实验2、生产者/消费者问题的实现

- 实验目的

① 掌握进程、线程的概念，熟悉相关的控制语。

② 掌握进程、线程间的同步原理和方法。

③ 掌握进程、线程间的互斥原理和方法。

④ 掌握使用信号量原语解决进程、线程间互斥和同步方法。

- 实验内容

①有一群生产者任务在生产产品，并将这些产品提供给消费者任务去消费。为使生产者任务与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池：

- 生产者任务从文件中读取一个数据，并将它存放到一个缓冲区中
- 消费者任务从一个缓冲区中取走数据，并输出此数据（printf）
- 生产者和消费者之间必须保持同步原则：不允许消费者任务到一个空缓冲区去取产品；也不允许生产者任务向一个已装满产品且尚未被取走的缓冲区中投放产品。

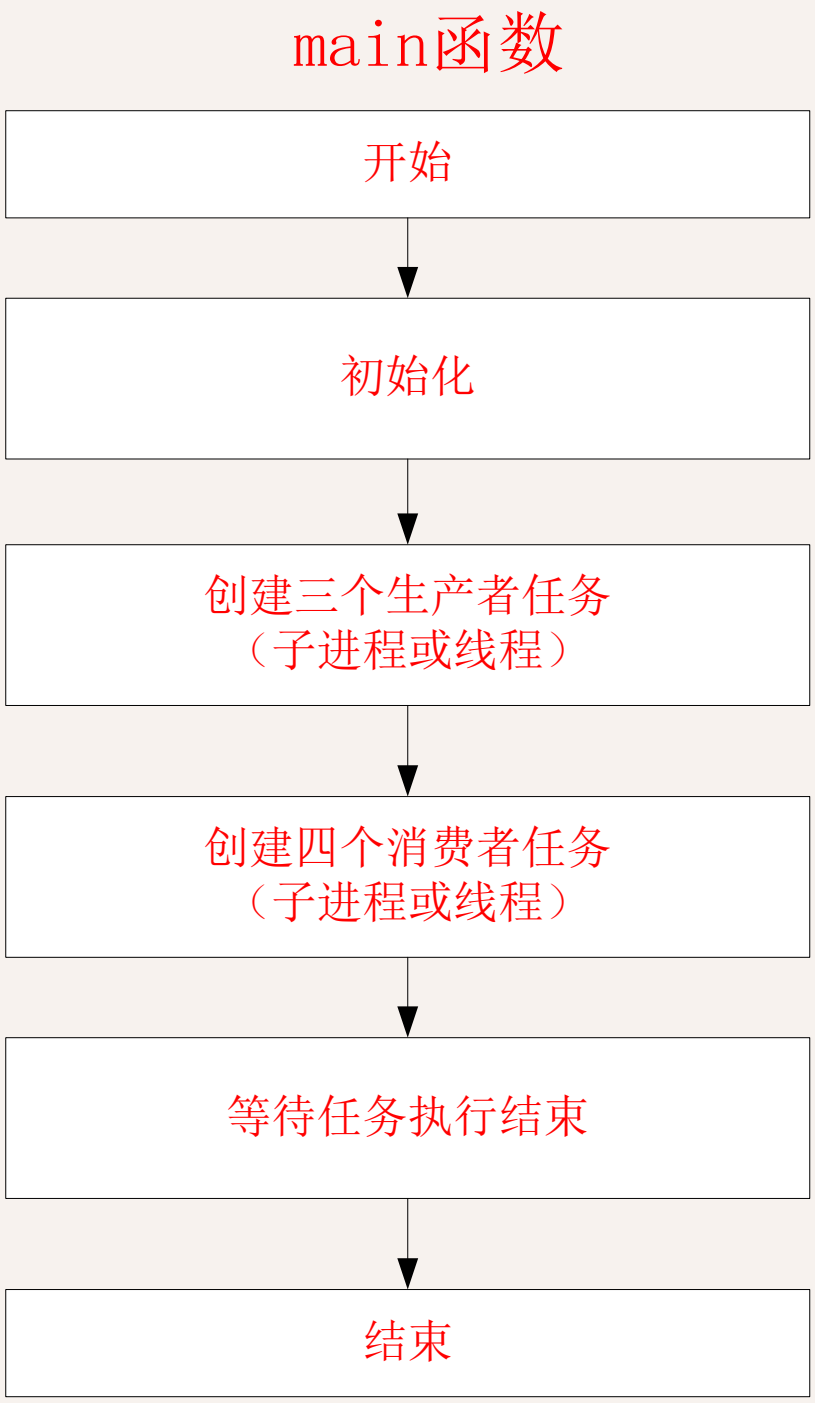
- 实验内容

②创建3个进程（或者线程）作为生产者任务，4个进程（或者线程）作为消费者任务

