操作系统实验报告

实验 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。
- 代码实现

```cpp
#include<unistd.h>
#include<iostream>
#include<fstream>

int qwq = 0;
std::string awa = "";

int main() {
    auto x = vfork();
    if( x == 0 ) {
        std::ifstream in("os1.in");
        in >> awa;
        in.close();
        qwq = 1;
        std::cout << "child ended\n";

    } else {
        while( !qwq ) ;
        std::cout << "main proecess run\n";
        std::cout << awa << "\n";
        std::cout << "main proecess ended\n";
    }
    return EXIT_SUCCESS;
}
```

- 实验结果

```
终端
~/project/XDU-CS-Experiments/Operating System $ g++ os1.cpp -o os
~/project/XDU-CS-Experiments/Operating System $ ./os
child ended
main proecess run
TaT
main proecess ended
*** stack smashing detected ***: terminated
已放弃（核心已转储）
```

- 实验心得
  使用 `vfork` 时修改静态区变量是未定义行为，会导致 stack smashing。

实验 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。
- 代码实现

```cpp
#include<thread>
#include<mutex>
#include<bits/stdc++.h>

static int test = 1;
std::mutex mt;

void thread1(int n) {
    while(test <= n) {
        if( mt.try_lock() ) {
            std::cout << "Thread1 " << test << "\n";
            sleep(0.1); test ++;
            mt.unlock();
        }
    }
}

int main() {
    int n = 20;
    std::thread t1(thread1, n);
    while(test <= n) {
        if( mt.try_lock() ) {
            std::cout << "MainThread " << test << "\n";
            test ++;
            mt.unlock();
        }
    }
    return 0;
}
```

- 实验结果

```
~/project/XDU-CS-Experiments/Operating System $ g++ os2.cpp -o os
~/project/XDU-CS-Experiments/Operating System $ ./os
MainThread 1
MainThread 2
MainThread 3
Thread1 4
MainThread 5
Thread1 6
MainThread 7
Thread1 8
Thread1 9
MainThread 10
Thread1 11
Thread1 12
MainThread 13
Thread1 14
MainThread 15
Thread1 16
MainThread 17
Thread1 18
MainThread 19
Thread1 20
```

- 实验心得
  建议使用 `std::mutex` 而不是自己重复发明互斥量。

实验 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。
- 代码实现

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

void sig_handle(int sig) {
    std::cout << "get sigint\n";
    signal(SIGINT, SIG_DFL);
}

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

- 实验结果

```
~/project/XDU-CS-Experiments/Operating System $ g++ os3.cpp -o os
~/project/XDU-CS-Experiments/Operating System $ ./os
fa : 74662
ch : 74663
get sigint
```

- 实验心得
  感觉信号处理是一种很古老的异步方式。

实验 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。
- 代码实现

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


int main() {
    auto fa = getpid();

    int _pipe[2];
    int ret = pipe(_pipe);
    if(ret < 0) {
        std::cout << "pipe error\n";
        return 0;
    }

    auto x = fork();

    if( x == 0 ) {
        close(_pipe[1]);
        char s[100];
        memset(s, '\0', sizeof(s));
        read(_pipe[0], s, sizeof(s));
        std::cout << "fa : " << getpid() <<  " read " << s << "\n";
    } else {
        close(_pipe[0]);
        char* msg = NULL;
        msg = "awawawa";
        write(_pipe[1], msg, strlen(msg));
        std::cout << "ch : " << getpid() << " write successfully\n";
    }
    return 0;
}
```

- 实验结果

```
~/project/XDU-CS-Experiments/Operating System $ g++ os4.cpp -o os
os4.cpp: In function ‘int main()’:
os4.cpp:26:15: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   26 |         msg = "awawawa";
      |               ^~~~~~~~~
~/project/XDU-CS-Experiments/Operating System $ ./os
ch : 74874 write successfully
fa : 74875 read awawawa
```

- 实验心得
  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。

实验 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。
- 代码实现

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

#define PATH "/tmp/tmp_fifo.tmp"

int main() {
    auto r = mkfifo( PATH, S_IFIFO | 0666 );
    auto x = fork();
    if( x == 0 ) {
        std::ofstream out(PATH);
        for( int i = 1; i <= 10; i ++ ) {
            out << i << "\n" << std::flush;
            std::cout << "child " << getpid() << " write " << i << "\n";
            sleep(1);
        }
        out << "END\n" << std::flush;
        std::cout << "child ends\n";
    } else {
        std::ifstream in(PATH);
        std::string s;
        while( getline(in, s) and s != "END" ) {
            std::cout << "father " << getpid() << " read " << s << "\n";
        }
        std::cout << "father ends\n";
    }
}
```

- 实验结果

```
~/project/XDU-CS-Experiments/Operating System $ g++ os5.cpp -o os
~/project/XDU-CS-Experiments/Operating System $ ./os
child 74929 write 1
father 74928 read 1
child 74929 write 2
father 74928 read 2
child 74929 write 3
father 74928 read 3
child 74929 write 4
father 74928 read 4
child 74929 write 5
father 74928 read 5
child 74929 write 6
father 74928 read 6
child 74929 write 7
father 74928 read 7
child 74929 write 8
father 74928 read 8
child 74929 write 9
father 74928 read 9
child 74929 write 10
father 74928 read 10
child ends
father ends
```

- 实验心得
  文件的本质是流，所以也可以用文件系统维护进程间的管道。

实验 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:

