实验目的

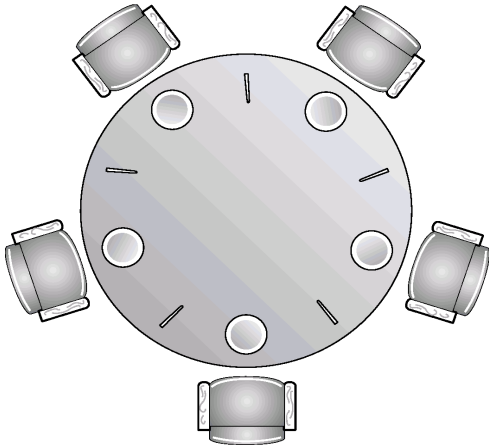
- 实现哲学家就餐问题，要求不能出现死锁。
- 通过本实验熟悉Linux系统的基本环境，了解Linux下进程和线程的实现。

实验内容

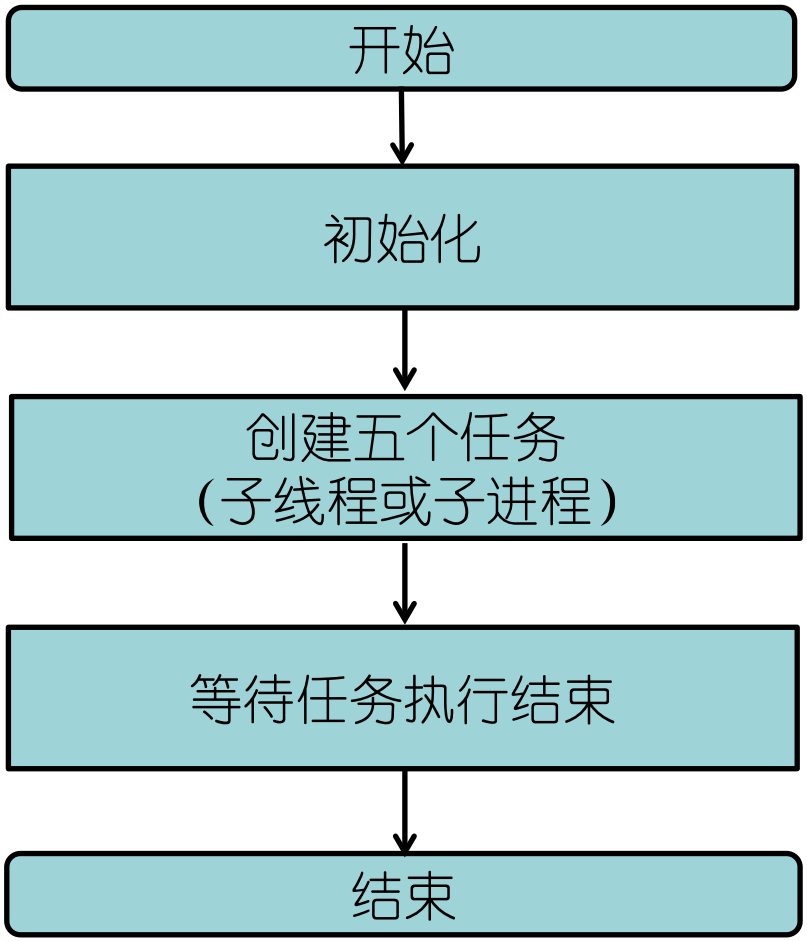
- 在linux系统下实现教材2.5.2节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。**连续运行30次以上**都未出现死锁现象。

实验原理-问题描述：

五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取其左右最靠近它的筷子，只有他拿到两只筷子时才能进餐。进餐毕，放下筷子继续思考。