③创建一个文件作为数据源，文件中事先写入一些内容作为数据（字符串或者数值）

④生产者和消费者任务（进程或者线程）都具有相同的优先级

- 实验原理：


main函数
开始
初始化
创建三个生产者任务
（子进程或线程）
创建四个消费者任务
（子进程或线程）
等待任务执行结束
结束

- 实验原理

n个缓冲区的缓冲池作为一个临界资源：

- ✓ 当生产者任务从数据源—文件中读取数据后将会申请一个缓冲区，并将此数据放入缓冲区中。
- ✓ 消费者任务从一个缓冲区中取走数据，并将其中的内容打印输出。

当一个生产者任务正在访问缓冲区时，其他生产者和消费者任务不能访问缓冲区

当一个消费者任务正在访问缓冲区时，其他其他生产者和消费者任务不能访问缓冲区

- ◆ 使用互斥量实现对缓冲池的互斥访问！！

- 实验原理

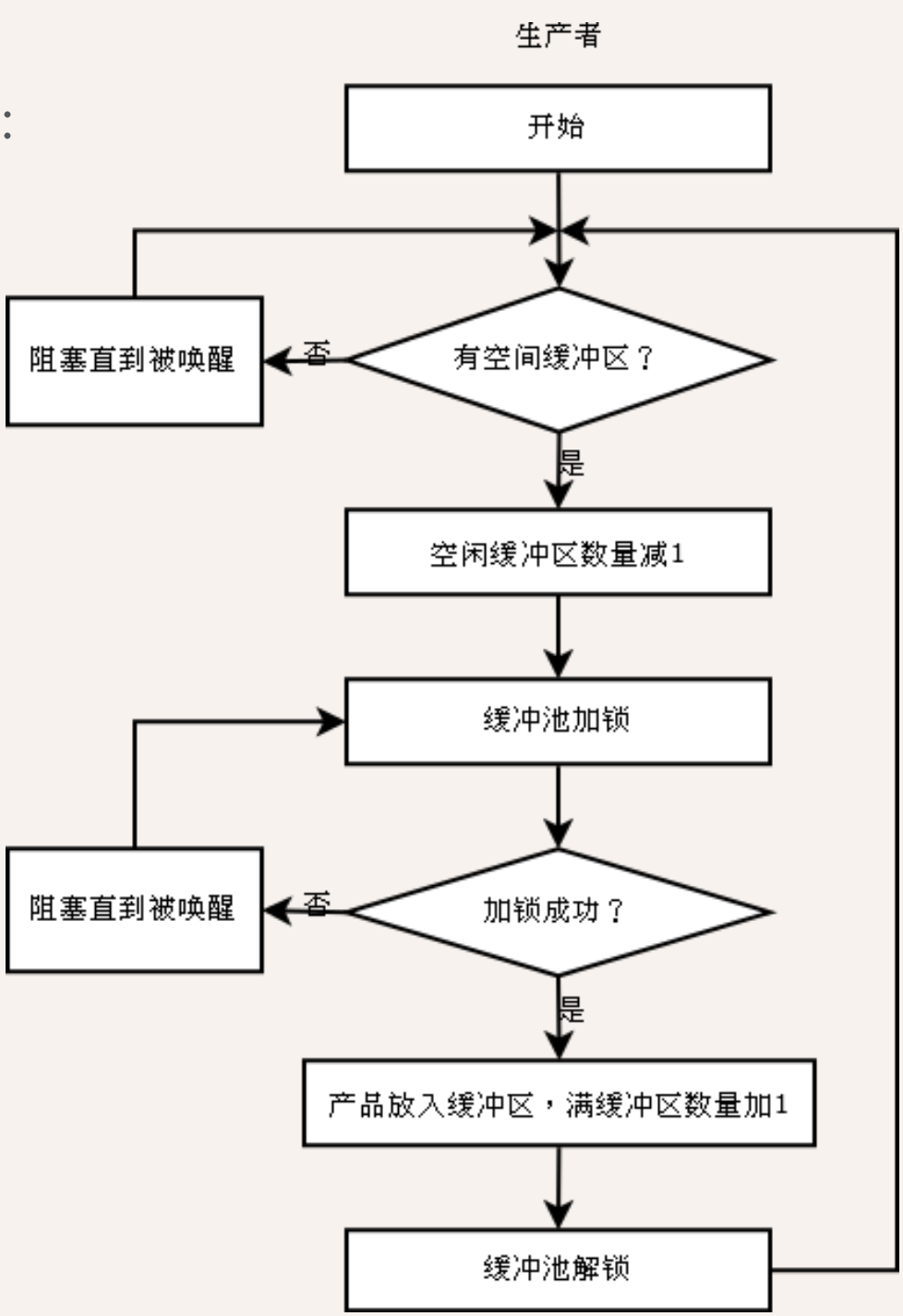
生产者任务在向缓冲池放入数据之前需要判断缓冲池中是否还有空的缓冲区，如果有则向空的缓冲区写入，如果没有则等待

