

一、实验目的：

1. 掌握Linux平台下应用程序开发的基本过程；
2. 掌握Linux进程控制相关系统调用函数的使用；
3. 掌握Linux进程通信相关的系统调用函数的使用；
4. 掌握Linux线程、信号、信号量相关系统调用。

二、准备知识：

1. Linux平台下C/C++应用程序开发的基本工具：GCC编译器、GDB调试工具；
2. Linux进程控制相关系统调用：fork()、wait()、waitpid()、kill()、exit()、system()、exec()系列函数；
3. Linux进程通信相关系统调用：无名管道通信（pipe()、read()、write()、close()）、有名管道通信（mknod()、open()、read()、write()、close()）、消息队列通信（msgget()、msgsnd()、msgrcv()、msgctl()）、共享存储（shmget()、shmat()、shmdt()）、信号（kill()、alarm()、signal()）、信号量（semget()、semop()）；
4. Linux线程相关系统调用：pthread_create()、pthread_exit()、pthread_join()。

三、实验内容：

本次实验分为两个部分：

1. 实现多进程间的通信
设计三个进程，P1和P2负责从键盘接收字符串，均发送给P3，P3接收到字符串，根据发送方分别显示"P3 received *** from P1（或P2）"。使用管道通信、消息队列和共享存储三种通信方式实现。
2. 实现哲学家就餐问题的解决方案
选取其中一种编码实现，哲学家就餐使用进程或线程实现均可。

四、实验设计：

1. 通信细节总结

在多进程间进行通信是实现复杂应用程序的必要手段之一。在Linux系统中，有多种不同的进程间通信（IPC）机制可供选择，其中常用的包括管道通信、消息队列通信、共享存储+信号量通信等。

1.1 管道通信（有名管道）

管道是一种基于文件系统的通信方式，可以用来在具有亲缘关系的进程之间传递数据。管道又分为有名管道和无名管道两种。

有名管道是一种特殊的文件，可以在不具有亲缘关系的进程之间传递数据。有名管道的使用需要手动序列化和反序列化数据，相对于直接内存访问，会增加一定的开销。有名管道常用于构建分布式系统，如消息中间件、任务队列等。