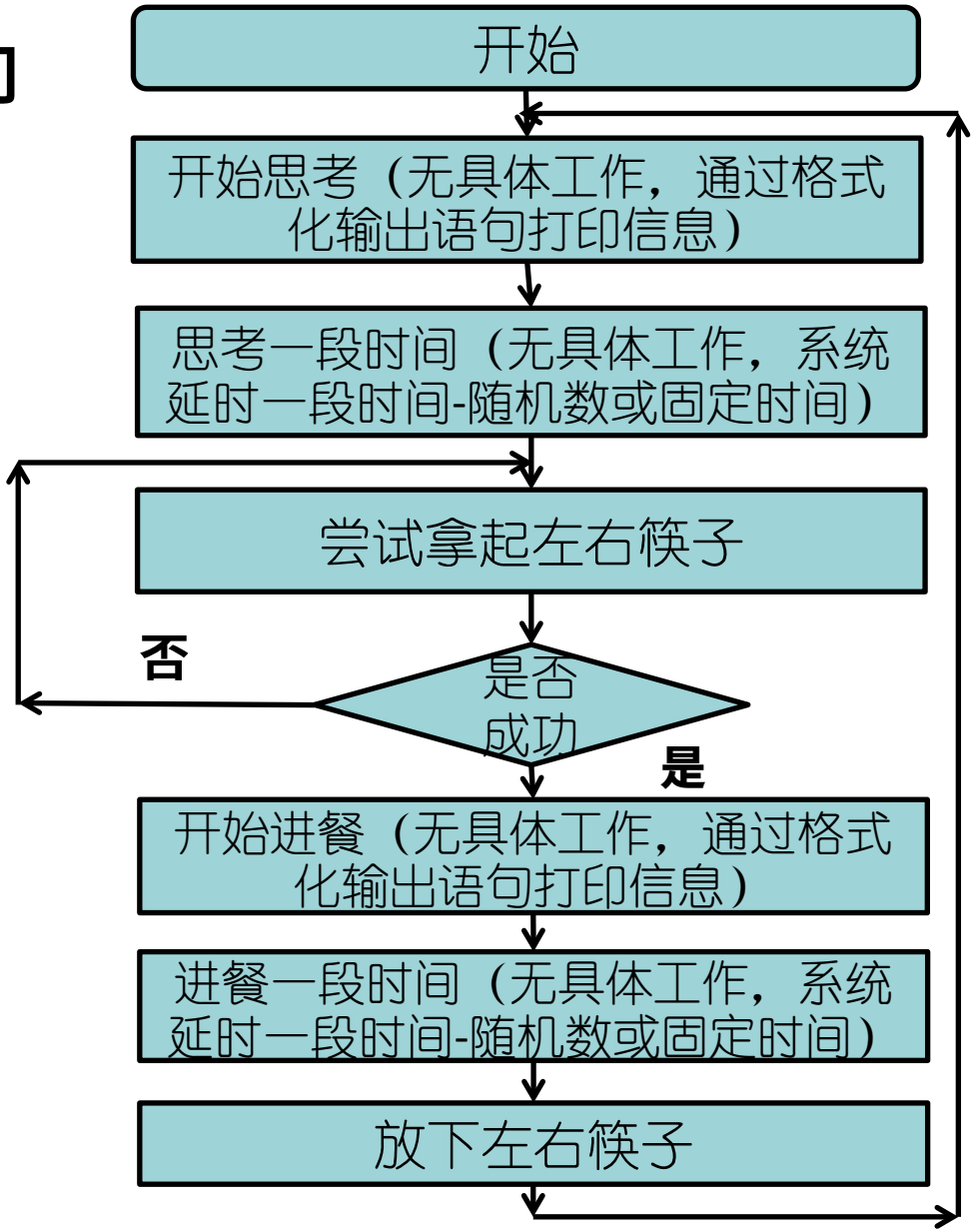
实验原理：哲学家进餐问题的程序实现



main函数

实验原理：哲学家进餐问题的程序实现

每一个哲学家函数



实验原理

拿起筷子和放下筷子的操作如何实现：

- ✓筷子是临界资源，拿起筷子操作就哲学家（不同任务）之间是对临界资源的互斥访问，放下筷子就是对临界资源的释放，在Linux中可以通过多种机制来实现
  - ✓互斥量（加锁，解锁），适用于线程
  - ✓POSIX信号量（P操作，V操作），无名信号量适用于线程，命名信号量适用于进程/线程
  - ✓XSI信号量集（P操作，V操作），同时适用于进程与线程