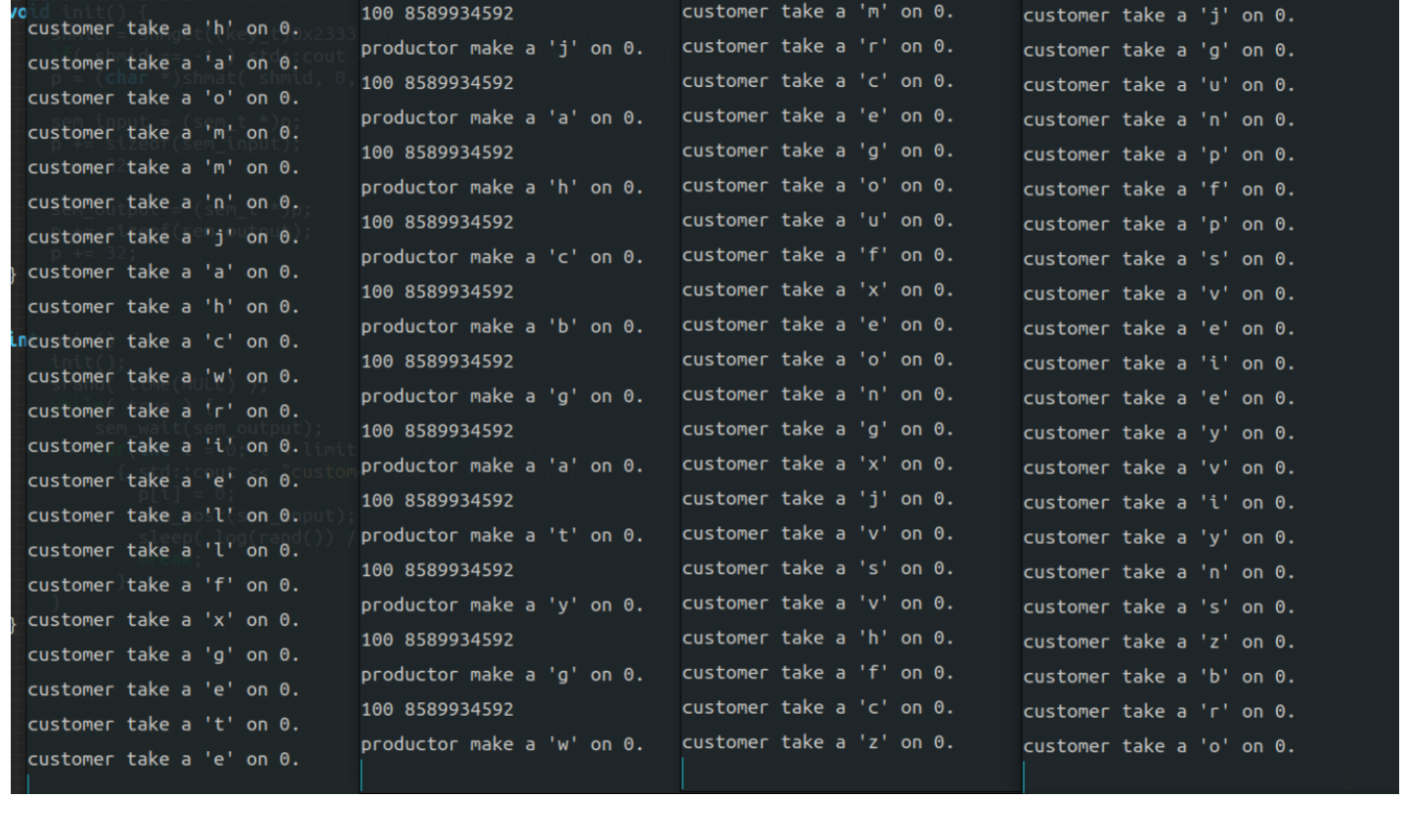
  - 生产者进程生产产品，消费者进程消费产品。
  - 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区。
  - 当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来。

- 代码实现

```cpp
/*productor.cpp*/
  #include <sys/ipc.h>
  #include <sys/shm.h>
  #include <bits/stdc++.h>
  #include <semaphore.h>

  const int limit = 100;

  int shmid;
  char *p;

  union qwq {
      sem_t sem;
      long long awa;
  };

  qwq * sem_input;
  qwq * sem_output;

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

      sem_input = new(p) qwq;
      if( sem_init( &(sem_input -> sem), 1, limit ) < 0 ) std::cout << "create sem_input
      p += sizeof(sem_input);
      p += 32;
      std::cout << sem_input -> awa << "\n";

      sem_output = new(p) qwq;
      if( sem_init( &(sem_output -> sem), 1, 0 ) < 0 ) std::cout << "create sem_output f
      p += sizeof(sem_output);
      p += 32;
      std::cout << sem_output -> awa << "\n";
  }

  int main() {
      init();
      srand( time(NULL) );
      while( true ) {
          std::cout << sem_input -> awa << " " << sem_output -> awa << "\n";
          sem_wait(&(sem_input -> sem));
          for(int i = 0; i < limit; i ++) if( p[i] == 0 )
            { p[i] = rand() % 26 + 'a';
              std::cout << "productor make a '" << p[i] << "' on " << i << ".\n";
              sem_post(&(sem_output -> sem));
              sleep( log(rand()) / 25.0 );
              break;
            }
      }
  }
```

```cpp
/*customer.cpp*/
  #include <sys/ipc.h>
  #include <sys/shm.h>
  #include <bits/stdc++.h>
  #include <semaphore.h>

  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 ) std::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 )
            { std::cout << "customer take a '" << p[i] << "' on " << i << ".\n";
              p[i] = 0;
              sem_post(sem_input);
              sleep( log(rand()) / 25.0 );
              break;
            }
      }
  }
```

- 实验结果



- 实验心得
  `sizeof()` 不能准确的计算 `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。