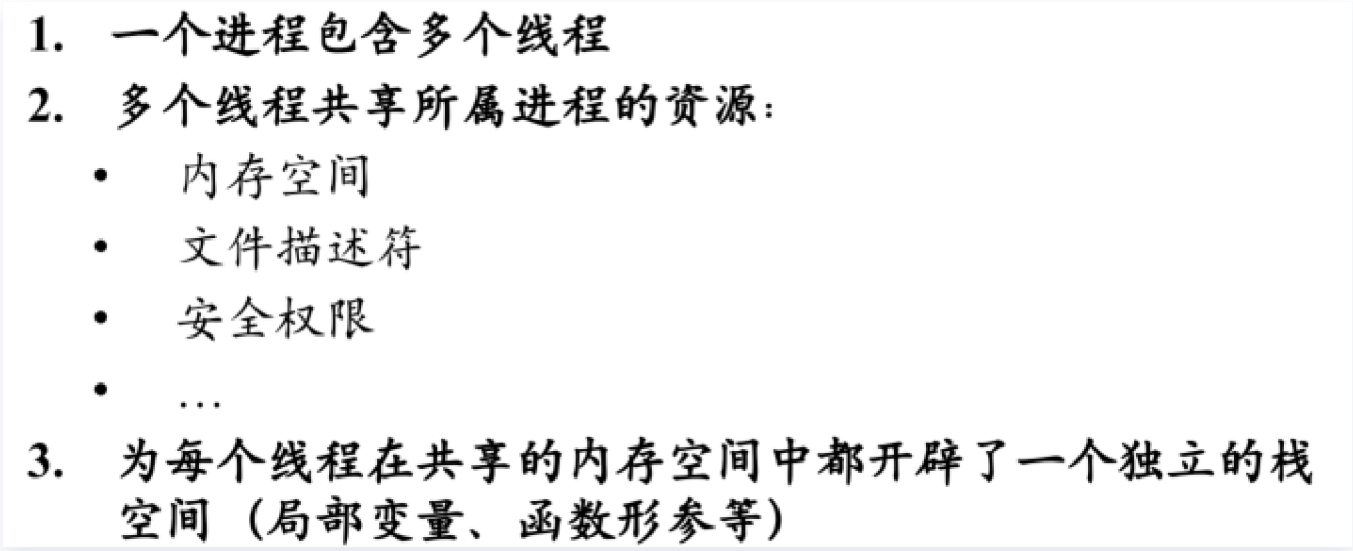
Linux 多线程编程

线程的基本概念



- 进程：程序执行和资源分配的基本单位；
- 线程：轻量级进程，处理器调度的基本单位；目的：加快上下文切换速度、节省系统资源
- 进程的数据区：数据段、代码段和堆栈段；
- 线程的数据区：线程控制表（记录相关执行状态和存储变量），但共享一个用户地址空间，即共享进程的资源和地址空间（一个线程对系统资源的操作都会对其他线程产生影响）
- 线程分类：
  - 用户级线程：解决上下文切换问题，调度算法以及过程由用户自行选择决定；
  - 核心级线程：不同进程中的线程按照同一相对优先调度方法进行调度；

线程的创建

1. 线程标识：一个pthread_t 类型的变量
2. 线程属性：一个pthread_attr_t 类型的结构体
3. 创建线程：确定调用该线程函数的入口点

int pthread_create(pthread_t* tid, pthread_attr_t *attr, void*(*start_routine)(void*), void *arg);

- tid：输出参数，用于返回所创建现成的标识；
- attr：用于设定线程属性，大多数情况下传NULL；
- start_routine：用于指定线程主函数
- arg：为线程主函数传递的参数

线程的退出

1. 三种退出方式：
   - 线程主函数执行完毕，自动退出；
   - 线程执行过程中调用了pthread_exit();(千万别调exit)
   - 其他线程利用pthread_cancel() 要求该线程强制退出
2. pthread_cancel(pthread_t tid)向目标线程发Cancel信号，但如何处理Cancel信号则由目标线程自己决定，或者忽略、或者立即终止、或者继续运行至Cancelation-point（取消点）
3. int pthread_setcancelstate(int state, int*oldstate);设置本线程对取消信号的反应。(CANCEL_ENABLE/DISABLE)
4. pthread_setcanceltype()函数设置取消类型(立即取消、或运行到下一个取消点)
5. pthread_setcancel()函数设置取消点；

