

# 操作系统实验二

汪航 2021211114

2023 年 10 月 20 日

## 1 实验内容

本次实验编写了三道程序，其中前两道程序分别使用了开辟子进程和内存共享实现父子进程通信的方法实现了 Collatz 猜想；第三道程序实现了父子进程双向、异步的管道通信。

## 2 开发环境

Linux(WSL 下的 Ubuntu)

## 3 程序一

### 3.1 主要内容

使用系统调用 `fork()` 编写程序生成 Collatz 序列并输出，使用命令行提供输入数字。

使用主要系统调用 `fork()`, `wait()`。

### 3.2 程序设计思路

采用类似于递归的设计思想，对于一个正整数  $N$ ，处理该数的进程先将  $N$  打印出来，然后对  $N$  进行变换：如果是个奇数，则下一步  $N$  变成  $3N+1$ ；如果是个偶数，则  $N$  变成  $N/2$ 。做完变换父进程开辟一个子进程，子进程

的  $N$  值复制自修改过的父进程的  $N$  值，父进程等待子进程执行完便退出。该过程一直执行至  $N==1$ ，此时进程直接退出，由此等待子进程的父进程全部退出，程序执行完毕。

代码实现：

```
1  /*homework.c*/
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  int main(int argc, char* argv[])
7  {
8      int N=atoi(argv[1]);
9      pid_t pid;
10     while (1)
11     {
12         pid = fork();
13
14         /*fork child process*/
15         if (pid == 0)
16         {
17             if (N % 2 == 0)
18             {
19                 printf("%d ", N);
20                 fflush(stdout);
21                 N = N / 2;
22             }
23             else if (N % 2 != 0)
24             {
25                 if (N == 1)
26                 {
27                     printf("%d ", N);
28                     fflush(stdout);
29                     break;
30                 }
31                 else
32                 {
33                     printf("%d ", N);
34                     fflush(stdout);
35                     N = 3 * N + 1;
36                 }
37             }
38         }
39         else if (pid > 0)
40         {
41             wait(NULL);
```

```
42     fflush(stdout);
43         break;
44     }
45     else
46     {
47         fprintf(stderr, "Fork Failed");
48         return 1;
49     }
50 }
51 return 0;
52 }
```

注意到每次 `printf()` 后便会使用 `fflush()` 清空输出区, 这是因为进程的输出是异步的, 若不及时清空输出缓存区则会有相同内容的重复输出。

### 3.3 程序运行结果

输入: `./homework 35`

输出: 35 106 53 160 80 40 20 10 5 16 8 4 2 1

输入: `./homework 41`

输出: 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91  
274 137 412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593  
1780 890 445 1336 668 334 167 502 251 754 377 1132 566 283 850 425 1276  
638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822  
911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433  
1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10  
5 16 8 4 2 1

结果分析表明, 该程序能通过复制子进程的方式正确输出 Collatz 序列且能正确地终止。同时表明 Collatz 序列的逆序列是一个固定值的序列, 以一开头, 可能包含所有的正整数。

## 4 程序二

### 4.1 主要内容

以共享内存技术编程实现 Collatz 猜想。程序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。

使用主要系统调用：fork(), shm\_open(), ftruncate(), mmap(), fork(), wait(), sprintf(), shm\_unlink()。

### 4.2 程序设计思路

程序分为两个进程：父进程和子进程。父进程先获得输入需要的 N 值并创建共享内存对象，使指针 ptr 指向被创建的共享内存对象然后创建子进程并等待，子进程使用 while 循环输出对应的 Collatz 序列到共享内存对象中然后退出。父进程等子进程执行完后输出共享内存对象中的序列值后删除共享内存对象，程序结束。

代码实现：

```
1  /*homework2.c*/
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<string.h>
5  #include<fcntl.h>
6  #include<sys/shm.h>
7  #include<sys/stat.h>
8
9  #include<sys/mman.h>
10 #include<unistd.h>
11 #include<math.h>
12 /*gcc homework2.c -o homework2 -lm*/
13 int main(int argc, char* argv[])
14 {
15     /*name of the shared memory object*/
16     const char *name = "OS";
17     const int SIZE = 4096;
18     /*pointer to shared memory object*/
19     char *ptr;
20     int N = atoi(argv[1]);
21     /*shared memory file descriptor*/
22     int fd;
```

```
23  /*create the shared memory object*/
24  fd = shm_open(name, O_CREAT | O_RDWR, 0666);
25
26  /*configure the size of the shared memory object*/
27  ftruncate(fd, SIZE);
28
29  /*memory map the shared memory object*/
30  ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
31
32  pid_t pid;
33  pid = fork();
34  if(pid==0){
35      while(N!=1)
36      {
37          sprintf(ptr, "%d ", N);
38          ptr += ((int)(log10(N))+2);
39          if (N % 2 == 0)
40          {
41              N = N / 2;
42          }
43          else if (N % 2 != 0)
44          {
45              N = 3 * N + 1;
46          }
47      }
48      sprintf(ptr, "%d", N);
49      ptr += 1;
50  }
51  else if (pid > 0)
52  {
53      wait(NULL);
54      printf("%s", (char *)ptr);
55      shm_unlink(name);
56  }
57  else
58  {
59      fprintf(stderr, "Fork Failed");
60      return 1;
61  }
62  return 0;
63 }
```