1.1.1 函数及其参数

有名管道的创建：

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

参数说明：

- `pathname`：管道文件路径名。
- `mode`：管道文件权限。

有名管道的读写：

```
#include <fcntl.h>

int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

参数说明：

- `pathname`：管道文件路径名。
- `flags`：打开文件的模式，可以是 `O_RDONLY`（只读）、`O_WRONLY`（只写）或 `O_RDWR`（读写）等。
- `fd`：文件描述符。
- `buf`：读写缓冲区。
- `count`：读写数据的字节数。

1.1.2 注意事项

- 创建有名管道需要提供合法的文件路径名，并且需要保证文件路径名的唯一性。
- 使用有名管道进行通信时，需要手动序列化和反序列化数据，即将数据转换为字节流进行传输，接收方再将字节流转换为原始数据。
- 有名管道的读写操作会阻塞进程，因此需要在读写操作前后进行合理的处理，避免进程阻塞导致程序死锁或效率低下。

1.1.3 通信过程

有名管道通信的一般流程如下：

1. 创建有名管道文件，使用 `mkfifo` 函数。
2. 打开管道文件，获取文件描述符，使用 `open` 函数。
3. 进行读写操作，使用 `read` 和 `write` 函数。
4. 关闭管道文件，释放资源，使用 `close` 函数。

1.2 消息队列通信

消息队列是一种通过内核实现的通信方式，可以实现多个进程之间的异步通信。消息队列直接支持结构化数据传输，适用于需要传输大量数据的场景。消息队列的使用需要先创建消息队列，然后发送和接收消息。消息队列通信方式比有名管道更加高效，但相对复杂一些。

1.2.1 函数及其参数

消息队列的创建：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

参数说明：

- `key`：消息队列的键值，可以使用 `ftok` 函数生成。
- `msgflg`：消息队列的标志，可以是 `IPC_CREAT|IPC_EXCL`（如果消息队列已经存在，则返回错误）等。

消息队列的写入：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

参数说明：

- `msqid`：消息队列的标识符。
- `msgp`：指向消息缓冲区的指针。
- `msgsz`：消息的大小。
- `msgflg`：消息发送的标志，可以是 `IPC_NOWAIT`（如果消息队列已满，则立即返回错误）等。

消息队列的读取：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

参数说明：

- `msqid`：消息队列的标识符。
- `msgp`：指向消息缓冲区的指针。

- `msgsz`：消息缓冲区的大小。
- `msgtyp`：消息的类型，可以是具体的消息类型，也可以是 `IPC_NOWAIT` 等特殊标志。
- `msgflg`：消息接收的标志，可以是 `IPC_NOWAIT`（如果消息队列为空，则立即返回错误）等。

1.2.2 注意事项

- 消息队列通信方式支持结构化数据传输，可以直接传输数据对象，不需要手动序列化和反序列化。
- 消息队列通信方式相对于有名管道更加高效，但是相对复杂一些，需要先创建消息队列，然后进行消息的发送和接收。

1.2.3 通信过程

消息队列通信的一般流程如下：

1. 创建消息队列，使用 `msgget` 函数。
2. 发送消息，使用 `msgsnd` 函数。
3. 接收消息，使用 `msgrcv` 函数。
4. 关闭消息队列，释放资源，使用 `msgctl` 函数。

1.3 共享存储+信号量通信

共享存储+信号量通信是一种高效的进程间通信方式，它将共享内存区域映射到多个进程的虚拟地址空间，使得多个进程可以直接访问同一块物理内存。同时，信号量可以保证多个进程访问共享内存时的互斥性和同步性。共享存储+信号量通信需要考虑到互斥问题，否则可能会导致数据不一致或者进程崩溃等问题。

1.3.1 函数及其参数

共享内存的创建：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

参数说明：

- `key`：共享内存的键值，可以使用 `ftok` 函数生成。
- `size`：共享内存的大小。
- `shmflg`：共享内存的标志，可以是 `IPC_CREAT|IPC_EXCL`（如果共享内存已经存在，则返回错误）等。

共享内存的读写：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

参数说明：

- `shmid`：共享内存的标识符。
- `shmaddr`：共享内存的映射地址，如果为 `NULL`，则由内核自动选择一个可用地址。
- `shmflg`：共享内存的标志，可以是 `SHM_RDONLY`（只读模式）等。

信号量的创建和控制：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf *sops, size_t nsops);
```

参数说明：

- `key`：信号量的键值，可以使用 `ftok` 函数生成。
- `nsems`：信号量集中信号量的数量。
- `semid`：信号量的标识符。
- `semnum`：信号量在信号量集中的编号。
- `cmd`：对信号量的操作，可以是 `IPC_RMID`（删除信号量）等。
- `sops`：信号量操作的数组指针。
- `nsops`：信号量操作的数量。

1.3.2 注意事项

- 共享存储+信号量通信方式可以实现高效的进程间通信，但需要考虑到互斥问题，否则可能会导致数据不一致或者进程崩溃等问题。
- 共享内存的读写操作需要考虑数据的同步和互斥问题，可以使用信号量进行控制。
- 信号量的创建、控制和操作需要使用系统调用函数，相对复杂一些。

1.3.3 通信过程

共享存储+信号量通信的一般流程如下：

1. 创建共享内存，使用 `shmget` 函数。
2. 映射共享内存到进程的虚拟地址空间，使用 `shmat` 函数。

3. 创建信号量集，使用 `semget` 函数。
4. 设置信号量的初始值和操作，使用 `semctl` 函数。
5. 进行共享内存的读写操作，使用 `memcpy` 等内存操作函数。
6. 使用信号量进行同步和互斥，使用 `semop` 函数。
7. 解除共享内存的映射，使用 `shmdt` 函数。
8. 删除共享内存和信号量，使用 `shmctl` 和 `semctl` 函数。