消费者任务在从缓冲池读取数据之前需要判断缓冲池中是否有已经写入数据的缓冲区，如果有则读取已经写入数据的缓冲区，如果没有则等待

- 使用两个信号量，一个代表空缓冲区的数量，一个代表已经写入数据的缓冲区数量！！

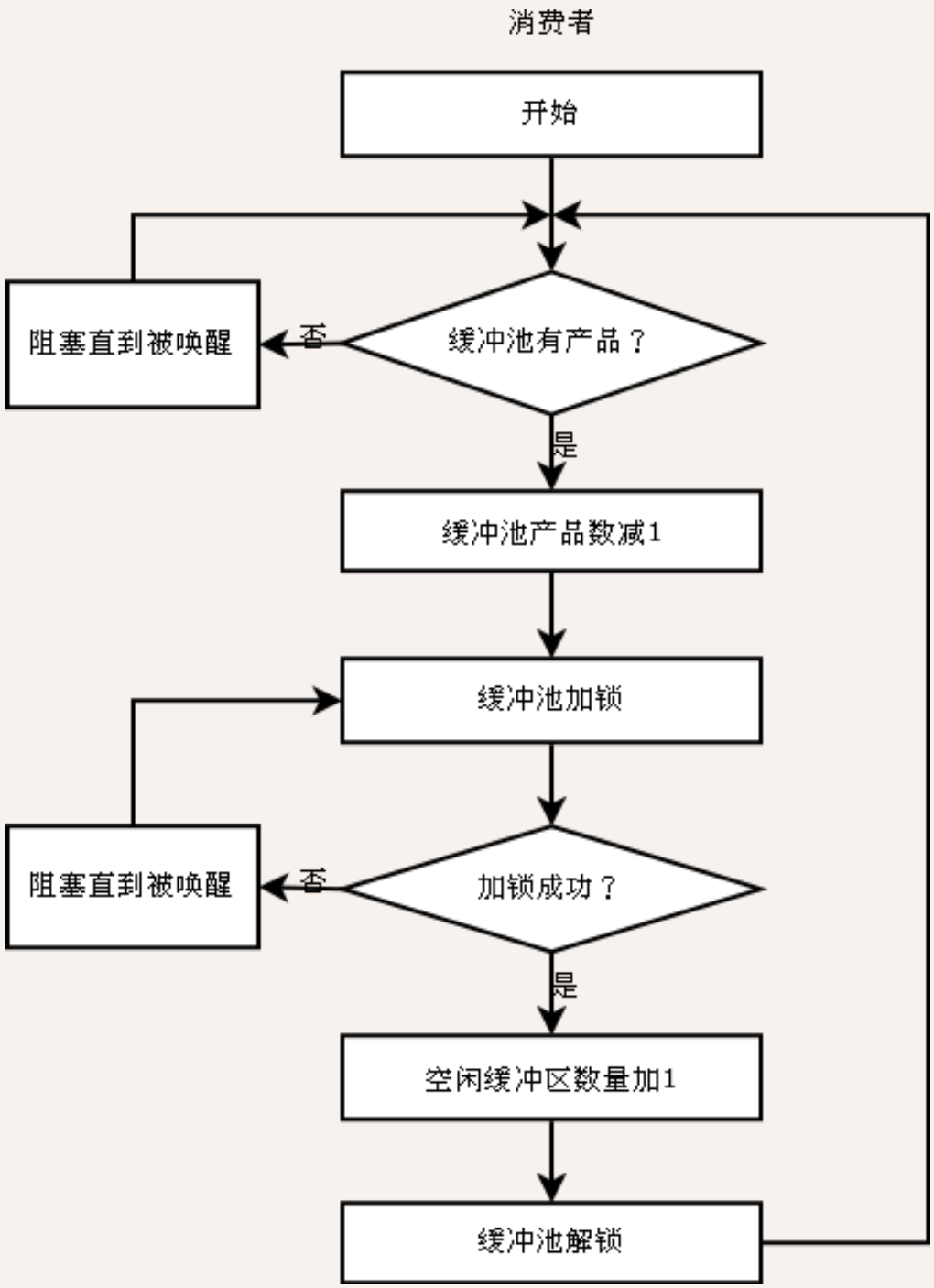
- 实验原理：

生产者任务


生产者
开始
有空间缓冲区？
否
阻塞直到被唤醒
是
空闲缓冲区数量减1
缓冲池加锁
加锁成功？
否
阻塞直到被唤醒
是
产品放入缓冲区，满缓冲区数量加1
缓冲池解锁

- 实验原理：

消费者任务


消费者
开始
缓冲池有产品？
否
阻塞直到被唤醒
是
缓冲池产品数减1
缓冲池加锁
加锁成功？
否
阻塞直到被唤醒
是
空闲缓冲区数量加1
缓冲池解锁

- 编程思路

- ✓ 分配具有n个缓冲区的缓冲池，作为临界资源。
- ✓ 定义两个记录型信号量empty 和full，empty信号量表示当前空的缓冲区数量，full表示当前写入数据的缓冲区（满缓冲区）的数量。
- ✓ 定义互斥量mutex，当某个任务访问缓冲区之前对互斥量加锁，在对缓冲区的操作完成后再释放此互斥量。以此实现多个任务对临界资源的互斥访问。
- ✓ 创建3进程（或者线程）作为生产者任务，4个进程（或者线程）作为消费者任务。创建一个文件作为数据源，文件中事先写入一些内容作为内容。

- 编程思路

✓ 生产者任务的工作内容：

从文件中读取数据

申请对empty信号量进行P操作

申请对mutex互斥量加锁

进入临界区操作将读取的数据放入到空缓冲区（放到哪个空缓冲区可以参考教材2.5.1中的输入指针）

对full信号量进行V操作

释放mutex互斥量。

- 编程思路

✓ 消费者任务的工作内容：

申请对full信号量进行P操作

申请对mutex互斥量加锁

进入临界区将写入的数据缓冲区读出并通过printf打印（读出哪个写入数据的缓冲区可以参考教材2.5.1中的输出指针）

对empty信号量进行V操作

释放mutex互斥量。

- 实验报告

说明实验过程，进行结果分析。

- 实验器材（设备、元器件）

