# 操作系统实验报告

顾芃骐 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。
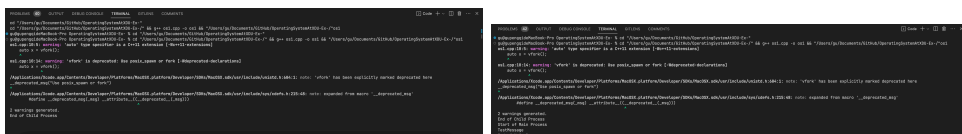
```cpp
#include<fstream>
#include<unistd.h>
#include<stdio.h>
#include<iostream>
using namespace std;
int main(){
    int flag = 0;
    string answer = "";
    auto x = vfork();
    if (!x) {
        ifstream in("os1.in");
        in >> answer;
        in.close();
        flag = 1;
        cout << "End of Child Process" << endl;
    } else {
        while (!flag);
        cout << "Start of Main Process" << endl;
        cout << answer << endl;
        cout << "End of Main Process" << endl;
    }
    return EXIT_SUCCESS;
}
```

## 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。

```cpp
#include <thread>
#include <mutex>
#include <bits/stdc++.h>
#include <unistd.h>
using namespace std;
static int Sample = 1;
mutex mutx;
void thread1(int n){
    while (Sample <= n)
    {
        if (mutx.try_lock())
        {
            cout << "Thread1 " << Sample << "\n";
            sleep(1);
            Sample++;
            mutx.unlock();
        }
    }
}
int main(){
    int n = 20;
    thread t1(thread1, n);
    while (Sample <= n){
        if (mutx.try_lock()){
            cout << "MainThread " << Sample << "\n";
            Sample++;
            mutx.unlock();
        }
    }
    return 0;
}
```
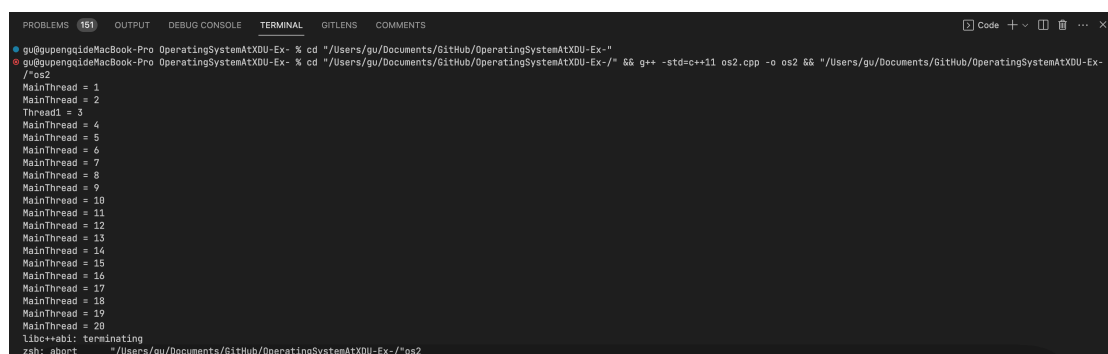
## 2.2　实验结果

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。

## 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"。
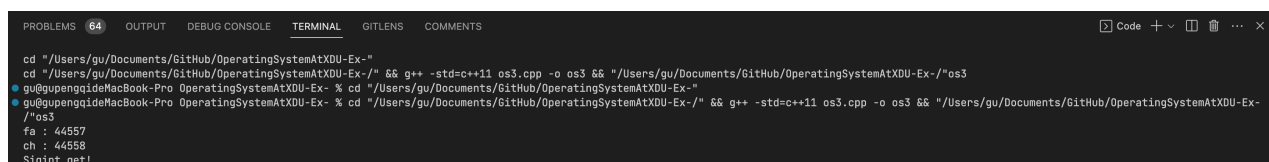
# 3 信号通信

- 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

```cpp
#include<unistd.h>
#include<stdlib.h>
#include<signal.h>
#include<bits/stdc++.h>

using namespace std;

void sig_handle(int sig) {
    cout << "Sigint get!\n";
    signal(SIGINT, SIG_DFL);
}

int main() {
    auto fa = getpid();
    auto x = fork();
    if( getpid() == fa ) {
        cout << "fa : " << getpid() << "\n";
        if(signal(SIGINT, sig_handle) == SIG_ERR) {
            cout << "ERROR!\n";
        }
        sleep(100);
    } else {
        cout << "ch : " << getpid() << "\n";
        kill( getppid(), SIGINT );
    }
}
```