- 线程调用 `pthread_exit()` 函数主动退出
  - `exit()` 函数用于使调用进程终止，在调用 `exit()` 后该进程中的所用线程都终止了，因此不能使用 `exit()` 函数；
  - `pthread_cancel()` 函数用于在别的线程中终止另一个线程的执行

线程资源的回收

1. 一个线程退出后其部分资源并不能被OS回收，必须等到其他线程（一般是主线程）获得其退出状态并最终回收剩余资源。
2. pthread_join()可以用于将当前线程挂起来等待指定线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。
3. int pthread_join(pthread_t tid, void **retval);
4. 线程也可以利用pthread_detach解除自己与所属进程之间的绑定。分离之后，线程结束，资源被全部回收。
5. 创建线程时也可以设定线程的分离属性。

- **phtread_create()函数语法：**

所需头文件	#include <pthread.h>
函数原型	int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))
函数传入值	thread：线程标识符
	attr：线程属性设置（其具体设置参见 6.4.3 小节），通常取为 NULL
	start_routine：线程函数的起始地址，是一个以指向 void 的指针作为参数和返回值的函数指针
	arg：传递给 start_routine 的参数
函数返回值	成功：0
	出错：返回错误码

- **phtread_exit()函数语法：**

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	retval：线程结束时的返回值，可由其他函数如 pthread_join()来获取

- **pthread_join()函数语法：**

所需头文件	#include <pthread.h>
函数原型	int pthread_join ((pthread_t th, void **thread_return))
函数传入值	th：等待线程的标识符
	thread_return：用户定义的指针，用来存储被等待线程结束时的返回值（不为 NULL 时）
函数返回值	成功：0
	出错：返回错误码

- **pthread_cancel()函数语法：**

所需头文件	#include <pthread.h>
函数原型	int pthread_cancel((pthread_t th)
函数传入值	th：要取消的线程的标识符
函数返回值	成功：0
	出错：返回错误码

- `thread_return`：被等待线程结束后，返回一些参数值给等待进程；

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREAD_NUMBER 3 /*线程数*/
#define REPEAT_NUMBER 5 /*每个线程中的小任务数*/
#define DELAY_TIME_LEVELS 10.0 /*小任务之间的最大时间间隔*/
void *thrd_func(void *arg)
{ /* 线程函数例程 */
    int thrd_num = (int)arg;
    int delay_time = 0;
    int count = 0;
    printf("Thread %d is starting\n", thrd_num);
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay = %d\n", thrd_num, count, delay_time);
    }
    printf("Thread %d finished\n", thrd_num);
    pthread_exit(NULL);
}
int main(void)
{
```

```
    pthread_t thread[THREAD_NUMBER];
    int no = 0, res;
    void * thrd_ret;
    srand(time(NULL));
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        /* 创建多线程 */
        res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
        if (res != 0)
        {
            printf("Create thread %d failed\n", no);
            exit(res);
        }
    }
    printf("Create treads success\n Waiting for threads to finish...\n");
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        /* 等待线程结束 */
        res = pthread_join(thread[no], &thrd_ret);
        if (!res)
        {
            printf("Thread %d joined\n", no);
        }
        else
        {
            printf("Thread %d join failed\n", no);
        }
    }
    return 0;
}
```

同步与互斥

互斥锁机制

在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。 若其他线程希望上锁一个已经被上锁的互斥锁， 则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。

互斥锁机制：

- 互斥锁初始化：pthread_mutex_init()
- 互斥锁上锁：pthread_mutex_lock()
- 互斥锁判断上锁：pthread_mutex_trylock()
- 互斥锁解锁：pthread_mutex_unlock()
- 消除互斥锁：pthread_mutex_destroy()

互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。

– pthread_mutex_init()函数语法：

所需头文件	#include <pthread.h>	
函数原型	int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)	
函数传入值	mutex：互斥锁	
函数传入值	Mutexattr	PTHREAD_MUTEX_INITIALIZER：创建快速互斥锁
		PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP：创建递归互斥锁
		PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP：创建检错互斥锁
函数返回值	成功：0	
	出错：返回错误码	

– pthread_mutex_lock()函数语法：

所需头文件	#include <pthread.h>
函数原型	int pthread_mutex_lock(pthread_mutex_t *mutex,) int pthread_mutex_trylock(pthread_mutex_t *mutex,) int pthread_mutex_unlock(pthread_mutex_t *mutex,) int pthread_mutex_destroy(pthread_mutex_t *mutex,)
函数传入值	mutex：互斥锁
函数返回值	成功：0
	出错：-1