① 学生每人一台PC，安装WindowsXP/2000操作系统。

② 局域网络环境。

③ 个人PC安装VMware虚拟机和Ubuntu系统。

- 实验内容：

熟悉Ubuntu系统下的多线程编程。

① 使用“Ctrl+Alt+T”打开终端；

② 使用gedit或vim命令打开文本编辑器进行编码：“gedit 文件名.c”

③ 编译程序：“gcc文件名.c -o 可执行程序名”（如果只输入gcc文件名.c，默认可执行程序名为a.out）使用线程库时，gcc编译需要添加-lpthread

④ 执行程序：./可执行程序名

线程的创建

- pthread_create函数用于创建一个线程
- 函数原型

```
int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *),
                   void *restrict arg);
```

（pthread.h）

- 调用pthread_create函数的线程是所创建线程的父线程

pthread_create函数

- 参数

  tidp：指向线程ID的指针，当函数成功返回时将存储所创建的子线程ID

  attr：用于定制各种不同的线程属性（一般直接传入空指针NULL，采用默认线程属性）

  start_rtn：线程的启动例程函数指针，新创建的线程执行该函数代码

  arg：向线程的启动例程函数传递信息的参数

- 返回值

  成功返回0，出错时返回各种错误码

线程的终止

- 线程的三种终止方式

  线程从启动例程函数中返回，函数返回值作为线程的退出码

  线程被同一进程中的其他线程取消

  线程调用pthread_exit函数终止执行

获取线程ID

- pthread_self函数可以让调用线程获取自己的线程ID
- 函数原型

pthread_t pthread_self();（pthread.h）

- 返回调用线程的线程ID

pthread_exit函数

- 线程终止函数

void pthread_exit(void *rval_ptr);

