采用share memory 做ipc通信

介绍

1. 共享内存就是允许两个不相关的进程访问同一个逻辑内存；共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式； 不同进程之间共享的内存通常安排为同一段物理内存，进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址； 而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

2. 共享内存是最快的可用IPC形式。

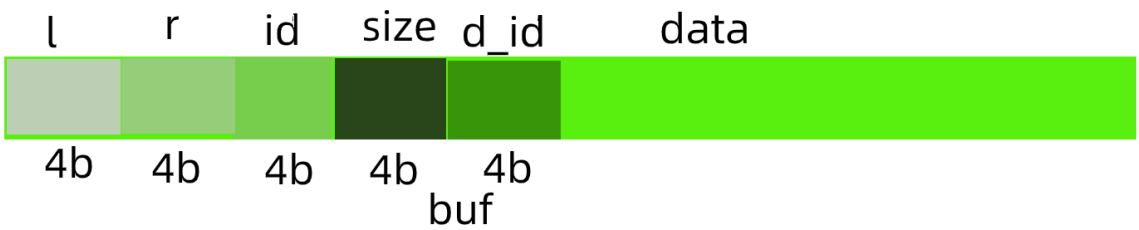
3. 共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取；通常需要用其他的机制来同步对共享内存的访问，例如信号量、互斥锁。

4. 主要有两类实现 System V， Posix两种实现， 两者间的区别及简单实现。

本文通过Posix和信号量实现ipc通信队列并支持多生产者和多消费者。

实现过程

原理



write:对前当前要写的数据增加size和d_id=id+1,读取l,r并计算剩余空间，如果不够则l通过size右移到下个data的起始位置直到剩余空间够写当前数据并更新l，r, id到buf。
read:自己保留一个read l,read_id,读取l,r.如果l_id>read_id则说明当前读取数据已被覆盖，read_l移动到l,读取数据;如果r_id>read_id>=l_id则读取read_l的size移动下一个数据，读取数据;如果read_id>=id则说明无新数据到来。

- 解释 l为第一个数据的起始位置， r为最后一个数据的起始位置，id为当前写入的数据id（默认从0递增），data为原始数据，d_id为当前写入数据的id值，size为当前data+d_id+size之后的大小。buf为整个共享内存，l，r,id为header; size,d_id为数据的header。除了buf header剩余的空间为数据区，本质是个循环队列，这里大小为queue_size = buf_size - 12。

- 过程

1. **写操作**，读取l,r,id,封装当前数据（增加d_id=id+1和size）,判断(r-l+queue_size)%queue_size计算当前剩余空间是否能够放下当前数据，如果不够，(l + l_size)%queue_size,剩余空间增加l_size。循环之，直到剩余空间能够放下当先数据，然后复制到buf，得到新的r。更新l，r，id=id+1到buf header。写操作完成。

2. **读操作**, 读取l，r，id。read_index,read_fid（为已经读取的数据位置，数据的id，由read方维护），计算r处的r_id,l处的l_id.如果r_id<=read_fid,则当前无新数据可读;如果l_id<=read_id，则read_index移动read_size到下一个数据的起点，读取数据即可read_index=(read_size + read_index)%queue_size,read_id = read_index_id;如果l_id>read_id则数据已经被覆盖，read_index=l_index，read_id=l_id。至次读取数据完毕。

3. 由于共享内存没有同步操作，所以每次读写操作之前都需要通过信号量来同步，确保每次读写分离。

主要函数解释

- prepare() 创建或挂载共享内存，创建或绑定信号量

```
void parare(){
    int fd = shm_open(topic, O_CREAT | O_EXCL | O_RDWR, 0644);

    if (fd == -1){
        fd = shm_open(topic, O_EXCL | O_RDWR, 0644);
        if( fd == -1) {
            perror("shm_open");
            exit(EXIT_FAILURE);
        }
    }

    if( ftruncate(fd, shm_size) == -1){
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }
    buf = (uint8_t*)mmap(NULL, shm_size, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED, fd, 0);
    if (buf == MAP_FAILED){
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fd);

    sem = sem_open(topic, O_CREAT | O_RDWR  | O_EXCL, 0644, 1);
    if( sem == SEM_FAILED){

        sem = sem_open(topic, O_RDWR  | O_EXCL, 0644, 1);

        if(sem == SEM_FAILED) {
            perror("sem_open");
            exit(EXIT_FAILURE);
        }
    }

}
```