2. 通信对比分析

2.1 共享存储

- 优点：
 - 数据持久化存储，不会因为进程退出而丢失数据。
 - 不需要拷贝副本，多个进程可以直接访问同一块物理内存，提高了通信效率。
- 缺点：
 - 需要考虑互斥问题，否则可能会导致数据不一致或者进程崩溃等问题。

2.2 消息队列

- 优点：
 - 直接支持结构化数据传输，适用于需要传输大量数据的场景。
 - 可以实现多个进程之间的异步通信。
- 缺点：
 - 相对于共享存储，消息队列通信方式稍微复杂一些。

2.3 有名管道

- 优点：
 - 可以在不具有亲缘关系的进程之间传递数据。
 - 可以用于构建分布式系统，如消息中间件、任务队列等。
- 缺点：
 - 因为本质上是基于文件系统的通信，所以不支持直接结构化数据传输，需要手动序列化和反序列化。
 - 相对于直接内存访问，会增加一定的开销。

2.4 无名管道

- 优点：
 - 可以用于实现一些简单的进程间通信，如shell中的管道符号“|”。
 - 可以用于进程的协作，如在一个进程中创建一个子进程，使用无名管道来传递数据和命令。
- 缺点：

- 仅适用于具有父子关系的进程之间的通信，且仅支持单向通信。
- 容量有限，一旦管道被填满，写入端的进程会被阻塞。
- 不支持多个进程同时写入，容易引起进程间竞争问题。

3. 通信共性总结

这三种通信方式其实宏观上的行为很类似，我们可以将它们的通信过程概括为以下步骤：

对于消息发送：

1. 初始化通信方式，创建通信对象。
2. 将消息写入通信对象中。
3. 关闭通信对象，释放资源。

对于接收消息：

1. 初始化通信方式，创建通信对象。
2. 从通信对象中读取消息。
3. 关闭通信对象，释放资源。

下图是我针对三种不同的通信做的总结图示，不同的颜色代表不同的通信模式：



五、代码实现和运行截图：

通过第四节分析，我们可以知道三种通信模式本质上具有相似性，因此我将其整合到一个文件之中了，具体的代码如下：

发送方：

```
/*
stdio.h：提供了输入输出函数，如 printf()、scanf() 和 fgets() 等。
stdlib.h：提供了标准库函数，如 exit()、malloc() 和 free() 等。
string.h：提供了字符串处理函数，如 strcpy()、strcat() 和 strlen() 等。
unistd.h：提供了一些系统调用，如 fork()、exec()、wait() 和 pipe() 等。
sys/types.h：定义了一些系统数据类型，如 pid_t 和 key_t 等。
sys/ipc.h：提供了一些 IPC 相关函数和数据类型，如 ftok() 和 key_t 等。
sys/msg.h：提供了消息队列相关函数和数据类型，如 msgget()、msgsnd()、msgrcv() 和 msgctl() 等。
fcntl.h：提供了文件控制操作函数，如 open() 和 fcntl() 等。
errno.h：定义了错误码，如 errno 和 EINTR 等。
sys/stat.h：定义了文件状态的数据类型和相关常量，如 struct stat 和 S_IRUSR 等。
sys/sem.h：提供了信号量相关函数和数据类型，如 semget()、semop() 和 semctl() 等。
sys/shm.h：提供了共享内存相关函数和数据类型，如 shmget()、shmat() 和 shmdt() 等。
*/
#include <stdio.h> //输入输出 printf()、scanf()、fgets()
#include <stdlib.h> //系统库函数 exit()、malloc()、free()
#include <unistd.h> //Unix系统相关函数 fork()、exec()、wait()、pipe()
#include <sys/msg.h> //消息队列 msgget()、msgsnd()、msgrcv()、msgctl()
#include <fcntl.h> //文件控制函数
#include <errno.h> //errno.h错误号定义
#include <sys/stat.h> //stat结构的定义
#include <sys/types.h> //支持Unix基本类型的定义
#include <sys/ipc.h> //IPC相关函数和结构定义
#include <sys/sem.h> //共享内存及信号量
#include <sys/shm.h> //共享内存
#include "message.pb.h"

/*-----使用管道-----*/
#define PUBLIC_PIPE "/tmp/public_pipe"
inline int create_pipe(const char* path, const int oflag) {
    if (access(path, F_OK) == -1) {
        // 创建管道，0666表示所有进程都具有读写权限
        if (mkfifo(path, 0666) < 0) {
            perror("mkfifo error");
            exit(1);
        }
    }
    // 打开管道
    int fd = open(path, oflag);
    if (fd < 0) {
        perror("open error");
        exit(1);
    }
}
```



```

    return fd;
}

void UsingPipe(){
    MyMessage message;
    message.set_pid(getpid());
    printf("success: %d\n", message.pid());
    int fd = create_pipe(PUBLIC_PIPE, O_WRONLY);

    while (1) {
        printf("%d input a string\n", getpid());
        std::string input;
        std::getline(std::cin, input);
        message.set_data(input);
        std::string serialized_message = message.SerializeAsString();
        write(fd, serialized_message.c_str(), serialized_message.size());
    }
    close(fd);
}

/*-----使用消息队列-----*/
#define MSG_SIZE 50
typedef struct message {
    long type;
    int pid;
    char text[MSG_SIZE];
} Message;

void UsingMessageQueue(){
    key_t key;
    int msgid;
    Message message;
    char data_buffer[MSG_SIZE];

    // 生成key
    // 将当前目录和一个字符 'a' 转换成一个唯一的键值，该键值将作为消息队列的标识符
    if ((key = ftok(".", 'a')) < 0) {
        perror("ftok error");
        exit(1);
    }

    // 创建消息队列
    if ((msgid = msgget(key, IPC_CREAT | 0666)) < 0) {
        perror("msgget error");
        exit(1);
    }
    message.pid = getpid();
    // 循环读取用户输入
    while (1) {
        printf("%d input a string\n", getpid());
        fgets(data_buffer, MSG_SIZE, stdin);
        if (strcmp(data_buffer, "q!\n") == 0) {