实验原理

拿起拿起筷子操作导致死锁的原因：

- ✓筷子需要互斥访问
- ✓任务无法抢占其他任务已经拿起的筷子
- ✓任务拿起筷子直到进餐完毕后才放下
- ✓圆桌导致可能出现循环等待

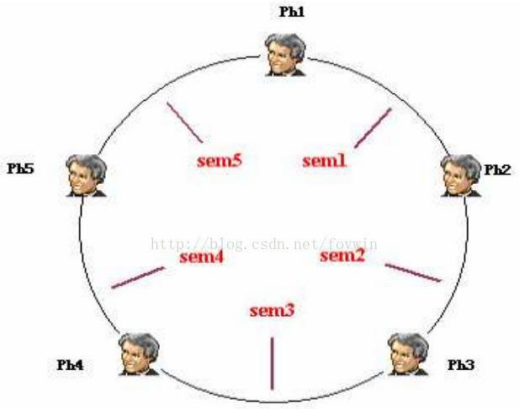


图 4-1

实验原理

预防死锁的实现方法1：破坏“ 请求和保持条件 ”**（作为参考，最终实验不能按照此方法实现！）**

- ✓任务如果无法同时拿起两支筷子，则放下已经拿起的筷子，等待一段时间再尝试
- ✓利用POSIX API中的非阻塞操作实现对能否拿起筷子的判断
  - ✓pthread_mutex_trylock操作（非阻塞加锁）
  - ✓sem_trywait操作（非阻塞P操作）
  - ✓ XSI信号量集中设置IPC_NOWAIT参数（非阻塞V操作）

实验原理

预防死锁的实现方法2：破坏“ 循环等待条件 ”

- ✓对哲学家编号，奇偶号哲学家拿起筷子的顺序不同
- ✓……

实验原理

避免死锁的实现方法

- ✓再创建一个任务，哲学家拿起筷子时向该任务发起申请，由该任务对当前筷子的分配情况进行判断，判定系统是否由安全状态向不安全状态转换，从而允许或拒绝该次申请
- ✓……

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** ” **cd Desktop**

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**，默认生成可执行程序名为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**

pthread_create函数

- 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);
```

⑩头文件：pthread.h

- 调用pthread_create函数的线程是所创建线程的父线程

pthread_create参数和返回值

- **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

pthread_create传递多个参数

```
struct mypara
{
    char str_path[1024];
    char des_path[1024];
};

struct mypara para;

/*创建线程一*/
ret=pthread_create(&id,NULL,(void  *) thread1,&(para));

/*等待线程结束*/
pthread_join(id,NULL);
```

pthread_self函数-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();
```

头文件：pthread.h

- 返回调用线程的线程ID

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（函数体中的代码执行完后自行结束）
- 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

pthread_exit函数-线程终止

- 线程终止函数

```
void pthread_exit(void *rval_ptr);
```

头文件：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）

例子：pthread_exit(NULL);//如果线程不需要返回任何数据

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);

