操作系统
实验报告

学　　号_____19071125_____

姓　　名_____卢雨轩_____

提交日期_____2021.12.31_____

报告格式(5)	报告内容(15)	教师评语	报告成绩
要求：项目完整、图表符合规范、没有错别字、符合模板规范。(满分5分；实际得分_____)	□图表内容符合计算机学科标准(满分4分；实际得分_____) □可否体会设计思路实现方案/设计思路实现方案(满分10分；实际得分_____) □是否提交了电子版的代码(满分10分；实际得分_____)		

教师签字:_____

目录

实验二——进程创建与管道通信

一、 实验目的

1、 加深对进程概念的理解，明确进程与程序的区别；
2、 进一步认识并发执行的实质；
3、 学习在 Linux 操作系统中父子进程之间进行管道通信的方法。

二、 实验内容

1、 学习使用 fork() 系统调用创建子进程，体会父子进程之间的并发关系。

请编写一段程序，使用系统调用 fork() 创建两个子进程，实现当此程序运行时，在系统中有一个父进程和两个子进程在活动。父进程的功能是输出一个字符“a”；两个子进程的功能是分别输出一个个字符“b”和一个字符“c”。

多次运行这个程序，试观察记录屏幕上的显示结果，并分析原因。

另外，为了更好地展示进程之间的父子关系，大家可以使用 getpid() 系统调用来获取当前进程的 PID，并用 getppid() 用于获取当前进程的父进程的 PID。

2、 继续体会进程之间的并发关系

修改刚才的程序，将每一个进程输出一个字符改为用一个循环输出 1000 个字符（父进程输出 1000 个“a”，子进程分别输出 1000 个“b”和“c”），再观察程序执行时屏幕上出现的现象，并分析原因。

3、 进程的管道通信

编写程序实现进程的管道通信。父进程使用系统调用 pipe() 创建一个无名管道，二个子进程分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示出来。

补充材料中给出了管道通信实现过程中需要使用的系统调用的说明，请仔细阅读。

三、 实验设计

1、 学习使用 fork() 系统调用创建子进程

实验设计：在父进程中创建子进程，打印 pid 与 ppid 后退出。父进程通过两次 wait 调用等待两个子进程都退出后退出。

实验代码：

```
#include <cerrno>
#include <iostream>
```

```
#include <unistd.h>
#include <sys/wait.h>
using namespace std;
int main() {
  pid_t c1 = fork();
  if (c1 == 0) {
    cout << "This is child " << getpid();
    cout << "!" << endl;
    cout << "Parent:" << getppid() << endl;
    cout << "b" << endl;
    exit(0);
  }
  if (c1 <= 0) {
    cerr << "ERROR !" << c1 << endl;
    exit(-1);
  }

  pid_t c2 = fork();
  if (c2 == 0) {
    cout << "This is child " << getpid();
    cout << "!" << endl;
    cout << "Parent:" << getppid() << endl;
    cout << "c" << endl;
    exit(0);
  }
  if (c2 <= 0) {
    cerr << "ERROR !" << c2 << endl;
    exit(-1);
  }
  cout << "a" << endl;
  wait(nullptr);
  wait(nullptr);
}
```

2、 继续体会进程之间的并发关系

实验设计：在父进程中创建子进程，打印字符后退出。父进程通过两次 wait 调用等待两个子进程都退出后退出。

同时，为了试运行结果更为明显，在打印每个字母后均刷新缓冲区。

```
#include <cerrno>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
```

```
using namespace std;
int main() {
  pid_t c1 = fork();
  if (c1 == 0) {
    cout << "This is child " << getpid();
    cout << "!" << endl;
    cout << "Parent:" << getppid() << endl;
    for (int i = 0; i < 1000; ++i) {
      cout << "b";
      cout.flush();
    }
    exit(0);
  }
  if (c1 <= 0) {
    cerr << "ERROR !" << c1 << endl;
    exit(-1);
  }

  pid_t c2 = fork();
  if (c2 == 0) {
    cout << "This is child " << getpid();
    cout << "!" << endl;
    cout << "Parent:" << getppid() << endl;
    for (int i = 0; i < 1000; ++i) {
      cout << "c";
      cout.flush();
    }
    exit(0);
  }
  if (c2 <= 0) {
    cerr << "ERROR !" << c2 << endl;
    exit(-1);
  }
  for (int i = 0; i < 1000; ++i) {
    cout << "a";
    cout.flush();
  }
  wait(nullptr);
  wait(nullptr);
}
```