```

```

        break;
    }

    // 发送消息
    message.type = 1;
    strncpy(message.text, data_buffer, MSG_SIZE);
    if (msgsnd(msgid, &message, MSG_SIZE, 0) < 0) {
        perror("msgsnd error");
        exit(1);
    }
}

}

/*-----使用共享存储-----*/
// 定义一个共享内存区域，用于存储消息
typedef struct {
    int write_index;
    Message messages[10];
} SharedMemory;

// 定义一个互斥锁
static struct sembuf p = {0, -1, SEM_UNDO};
static struct sembuf v = {0, 1, SEM_UNDO};

void UsingSharedBuffer(){
    key_t key;
    int shmid, semid;
    char buffer[MSG_SIZE];

    // 生成key
    if ((key = ftok(".", 'a')) < 0) {
        perror("ftok error");
        exit(1);
    }

    // 创建共享内存段
    if ((shmid = shmget(key, sizeof(SharedMemory), IPC_CREAT | 0666)) < 0) {
        perror("shmget error");
        exit(1);
    }

    // 将共享内存段映射到进程的地址空间中
    SharedMemory* shm_ptr = static_cast<SharedMemory*>(shmat(shmid, NULL, 0));
    if (shm_ptr == (SharedMemory *) -1) {
        perror("shmat error");
        exit(1);
    }

    // 创建信号量
    if ((semid = semget(key, 1, IPC_CREAT | 0666)) < 0) {
        perror("semget error");
        exit(1);
    }
}

```

```

}

// 初始化信号量
if (semctl(semid, 0, SETVAL, 1) < 0) {
    perror("semctl error");
    exit(1);
}

// 循环读取用户输入
while (1) {
    printf("%d input a string\n", getpid());
    fgets(buffer, MSG_SIZE, stdin);

    // 获取锁
    if (semop(semid, &p, 1) < 0) {
        perror("semop error");
        exit(1);
    }

    // 写入数据到共享内存
    strncpy(shm_ptr->messages[shm_ptr->write_index].text, buffer, MSG_SIZE - 1);
    shm_ptr->messages[shm_ptr->write_index].pid = getpid();
    shm_ptr->write_index = (shm_ptr->write_index + 1) % 10;

    // 释放锁
    if (semop(semid, &v, 1) < 0) {
        perror("semop error");
        exit(1);
    }
}
}

int main() {
    int type = 0;
    printf("Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): ");
    scanf("%d", &type);
    switch (type) {
        case 0:
            UsingPipe();
            break;
        case 1:
            UsingMessageQueue();
            break;
        case 2:
            UsingSharedBuffer();
            break;
        default:
            break;
    }
    return 0;
}

