

# 操作系统实验报告

顾芄骐 20079100001

2023 年 4 月 11 日

## 1 进程的建立

- 实验目的：学会通过基本的 Windows 或者 Linux 进程控制函数，由父进程创建子进程，并实现父子进程协同工作。
- 实验软件环境：Ubuntu22.04 & GNU-C++17
- 实验内容：创建两个进程，让子进程读取一个文件，父进程等待子进程读取完文件后继续执行，实现进程协同工作。进程协同工作就是协调好两个进程，使之安排好先后次序并以此执行，可以用等待函数来实现这一点。当需要等待子进程运行结束时，可在父进程中调用等待函数。

### 1.1 代码实现

在“*os1.cpp*”中写如下代码，并创建名为“*os1.in*”的文件，在后者中写入且仅写入“TextMessage”作为测试输出内容。

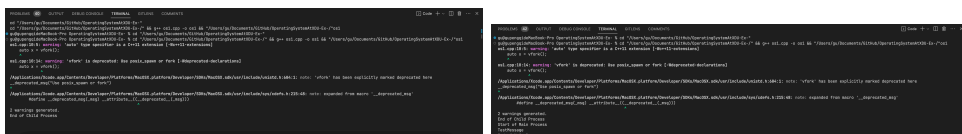
```

1 #include<fstream>
2 #include<unistd.h>
3 #include<stdio.h>
4 #include<iostream>
5 using namespace std;
6 int main(){
7     int flag = 0;
8     string answer = "";
9     auto x = vfork();
10    if (!x) {
11        ifstream in("os1.in");
12        in >> answer;
13        in.close();
14        flag = 1;
15        cout << "End of Child Process" << endl;
16    } else {
17        while (!flag);
18        cout << "Start of Main Process" << endl;
19        cout << answer << endl;
20        cout << "End of Main Process" << endl;
21    }
22    return EXIT_SUCCESS;
23 }

```

### 1.2 实验结果

运行上述代码，可得到如下左图所示的实验结果。片刻之后，终端显示如右所示的结果。



### 1.3 实验结果分析

在 1.1 所示代码中，子进程读取了文件“*os1.in*”；读取完成后，父进程继续执行，并显示了开始、结束和子进程所读取的文件内容。需要注意的是，使用 `vfork` 时修改静态区变量是未定义行为，会引发“栈溢出（stack smashing）”。

## 2 线程共享进程数据

- 实验目的：了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。
- 实验软件环境: Ubuntu22.04 & gnu-c++17
- 实验内容: 在进程中定义全局共享数据，在线程中直接引用该数据进行更改并输出该数据。

### 2.1 代码实现

在“*os2.cpp*”中写如下代码。

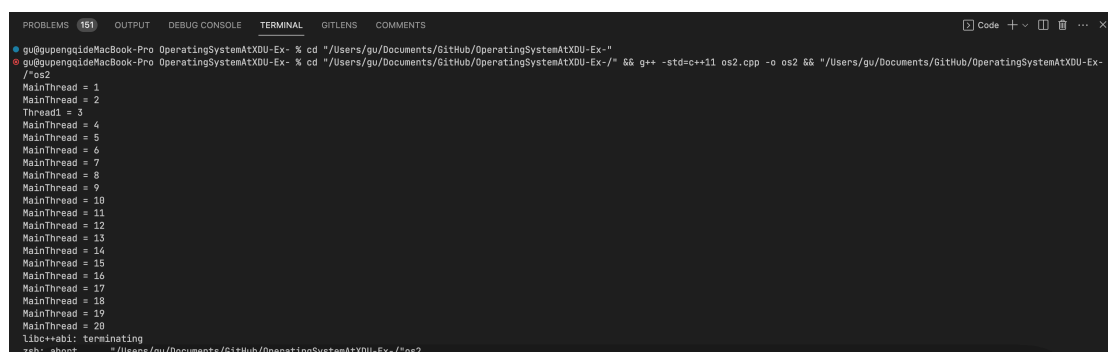
```
1 #include <thread>
#include <mutex>
#include <bits/stdc++.h>
#include <unistd.h>
5 using namespace std;
static int Sample = 1;
mutex mtx;
void thread1(int n){
9     while (Sample <= n)
    {
        if (mtx.try_lock()){
13             cout << "Thread1 " << Sample << "\n";
            sleep(1);
            Sample++;
            mtx.unlock();
17        }
    }
}
int main(){
21     int n = 20;
    thread t1(thread1, n);
    while (Sample <= n){
        if (mtx.try_lock()){
25             cout << "MainThread " << Sample << "\n";
            Sample++;
            mtx.unlock();
29        }
    }
    return 0;
}
```