编译该文件需要使用 `gcc homework2.c -o homework2 -lm` 指令。若不加后缀 `-lm` 会导致 `log10()` 函数未正确加载。

### 4.3 程序运行结果

输入: `./homework 35`

输出: 35 106 53 160 80 40 20 10 5 16 8 4 2 1

输入: `./homework 41`

输出: 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91  
274 137 412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593  
1780 890 445 1336 668 334 167 502 251 754 377 1132 566 283 850 425 1276  
638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822  
911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433  
1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10  
5 16 8 4 2 1

结果分析表明, 该程序能够通过父子进程通过共享内存通信的方式实现 Collatz 序列的输出。

## 5 程序三

### 5.1 主要内容

程序创建父子进程, 父子进程之间通过管道进行通信, 父进程给子进程发送“I am here”, 子进程收到后, 将它改变为: “i AM hERE”之后, 再发给父进程。

### 5.2 程序设计思路

如上。需要注意的一点是普通管道是单工管道, 一方只能读另一方只能写。因此需要开辟两个管道, 让父子进程分别作为第一第二个管道的写端, 第二第一个管道的读端。

代码实现:

```
1 /*homework3.c*/  
2 #include<sys/types.h>  
3 #include<stdio.h>  
4 #include<string.h>
```

```
5 #include<unistd.h>
6 #include<stdlib.h>
7
8 #define BUFFER_SIZE 25
9 #define READ_END 0
10 #define WRITE_END 1
11
12 int main()
13 {
14     char write_msg[BUFFER_SIZE] = "I am Here";
15     char read_msg[BUFFER_SIZE];
16     int fd[2];
17     int fd2[2];
18     pid_t pid;
19     /*create the pipe*/
20     if(pipe(fd)==-1){
21         fprintf(stderr, "Pipe failed");
22         return 1;
23     }
24     if(pipe(fd2)==-1)
25     {
26         fprintf(stderr, "Pipe failed");
27     }
28     /*fork a child process*/
29     pid = fork();
30     if(pid<0)
31     { /*fork failed*/
32         fprintf(stderr, "Fork failed");
33         return 1;
34     }
35     if(pid>0){
36         close(fd[READ_END]);
37         write(fd[WRITE_END],write_msg,strlen(write_msg)+1);
38         close(fd[WRITE_END]);
39         wait(NULL);
40         close(fd2[WRITE_END]);
41         read(fd2[READ_END], read_msg, BUFFER_SIZE);
42         printf("parent receive:%s\n", read_msg);
43         close(fd2[READ_END]);
44     }
45     else
46     {
47         close(fd[WRITE_END]);
48         read(fd[READ_END], read_msg, BUFFER_SIZE);
49         printf("child recieve:%s\n", read_msg);
50         strcpy(read_msg, "i AM hERE");
```

```

51     close(fd[READ_END]);
52     close(fd2[READ_END]);
53     write(fd2[WRITE_END], read_msg, strlen(read_msg)+1);
54     close(fd2[WRITE_END]);
55     return 0;
56 }
57 return 0;
58 }

```

### 5.3 程序运行结果

输出: child recieve:I am Here

parent receive:i AM hERE

结果分析表明, 父子进程之间通过普通管道成功地实现了通信。

### 5.4 引申

在程序三中, 若是单纯地使用指针分配内存实现父子进程通信, 是否可行呢?

对程序三做出稍许修改, 得到以下程序, 用于测试猜想:

```

1  #include<sys/types.h>
2  #include<stdio.h>
3  #include<string.h>
4  #include<unistd.h>
5  #include<stdlib.h>
6
7  #define BUFFER_SIZE 25
8  #define READ_END 0
9  #define WRITE_END 1
10
11 int main()
12 {
13     char write_msg[BUFFER_SIZE] = "I am Here";
14     char* read_msg=(char*)malloc((sizeof(char))*BUFFER_SIZE);
15     strcpy(read_msg, "silly");
16     printf("%p:", read_msg);
17     printf("%s\n", read_msg);
18     int fd[2];
19     int fd2[2];
20     pid_t pid;
21     /*create the pipe*/
22     if(pipe(fd)==-1){