3、 进程的管道通信

首先，父进程通过 pipe 调用开启管道，并通过 `fdopen()` 函数将文件描述符转换为 c 语言的`FILE*`，以方便后续通过 `fwrite` 写入。

然后，开启子进程，并写入管道通讯。父进程通过 wait 系统调用等待子进程退出后，打印管道中读取的结果，关闭管道并退出。

```
#include <cerrno>
#include <fstream>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;
int main() {
  int fd[2];
  if (pipe(fd) == -1) {
    cerr << "ERROR";
    exit(-1);
  }
  FILE *wr, *re;
  wr = fdopen(fd[1], "w");
  re = fdopen(fd[0], "r");

  pid_t c1 = fork();
  if (c1 == 0) {
    close(fd[0]);
    cout << "This is child1 " << getpid();
    cout << "!" << endl;
    cout << "Parent:" << getppid() << endl;
    cout << "b" << endl;
//    fprintf(wr, "Child 1 is sending a message! \n");
        write(fd[1], "Child 1 is sending a message! \n", 31);
    cout << "b exit" << endl;
    exit(0);
  }
  if (c1 <= 0) {
    cerr << "ERROR !" << c1 << endl;
    exit(-1);
  }
  pid_t c2 = fork();
  if (c2 == 0) {
    close(fd[0]);
    cout << "This is child2 " << getpid();
    cout << "!" << endl;
```

```
    cout << "Parent:" << getppid() << endl;
    cout << "c" << endl;
    fprintf(wr, "Child 2 is sending a message! \n");
    // write(fd[1], "Child 2 is sending a message! \n", 31);
    cout << "c exit" << endl;
    exit(0);
  }

  close(fd[1]);

  if (c2 <= 0) {
    cerr << "ERROR !" << c2 << endl;
    exit(-1);
  }

  cout << "a" << endl;
  waitpid(c1, nullptr, 0);
  waitpid(c2, nullptr, 0);
  char *buf = new char[1000];
  while (fgets(buf, 1000, re)) {
    cout << buf;
  }
  close(fd[0]);
}
```

四、 实验结果与分析

1、 学习使用 fork() 系统调用创建子进程

运行结果：

```
a
This is child 21448!
Parent:21447
b
This is child 21449!
Parent:21447
c
```

可以看到，成功创建子进程。两个子进程的 pid 连续（因为创建时间几乎同时），并有同一个父进程 id。

2、 继续体会进程之间的并发关系

运行结果（省略了部分过长的输出）：

```
aaaaaaaaThis is child 21574!
aaaaaaaaaaaParent:21573
abababababababababababababababababababababababababababababab....
This is child 21575!
abababababababaParent:21573
bacabcabcabcabacbacbacbacbacbacbacbacbacbacbacbacbacbacbacbacbc....
```

可以看到，进程的相对执行速度没有任何关系。所以，在编写程序的时候，**不应该对进程之间的相对执行速度做任何假设**。

3、 进程的管道通信

运行结果：

```
a
This is child1 22232!
Parent:22231
b
b exit
This is child2 22233!
Parent:22231
c
c exit
Child 1 is sending a message!
Child 2 is sending a message!
```

可以看到，子进程打印 pid 和 ppid 后退出，父进程成功接受到子进程消息后退出。

五、 思考与总结

本次实验中，最大的收获就是证实了**不能对进程之间的相对执行速度做任何假设**。在以后任何涉及同步性问题的程序设计中都一定要考虑这一点。

实验三——线程管理

一、 实验目的

1、 编写 Linux 环境下的多线程程序，了解多线程的程序设计方法，掌握最常用的三个函数 pthread_create，pthread_join 和 pthread_exit 的用法；
2、 掌握向线程传递参数的方法。

二、 实验内容

1、 创建线程。
在这个任务中，需要在主程序中创建两个线程 myThread1 和 myThread2，每个线程打印一句话。提示：先定义每个线程的执行体，然后在主函数中使用 pthread_create 负责创建两个线程。整个程序等待子线程结束后再退出。
2、 向线程传递参数。
在上一个程序的基础上，分别向两个线程传递一个字符和一个整数，并让线程负责将两个参数的值打印出来。
3、【选作题】使用两个线程实现数组排序。
主程序中用数组 data[10] 保存 10 个整数型数据，创建两个线程，一个线程将这个数组中的数据从大到小排列输出；另一个线程求出数组中所有数据的和。

