

重庆理工大学/电气学院

CHONGQING UNIVERSITY OF TECHNOLOGY

# 嵌入式Linux系统开发教程

## —基于ARM处理器通用平台 (arm9-arm11- cortexA系列)

丛超

2025年5月





1

Linux多线程概念

2

线程函数

3

4



01

# Linux多线程概念

早期操作系统中并没有线程这一概念，无论是分配资源还是调度分派，都以进程为最小单位，随着计算机技术的发展，人们逐渐发现了进程作为系统调度分派单位时存在的一些弊端，人们意识到操作系统应能调度一个更小的单位，以减少消耗，提高效率，由此，线程应运而生。

与进程不同，线程（Thread）是系统调度分派的最小单位，与进程相比，线程没有独立的地址空间，多个线程共享一段地址空间，因此线程消耗更少的内存资源，线程间通信也更为方便，有时线程也被称为轻量级进程（Light Weight Process, LWP）。



Linux系统中的线程借助进程机制实现，线程与进程联系密切：

- 进程可以蜕变成线程，当在一个进程中创建一个线程时，原有的进程就会变成线程，两个线程共用一段地址空间；
- 线程又被称为轻量级进程，线程的TCB（Thread Control Block，线程控制块）与进程的PCB相同，因此也可以将TCB视为PCB；
- 对内核而言，线程与进程没有区别，cpu会为每个线程与进程分配时间片，并通过PCB来调度不同的线程和进程。

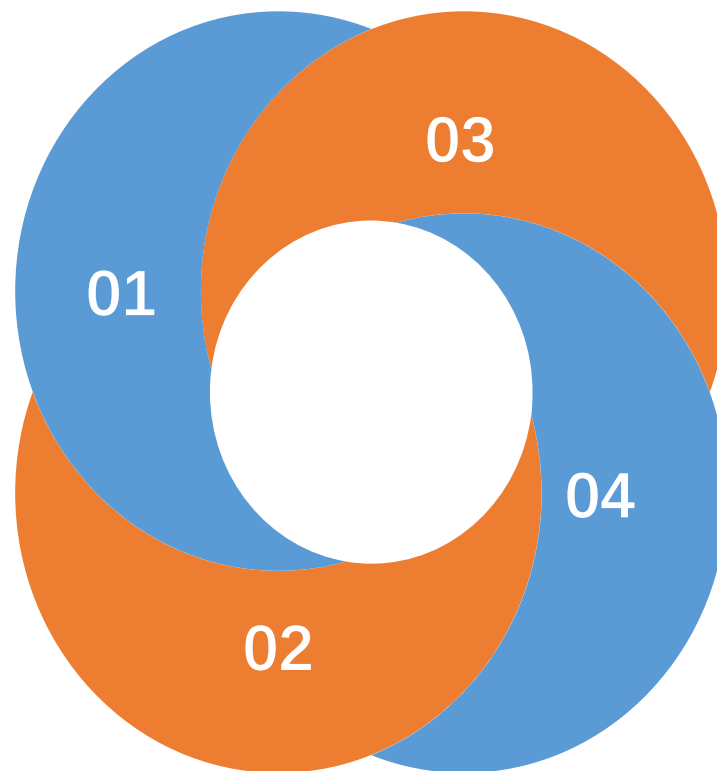
线程的生命周期包括：就绪、运行、阻塞、终止。

(1) 就绪

线程能够运行，在等待可用的处理器，可能刚刚启动，或者刚刚从阻塞中恢复，或者被其它线程抢占。

(2) 运行

线程正在运行。在单处理器系统中，只能有一个线程处于运行状态，在多处理器系统中，可能有多个线程处于运行态。



(3) 阻塞

线程由于等待“处理器”外的其它条件而无法运行，如：条件变量的改变，加锁互斥量或者等待I/O操作结束。

(4) 终止

线程从线程函数中返回，或者调用 `pthread_exit`，或者被取消，或随进程的终止而终止。线程终止后会完成所有的清理工作。

## 8.2.1线程创建

表头文件

```
#include<pthread.h>
```

定义函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine)  
(void *), void *arg);
```

函数说明 参数thread表示待创建线程的线程id指针；参数attr用于设置待创建线程的属性；参数start\_routine是一个函数指针，该函数为待创建线程的执行函数，线程创建成功后将会执行该函数中的代码；参数arg为要传给线程执行函数的参数。

返回值 线程创建成功返回0，发生错误时返回错误码。

因为pthread的库不是linux系统的库，所以在进行编译的时候要加上-lpthread，如：