（pthread.h）

- 参数

rval_ptr：该指针将传递给pthread_join函数

（与exit函数参数用法类似）

父线程等待子线程终止

- 父线程等待自线程终止函数原型

```
int pthread_join(pthread_t thread,void **rval_ptr);
```

（pthread.h）

- 调用该函数的父线程将一直被阻塞，直到指定的子线程终止

  从启动例程函数中返回

  被同一进程中的其他线程取消

  调用pthread_exit终止执行

pthread_join函数

- 参数

  thread：需要等待的子线程ID

  rval_ptr：若不关心线程返回值，可直接将该参数设置为空指针NULL

  - 若线程从启动例程返回，rval_ptr将包含返回码
  - 若线程被取消，由rval_ptr指定的内存单元就置为PTHREAD_CANCELED
  - 若线程由于pthread_exit终止，rval_ptr即pthread_exit的参数

- 返回值

  成功返回0，否则返回错误编号

创建并等待子线程代码示例

```
void *childthread(void){
int i;
for(i=0;i<10;i++){
    printf("childthread message\n");
    sleep(100);}}
int main(){
    pthread_t tid;
    printf("create childthread\n");
    pthread_create(&tid,NULL,(void *) childthread,NULL);
    pthread_join(tid,NULL);
    printf("childthread exit\n"); }
```

互斥量

- 确保同一时间里只有一个线程访问共享资源或临界区域
- 互斥量（mutex）本质上是一把锁

  在访问共享资源后临界区域前，对互斥量进行加锁

  在访问完成后释放互斥量上的锁

  对互斥量进行加锁后，任何其他试图再次对互斥量加锁的线程将会被阻塞，直到锁被释放

互斥量的初始化

- 静态初始化

pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER

互斥量的初始化

- 动态初始化

函数原型（pthread.h）

```
int pthread_mutex_init(pthread_mutex_t *mutex,const
  pthread_mutexattr_t *attr);
```

参数与返回值

- mutex：即互斥量，类型是pthread_mutex_t

  注意：mutex必须指向有效的内存区域
- attr：设置互斥量的属性，通常可采用默认属性，即可将attr设为NULL。
- 成功返回0，出错返回错误码

互斥量的销毁

- 互斥量在使用完毕后，必须要对互斥量进行销毁，以释放资源
- 函数原型（pthread.h）

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- 参数与返回值
  mutex：即互斥量
  成功返回0，出错返回错误码

互斥量的加锁和解锁操作

- 在对共享资源访问之前和访问之后，需要对互斥量进行加锁和解锁操作
- 函数原型（pthread.h）

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
Int pthread_mutex_unlock(pthread_mutex_t
  *mutex);
```

- 当调用pthread_mutex_lock时，若互斥量已被加锁，则调用线程将被阻塞

尝试加锁

- 函数原型（pthread.h）

```
int pthread_mutex_trylock(pthread_mutex_t
*mutex);
```

- 调用该函数时，若互斥量未加锁，则锁住该互斥量，返回0；若互斥量已加锁，则函数直接返回失败（不会阻塞调用线程）

互斥量的操作顺序

- 定义一个互斥量变量
- 调用pthread_mutex_init初始化互斥量
- 调用pthread_mutex_lock或者pthread_mutex_trylock对互斥量进行加锁操作
- 调用pthread_mutex_unlock对互斥量解锁
- 调用pthread_mutex_destroy销毁互斥量

- 示例

```
//…
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
//do something
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
//…
```

- POSIX信号量

#include<semaphore.h>

信号量数据类型：sem_t

- 主要函数：

①sem_init(sem_t *sem, int pshared, unsigned int value);//初始化一个无名信号量

②sem_destroy(sem_t *sem);//销毁一个无名信号量

返回值：成功返回 0；错误返回 -1，并设置 errno 。

③ sem_post(sem_t *sem);//信号量P操作，信号量值加1。若有线程阻塞于信号量sem，则调度器会唤醒对应阻塞队列中的某一个线程。

④ sem_wait(sem_t *sem);//信号量V操作，若信号量值小于0，则线程阻塞于信号量sem，直到sem大于0。否则信号量值减1。

⑤ sem_trywait(sem_t *sem);//功能同sem_wait()，但此函数不阻塞，若sem小于0，直接返回。

返回值：成功返回0，错误返回-1，并设置errno 。

- 示例

```
sem_t sem;
sem_init(&sem, 0, 1);//初始化一个值为1的信号量
sem_wait(&sem);//信号量P操作
//do somthing
sem_post(&sem);//信号量V操作
sem_destroy(&sem);//销毁一个无名信号量
```