三、 实验设计

1、 创建线程、向进程传递参数

实验设计：
创建线程，传入线程参数。在线程内部将参数类型从`void*`转换回 `thread_arg*`后输出。
主进程创建完线程后等待线程退出。
实验代码：

```
#include "pthread.h"
#include <cerrno>
#include <iostream>
#include <sys/wait.h>

struct thread_arg {
  int num;
  char ch;
};

void *thread1(void *args) {
```

```
  auto argv = reinterpret_cast<thread_arg *>(args);
  std::cout << "Hello world from thread1!" << std::endl;
  std::cout << "Thread 1 arguments: " << argv->num << " " << argv->ch
            << std::endl;
  return nullptr;
}

void *thread2(void *args) {
  auto argv = reinterpret_cast<thread_arg *>(args);
  std::cout << "Hello world from thread2!" << std::endl;
  std::cout << "Thread 2 arguments: " << argv->num << " " << argv->ch
            << std::endl;
  return nullptr;
}

int main() {
  std::cout << "Main thread booted!" << std::endl;
  pthread_t t1, t2;
  thread_arg arg1{1, 'a'}, arg2{2, 'b'};
  pthread_create(&t1, nullptr, thread1, &arg1);
  pthread_create(&t2, nullptr, thread2, &arg2);
  pthread_join(t1, nullptr);
  pthread_join(t2, nullptr);
  std::cout << "Other thread exited, main thread quitting!" << std::endl;
  return 0;
}
```

2、 选做题——使用两个线程实现数组排序

首先，为了方便后续开发，定义调试宏：

```
#define F(arg, error_value, fail)                                    \
  ({                                                                 \
    auto ret = arg;                                                  \
    if (ret == error_value) {                                        \
      std::cout << #arg << std::strerror(errno) << std::endl;        \
      fail;                                                          \
    }                                                                \
    ret;                                                             \
  })
```

这个宏定义了一个 expression statement，即可以在表达式中执行的语句。首先运行被调用的函数。如果返回值是给定的错误值，则调用给定的语句报错，否则返回返回值。

如，调用 `sorted = F(sem_open("sorted",O_CREAT | O_EXCL, 0, 0) , SEM_FAILED, return -1);` 即调用 `sem_open`，如果返回值是 `SEM_FAILED` 就执行`return -1`，否则将返

回值放入 sorted 中。

其余设计：主进程首先开启一个信号量，并开启两个线程，讲数据指针和信号量作为参数传入。排序线程首先进行排序，然后对信号量做 sem_post。求和线程先等待信号量（sem_wait），然后求和后返回结果。主进程接受求和线程返回的值，打印后退出。

实验代码：

```
#include "pthread.h"
#include <cerrno>
#include <iostream>
#include <sys/wait.h>
#include <semaphore.h>
#define F(arg, error_value, fail)                                    \
  ({                                                                 \
    auto ret = arg;                                                  \
    if (ret == error_value) {                                        \
      std::cout << #arg << std::strerror(errno) << std::endl;        \
      fail;                                                          \
    }                                                                \
    ret;                                                             \
  })
sem_t *sorted;
struct thread_arg {
  int data[10];
};

void *sort(void *args) {
  auto argv = reinterpret_cast<thread_arg *>(args);
  bool flag = false;
  int sorted_count = 0;
  while (!flag) {
    flag = true;
    for (int i = 1; i < 10 - sorted_count; ++i) {
      if (argv->data[i - 1] > argv->data[i]) {
        std::swap(argv->data[i - 1], argv->data[i]);
        flag = false;
      }
    }
    sorted_count++;
  }
  F(sem_post(sorted), -1, return nullptr);
  return nullptr;
}
```

```
void *sum(void *args) {
  sem_wait(sorted);
  std::cout<<"Got sorted signal!"<<std::endl;
  auto argv = reinterpret_cast<thread_arg *>(args);
  auto ret = new int;
  int &sum = *ret;
  for (int i : argv->data) {
    sum += i;
  }
  return ret;
}

int main() {
  sem_unlink("sorted");
  sorted = F(sem_open("sorted",O_CREAT | O_EXCL, 0, 0) , SEM_FAILED, return -1);
  std::cout << "Main thread booted!" << std::endl;
  pthread_t t1, t2;
  thread_arg arg{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  pthread_create(&t1, nullptr, sort, &arg);
  pthread_create(&t2, nullptr, sum, &arg);
  int *ret;
  pthread_join(t1, nullptr);
  pthread_join(t2, reinterpret_cast<void **>(&ret));
  std::cout << "Sum: " << *ret << std::endl;
  for (auto i : arg.data) {
    std::cout << i << ' ';
  }
  std::cout << std::endl;
  std::cout << "Other thread exited, main thread quitting!" << std::endl;
  return 0;
}
```