```
# gcc filename -lpthread
```

**Example8.2.1-1:** 使用pthread\_create()函数创建线程，并使原线程与新线程分别打印自己的线程id

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
void *tfn(void *arg)
{
    printf("tfn--pid=%d,tid=%lu\n", getpid(), pthread_self());
    return (void*)0;
}
int main()
{
    pthread_t tid;
    printf("main--pid=%d,tid=%lu\n", getpid(), pthread_self());
    int ret = pthread_create(&tid, NULL, tfn, NULL);
    if (ret != 0){
        fprintf(stderr, "pthread_create error:%s\n", strerror(ret));
        exit(1);
    }
    sleep(1);
    return 0;
}
```



### Example8.2.1-2: 交替创建线程1与线程2

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <unistd.h>
4.  void *myThread1(void)
5.  {
6.      int i;
7.      for (i=0;i<100;i++)
8.      {
9.          printf("This is the 1st pthread, created by ziecekey.\n");
10.         sleep(1); //线程休眠1秒后继续运行
11.     }
12. }
13. void *myThread2(void)
14. {
15.     int i;
16.     for (i=0;i<100;i++)
17.     {
18.         printf("This is the 2st pthread, created by ziecekey.\n");
19.         sleep(1);
```

```
20.     }
21. }
22. int main()
23. {
24.     int i=0,ret=0;
25.     pthread_t id1,id2;
26.     ret = pthread_create(&id1,NULL,(void*)myThread1,NULL); //创建线程1
27.     if (ret)
28.     {
29.         printf("Create pthread error!\n");
30.         return 1;
31.     }
32.     ret = pthread_create(&id2,NULL,(void*)myThread2,NULL); //创建线程2
33.     if (ret)
34.     {
35.         printf("Create pthread error!\n");
36.         return 1;
37.     }
38.     pthread_join(id1, NULL); //挂起线程1
39.     pthread_join(id2, NULL); //挂起线程2
40.     return 0;
41. }
```

## 8.2.2线程退出

表头文件

```
#include <pthread.h>
```

定义函数

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

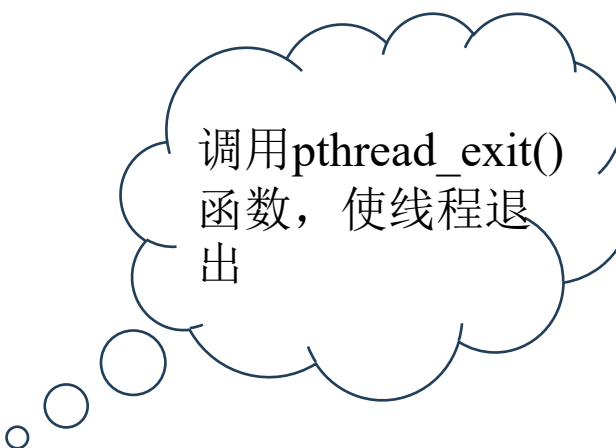
函数说明 用于终止调用线程。rval\_ptr是线程结束时的返回值，可由其他函数如pthread\_join()来获取。

线程的正常退出方式有：

- (1) 线程从启动例程中返回；
- (2) 线程可以被另一个进程终止；
- (3) 线程自己调用pthread\_exit函数。

**Example 8.2.2-1:**

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #include <unistd.h>
4. #include <stdlib.h>
5. void *tfn(void *arg)
6. {
7.     long int i;
8.     i = (long int)arg;
9.     if (i == 2)
10.         pthread_exit(NULL);
11.     sleep(i);
12.     printf("I'm %ldth thread, Thread_ID = %lu\n", i
13. + 1, pthread_self());
14.     return NULL;
15. }
```



调用pthread\_exit()  
函数，使线程退出

//通过i来区别每个线程

```
15. int main(int argc, char *argv[])
16. {
17.     long int n = 5, i;
18.     pthread_t tid;
19.     if (argc == 2)
20.         n = atoi(argv[1]);
21.     for (i = 0; i < n; i++) {
22.         //将i转换为指针，在tfn中再强转回整形
23.         pthread_create(&tid, NULL, tfn, (void *)i);
24.     }
25.     sleep(n);
26.     printf("I am main, I'm a thread!\n"
27.         "main_thread_ID = %lu\n", pthread_self());
28.     return 0;
29. }
```