```

```
}
```

接受方：

```
#include <stdio.h> //输入输出 printf()、scanf()、fgets()
#include <stdlib.h> //系统库函数 exit()、malloc()、free()
#include <unistd.h> //Unix系统相关函数 fork()、exec()、wait()、pipe()
#include <sys/msg.h> //消息队列 msgget()、msgsnd()、msgrcv()、msgctl()
#include <fcntl.h> //文件控制函数
#include <errno.h> //errno.h错误号定义
#include <sys/stat.h> //stat结构的定义
#include <sys/types.h> //支持Unix基本类型的定义
#include <sys/ipc.h> //IPC相关函数和结构定义
#include <sys/sem.h> //共享内存及信号量
#include <sys/shm.h> //共享内存
#include "message.pb.h"

// Pipe
#define PUBLIC_PIPE "/tmp/public_pipe"

inline int create_pipe(const char* path, const int oflag) {
    if (access(path, F_OK) == -1) {
        // 创建管道, 0666表示所有进程都具有读写权限
        if (mkfifo(path, 0666) < 0) {
            perror("mkfifo error");
            exit(1);
        }
    }
    // 打开管道
    int fd = open(path, oflag);
    if (fd < 0) {
        perror("open error");
        exit(1);
    }
    return fd;
}

void UsingPipe() {
    MyMessage message;
    int fd = create_pipe(PUBLIC_PIPE, O_RDONLY);
    printf("success: %d\n", getpid());
    while (1) {
        // 从管道中读取数据
        char buffer[1024];
        ssize_t size = read(fd, buffer, sizeof(buffer));
        if (size <= 0) {
            perror("read error");
            exit(1);
        }
        // 解析收到的消息
    }
}
```

```

        std::string serialized_message(buffer, size);
        message.ParseFromString(serialized_message);
        printf("received message: %s from %d \n", message.data().c_str(),
message.pid());
        if (strcmp(message.data().c_str(), "end") == 0) {
            break;
        }
    }

    close(fd);
}

#define MSG_SIZE 50
typedef struct message {
    long type;
    int pid;
    char text[MSG_SIZE];
} Message;

void UsingMessageQueue(){
    key_t key;
    int msgid;
    Message message;

    // 生成key
    // 将当前目录和一个字符 'a' 转换成一个唯一的键值，该键值将作为消息队列的标识符
    if ((key = ftok(".", 'a')) < 0) {
        perror("ftok error");
        exit(1);
    }

    // 获取消息队列
    if ((msgid = msgget(key, 0666)) < 0) {
        perror("msgget error");
        exit(1);
    }

    // 循环接收消息
    while (1) {
        if (msgrcv(msgid, &message, MSG_SIZE, 1, 0) < 0) {
            perror("msgrcv error");
            exit(1);
        }
        printf("Received message from pid %d: %s", message.pid, message.text);
        if (!strcmp(message.text, "end")) {
            break;
        }
    }

    // 删除消息队列