四、 实验结果与分析

1、 创建进程与传递参数

运行结果:

```
Main thread booted!
Hello world from thread1!
Thread 1 arguments: 1 a
Hello world from thread2!
Thread 2 arguments: 2 b
Other thread exited, main thread quitting!
```

可以看到，两个进程分别打印了参数的值，主进程等待两个子进程退出后退出。

2、 使用两个进程进行排序和求和

```
Main thread booted!
Got sorted signal!
Sum: 55
1 2 3 4 5 6 7 8 9 10
Other thread exited, main thread quitting!
```

可以看到，求值正确，且排序结果正确，没有出现数据竞争。

五、 思考与总结

线程可以共享进程的地址空间，在很大程度上可以加快程序的运行。但是，也正因为共享地址空间，进程稍微使用不当就会导致同步性问题。

rust 是一个在编译时进行内存安全检查，实现内存安全的编程语言。通过严格的作用域、所有权、生命周期控制，可以保证程序只要能通过编译就不存在数据竞争。如，静态（static）作用域的变量必须是常量。

在编写程序的过程中，解决同步性问题，可以通过信号量与锁解决，也可以采用 rust 的 RAII 解决。

实验四——利用信号量实现线程互斥与同步

一、 实验目的

1、 学习 UNIX 类（System V）操作系统信号量机制；
2、 编写 Linux 环境下利用信号量实现进程控制的方法，掌握相关系统调用的使用方法。

二、 实验内容

1、 生产者消费者问题

有数据文件 1.dat 和 2.dat 分别存放了 10 个整数。创建 4 个线程，其中两个线程 read1 和 read2 负责分别从文件 1.dat 和 2.dat 中读取一个整数到公共的缓冲区，另两个线程 operate1 和 operate2 分别从缓冲区读取数据作加运算和乘运算。使用信号量控制这些线程的执行，保证缓冲区中的数据只能被计算一次（加或者乘），计算完成之后才能继续进行数据的读取工作。

2、 严格限制的生产者消费者问题

如果严格限制加和乘的两个操作数必须分别来自 1.dat 和 2.dat，且加法和乘法要严格交叉工作，应该如何修改上面的程序?

三、 实验设计

1、 生产者消费者问题

开启一个长度为 2 的环形缓冲区，用 `buf_ready` 信号量维护缓冲区中的数的数量，用 `buf_avail` 信号量维护缓冲区中剩余空间的数量，用 `buf_lock` 信号量维护缓冲区的操作的原子性。

生产者在获取到 `buf_avail` 信号后，获取 `buf_lock` 信号，写入缓冲区，并提供 `buf_ready` 信号。

消费者在获取到 `buf_ready` 信号后，获取 `buf_lock` 信号，读出缓冲区，并提供 `buf_avail` 信号。

通过 `worker` 信号量保证同时只有一个消费者在执行，不会产生死锁。

实验代码：

```
#include <atomic>
#include <cerrno>
#include <cstring>
#include <fcntl.h>
#include <fstream>
```

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <string>
#include <sys/wait.h>
#define F(arg, error_value, fail)                                   \
  ({                                                                \
    auto ret = arg;                                                 \
    if (ret == error_value) {                                       \
      std::cout << #arg << std::strerror(errno) << std::endl;       \
      fail;                                                         \
    }                                                               \
    ret;                                                            \
  })

int buf[2];
int buf_tail = 0;
int buf_head = 0;
int buf_size = 0;
sem_t *buf_lock;
sem_t *buf_avail;
sem_t *buf_ready;
sem_t *worker;
//sem_t *worker[2];
std::atomic_int processed_count;
pthread_t reader1, reader2, operate1, operate2;
struct reader_args {
  std::string filename;
};

void *read(void *args) {
  auto argv = reinterpret_cast<reader_args *>(args);
  std::ifstream fin(argv->filename);
  int data;
  while (fin >> data) {
    F(sem_wait(buf_avail), -1, return nullptr);
    F(sem_wait(buf_lock), -1, return nullptr);
//    std::cout << "Got" << data << std::endl;
    buf[buf_tail] = data;
    buf_tail = (buf_tail + 1) % 2;
    buf_size ++;
    F(sem_post(buf_lock), -1, return nullptr);
```