  头文件：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

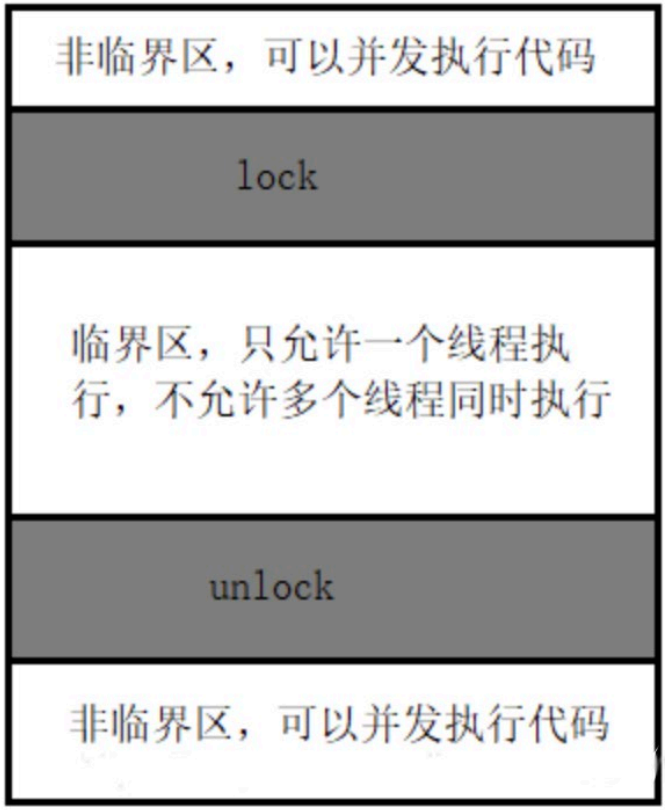
- 参数　　int pthread_join(pthread_t thread,void **rval_ptr);
  - **thread**：需要等待的子线程ID
  - **rval_ptr**：若不关心线程返回值，可直接将该参数设置为空指针NULL
    - 若线程从启动例程返回，rval_ptr将包含**返回码**
    - 若线程被取消，由rval_ptr指定的内存单元就置为**PTHREAD_CANCELED**
    - 若线程由于pthread_exit终止，**rval_ptr即pthread_exit的参数**
- 返回值
  - 成功返回0，否则返回错误编号

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

互斥量的销毁

在销毁互斥量的过程中需要注意几个问题：

①使用PTHREAD_ MUTEX_ INITIALIZER初始化的互斥量不需要销毁 （即静态分配）

②不要销毁一个已经加锁的互斥量

③已经销毁的互斥量，要确保后面不会有线程再尝试加锁

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

① 定义一个互斥量变量

② 调用pthread_mutex_init初始化互斥量

③ 调用pthread_mutex_lock或者pthread_mutex_trylock对互斥量进行加锁操作

④ 调用pthread_mutex_unlock对互斥量解锁

⑤ 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：

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

② sem_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 。

POSIX信号量

③ **sem_post(sem_t *sem);//**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);//**若sem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 。

POSIX信号量示例

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

错误码 / errno

- Linux中系统调用的错误都存储于 errno中，errno由操作系统维护，存储就近发生的错误，即下一次的错误码会覆盖掉上一次的错误。

PS: 只有当系统调用或者调用lib函数时出错，才会置位errno！

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
int main()
{ int tmp = 0;
for(tmp = 0; tmp <=256; tmp++) {
printf("errno:%2d\t%s\n",tmp,strerror(tmp)); }
return 0; }
```

显示每个错误码的含义

错误码 / errno

```
int main( ){
FILE * fp=fopen("txt","r");
if(fp==NULL )
   perror("错误是");
return 0;
}
```

容易错误的地方

1. 报错：Pthread_create未定义

- 答： 未加-lpthread 或 -lpthread未放在最后面
- 正确示例： **gcc test.c -o test –lpthread**
- 使用： **./test**

2.在Linux下用sleep(500)

· 答：在Linux C语言中 sleep的单位是秒(s)

· 例：sleep(5)停5秒

3.总是同一个线程在跑

· 答：sleep时间太短

容易错的地方——主线程传值

```
r(int j=0;j<5;j++){
 printf("j now is %d \n",j);
 if
thread_create(&phi[j],NULL,eat_think,&j)==0)
 {
    printf("create suc\n");
 }}
```

```
void *eat_think(void *arg){
        int p=*(int*)arg;
}
```