### 8.2.3线程等待

`pthread_join`是一个线程阻塞函数，调用后，则一直等待指定的线程结束才函数返回，被等待线程的资源就会被收回。

表头文件

```
#include <pthread.h>
```

定义函数

```
int pthread_join(pthread_t tid, void **rval_ptr)
```

函数说明 阻塞调用线程，直到指定的线程终止。`tid` 是等待退出的线程id；`rval_ptr`是用户定义的指针，用来存储被等待线程结束时的返回值（不为NULL时）。

**Example8.2.3-1: 随进程的终止而终止**

```
#include <pthread.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *run(void *buf){
    int i=0;
    while(i<10){
printf("-----pthread id =%ld,i=%d\n", pthread_self(),i);
        usleep(1000000);
        i++;
    }
}
```

```
int main(int argc,char **argv){
    pthread_t tid;
    pthread_create(&tid,NULL,run,NULL); //创建线程
    sleep(1);
}
```

- **Example 8.2.3-2:** 从线程函数中返回而终止

```
#include <pthread.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *run(void *buf){
    int i=0;
    while(i<10){
        printf("-----pthread id =%ld,i=%d\n", pthread_self(),i);
        usleep(1000000);
        if (i==5) return NULL; //return的作用是结束函数
        i++;
    }
}
```

```
int main(int argc,char **argv){
    pthread_t tid;
    pthread_create(&tid,NULL,run,NULL); //创建线程
    pthread_join(tid,NULL);
}
```





### 8.2.4线程标识获取

表头文件

```
#include <pthread.h>
```

定义函数

```
pthread_t pthread_self(void)
```

函数说明 获取调用线程的 thread identifier。

**Example 8.2.4-1:**

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> /*getpid()*/
void *create(void *arg)
{
    printf("New thread .... \n");
    printf("This thread's id is %u \n",(unsigned int)pthread_self());
    printf("The process pid is %d\n",getpid());
    return (void *)0;
}
```

```
int main(int argc,char *argv[])
{
    pthread_t tid;
    int error;
    printf("Main thread is starting ... \n");
    error = pthread_create(&tid, NULL, create,NULL);
    if(error)
    {
        printf("thread is not created ... \n");
        return -1;
    }
    printf("The main process's pid is %d \n",getpid());
    sleep(1);
    return 0;
}
```

## 8.2.5线程清除

线程终止有两种情况：正常终止和非正常终止。线程主动调用`pthread_exit`或者从线程函数中`return`都将使线程正常退出，这是可预见的退出方式；非正常终止是线程在其他线程的干预下，或者由于自身运行出错（比如访问非法地址）而退出，这种退出方式是不可预见的。

在POSIX线程API中提供了一个`pthread_cleanup_push()/pthread_cleanup_pop()`函数对用于自动释放资源。从`pthread_cleanup_push`的调用点到`pthread_cleanup_pop`之间的程序段中的终止动作（包括调用`pthread_exit()`和异常终止，不包括`return`）都将执行`pthread_cleanup_push()`所指定的清理函数。



# 1. pthread\_cleanup\_push

表头文件

```
#include <pthread.h>
```

定义函数

```
void pthread_cleanup_push(void (*rtn)(void *), void *arg)
```

函数说明 将清除函数压入清除栈。rtn是清除函数； arg是清除函数的参数。

## 2. pthread\_cleanup\_pop

表头文件

```
#include <pthread.h>
```

定义函数

```
void pthread_cleanup_pop(int execute)
```

函数说明 将清除函数弹出清除栈，执行到pthread\_cleanup\_pop()时，参数execute决定是否在弹出清理函数的同时执行该函数。execute非0时，执行；execute为0时，不执行。

### 3.thread\_cancel

表头文件

```
#include <pthread.h>
```

定义函数

```
int pthread_cancel(pthread_t thread);
```

参数: thread指定要退出的线程ID。

函数说明 取消线程，该函数在其它线程中调用，用来强行杀死指定的线程。



**Example8.2.5-2: 线程清除**

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *run(void *buf){
    int i=0;
    while(1){
        printf("-----pthread id =%lu,i=%d\n",
pthread_self(),i);
        usleep(1000000);
        i++;
    }
    return NULL;
}
```

```
int main(int argc,char **argv){
    pthread_t tid;
    pthread_create(&tid,NULL,run,NULL); //创建线程
    sleep(5);
    pthread_cancel(tid);
    pthread_join(tid,NULL);
}
```



02

线程函数

### 8.3.1 线程函数传参

#### 传递简单的数据指针

在例Example8.2.1-1中，函数`pthread_create()`中参数`arg`被传递`tfn`到函数中。其中`tfn`的形参为`void*`类型，该类型为任意类型的指针。所以任意一种类型都可以通过地址将数据传送到线程函数中。