```
    F(sem_post(buf_ready), -1, return nullptr);
  }
  return nullptr;
}

enum class op { ADD, MUL };

struct operator_args {
  op ope;
  sem_t *worker_sem;
  sem_t *another_worker_sem;
};

void *operate(void *args) {
  auto argv = reinterpret_cast<operator_args *>(args);
  int bbuf[2];
  while (true) {
    F(sem_wait(worker), -1, return nullptr);
    if (processed_count >= 10) {
      F(sem_post(worker), -1, return nullptr);
      return nullptr;
    }
    F(sem_wait(buf_ready), -1, return nullptr);
    F(sem_wait(buf_lock), -1, return nullptr);
    bbuf[0] = buf[ (buf_head + 1) % 2 ];
    buf_head += 1;
    buf_head %= 2;
    F(sem_post(buf_lock), -1, return nullptr);
    F(sem_post(buf_avail), -1, return nullptr);
    F(sem_wait(buf_ready), -1, return nullptr);
    F(sem_wait(buf_lock), -1, return nullptr);
    bbuf[1] = buf[ (buf_head + 1) % 2 ];
    buf_head += 1;
    buf_head %= 2;
    F(sem_post(buf_lock), -1, return nullptr);
    F(sem_post(buf_avail), -1, return nullptr);
    if (argv->ope == op::MUL) {
      std::cout << bbuf[0] << "*" << bbuf[1] << "=" << bbuf[0] * bbuf[1]
                << std::endl;
    } else {
      std::cout << bbuf[0] << "+" << bbuf[1] << "=" << bbuf[0] + bbuf[1]
                << std::endl;
```

```
    }
    if (++processed_count >= 10) {
      F(sem_post(worker), -1, return nullptr);
      return nullptr;
    }
    F(sem_post(worker), -1, return nullptr);
  }
  return nullptr;
}

int main() {
  std::cout << "Main thread booted!" << std::endl;
  sem_unlink("worker");
  sem_unlink("avail");
  sem_unlink("ready");
  sem_unlink("buf");
  worker = F(sem_open("worker",O_CREAT | O_EXCL, 0, 1) , SEM_FAILED, return -1);
  buf_avail = F(sem_open("avail",O_CREAT | O_EXCL, 0, 2) , SEM_FAILED, return -1);
  buf_ready = F(sem_open("ready",O_CREAT | O_EXCL, 0, 0) , SEM_FAILED, return -1);
  buf_lock = F(sem_open("buf",O_CREAT | O_EXCL, 0, 1) , SEM_FAILED, return -1);
  reader_args r1{
      "1.dat",
  };
  reader_args r2{
      "2.dat",
  };
  operator_args o1{op::MUL},
      o2{op::ADD};
  pthread_create(&reader1, nullptr, read, &r1);
  pthread_create(&reader2, nullptr, read, &r2);
  pthread_create(&operate1, nullptr, operate, &o1);
  pthread_create(&operate2, nullptr, operate, &o2);
  pthread_detach(reader1);
  pthread_detach(reader2);
  pthread_join(operate1, nullptr);
  pthread_join(operate2, nullptr);
  return 0;
}
```

2、 严格限制的生产者消费者问题

给每个生产者分配唯一的缓冲区，并让两个消费者互相唤醒对方，来保证每次使用的两个操作数分别来自两个文件且加法乘法严格交替工作。

实验代码：

```
#include <atomic>
#include <cerrno>
#include <cstring>
#include <fcntl.h>
#include <fstream>
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <string>
#include <sys/wait.h>
#define F(arg, error_value, fail)                                   \
  ({                                                                \
    auto ret = arg;                                                 \
    if (ret == error_value) {                                       \
      std::cout << #arg << std::strerror(errno) << std::endl;       \
      fail;                                                         \
    }                                                               \
    ret;                                                            \
  })
int buf[2];
sem_t *avail[2];
sem_t *ready[2];
sem_t *worker[2];
std::atomic_int processed_count;
pthread_t reader1, reader2, operate1, operate2;
struct reader_args {
  std::string filename;
  int *target;
  sem_t *target_avail; // available for writing
  sem_t *target_ready; // ready for operating
};

void *read(void *args) {
  auto argv = reinterpret_cast<reader_args *>(args);
  std::ifstream fin(argv->filename);
  int data;
  while (fin >> data) {
    F(sem_wait(argv->target_avail), -1, return nullptr);
    *argv->target = data;
    F(sem_post(argv->target_ready), -1, return nullptr);
  }
```