```

```

    if (msgctl(msgid, IPC_RMID, NULL) < 0) {
        perror("msgctl error");
        exit(1);
    }
}

// 定义一个共享内存区域，用于存储消息
typedef struct {
    int write_index;
    Message messages[10];
} SharedMemory;

// 定义一个互斥锁
static struct sembuf p = {0, -1, SEM_UNDO};
static struct sembuf v = {0, 1, SEM_UNDO};

void UsingSharedBuffer() {
    key_t key;
    int shmid, semid;

    // 生成key
    if ((key = ftok(".", 'a')) < 0) {
        perror("ftok error");
        exit(1);
    }

    // 获取共享内存段
    if ((shmid = shmget(key, sizeof(SharedMemory), 0666)) < 0) {
        perror("shmget error");
        exit(1);
    }

    // 将共享内存段映射到进程的地址空间中
    SharedMemory* shm_ptr = static_cast<SharedMemory*>(shmat(shmid, NULL, 0));
    if (shm_ptr == (SharedMemory *) -1) {
        perror("shmat error");
        exit(1);
    }

    // 获取信号量
    if ((semid = semget(key, 1, 0666)) < 0) {
        perror("semget error");
        exit(1);
    }

    // 循环接收消息
    while (1) {
        // 获取锁
        if (semop(semid, &p, 1) < 0) {
            perror("semop error");

```

```

        exit(1);
    }

    // 读取消息
    int read_index = (shm_ptr->write_index + 9) % 10;
    // printf("read_index: %d\n", read_index);
    if (strcmp(shm_ptr->messages[read_index].text, "") != 0) {
        printf("pid %d got message from pid %d: %s", getpid(), shm_ptr->messages[read_index].pid, shm_ptr->messages[read_index].text);
        strcpy(shm_ptr->messages[read_index].text, "");
    }

    // 释放锁
    if (semop(semid, &v, 1) < 0) {
        perror("semop error");
        exit(1);
    }

    // usleep(1000000);
}

// 解除共享内存段的映射
if (shmdt(shm_ptr) < 0) {
    perror("shmdt error");
    exit(1);
}

// 删除共享内存段
if (shmctl(shmid, IPC_RMID, NULL) < 0) {
    perror("shmctl error");
    exit(1);
}
}

int main() {
    int type = 0;
    printf("Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): ");
    scanf(" %d", &type);
    switch (type) {
        case 0:
            UsingPipe();
            break;
        case 1:
            UsingMessageQueue();
            break;
        case 2:
            UsingSharedBuffer();
            break;
        default:

```

```
        break;
    }
    return 0;
}

// g++ -o b message.pb.cc B.cpp -lprotobuf
```

5.1 管道通信（有名管道）

发送放选择模式0，然后依次发生1-7的字符：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$ ./a
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 1
6408 input a string
6408 input a string
1
6408 input a string
2
6408 input a string
3
6408 input a string
4
6408 input a string
5
6408 input a string
6
6408 input a string
7
6408 input a string
```

下面是接受放得到的信息：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$
./b
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 1
Received message from pid 6408: null
Received message from pid 6408: 1
Received message from pid 6408: 2
Received message from pid 6408: 3
Received message from pid 6408: 4
Received message from pid 6408: 5
Received message from pid 6408: 6
Received message from pid 6408: 7
```


5.2 消息队列通信

发送方选择模式1，然后依次发生1-7的字符：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$ ./a
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 0
success: 6550
6550 input a string
6550 input a string
1
6550 input a string
1
6550 input a string
2
6550 input a string
3
6550 input a string
4
6550 input a string
5
6550 input a string
6
6550 input a string
7
6550 input a string
```

下面是接受方得到的信息：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$
./b
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 0
success: 6554
received message: from 6550
received message: 1 from 6550
received message: 1 from 6550
received message: 2 from 6550
received message: 3 from 6550
received message: 4 from 6550
received message: 5 from 6550
received message: 6 from 6550
received message: 7 from 6550
```

5.3 共享存储+信号量通信

发送放选择模式2，然后依次发生1-7的字符：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$ ./a
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 1
6408 input a string
6408 input a string
1
6408 input a string
2
6408 input a string
3
6408 input a string
4
6408 input a string
5
6408 input a string
6
6408 input a string
7
6408 input a string
```

下面是接受放得到的信息：

