。

1.2 实验环境

- Ubuntu 22.04.1 LTS
- gcc 11.4.0
- Visual Studio Code

二、实验分析与设计

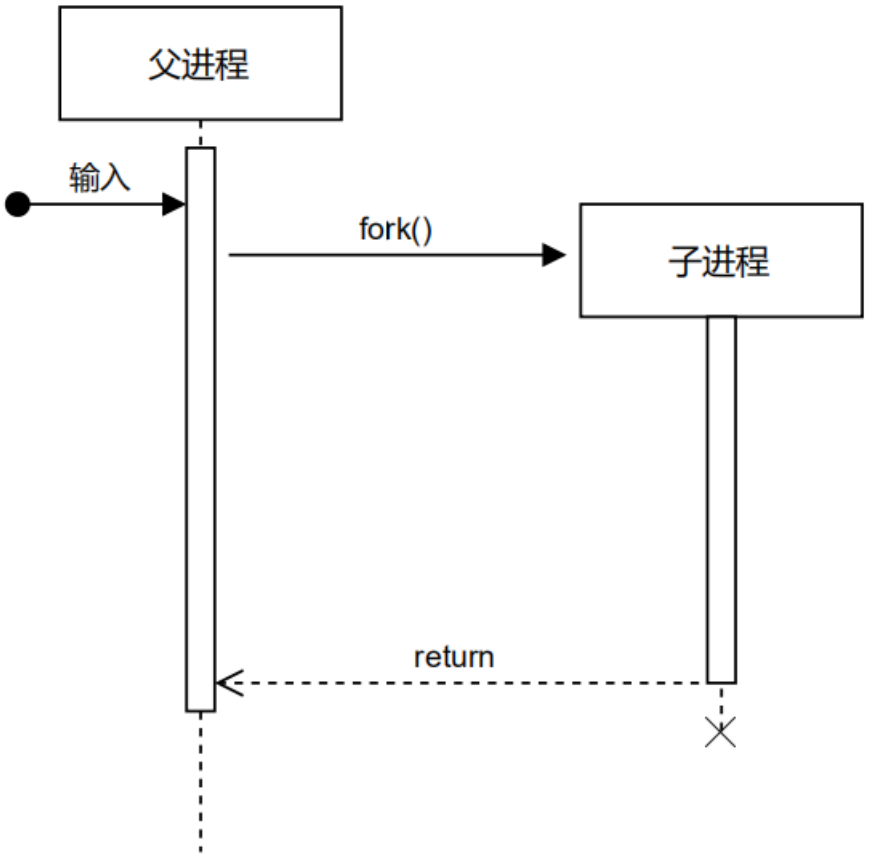
2.1 实验一

2.1.1 相关 API

- pid_t fork(void); 创建子进程，其中子进程是调用该函数的进程的拷贝。子进程与父进程有各自单独的地址空间，在调用之后其中的内容是相同的。如果创建成功，该函数返回给父进程以子进程的 PID，给子进程返回 0。
- pid_t wait(int *wstatus); 暂停调用该函数的进程直到所有子进程都终止，当成功时，返回终止的子进程的 PID，失败时返回-1。
- collatz_sequence(int start); 接受一个 int 类型的初始值，用于生成 Collatz 猜想的数列。

2.1.2 设计概述

如图所示，父进程首先通过命令行参数读取 n，之后通过 fork() 函数创建子进程，并调用 wait()函数等待子进程结束。
子进程得到与父进程数据段地址空间的一份副本，其中包含 n 的值。之后子进程通过调用 `collatz_sequence()`函数循环计算序列并且输出，之后通过 return 0 终止。
父进程回收子进程后，输出一行字符串之后终止



2.1.3 代码实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

// generate Collatz sequence
void collatz_sequence(int pid, int start)
{
   if (pid == 0)
   {
      printf("Child:");
   }
   while (start != 1)
   {
      printf(" %d", start);

      if (start % 2 == 0)
      {
         start = start / 2;
      }
      else
      {
         start = 3 * start + 1;
      }
   }

   printf(" 1\n");
}

int main(int argc, char *argv[])
{
   if (argc != 2)
   {
      printf("Usage: %s <start_number>\n", argv[0]);
      return 1;
   }

   int start = atoi(argv[1]);

   if (start <= 0)
   {
      printf("The number must be positive!\n");
```

```
    return 1;
  }

  pid_t pid = fork();

  if (pid == -1)
  {
    perror("Fork failed");
    return 1;
  }
  else if (pid == 0)
  {
    // generate sequence in child process
    collatz_sequence(pid, start);
  }
  else
  {
    // wait for child
    wait(NULL);
    printf("\nParent: Done.\n");
  }