```
  return nullptr;
}

enum class op { ADD, MUL };

struct operator_args {
  op ope;
  sem_t *worker_sem;
  sem_t *another_worker_sem;
};

void *operate(void *args) {
  auto argv = reinterpret_cast<operator_args *>(args);
  while (true) {
    F(sem_wait(argv->worker_sem), -1, return nullptr);
    F(sem_wait(ready[0]), -1, return nullptr);
    F(sem_wait(ready[1]), -1, return nullptr);
    if (argv->ope == op::MUL) {
      std::cout << buf[0] << "*" << buf[1] << "=" << buf[0] * buf[1]
                << std::endl;
    } else {
      std::cout << buf[0] << "+" << buf[1] << "=" << buf[0] + buf[1]
                << std::endl;
    }
    F(sem_post(avail[0]), -1, return nullptr);
    F(sem_post(avail[1]), -1, return nullptr);
    F(sem_post(argv->another_worker_sem), -1, return nullptr);
    if (++processed_count >= 9) {
      return nullptr;
    }
  }
  return nullptr;
}

int main() {
  std::cout << "Main thread booted!" << std::endl;
  sem_unlink("avail_0");
  sem_unlink("avail_1");
  sem_unlink("ready_0");
  sem_unlink("ready_1");
  sem_unlink("worker_0");
  sem_unlink("worker_1");
```

```
    avail[0] =
        F(sem_open("avail_0", O_CREAT | O_EXCL, 0, 1), SEM_FAILED, return -1);
    avail[1] =
        F(sem_open("avail_1", O_CREAT | O_EXCL, 0, 1), SEM_FAILED, return -1);
    ready[0] =
        F(sem_open("ready_0", O_CREAT | O_EXCL, 0, 0), SEM_FAILED, return -1);
    ready[1] =
        F(sem_open("ready_1", O_CREAT | O_EXCL, 0, 0), SEM_FAILED, return -1);
    worker[0] =
        F(sem_open("worker_0", O_CREAT | O_EXCL, 0, 1), SEM_FAILED, return -1);
    worker[1] =
        F(sem_open("worker_1", O_CREAT | O_EXCL, 0, 0), SEM_FAILED, return -1);
    reader_args r1{
        "1.dat",
        &buf[0],
        avail[0],
        ready[0],
    };
    reader_args r2{
        "2.dat",
        &buf[1],
        avail[1],
        ready[1],
    };
    operator_args o1{op::MUL, worker[0], worker[1]},
        o2{op::ADD, worker[1], worker[0]};
    pthread_create(&reader1, nullptr, read, &r1);
    pthread_create(&reader2, nullptr, read, &r2);
    pthread_create(&operate1, nullptr, operate, &o1);
    pthread_create(&operate2, nullptr, operate, &o2);
//  pthread_detach(reader1);
//  pthread_detach(reader2);
    pthread_join(operate1, nullptr);
    pthread_join(operate2, nullptr);
    return 0;
}
```

四、 实验结果与分析

1、 生产者消费者问题

两个输入文件分别为 10-1 和 1-10。

```
    Main thread booted!
9*10=90
1+8=9
2*7=14
2+5=7
4*5=20
5+4=9
6*3=18
7+2=9
8*1=8
10+9=19
```

可以看到，每个数字均只出现一次，并且执行速度随机。

2、 严格限制的生产者消费者问题

```
Main thread booted!
1*10=10
2+9=11
3*8=24
4+7=11
5*6=30
6+5=11
7*4=28
8+3=11
9*2=18
10+1=11
```

可以看到，每次运算的两个操作数均来自两个不同的文件，并乘法和加法严格交替工作。

五、 思考与总结

所有同步性问题的解决方案都是用小的原子性操作保护大的操作，使得其也拥有原子性。操作系统通过硬件提供的原子赋值指令或关中断等方式提供信号量和锁等原子性操作，而使用信号量使得环形缓冲区的 push 和 pop 操作也具有原子性。

实验五——基于消息队列和共享内存的进程间通信

一、 实验目的

1、 了解和熟悉 Linux 支持的消息通信机制及其使用方法
2、 解和熟悉 Linux 系统的共享存储区的原理及使用方法。

二、 实验内容

1、 消息的创建、发送和接收

主函数中使用 fork() 系统调用创建两个子进程 sender 和 receiver。sender 负责接收用户输入的一个字符串，并构造成一条消息，传送给 receiver。Receiver 在接收到消息后在屏幕上显示出来。此过程一直继续直到用户输入 exit 为止。在程序设计过程中使用 msgget()、msgsnd()、msgrcv()、msgctl()。

2、 共享存储取得创建、附接和断接

主函数中使用 fork() 系统调用创建两个子进程 sender 和 receiver。Sender 创建共享内存区域并从用户输入得到一个整数放入共享内存区域，receiver 负责取出此数并将此整数的平方计算出来。要求程序可以计算 10 个整数的平方值。在程序设计过程中使用 shmget()、shmat()、shmctl()。

三、 实验设计

1、 消息的创建、发送和接收

主进程通过 fork 创建两个子进程。发送者进程通过 msgget 获取消息队列，之后获取用户输入并通过 msgsnd 发送给接收者进程。接收者进程通过 msgget 获取消息队列，并通过 msgrcv 接受消息，打印在控制台中。

用户输入 exit 后，发送者退出。主进程使用 wait 等待发送者退出后，关闭消息队列，并退出。接收者进程由于主进程退出而自然退出。