```
/*thread_mutex.c*/
```

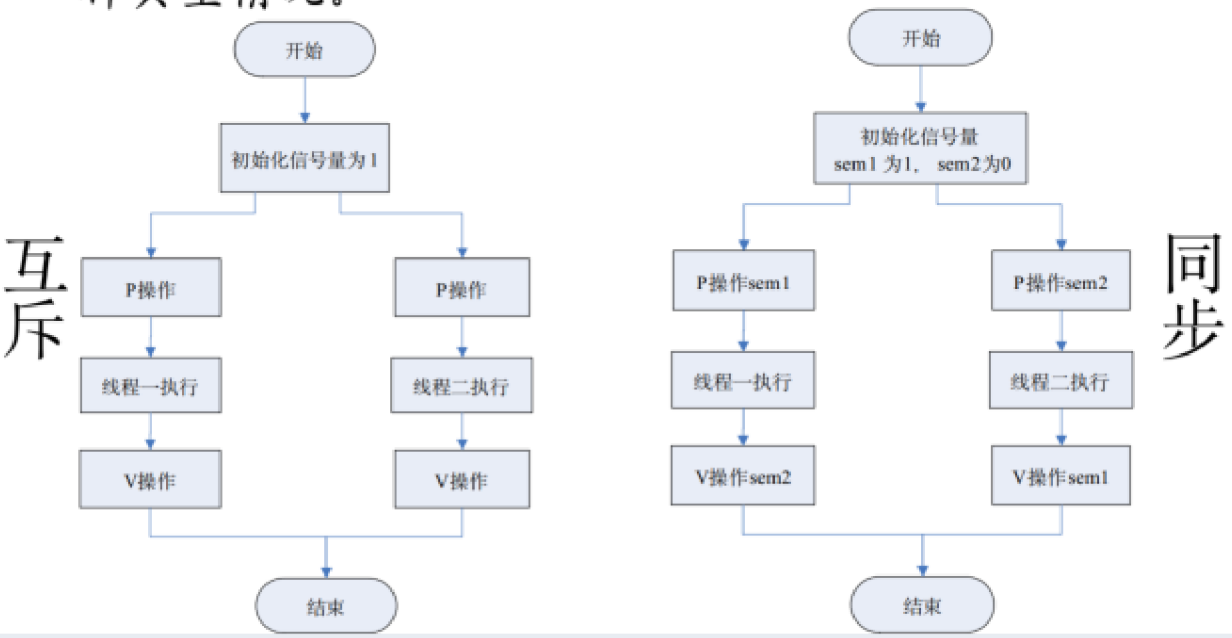
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREAD_NUMBER 3 /* 线程数 */
#define REPEAT_NUMBER 3 /* 每个线程的小任务数 */
#define DELAY_TIME_LEVELS 10.0 /*小任务之间的最大时间间隔*/
pthread_mutex_t mutex;
void *thrd_func(void *arg)
{
    int thrd_num = (int)arg;
    int delay_time = 0, count = 0;
    int res;
    /* 互斥锁上锁 */
    res = pthread_mutex_lock(&mutex);
    if (res)
    {
        printf("Thread %d lock failed\n", thrd_num);
        pthread_exit(NULL);
    }
    printf("Thread %d is starting\n", thrd_num);
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay = %d\n", thrd_num, count, delay_time);
    }
    printf("Thread %d finished\n", thrd_num);
    pthread_exit(NULL);
}
int main(void)
{
    pthread_t thread[THREAD_NUMBER];
    int no = 0, res;
    void * thrd_ret;

    srand(time(NULL));
    /* 互斥锁初始化 */
    pthread_mutex_init(&mutex, NULL);
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
        if (res != 0)
        {
            printf("Create thread %d failed\n", no);
```

```
            exit(res);
        }
    }
    printf("Create treads success\n Waiting for threads to finish...\n");
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        res = pthread_join(thread[no], &thrd_ret);
        if (!res)
        {
            printf("Thread %d joined\n", no);
        }
        else
        {
            printf("Thread %d join failed\n", no);
        }
        /* 互斥锁解锁 */
        pthread_mutex_unlock(&mutex);
    }
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

信号量机制

- sem_init()用于创建一个信号量，并初始化它的值。
- sem_wait()和sem_trywait()都相当于P操作，在信号量大于零时它们都能将信号量的值减一，两者的区别在于若信号量小于零时，sem_wait()将会阻塞进程，而sem_trywait()则会立即返回。
- sem_post()相当于V操作，它将信号量的值加一同时发出信号来唤醒等待的进程。
- sem_getvalue()用于得到信号量的值。
- sem_destroy()用于删除信号量。

- 信号量线程控制
  - PV原子操作主要用于进程或线程间的同步和互斥这两种典型情况。