```
star@chase:~/Programming/Linux Programming/Experiments/EX1_进程通信/merge$
./b
Please input your choice (0: UsingPipe, 1: UsingMessageQueue, 2: UsingSharedBuffer): 1
Received message from pid 6408: null
Received message from pid 6408: 1
Received message from pid 6408: 2
Received message from pid 6408: 3
Received message from pid 6408: 4
Received message from pid 6408: 5
Received message from pid 6408: 6
Received message from pid 6408: 7
```

六 哲学家就餐问题的解决方案

6.1 实验原理

哲学家就餐问题是一个经典的并发编程问题，它描述了五个哲学家围坐在桌子旁，每个哲学家左右两边各放着一把叉子，而每个哲学家必须用左右两边的叉子才能进餐。问题在于，如果每个哲学家都先拿起自己左边的叉子，那么右边的叉子就会被邻座的哲学家占用，导致死锁。因此，需要一种策略来保证每个哲学家都能就餐，同时避免死锁。

常见的解决方案有以下几种：

1. Chandy/Misra解法

这种解法通过使用消息传递来解决哲学家就餐问题。每个哲学家都有一个消息队列，当一个哲学家想要就餐时，将向左右两边的哲学家发送请求消息，请求拿起相应的叉子。如果左右两边的哲学家都没有在吃饭，就可以拿起相应的叉子。否则，就等待哲学家吃完饭之后放下叉子，再重新尝试拿起叉子。这种解法可以保证就餐的公平性，但是实现起来比较复杂，且容易发生消息丢失等问题。

2. Dijkstra解法

这种解法通过引入一个额外的资源（如服务员）来解决哲学家就餐问题。每个哲学家在就餐时需要先向服务员请求获取一个许可，只有当服务员发放许可时，哲学家才能拿起左右两边的叉子进餐。这种解法可以避免死锁，但是需要引入额外的资源，实现起来比较复杂。

3. 使用信号量的解法

这种解法通过使用信号量来实现资源的互斥和同步，保证每个哲学家都能就餐。具体实现方法是，每个叉子对应一个信号量，每个哲学家线程在尝试拿起左边的叉子和右边的叉子时，都先尝试获取对应的信号量。如果两个信号量都能够获取成功，就可以开始进餐；否则，就释放已获取的信号量，并等待一段时间再重新尝试。这种解法实现简单，且能够避免死锁。

6.2 实验过程

首先，在代码实现中定义了一个Philosopher类，表示哲学家。Philosopher类具有三个成员变量：哲学家的编号id，左边的叉子left_fork和右边的叉子right_fork，以及哲学家的状态state，状态分为三种：思考、饥饿和进餐。

在Philosopher类中，定义了一个TryToEat方法，用于尝试进餐。在这个方法中，哲学家首先尝试获取左边的叉子，如果成功获取就打印一条信息，否则就等待一段时间后重试。如果左边的叉子获取成功，就尝试获取右边的叉子，如果两个叉子都获取成功，就开始进餐，打印一条信息，并等待一段时间。进餐结束后，哲学家先放下右边的叉子，再放下左边的叉子，并将状态设置为思考。

在主函数中，创建了五个mutex对象，表示五把叉子，然后创建了五个Philosopher对象，每个对象持有左右两边的叉子。接着，创建了五个线程，分别启动每个Philosopher对象的Run方法。最后，等待五个线程结束。

每个哲学家在思考一段时间后会尝试进餐，然后先尝试获取左边的叉子，再尝试获取右边的叉子，如果两个叉子都拿到了，就开始进餐。进餐结束后，放下叉子，并将状态设置为思考。

由于使用了互斥锁（mutex）来保证叉子的互斥访问，因此同一时间只有一个哲学家能够拿到同一把叉子。同时，每个哲学家的状态都是互相独立的，因此不存在死锁等问题。

6.3 代码实现

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
```

```

using namespace std;

const int kNumPhilosophers = 5;

// 哲学家的状态
enum class PhilosopherState {
    kThinking,
    kHungry,
    kEating
};

// 哲学家类
class Philosopher {
public:
    Philosopher(int id, mutex& left_fork, mutex& right_fork)
        : id_(id), left_fork_(left_fork), right_fork_(right_fork),
        state_(PhilosopherState::kThinking) {}

    // 哲学家线程的入口函数
    void Run() {
        while (true) {
            // 思考一段时间
            cout << "Philosopher " << id_ << " is thinking." << endl;
            this_thread::sleep_for(chrono::seconds(1));

            // 想要进餐
            state_ = PhilosopherState::kHungry;
            cout << "Philosopher " << id_ << " is hungry." << endl;
            TryToEat();
        }
    }

private:
    // 尝试进餐
    void TryToEat() {
        // 先尝试获取左边的叉子
        left_fork_.lock();
        cout << "Philosopher " << id_ << " picks up left fork." << endl;
        // 再尝试获取右边的叉子
        if (right_fork_.try_lock()) {
            cout << "Philosopher " << id_ << " picks up right fork." << endl;
            // 如果两个叉子都拿到了, 开始进餐
            state_ = PhilosopherState::kEating;
            cout << "Philosopher " << id_ << " starts eating." << endl;
            this_thread::sleep_for(chrono::seconds(1));
            // 进餐结束, 放下叉子
            right_fork_.unlock();
            cout << "Philosopher " << id_ << " puts down right fork." << endl;
        }
        left_fork_.unlock();
        cout << "Philosopher " << id_ << " puts down left fork." << endl;
        state_ = PhilosopherState::kThinking;
    }
};