```
#include <cerrno>
#include <cstring>
#include <iostream>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
#define F(arg, error_value, fail)                                   \
  ({                                                                \
    auto ret = arg;                                                 \
    if (ret == error_value) {                                       \
      std::cout << #arg << std::strerror(errno) << std::endl;       \
      fail;                                                         \
    }                                                               \
    ret;                                                            \
  })
int queue;
struct my_message {
  long int type;
  char data[1024];
};

int sender() {
  std::string str;
  queue = msgget((key_t)0xdeadbeef, 0666 | IPC_CREAT);
  while (std::cin >> str) {
    if (str == "exit") {
      return 0;
    }
    if (str.size() > 1023) {
      std::cout << "TOO BIG" << std::endl;
      continue;
    }
    my_message msg{1};
    for (int i = 0; i < str.size(); ++i) {
      msg.data[i] = str[i];
    }
    msg.data[str.size()] = '\0';
    F(msgsnd(queue, &msg, 1024, 0), -1, return -1);
  }
  return 0;
}

int receiver() {
  my_message msg{};
  queue = F(msgget((key_t)0xdeadbeef, 0666 | IPC_CREAT), -1, return -1);
  while (msgrcv(queue, &msg, 1024, 0, 0) != -1) {
    std::cout << "received " << msg.data << std::endl;
  }
```

```
  return 0;
}

int main() {
  queue = F(msgget((key_t)0xdeadbeef, 0666 | IPC_CREAT), -1, exit(-1));
  pid_t t;
  if ((t = fork()) == 0) {
    exit(sender());
  } else if (t < 0) {
    std::cout << std::strerror(errno) << std::endl;
    exit(-1);
  }
  if ((t = fork()) == 0) {
    exit(receiver());
  } else if (t < 0) {
    std::cout << std::strerror(errno) << std::endl;
    exit(-1);
  }
  wait(nullptr);
  msgctl(queue, IPC_RMID, nullptr);
}
```

2、 共享存储取得创建、附接和断接

主进程通过 fork 创建两个子进程。两个子进程先是通过 shmget 和 shmat 进行共享内存的获取与附接，之后通过 sem_open 获取命名信号量作为同步控制。其中，clear 信号量代表内存中数据空，ready 信号量代表内存中有数据。

发送者等到 clear 信号量之后给共享内存赋值，并发送 ready 信号量。接收者接受 ready 信号量后，从共享内存取出值，运算后设置 clear 信号量。

```
#include <cerrno>
#include <cstring>
#include <iostream>
#include <semaphore.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#define F(arg, error_value, fail)                                   \
  ({                                                                \
    auto ret = arg;                                                 \
    if (ret == error_value) {                                       \
```

```
    std::cout << #arg << " " << __LINE__ << " " << std::strerror(errno)      \
              << std::endl;                                                  \
    fail;                                                                    \
  }                                                                          \
  ret;                                                                       \
})
struct shm {
  int data;
};
int sender() {
  int mem = F(shmget((key_t)0xc0ffee, sizeof(shm), IPC_CREAT), -1, return -1);
  shm *mm = reinterpret_cast<shm *>(shmat(mem, nullptr, 0));
  sem_t *clear, *ready;
  clear =
      F(sem_open("leo_osexp_clear", O_CREAT, 0666, 1), SEM_FAILED, return -1);
  ready =
      F(sem_open("leo_osexp_ready", O_CREAT, 0666, 0), SEM_FAILED, return -1);
  int data;
  for(int i = 0; i < 10; ++ i) {
    std::cin >> data;
    sem_wait(clear);
    mm->data = data;
    F(sem_post(ready), -1, return -1);
  }
  return 0;
}

int receiver() {
  int mem = F(shmget((key_t)0xc0ffee, sizeof(shm), IPC_CREAT), -1, return -1);
  shm *mm = reinterpret_cast<shm *>(shmat(mem, nullptr, 0));
  sem_t *clear, *ready;
  clear =
      F(sem_open("leo_osexp_clear", O_CREAT, 0666, 1), SEM_FAILED, return -1);
  ready =
      F(sem_open("leo_osexp_ready", O_CREAT, 0666, 0), SEM_FAILED, return -1);
  if (clear == SEM_FAILED || ready == SEM_FAILED) {
    std::cout << 4 << std::strerror(errno) << std::endl;
    exit(-1);
  }
  while (true) {
    F(sem_wait(ready), -1, return -1);
    std::cout << "received " << mm->data * mm->data << std::endl;
```