  return 0;
}
```

2.1.4 编译执行

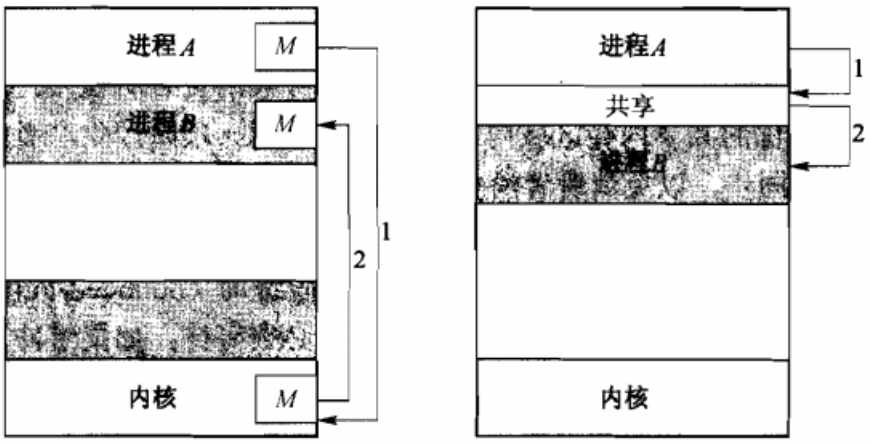
```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc a.c -o a
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./a 20
Child: 20 10 5 16 8 4 2 1

Parent: Done.
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./a -10
The number must be positive!
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$
```

如果参数合法，会输出相应的序列与进程名（Parent 或 Child），否则会报错。

2.2 实验二

2.2.1 相关 API

如图所示，进程之间的通信有两种方式：消息传递和共享内存。消息传递就是利用操作系统内核的一些函数比如 send()、receive()来像网络通信一样进行数据交流；共享内存则是专门开辟一块空间，两个进程都可以访问、改写这里面的数据，这样就实现了数据的交流。



（a）消息传递 （b）共享内存

LINUX 共享内存的 POSIX 接口是 shm_open 这个函数。其实这个接口就是在/dev/shm 目录创建一个文件，然后使用这个文件来实现共享内存的功能。

- int shm_open(const char *name, int oflag, mode_t mode); 创建或打开一个共享内存，成功返回一个整数的文件描述符，错误返回-1。name：共享内存区的名字；标志位：open 的标志一样；权限位。
- int ftruncate(int fd, off_t length); 通过该函数给文件（即之前创建的共享内存对象）分配空间，fd 表示文件描述符，length 为裁切的长度。如果成功，返回 0。
- int munmap(void *addr, size_t length); 删除给定地址的映射，之后对于给定地址的引用会无效。当进程终止时，映射也会被自动删除。
- int shm_unlink(const char *name); 移除一个共享内存对象名称，如果所有的进程都已经删除了对该对象的映射，那么该内存区域会被释放。

2.2.2 设计概述

流程仍然如图 1 所示，父进程首先读取 n，创建共享内存对象并且分配空间，创建对共享内存的映射，之后通过 fork() 函数创建子进程，并调用 wait() 函数等待子进程结束。

子进程得到了 n 和共享内存的地址，计算循环计算序列并且以字符串的形式打印到共享内存中，之后删除映射并通过 return 0 终止。

父进程回收子进程后，向标准输出打印共享内存中的字符串，之后删除映射并且移除共享内存对象（此时会释放共享内存），最后终止。

2.2.3 代码实现

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
```

```c
// shared memory size
#define SHM_SIZE 1024

// generate collatz_sequence and add to shared memory
void collatz_sequence(int start, int *shm_ptr)
{
   int index = 0;

   while (start != 1)
   {
      shm_ptr[index] = start;
      index++;

      if (start % 2 == 0)
      {
         start = start / 2;
      }
      else
      {
         start = 3 * start + 1;
      }
   }

   shm_ptr[index] = 1;
}

int main(int argc, char *argv[])
{
   if (argc != 2)
   {
      printf("Usage: %s <start_number>\n", argv[0]);
      return 1;
   }

   int start = atoi(argv[1]);

   if (start <= 0)
   {
      printf("The number must be positive!");
      return 1;
   }

   // create shared memory
```

```
  int shm_fd = shm_open("/collatz_shm", O_CREAT | O_RDWR, 0666);
  ftruncate(shm_fd, SHM_SIZE);

  // map shared memory to process address space
  int *shm_ptr = (int *)mmap(0, SHM_SIZE, PROT_WRITE | PROT_READ,
MAP_SHARED, shm_fd, 0);

  pid_t pid = fork();

  if (pid == -1)
  {
    perror("Fork failed");
    return 1;
  }
  else if (pid == 0)
  {
    // generate sequence and write to shm in child
    collatz_sequence(start, shm_ptr);
  }
  else
  {
    // wait for child
    wait(NULL);

    // print
    printf("------sequence from shared memory------\n");
    int index = 0;
    while (shm_ptr[index] != 0)
    {
      printf("%d ", shm_ptr[index]);
      index++;
    }
    printf("\n");
    printf("----------------- END -----------------\n");

    // delete
    munmap(shm_ptr, SHM_SIZE);
    close(shm_fd);
    shm_unlink("/collatz_shm");
  }

  return 0;
}
```