**Example8.3.1-1:** 传递字符串指针

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *run(void *buf){
    char *str=(char*)buf;
    printf("-----pthread id=%lu,%s\n", pthread_self(),str);
    int i=0;
    while(1){
        printf("-----pthread id =%lu,i=%d\n", pthread_self(),i);
        usleep(1000000);
        i++;
    }
}
```

```
int main(int argc,char **argv){
    pthread_t tid;
    char buf[256]="abcdefg";
    pthread_create(&tid,NULL,run,buf);
    void *val; //val是变量,指针变量就是整型变量
    pthread_join(tid,&val);
    printf("%d\n",val);
}
```

数组做实参时，传入的是数组的首地址，即传入多个相同类型数据的首地址；

**Example 8.3.1-2: 多线程中结构体作参数**

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
struct STU{ //结构体STU声明
    int runn;
    int num;
    char name[32];
};
void * run(void *buf){ //线程函数
    struct STU *p=buf;
    printf("id=%lu,num=%d,name=%s\n",pthread_self(),p->num,p-
>name);
    int i=0;
    while(i<p->runn){
        printf("-----id=%lu,i=%d\n",pthread_self(),i++);
        usleep(1000000);
    }
}
```

```
int main(int argc,char **argv){
    struct STU st[]={
        {10,12,"aaaa"},
        {20,23,"bbbb"}
    };
    pthread_t pid[2];
    int i;
    for(i=0;i<2;i++){
        pthread_create(&pid[i],NULL,run,&st[i]);
    }
    printf("%lu,%lu\n",pid[0],pid[1]);
    pthread_join(pid[0],NULL);
    pthread_join(pid[1],NULL);
    return 0;
}
```

结构体做实参，传入的是结构体的地址，  
即传入多个不同类型数据的结构地址。

### 8.3.2 绑定属性

关于线程的绑定，牵涉到另外一个概念：轻进程(LWP: Light Weight Process)。轻进程可以理解为内核线程，它位于用户层和系统层之间。系统对线程资源的分配、对线程的控制是通过轻进程来实现的，一个轻进程可以控制一个或多个线程。默认状况下，启动多少轻进程、哪些轻进程来控制哪些线程是由系统来控制的，这种状况即称为非绑定的。

绑定状况下，则顾名思义，即某个线程固定的"绑"在一个轻进程之上。被绑定的线程具有较高的响应速度，这是因为CPU时间片的调度是面向轻进程的，绑定的线程可以保证在需要的时候它总有一个轻进程可用。通过设置被绑定的轻进程的优先级和调度级可以使得绑定的线程满足诸如实时反应之类的要求。





设置线程绑定状态的函数为pthread\_attr\_setscope。

函数原型为：

```
int pthread_attr_setscope(pthread_attr_t *tattr, int scope);
```

它有两个参数，第一个是指向属性结构的指针，第二个是绑定类型，常用结构为：

```
#include <pthread.h>
```

```
pthread_attr_t tattr;
```

```
int ret;
```

```
/*绑定线程*/
```

```
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
```

```
/*非绑定线程*/
```

```
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

绑定类型有两个取值：

(1) 绑定：PTHREAD\_SCOPE\_SYSTEM

(2) 非绑定：PTHREAD\_SCOPE\_PROCESS。

返回值：pthread\_attr\_setscope()成功完成后将返回0。其他任何返回值都表示出现了错误。

下面程序段使用了三个函数调用：用于初始化属性的调用、用于根据缺省属性设置所有变体的调用，以及用于创建pthreads的调用。

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* 缺省属性初始化 */
ret = pthread_attr_init (&tattr);

/* 边界设置 */
ret = pthread_attr_setscope(&tattr,PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

### 8.3.3 分离属性

线程的分离状态决定一个线程以什么样的方式来终止自己。线程的默认属性为非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当 `pthread_join()` 函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。而分离线程不是这样子的，它没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。



设置线程分离状态的函数为pthread\_attr\_setdetachstate。

函数原型为:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr, int detachstate);
```