```
    F(sem_post(clear), -1, return -1);
  }
}

int main() {
  sem_unlink("leo_osexp_clear");
  sem_unlink("leo_osexp_ready");
  int mem = F(shmget((key_t)0xc0ffee, sizeof(shm), IPC_CREAT | 0666), -1, return -1);
  shm *mm = reinterpret_cast<shm *>(shmat(mem, nullptr, 0));
  if (reinterpret_cast<intptr_t>(mm) == -1) {
    std::cout << 6 << std::strerror(errno) << std::endl;
    exit(-1);
  }
  pid_t t;
  if ((t = fork()) == 0) {
    exit(sender());
  } else if (t < 0) {
    std::cout << std::strerror(errno) << std::endl;
    exit(-1);
  }
  if ((t = fork()) == 0) {
    exit(receiver());
  } else if (t < 0) {
    std::cout << std::strerror(errno) << std::endl;
    exit(-1);
  }
  wait(nullptr);
  shmdt(mm);
  shmctl(mem, IPC_RMID, nullptr);
  sem_unlink("leo_osexp_clear");
  sem_unlink("leo_osexp_ready");
}
```

四、 实验结果与分析

1、 消息的创建、发送和接收

运行结果如下，其中输入的内容用绿色标出：

```
hi
received hi
I'm Leo!
received I'm
received Leo!
```

```
This is Operating System Exp!
received This
received is
received Operating
received System
received Exp!
exit
```

可以看到接收者正确接受到所有内容，程序正常在输入 exit 后退出。

2、 共享存储取得创建、附接和断接

运行结果如下，其中输入内容用绿色标出：

```
1
received 1
2
received 4
3
received 9
4
received 16
5
received 25
6
received 36
7
received 49
8
received 64
9
received 81
10
received 100
```

可以看到接收者正确接受到所有内容，程序正常在输入 10 组数字后退出。

五、 思考与总结

进程间通讯有多种方式。共享内存能够减少内存复制的次数，但是需要额外的同步控制语句。消息队列自带同步控制，但是可能会把变量复制多次。

实验六——使用信号进行进程间通信

一、 实验目的

1、 学习操作系统信号机制的实现方法；
2、 编写 Linux 环境下利用信号实现进程间通信的方法，掌握注册信号处理程序及信号的发送和接收的一般过程。

二、 实验内容

1、 使用信号量完成进程间通信

编写一个程序，完成下列功能：实现一个 SIGINT、用户自定义信号的处理程序，注册该信号处理程序。主函数中创建一个子进程，令父子进程都进入等待状态。SIGINT、用户自定义信号的处理程序完成的任务包括：

(1) 打印接收到的信号的编号
(2) 打印进程 PID。编译并运行该程序，然后在键盘上敲 Ctrl + C，观察出现的现象，并解释其含义。

三、 实验设计

首先，主进程通过 signal 调用注册 sigint 信号的处理器，之后创建子进程，最后分别调用 pause 等待信号到来。

```
#include <iostream>
#include <csignal>
#include <unistd.h>
#include <cstring>
#include <cerrno>
#include <sys/wait.h>

void sigint(int _){
  std::cout<< "SIGINT! Current pid: " << getpid() << ", father pid:" <<
  ↪  getppid() << std::endl;
  exit(1);
}

int main(){
  signal(SIGINT, sigint);
  pid_t t;
  if((t = fork()) == 0){
    // child
```

```
    pause();
    return 0;
  } else if(t < 0) {
    std::cout << std::strerror(errno) << std::endl;
    return -1;
  }
  pause();
  return 0;
}
```

四、 实验结果与分析

```
^CSIGINT! Current pid: 27494, father pid:23812
SIGINT! Current pid: 27496, father pid:27494
```

可以看到，按下 Ctrl+C 后，前台进程组中的两个进程（父、子进程）均收到 SigInt 信号。分别打印 pid 和 ppid 后退出。

同时，可以看到，fork 出的子进程会继承父进程的信号处理函数。