- write(uint8_t* data, uint32_t size) 读取数据

```
void write(uint8_t* data, uint32_t size){
    if( sem_wait(sem) == -1) {
        perror("sem_wait");
        exit(EXIT_FAILURE);
    }

    uint32_t l = _uint8_2_uint32(0); //
    uint32_t r = _uint8_2_uint32(4); //
    uint32_t fid = _uint8_2_uint32(8);

    // 构造数据
    uint8_t* tmp = (uint8_t*)malloc(size + 8);
    memcpy(tmp+8, data, size);
    size += 8;
    fid += 1;
    for( int i = 0; i < 4; i++ ) tmp[3-i] = size >> (i * 8) & 0xFF;
    for( int i = 0; i < 4; i++ ) tmp[7-i] = fid >> (i * 8) & 0xFF;

    int real_size = (( r - l ) + queue_size) % queue_size;
    int remain = queue_size - real_size - 1;

    while( remain < size) {
        uint32_t s = _uint8_2_uint32(l + 1 + head_size);
        l = (l + s)%queue_size;
        remain += s;
    }

    if( queue_size - r - 1 < size) {
        int dt = size - queue_size + r + 1;
        memcpy(buf + r + head_size + 1, tmp, queue_size - r - 1);
        memcpy(buf + head_size, tmp + queue_size - r -1, dt);
        r = dt - 1;
    }else{
        memcpy(buf + r + head_size + 1, tmp, size);
        r = r + size;
    }

    for( int i = 0; i < 4; i++ ) buf[3-i] = l >> (i * 8) & 0xFF;
    for( int i = 0; i < 4; i++ ) buf[7-i] = r >> (i * 8) & 0xFF;
    for( int i = 0; i < 4; i++ ) buf[11-i] = fid >> (i * 8) & 0xFF;
    printf("write fid is %d\n", fid);

    if( sem_post(sem) == -1) {
        perror("sem_post");
        exit(EXIT_FAILURE);
    }

}
```

- uint_8* read() 读取数据

```

```

```
uint8_t* read(){

    if( sem_wait(sem) == -1) {
        perror("sem_wait");
        exit(EXIT_FAILURE);
    }


    uint32_t l = _uint8_2_uint32(0); //
    uint32_t r = _uint8_2_uint32(4); //
    uint32_t last_fid = _uint8_2_uint32(8);

    if( read_index == -1) {
        read_index = (l + 1)%queue_size;
    }else{
        if(last_fid <= read_fid) {
            if( sem_post(sem) == -1) {
                perror("sem_wait");
                exit(EXIT_FAILURE);
            }
            return NULL;
        }
        uint32_t mi_fid = _uint8_2_uint32(l + 1 + head_size + 4);

        if( mi_fid <= read_fid) {
            uint32_t s = _uint8_2_uint32(read_index + head_size);
            read_index = (s + read_index) % queue_size;

        }else read_index = (l + 1)%queue_size;
    }
    uint32_t s = _uint8_2_uint32(read_index + head_size);
    uint8_t* data = (uint8_t*)malloc(s);

    if( s + read_index > queue_size){
        uint32_t dt = s + read_index - queue_size;

        memcpy(data, buf + head_size + read_index, s - dt);
        memcpy(data + s - dt, buf + head_size, dt);

    }else{
        memcpy(data, buf + head_size + read_index, s);
    }
    read_fid = _uint8_2_uint32(read_index + head_size + 4);
    printf("read fid os %d\n", read_fid);

    if( sem_post(sem) == -1) {
        perror("sem_post");
        exit(EXIT_FAILURE);
    }
    return data;
    }
```

- python接口

```
extern "C"{
    ShmFlow* shmflow = NULL;
    void init_flow(char* topic, int size){
        if(shmflow == NULL)  shmflow = new ShmFlow(topic, size);
        printf("shmflow init ok\n");
    }

    void write_data(char* data, int size) {
        uint8_t* t = (uint8_t*)data;
        shmflow->write(t, size);
    }

    int read_data(char* data){
        uint8_t* res = shmflow->read();
        if( res == NULL) return 0;
        uint32_t tmp = 0;
        for( uint32_t i = 0; i < 4; i++) tmp = (tmp << 8) + res[i];
        memcpy(data, res, tmp);
        return 1;
    }

}
```

- python调用示例 读取数据

```
def test_shmflow_read():
    import cv2
    import numpy as np

    shmflow_dll =
cdll.LoadLibrary("/home/cao/CLionProjects/pcv/build/lib/libshmflow.so")

    topic = c_char_p(bytes("video", 'utf-8'))
    shmflow_dll.init_flow(topic, 1280*3*720*10)


    while True:
        d = create_string_buffer(1280 * 3 * 720 * 10)
        res = shmflow_dll.read_data(d)
        if res == 0:
            continue
        size = int.from_bytes(d.raw[:4], byteorder="big", signed=False)
        fid = int.from_bytes(d.raw[4:8], byteorder="big", signed=False)
        print(fid, size)
        d = d.raw[8:size]
        png = np.fromstring(d, dtype=np.uint8).reshape((720, 1280, 3))
        cv2.imshow("----", png)
```

```
        cv2.waitKey(110)
```

写数据

```
def test_shmflow_write():
    import cv2
    img = cv2.imread("/home/cao/pcview-
v2/x1d3_view/pcview_data/20200527112701/cap-fig/20200527-113043-314863-
cap.png")

    shmflow_dll =
cdll.LoadLibrary("/home/cao/CLionProjects/pcv/build/lib/libshmflow.so")

    topic = c_char_p(bytes("video", 'utf-8'))
    shmflow_dll.init_flow(topic, 1280 * 3 * 720 * 10)
    img = cv2.resize(img, (1280, 720))
    img = img.tostring()
    # t = np.fromstring(img, dtype=np.uint8).reshape((720, 1280, 3))
    # cv2.imshow("---", t)
    # cv2.waitKey(1000)
    cp = c_char_p(img)
    shmflow_dll.write_data(cp, 1280*720*3)
```

亟待完善

写和写互斥，写和读互斥，读和读不互斥

感谢您阅读此文档