```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define THREAD_NUMBER 3 /* 线程数 */
#define REPEAT_NUMBER 3 /* 每个线程中的小任务数 */
#define DELAY_TIME_LEVELS 10.0 /*小任务之间的最大时间间隔*/
sem_t sem[THREAD_NUMBER];
void *thrd_func(void *arg)
{
    int thrd_num = (int)arg;
    int delay_time = 0;
    int count = 0;
    /* 进行 P 操作 */
    sem_wait(&sem[thrd_num]);
    printf("Thread %d is starting\n", thrd_num);
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay = %d\n",
        thrd_num, count, delay_time);
    }
    printf("Thread %d finished\n", thrd_num);
```

```c
    pthread_exit(NULL);
}
int main(void)
{
    pthread_t thread[THREAD_NUMBER];
    int no = 0, res;
    void * thrd_ret;
    srand(time(NULL));
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        sem_init(&sem[no], 0, 0);
        res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
        if (res != 0)
        {
            printf("Create thread %d failed\n", no);
            exit(res);
        }
    }
    printf("Create treads success\n Waiting for threads to finish...\n");
    /* 对最后创建的线程的信号量进行 V 操作 */
    sem_post(&sem[THREAD_NUMBER - 1]);
    for (no = THREAD_NUMBER - 1; no >= 0; no--)
    {
        res = pthread_join(thread[no], &thrd_ret);
        if (!res)
        {
            printf("Thread %d joined\n", no);
        }
        else
        {
            printf("Thread %d join failed\n", no);
        }
        /* 进行 V 操作 */
        sem_post(&sem[(no + THREAD_NUMBER - 1) % THREAD_NUMBER]);
    }
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        /* 删除信号量 */
        sem_destroy(&sem[no]);
    }
    return 0;
}
```

线程的属性

线程常用属性包括：

1. 绑定属性
2. 分离属性
3. 堆栈地址和大小
4. 运行优先级
5. 系统默认的属性为非绑定、非分离、缺省1M的堆栈以及与父进程同样级别的优先级。

- **pthread_create()函数的第二个参数（pthread_attr_t *attr）表示线程的属性。如果该值设为NULL，就是采用默认属性**
- **绑定属性**
  - 绑定属性就是指一个用户线程固定地分配给一个内核线程，因为CPU时间片的调度是面向内核线程（也就是轻量级进程）的，因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。而与之对应的非绑定属性就是指用户线程和内核线程的关系不是始终固定的，而是由系统来控制分配的。