2.2.4 编译执行

```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc b.c -o b
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./b 20
------sequence from shared memory------
20 10 5 16 8 4 2 1
---------------- END -----------------
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./b -10
The number must be positive!
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$
```

这里父进程成功创建了共享内存和子进程，子进程计算了序列并且输出到共享内存，父进程打印了共享内存的内容。

2.3 实验三：普通管道通信

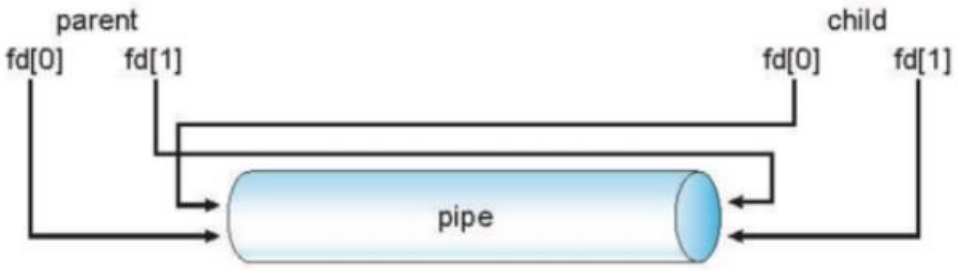
2.3.1 相关 API

- int pipe(int pipefd[2]); 创建一个单向的管道，pipefd 用于返回管道两端的文件描述符，分别为读取端和写入端。如果成功，函数返回 0。
- int close(int fd); 关闭一个文件描述符。
- ssize_t write(int fd, const void *buf, size_t count); 向文件描述符 fd 指明的文件写入从 buf 开始的 count 个字节。若成功，返回写入的字节数。
- ssize_t read(int fd, void *buf, size_t count); 从文件描述符 fd 指明的文件读取到从 buf 开始的缓冲区，最多读取 count 个字节。若成功，返回读取的字节数。

2.3.2 设计概述

一个普通的管道具有一个“write-end”写端和一个“read-end”读端。这里采用的是单向管道，即进程 A 建立一个管道并不断向 B 写，B 也可以同时用另一个管道不断向 A 写。管道一般分为两种：