### 2.2 实验结果

运行上述代码，可得到实验结果如下图所示。

### 2.3 实验结果分析

在此次实验结果中，当 `Sample` 值为 3 时，执行了 `thread1` 函数。线程中直接引用了这一数据进行更改并输出之。



```
PROBLEMS 151 OUTPUT DEBUG CONSOLE TERMINAL GITLENS COMMENTS
gu@guopengdaMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
gu@guopengdaMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"
os2
MainThread = 1
MainThread = 2
Thread1 = 3
MainThread = 4
MainThread = 5
MainThread = 6
MainThread = 7
MainThread = 8
MainThread = 9
MainThread = 10
MainThread = 11
MainThread = 12
MainThread = 13
MainThread = 14
MainThread = 15
MainThread = 16
MainThread = 17
MainThread = 18
MainThread = 19
MainThread = 20
libc++abi: terminating
^ah: abort "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os2
```

## 2.4 反思

本次实验代码中运行结果可能与“实验结果”章节中的不一致，有时会出现 `thread1` 函数不执行的情况，20 个输出结果均为“MainThread=x”。

## 3 信号通信

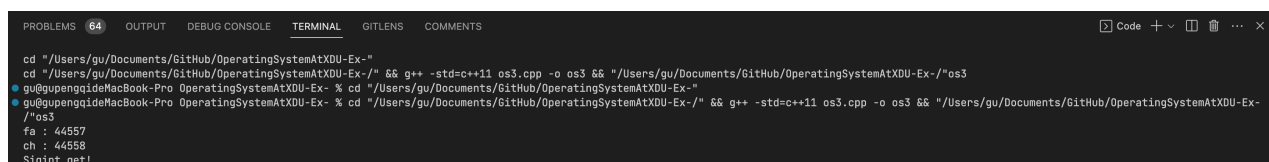
- 实验目的：利用信号通信机制在父子进程及兄弟进程间进行通信。
- 实验软件环境: Ubuntu22.04 & gnu-c++17
- 实验内容: 父进程创建一个有名事件，由子进程发送事件信号，父进程获取事件信号后进行相应的处理。

### 3.1 代码实现

```
1 #include<unistd.h>
2 #include<stdlib.h>
3 #include<signal.h>
4 #include<bits/stdc++.h>
5
6 using namespace std;
7
8 void sig_handle(int sig) {
9     cout << "Sigint get!\n";
10    signal(SIGINT, SIG_DFL);
11 }
12
13 int main() {
14     auto fa = getpid();
15     auto x = fork();
16     if( getpid() == fa ) {
17         cout << "fa : " << getpid() << "\n";
18         if(signal(SIGINT, sig_handle) == SIG_ERR) {
19             cout << "ERROR!\n";
20         }
21         sleep(100);
22     } else {
23         cout << "ch : " << getpid() << "\n";
24         kill( getppid(), SIGINT );
25     }
26 }
```

### 3.2 实验结果

运行上述代码，可得到如下图所示的实验结果。



```
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-" && g++ -std=c++11 os3.cpp -o os3 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os3
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/" && g++ -std=c++11 os3.cpp -o os3 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os3
fa : 44557
ch : 44558
Sigint get!
```