```

```

    }

    int id_; // 哲学家的编号
    mutex& left_fork_; // 左边的叉子
    mutex& right_fork_; // 右边的叉子
    PhilosopherState state_; // 哲学家的状态
};

int main() {
    // 创建五个叉子
    mutex forks[kNumPhilosophers];

    // 创建五个哲学家，每个哲学家持有左边和右边的两个叉子
    Philosopher philosophers[kNumPhilosophers] = {
        Philosopher(0, forks[0], forks[1]),
        Philosopher(1, forks[1], forks[2]),
        Philosopher(2, forks[2], forks[3]),
        Philosopher(3, forks[3], forks[4]),
        Philosopher(4, forks[4], forks[0])
    };

    // 创建五个哲学家线程，并启动
    thread philosopher_threads[kNumPhilosophers];
    for (int i = 0; i < kNumPhilosophers; ++i) {
        philosopher_threads[i] = thread(&Philosopher::Run, &philosophers[i]);
    }

    // 等待五个哲学家线程结束
    for (int i = 0; i < kNumPhilosophers; ++i) {
        philosopher_threads[i].join();
    }

    return 0;
}

```

6.4 实验结果

实验结果表明，每个哲学家都能够成功地进餐，且没有发生资源竞争等问题。

Philosopher Philosopher 0 is thinking.2 is thinking.

Philosopher Philosopher 3 is thinking.

Philosopher 4 is thinking.

1 is thinking.

Philosopher 0 is hungry.

Philosopher 0 picks up left fork.

Philosopher 0 picks up right fork.

Philosopher 0 starts eating.

Philosopher 2 is hungry.

Philosopher 2 picks up left fork.

Philosopher 2 picks up right fork.

Philosopher 2 starts eating.

Philosopher 3 is hungry.

Philosopher 4 is hungry.

Philosopher 4 picks up left fork.

Philosopher 4 puts down left fork.

Philosopher 4 is thinking.

Philosopher 1 is hungry.

Philosopher 0 puts down right fork. Philosopher

Philosopher 0 puts down left fork.

Philosopher 0 is thinking.

2 puts down right fork.

Philosopher 2 puts down left fork.

Philosopher 2 is thinking.

Philosopher 1 picks up left fork.

Philosopher 1 picks up right fork.

Philosopher 1 starts eating.

Philosopher 3 picks up left fork.

Philosopher 3 picks up right fork.

Philosopher 3 starts eating.

Philosopher 4 is hungry.

Philosopher 0 is hungry.

Philosopher 0 picks up left fork.

Philosopher 0 puts down left fork.

Philosopher 0 is thinking.

Philosopher 2 is hungry.

Philosopher 1 puts down right fork.

Philosopher 1 puts down left fork.

Philosopher 1 is thinking.

Philosopher 2 picks up left fork.

Philosopher 2 puts down left fork.

Philosopher 2 is thinking.

Philosopher 3 puts down right fork.

Philosopher 3 puts down left fork.4 picks up left fork.