## 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。



## 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

$$fa + 1 = ch \tag{1}$$

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。



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

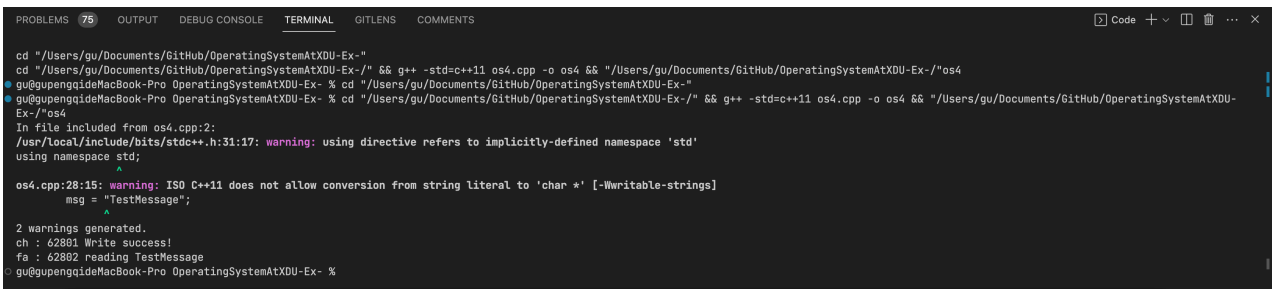- 实验内容：分别建立名为 Parent 的单文档应用程序和 Child 的单文档应用程序作为父子进程，由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

## 4.1 代码实现

```cpp
#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。

## 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：

$$fa - 1 = ch \tag{2}$$

## 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。

# 5  命名匿名管道通信

- 实验目的：学习使用命名匿名管道在多进程间建立通信。

- 实验软件环境：Ubuntu22.04 & gnu-c++17

- 实验内容：建立父子进程，由父进程创建一个命名匿名管道，由子进程向命名管道写入数据，由父进程从命名管道读取数据。

## 5.1  代码实现

```cpp
#include <unistd.h>
#include <sys/stat.h>
#include <cstring>
#include <iostream>
#include <fstream>

#define PATH "/tmp/tmp_fifo.tmp"

using namespace std;

int main()
{
    auto r = mkfifo(PATH, S_IFIFO | 0666);
    auto x = fork();
    if (x == 0)
    {
        ofstream out(PATH);
        for (int i = 1; i <= 10; i++)
        {
            out << i << "\n"
                << flush;
            cout << "Child " << getpid() << " write " << i << "\n";
            sleep(1);
        }
        out << "Fin.\n"
            << flush;
        cout << "Child ends!\n";
    }
    else
    {
        ifstream in(PATH);
        string s;
        while (getline(in, s) and s != "Fin.")
        {
            cout << "Father " << getpid() << " read " << s << "\n";
        }
        cout << "Father ends!\n";
    }
}
```
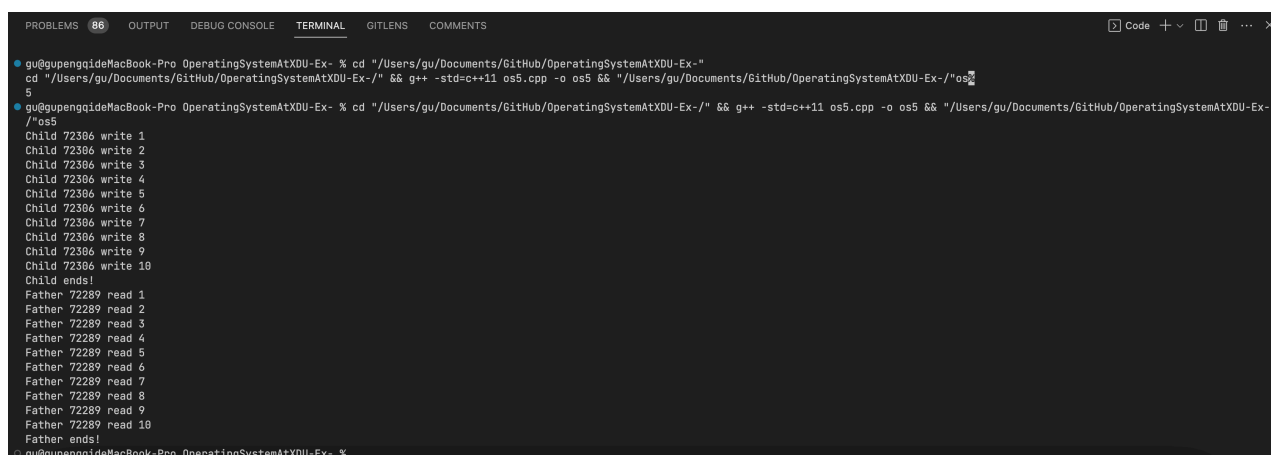
## 5.2  实验结果

运行上述代码，可得到如下图所示的实验结果。其中，子进程中的每一行（即输出"*Child ends!*"前的每一行）输出结果之间都间隔了一小段时间，父进程输出则一瞬间全部完成。

## 5.3   实验结果分析

　　父进程率先创建了命名匿名管道 PATH。子进程将测试数据 1 至 10 有序写入了该管道（因为 sleep 函数，连续两次写入完成 s 输出之间存在时间间隔），并在最终留下"*Fin.*"作为数据结束暗示。父进程读取这些数据并输出出来，在读取到"*Fin.*"后停止读取（输出"*Father ends!*"），此次命名匿名管道通信完成。

## 5.4   反思・心得

　　文件的本质是流，因此理论上也可以利用文件系统来维护多个进程之间的通信管道。

# 6 信号量实现进程同步

- 实验目的：进程同步是操作系统多进程/多线程并发执行的关键之一，进程同步是并发进程为了完成共同任务采用某个条件来协调他们的活动，这是进程之间发生的一种直接制约关系。本次试验是利用信号量进行进程同步。

- 实验软件环境：Ubuntu22.04 & gnu-c++17

- 实验内容：

    - 生产者进程生产产品，消费者进程消费产品。

    - 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区。

    - 当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来。

## 6.1 代码实现

本此实验采用生产者（os6_producer）和消费者（os6_consumer）两种代码，分别如下：