- **分离属性**
  - 分离属性是用来决定一个线程以什么样的方式来终止自己。

- **pthread_attr_init()**
  - pthread_attr_init()函数对属性进行初始化
  - pthread_attr_init()函数语法：

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_init(pthread_attr_t *attr)
函数传入值	attr：线程属性结构指针
函数返回值	成功：0
	出错：返回错误码

- **pthread_attr_setscope()**
  - pthread_attr_setscope()函数设置线程绑定属性
    pthread_attr_setscope()函数语法：

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int scope)	
函数传入值	attr：线程属性结构指针	
	scope	PTHREAD_SCOPE_SYSTEM：绑定
		PTHREAD_SCOPE_PROCESS：非绑定
函数返回值	成功：0	
	出错：-1	

- **pthread_attr_setdetachstate()**
  - pthread_attr_setdetachstate()函数设置线程分离属性
  - pthread_attr_setdetachstate()函数语法：

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int detachstate)	
函数传入值	attr：线程属性	
	detachstate	PTHREAD_CREATE_DETACHED：分离
		PTHREAD _CREATE_JOINABLE：非分离
函数返回值	成功：0	
	出错：返回错误码	

- **pthread_attr_setschedparam()**
  - pthread_attr_setschedparam()函数设置线程优先级
  - pthread_attr_setschedparam()函数语法：

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_setschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr：线程属性结构指针
	param：线程优先级
函数返回值	成功：0
	出错：返回错误码

- **pthread_attr_getschedparam()**
  - pthread_attr_getschedparam()函数获得线程优先级
  - pthread_attr_getschedparam()函数语法：

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_getschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr：线程属性结构指针
	param：线程优先级
函数返回值	成功：0
	出错：返回错误码

生产者消费者问题

问题要求

“生产者—消费者”问题描述如下。
有一个有限缓冲区和两个线程：生产者和消费者。他们分别不停地把产品放入缓冲区和从缓冲区中拿走产品。一个生产者在缓冲区满的时候必须等待，一个消费者在缓冲区空的时候也必须等待。另外，因为缓冲区是临界资源，所以生产者和消费者之间必须互斥执行。它们之间的关系如图 9.4 所示。

图 9.4　生产者消费者问题描述

这里要求使用有名管道来模拟有限缓冲区，并且使用信号量来解决“生产者—消费者”问题中的同步和互斥问题。

同步互斥关系分析

- 即存在互斥也存在同步关系
  - 生产者与消费者存取商品需要同步；
  - 使用缓冲区需要互斥进行；
- 信号量设置
  - 同步信号量：`avail` —初始值为 N、 `full` —初始值为 0
  - 互斥信号量：`mutex` —初始值为 1
- 流程图

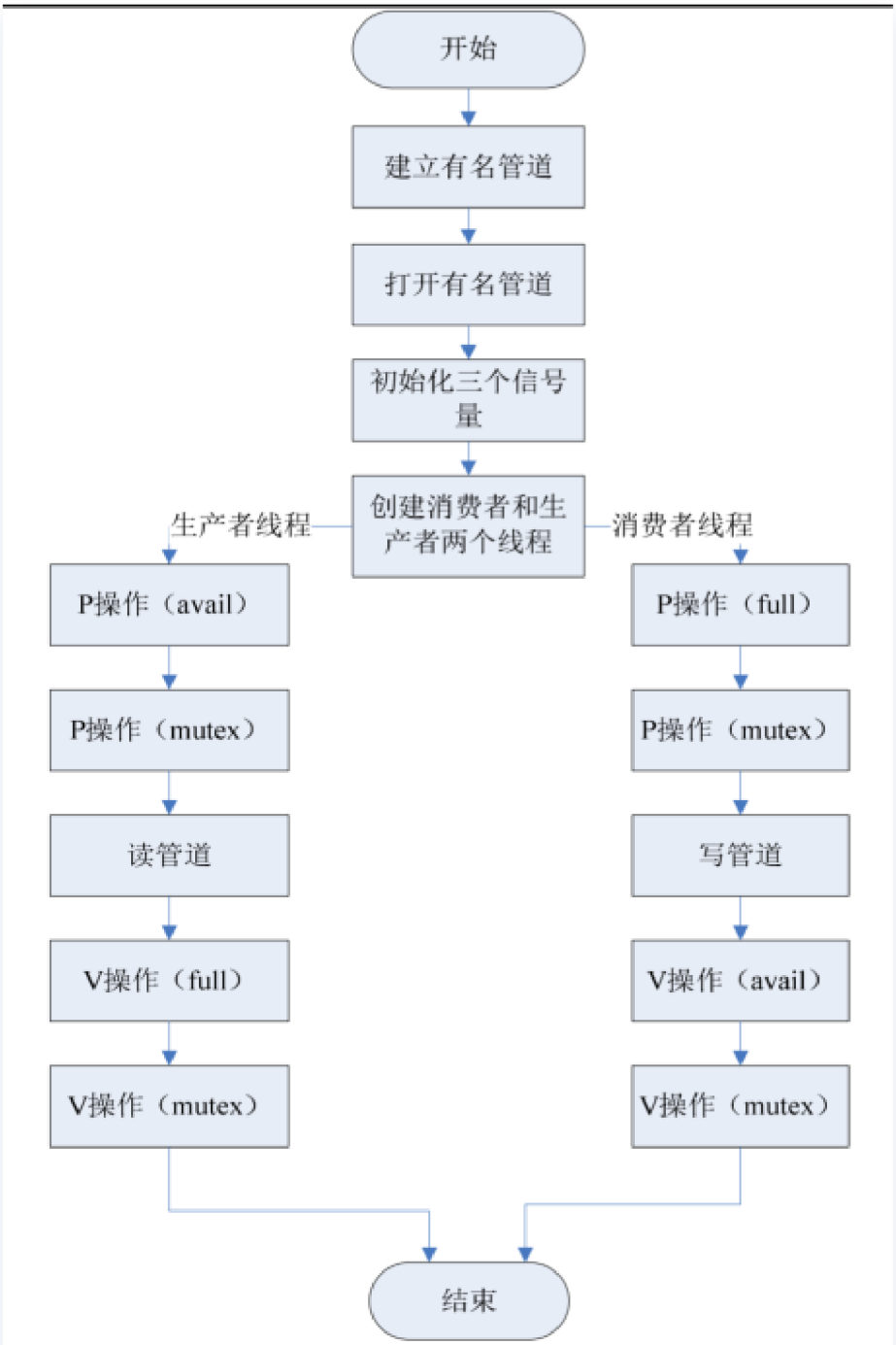


图 9.5 “生产者—消费者”实验流程图