- 普通管道：无法从创建它的进程外部访问。通常，父进程创建一个管道，并使用它与它创建的子进程通信。
- 命名管道：可以在没有父子关系的情况下被访问。

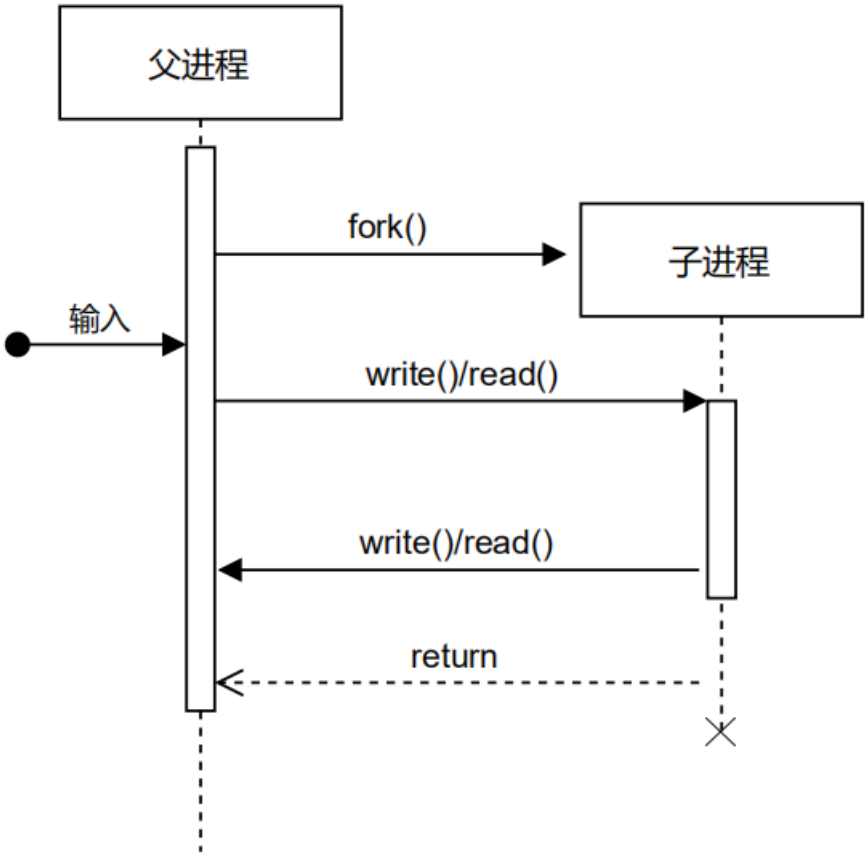


如图所示，父进程首先创建两个管道，之后通过 fork() 函数创建子进程。父进程创建管道，得到两个文件描述符指向管道的两端

父进程 fork()创建子进程，子进程也有两个文件描述符指向同一管道。

父进程关闭 fd[0]，子进程关闭 fd[1]，即父进程关闭管道读端，儿子进程关闭管道写端（因为管道只支持单向通信）。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实

现了进程间通信。



2.3.3 代码实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <ctype.h>

#define MAX_SIZE 256

int main()
{
   int pipe1[2]; // 管道 1，从父进程到子进程
   int pipe2[2]; // 管道 2，从子进程到父进程

   if (pipe(pipe1) == -1 || pipe(pipe2) == -1)
   {
      perror("Pipe creation failed");
      exit(1);
   }
```

```
  pid_t pid = fork();

  if (pid == -1)
  {
    perror("Fork failed");
    exit(1);
  }

  if (pid == 0)
  {
    // 子进程
    close(pipe1[1]); // 关闭管道 1 写入端
    close(pipe2[0]); // 关闭管道 2 读取端

    char message[MAX_SIZE];
    read(pipe1[0], message, sizeof(message)); // 从管道 1 中读取消息

    // 将消息中的字母大小写进行翻转
    for (int i = 0; message[i] != '\0'; i++)
    {
      if (islower(message[i]))
      {
        message[i] = toupper(message[i]);
      }
      else if (isupper(message[i]))
      {
        message[i] = tolower(message[i]);
      }
    }

    write(pipe2[1], message, strlen(message) + 1); // 将翻转后的消息
写入管道 2

    close(pipe1[0]);
    close(pipe2[1]);
  }
  else
  {
    // 父进程
    close(pipe1[0]); // 关闭管道 1 读取端
    close(pipe2[1]); // 关闭管道 2 写入端

    char message[256];
    printf("Enter a message in parent: ");
```

```
    fgets(message, sizeof(message), stdin); // 从用户输入中获取消息

    write(pipe1[1], message, strlen(message) + 1); // 将消息写入管道1

    read(pipe2[0], message, sizeof(message)); // 从管道 2 中读取翻转后的消息

    printf("Received message from child: %s", message);

    close(pipe1[1]);
    close(pipe2[0]);
  }

  return 0;
}
```

2.3.4 编译运行

```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc c.c -o c
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./c
Enter a message in parent: Hello! How's it going?
Received message from child: hELLO! hOW'S IT GOING?
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ _
```

主进程和子进程之间成功用管道交换信息。

三、测试与结果分析

3.1 实验一

```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc a.c -o a
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./a 20
Child: 20 10 5 16 8 4 2 1

Parent: Done.
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./a -10
The number must be positive!
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$
```

在实验一中，主进程成功创建了子进程，子进程计算并输出了序列。同时，程序进行了**非法输入处理**（输入不为正数时报错：The number must be positive）。

3.2 实验二

```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc b.c -o b
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./b 20
------sequence from shared memory------
20 10 5 16 8 4 2 1
---------------- END ----------------
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./b -10
The number must be positive!
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$
```

在实验二中，主进程成功创建了共享内存和子进程，子进程计算了序列并且输出到共享内存，主进程最后打印出共享内存的内容。同样，代码实现了**非法输入处理**。

3.3 实验三

```
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ gcc c.c -o c
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ ./c
Enter a message in parent: Hello! How's it going?
Received message from child: hELLO! hOW'S IT GOING?
arycra07@arycra07-virtual-machine:~/Desktop/os_lab1$ _
```

在实验三中，主进程成功地通过管道与子进程进行了通信，并获得了大小写相反的回复。

四、实验心得

由于本次实验使用 C 语言实现，我采用了 Linux 环境进行编程，使得我能够更方便地使用 gcc 进行编译执行。

我觉得最有意思的是第二部分的共享内存方法，两个进程开辟了一个类似信箱的“共享内存”，让它们不必进行任何直接交流便可以达到通信的效果。相比较下，管道通信就未免显得笨拙一点。

在整个实验过程中，我查阅了课本和多种资料，达到了复习提高的效果，也使我的 Linux-C 语言编程能力和英文文献阅读能力得到提高，收获颇丰。