```cpp
#include <sys/ipc.h>
#include <sys/shm.h>
#include <bits/stdc++.h>
#include <semaphore.h>
using namespace std;
const int limit = 100;
int shmid;
char *p;
sem_t * sem_input;
sem_t * sem_output;

void init() {
    shmid = shmget((key_t)0x2333, 1024, 0666|IPC_CREAT);
    if( shmid == -1 ) cout << "Create shared memory failed.\n", exit(-1);
    p = (char *)shmat( shmid, 0, 0 );

    sem_input = (sem_t *)p;
    p += sizeof(sem_input);
    p += 32;

    sem_output = (sem_t *)p;
    p += sizeof(sem_output);
    p += 32;
}

int main() {
    init();
    srand( time(NULL) );
    while( true ) {
        sem_wait(sem_output);
        for(int i = 0; i < limit; i ++) if( p[i] != 0 )
          { cout << "Customer take a '" << p[i] << "' on " << i << ".\n";
            p[i] = 0;
            sem_post(sem_input);
            sleep( log(rand()) / 25.0 );
            break;
          }
    }
}
```

os6_consumer.cpp

```cpp
#include <sys/ipc.h>
#include <sys/shm.h>
#include <bits/stdc++.h>
#include <semaphore.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

const int limit = 100;
using namespace std;
int shmid;
char *p;

union samp
{
    sem_t sem;
    long long tmp;
};

samp *sem_input;
samp *sem_output;

void init()
{
    shmid = shmget((key_t)0x2333, 1024, 0666 | IPC_CREAT);
    if (shmid == -1)
        cout << "Create shared memory failed.\n", exit(-1);
    p = (char *)shmat(shmid, 0, 0);
    memset(p, 0, 1024);

    sem_input = new (p) samp;
    if (sem_init(&(sem_input->sem), 1, limit) < 0)
        cout << "Create sem_input failed\n", exit(-1);
    p += sizeof(sem_input);
    p += 32;
    cout << sem_input->tmp << "\n";

    sem_output = new (p) samp;
    if (sem_init(&(sem_output->sem), 1, 0) < 0)
        cout << "Create sem_output failed\n", exit(-1);
    p += sizeof(sem_output);
    p += 32;
    cout << sem_output->tmp << "\n";
}

int main()
{
    init();
    srand(time(NULL));
    while (true)
    {
        cout << sem_input->tmp << " " << sem_output->tmp << "\n";
        sem_wait(&(sem_input->sem));
        for (int i = 0; i < limit; i++)
            if (p[i] == 0)
            {
                p[i] = rand() % 26 + 'a';
                cout << "Producer make a '" << p[i] << "' on " << i << ".\n";
```

```
61          sem_post(&(sem_output->sem));
            sleep(log(rand()) / 25.0);
            break;
        }
65    }
}
```

os6__producer.cpp

## 6.2 实验结果

先运行 os6__producer.cpp 文件中的代码，之后运行 os6__consumer.cpp 中的代码。随后，得到实验结果如下图所示：



## 6.3 反思 · 心得

sizeof() 函数不能准确计算 `sem_t *` 的大小，所以需要预留足够多的空间。`new(p) ObjectType` 可以表示从 p 位置分配内存给 `ObjectType` 对象，是非常好的选择。