```c
/*producer-customer.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>
#include <sys/ipc.h>
```

```c
#define MYFIFO "myfifo" /* 缓冲区有名管道的名字 */
#define BUFFER_SIZE 3 /* 缓冲区的单元数 */
#define UNIT_SIZE 5 /* 每个单元的大小 */
#define RUN_TIME 30 /* 运行时间 */
#define DELAY_TIME_LEVELS 5.0 /* 周期的最大值 */
int fd;
time_t end_time;
sem_t mutex, full, avail; /* 3 个信号量 */

/*生产者线程*/
void *producer(void *arg)
{
    int real_write;
    int delay_time = 0;
    while(time(NULL) < end_time)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX) / 2.0) + 1;
        sleep(delay_time);
        /*P 操作信号量 avail 和 mutex*/
        sem_wait(&avail);
        sem_wait(&mutex);
        printf("\nProducer: delay = %d\n", delay_time);
        /*生产者写入数据*/
        if ((real_write = write(fd, "hello", UNIT_SIZE)) == -1)
        {
            if(errno == EAGAIN)
            {
                printf("The FIFO has not been read yet.Please try later\n");
            }
        }
        else
        {
            printf("Write %d to the FIFO\n", real_write);
        }
        /*V 操作信号量 full 和 mutex*/
        sem_post(&full);
        sem_post(&mutex);
    }
    pthread_exit(NULL);
}
/* 消费者线程*/
void *customer(void *arg)
{
    unsigned char read_buffer[UNIT_SIZE];
```

```
    int real_read;
    int delay_time;
    while(time(NULL) < end_time)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        /*P 操作信号量 full 和 mutex*/
        sem_wait(&full);
        sem_wait(&mutex);
        memset(read_buffer, 0, UNIT_SIZE);
        printf("\nCustomer: delay = %d\n", delay_time);
        if ((real_read = read(fd, read_buffer, UNIT_SIZE)) == -1)
        {
            if (errno == EAGAIN)
            {
            printf("No data yet\n");
            }
        }
        printf("Read %s from FIFO\n", read_buffer);
        /*V 操作信号量 avail 和 mutex*/
        sem_post(&avail);
        sem_post(&mutex);
    }
    pthread_exit(NULL);
}
int main()
{
    pthread_t thrd_prd_id, thrd_cst_id;
    pthread_t mon_th_id;
    int ret;
    srand(time(NULL));
    end_time = time(NULL) + RUN_TIME;
    /*创建有名管道*/
    if((mkfifo(MYFIFO, O_CREAT|O_EXCL) < 0) && (errno != EEXIST))
    {
        printf("Cannot create fifo\n");
        return errno;
    }
    /*打开管道*/
    fd = open(MYFIFO, O_RDWR);
    if (fd == -1)
    {
        printf("Open fifo error\n");
        return fd;
```

```
    }
    /*初始化互斥信号量为 1*/
    ret = sem_init(&mutex, 0, 1);
    /*初始化 avail 信号量为 N*/
    ret += sem_init(&avail, 0, BUFFER_SIZE);
    /*初始化 full 信号量为 0*/
    ret += sem_init(&full, 0, 0);
    if (ret != 0)
    {
        printf("Any semaphore initialization failed\n");
        return ret;
    }
    /*创建两个线程*/
    ret = pthread_create(&thrd_prd_id, NULL, producer, NULL);
    if (ret != 0)
    {
        printf("Create producer thread error\n");
        return ret;
    }
    ret = pthread_create(&thrd_cst_id, NULL, customer, NULL);
    if(ret != 0)
    {
        printf("Create customer thread error\n");
        return ret;
    }
    pthread_join(thrd_prd_id, NULL);
    pthread_join(thrd_cst_id, NULL);
    close(fd);
    unlink(MYFIFO);
    return 0;
}
```