### 3.3 实验结果分析

父进程 `getpid` 创建了一个有名事件，由子进程 `ch` 完成后发送事件信号，父进程获取这一事件信号后进行相应的处理，反馈为“*Sigint get!*”。

每一次运行此代码得到的 **fa** 与 **ch** 值均不相同。例如，在上方实验结果之后重新运行此程序，得到“**fa** : 47122”“**ch** : 47123”和“*Sigint get!*”（下图）。然而所有的运行结果中，**fa** 与 **ch** 的值均满足规律

$$fa + 1 = ch \quad (1)$$

这是因为，代码中 **fa** 有关输出和 **ch** 有关输出总是在同一个条件语句中执行的。



```
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/" && g++ -std=c++11 os3.cpp -o os3 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os3
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/" && g++ -std=c++11 os3.cpp -o os3 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os3
fa : 44557
ch : 44558
Sigint get!
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/" && g++ -std=c++11 os3.cpp -o os3 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os3
fa : 47122
ch : 47123
Sigint get!
gu@gupengqideMacBook-Pro OperatingSystemAtXDU-Ex- % |
```

图 1: 重新运行此代码，得到的 **fa** 与 **ch** 数值并不相同

### 3.4 反思 · 心得

信号通信处理是一种很有效的异步处理方式；从此次实验结果来看运行速度很快。

## 4 匿名管道通信

- 实验目的：学习使用匿名管道在两个进程间建立通信。
- 实验软件环境：Ubuntu22.04 & gnu-c++17
- 实验内容：分别建立名为 Parent 的单文档应用程序和 Child 的单文档应用程序作为父子进程，由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

### 4.1 代码实现

```
#include<unistd.h>
#include<bits/stdc++.h>
using namespace std;

int main() {
    auto fa = getpid();
    int _pipeline[2];
    int ret = pipe(_pipeline);
    if(ret < 0) {
        cout << "Pipeline error!\n";
        return 0;
    }
    auto x = fork();
    if( x == 0 ) {
        close(_pipeline[1]);
        char s[100];
        memset(s, '\0', sizeof(s));
        read(_pipeline[0], s, sizeof(s));
        cout << "fa : " << getpid() << " reading " << s << "\n";
    }else {
        close(_pipeline[0]);
        char* msg = NULL;
        msg = "TestMessage";
        write(_pipeline[1], msg, strlen(msg));
        cout << "ch : " << getpid() << " Write success!\n";
    }
    return 0;
}
```

### 4.2 实验结果

运行上述代码，可得到如下图所示的实验结果。



```
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-"
cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/" && g++ -std=c++11 os4.cpp -o os4 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os4
gu@gupengqiMacBook-Pro OperatingSystemAtXDU-Ex- % cd "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-" && g++ -std=c++11 os4.cpp -o os4 && "/Users/gu/Documents/GitHub/OperatingSystemAtXDU-Ex-/"os4
In file included from os4.cpp:2:
/usr/local/include/bits/stdc++.h:31:17: warning: using directive refers to implicitly-defined namespace 'std'
using namespace std;
^
os4.cpp:28:15: warning: ISO C++11 does not allow conversion from string literal to 'char*' [-Wwriteable-strings]
    msg = "TestMessage";
    ^
2 warnings generated.
ch : 62801 Write success!
fa : 62802 reading TestMessage
gu@gupengqiMacBook-Pro OperatingSystemAtXDU-Ex- %
```

### 4.3 实验结果分析

父进程 `fa` 通过 `getpid` 指令构建了一个匿名管道，成功实现了利用数据“`TestMessage`”向此管道的写入（运行结果中的“*Write success*”）和读取（运行结果中的“*reading TestMessage*”）。与实验三相同，此实验中的运行结果各次之间 `fa` 与 `ch` 的值也不相同，但都满足下方等式：

$$fa - 1 = ch \quad (2)$$

### 4.4 反思·心得

匿名管道必须早于子进程建立，且只能用于有亲缘关系的进程间通信。