```



```
23     fprintf(stderr, "Pipe failed");
24     return 1;
25 }
26 if(pipe(fd2)==-1)
27 {
28     fprintf(stderr, "Pipe failed");
29 }
30 /*fork a child process*/
31 pid = fork();
32 if(pid<0)
33 { /*fork failed*/
34     fprintf(stderr, "Fork failed");
35     return 1;
36 }
37 if(pid>0){
38     close(fd[READ_END]);
39     write(fd[WRITE_END],write_msg,strlen(write_msg)+1);
40     close(fd[WRITE_END]);
41     wait(NULL);
42     fflush(stdout);
43     printf("%p:", read_msg);
44     printf("%s\n", read_msg);
45     fflush(stdout);
46     close(fd2[WRITE_END]);
47     read(fd2[READ_END], read_msg, BUFFER_SIZE);
48     printf("%p:parent receive:%s\n", read_msg,read_msg);
49     close(fd2[READ_END]);
50     printf("%s", write_msg);
51 }
52 else
53 {
54     close(fd[WRITE_END]);
55     read(fd[READ_END], read_msg, BUFFER_SIZE);
56     printf("%p:child recieve:%s\n", read_msg,read_msg);
57     strcpy(read_msg, "i AM HERE");
58     printf("%p:", read_msg);
59     printf("%s\n", read_msg);
60     close(fd[READ_END]);
61     close(fd2[READ_END]);
62     write(fd2[WRITE_END], read_msg, strlen(read_msg)+1);
63     fflush(stdout);
64     close(fd2[WRITE_END]);
65     return 0;
66 }
67 return 0;
68 }
```

上述程序为 `read_msg` 在堆区分配了内存,根据猜想,子进程修改了 `read_msg` 的值会立刻在父进程体现。猜想通过 58、59 行打印子进程中 `read_msg` 指向的地址中的值和打印 `read_msg` 指向的地址,43、44 行在父进程未从管道读取值前打印 `read_msg` 指向的地址和地址中存储的值验证。

#### 5.4.1 程序运行结果

```
输出 0x559c58ee52a0:silly
0x559c58ee52a0:child recieve:I am Here
0x559c58ee52a0:i AM hERE
0x559c58ee52a0:silly
0x559c58ee52a0:parent receive:i AM hERE
```

结果分析表明,子程序更改了 `read_msg` 的值为“i AM hERE”后,父进程的 `read_msg` 在未接收到管道值前的值仍未初始值“silly”。然而两个进程的 `read_msg` 指向的地址值都一样!

对此现象使用 ChatGPT 问答:

Q:”子进程中修改了这些内存中的数据,将会复制并分配新的内存空间来存储子进程的修改”我在我的程序里似乎看见了这句话的证据,但是我所打印出的父子进程的指针所指向的地址是一样的。这是为什么?

A:可能的情况是,尽管写时复制是在发生数据修改时才复制数据,但是这并不意味着每次都会生成新的物理内存页。实际上,写时复制的实现方式可能是将页标记为“只读”,并在进程试图写入这些页时产生一个页错误异常。在这种情况下,操作系统会为子进程分配新的物理页,并将数据从父进程的页复制到子进程的新页中,然后更新页表以反映这些更改。

这可能是你观察到父子进程指针指向相同地址的原因。尽管指针指向相同的虚拟地址,但实际上这些地址可能映射到不同的物理内存页。

请注意,具体的操作系统实现可能会有所不同。如果你想深入了解操作系统的内存管理和写时复制技术,请参考特定操作系统的文档或内核源代码。

Q:(char \*)mmap(0, SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);  
的指针是指向哪块内存区域的?为什么这个内存区域就能被父子进程共享?

A:这行代码使用了 `mmap` 系统调用,它是用于在进程的虚拟地址空间中创建一个新的内存映射区域的函数。这个内存映射区域可以与一个文件或者设备关联起来,也可以是匿名的。

在你的情况下,你使用了 `MAP_SHARED` 标志,这意味着映射区域会被共享。指向这块内存区域的指针可以在多个进程间共享,因为操作系统会为此映射区域维护一个页表,使得不同进程的虚拟地址都映射到同一块物理内存。这种共享内存区域的方法

使得多个进程可以访问并操作同一块内存，从而实现了进程间通信。

在父子进程间共享内存的情况下，`mmap` 函数创建的内存区域会被映射到父子进程的虚拟地址空间中相同的地址。因此，父子进程可以通过这个指针访问和修改同一块共享内存区域，实现了共享数据的目的。

需要注意的是，共享内存区域的并发访问需要通过同步机制来确保数据的一致性和正确性，避免出现竞争条件。

可以看见，两个进程所被分配到的内存为虚拟内存，两个进程的地址值虽然一样，但是并不指向相同的物理内存。因此需要管道或者共享内存等手段进行父子进程间的通信。