它有两个参数，第一个是指向属性结构的指针，第二个是分离类型，常用结构为:

```
#include <pthread.h>
```

```
pthread_attr_t tattr;
```

```
int ret;
```

```
/* 设置线程分离状态 */
```

```
ret=pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

分离参数可选为：

(1) 分离线程： PTHREAD\_CREATE\_DETACHED

(2) 非分离线程： PTHREAD\_CREATE\_JOINABLE

返回值：成功完成后将返回0。其他任何返回值都表示出现了错误。

非分离线程在终止后，必须要有一个线程用join来等待它。否则不会释放该线程的资源以供新线程使用，而这通常会导致内存泄漏。因此如果不希望线程被等待，请将该线程作为分离线程来创建。创建分离线程常用程序代码如下：

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;
/* 缺省属性初始化 */
ret = pthread_attr_init (&tattr);
ret = pthread_attr_setdetachstate (&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create (&tid,&tattr,start_routine,arg);
```

### 8.3.4 优先级属性

结构体中包含一个成员 `sched_priority`，该成员是一个整型变量，代表线程的优先级。用函数 `pthread_attr_getschedparam` 和函数 `pthread_attr_setschedparam` 进行存放，`pthread_attr_getschedparam` 将返回由 `pthread_attr_setschedparam()` 定义的调度参数。

`pthread_attr_setschedparam` 函数原型为：

```
int pthread_attr_setschedparam(pthread_attr_t *tattr, const struct sched_param *param);
```



常用结构为：

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;
newprio = 30;
/*设置优先级，其他不变*/
param.sched_priority = newprio;
/* 设置新的调度参数 */
ret = pthread_attr_setschedparam (&tattr, &param);
```

调度参数是在param结构中定义的。仅支持优先级参数。新创建的线程使用此优先级运行。



pthread\_attr\_getschedparam函数原型为:

```
int pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param);
```

常用结构为:

```
#include <pthread.h>
```

```
pthread_attr_t attr;
```

```
struct sched_param param;
```

```
int ret;
```

```
/* 获取现有的调度参数*/
```

```
ret = pthread_attr_getschedparam (&tattr,&param);
```



使用指定的优先级创建线程：创建线程之前，可以设置优先级属性。将使用在`sched_param`结构中指定的新优先级创建子线程。此结构还包含其他调度信息。创建子线程时建议执行以下操作：获取现有参数、更改优先级、创建子线程、恢复原始优先级。

**Example8.3.4-1: 创建优先级为50的线程**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
void * run(void *buf){
    int i=0;
    while(1){
        printf("-----id=%lu,i=%d\n",pthread_self(),i++);
        usleep(1000000);
    }
}
```

```
int main(int argc,char **argv){
    pthread_t pid;
    pthread_attr_t attr;
    struct sched_param param;
    //----设置线程属性----
    pthread_attr_init(&attr); //初始化线程属性
    param.sched_priority=50;
    pthread_attr_setschedparam(&attr,&param); //设置优先级
    //----创建线程----
    pthread_create(&pid,&attr,run,NULL);
    //----销毁线程属性----
    //----等待线程结束----
    pthread_join(pid,NULL);
    return 0;
}
```

### 8.3.5 线程栈属性

线程中有属于该线程的栈，用于存储线程的私有数据，用户可以通过Linux系统中的系统调用，对这个栈的地址、栈的大小以及栈末尾警戒区的大小等进行设置，其中栈末尾警戒区用于防止栈溢出时栈中数据覆盖附近内存空间中存储的数据。

一般情况下使用默认设置即可，但是当对效率要求较高，或线程调用的函数中局部变量较多、函数调用层次较深时，可以从实际情况出发，修改栈的容量。

Linux系统中用于修改和获取线程栈空间大小的函数为pthread\_attr\_setstacksize()和pthread\_attr\_getstacksize(), 这两个函数的声明分别如下:

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

```
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

参数attr代表线程属性; stacksize代表栈空间大小。若函数调用成功则返回0, 否则返回errno。

Linux中也提供了用于设置和获取栈地址、栈末尾警戒区大小的函数，它们的函数声明分别如下：

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

```
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

```
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

```
int pthread_attr_getguardsize(pthread_attr_t *attr, size_t *guardsize);
```

当改变栈地址属性时，栈警戒区大小通常会被清零。若函数调用成功则返回0，否则返回errno。

Linux系统中还提供了pthread\_attr\_setstack()函数和pthread\_attr\_getstack()函数，这两个函数可以在一次调用中设置或获取线程属性中的栈地址与栈容量，它们的函数声明分别如下：

```
int pthread_attr_setstack(pthread_attr_t *attr,void *stackaddr, size_t stacksize);  
int  pthread_attr_getstack(pthread_attr_t  *attr,void  **stackaddr,  size_t  
*stacksize);
```

其中的参数attr、stackaddr、stacksize分别代表线程属性、栈空间地址、栈空间容量。若函数调用成功则返回0，否则返回errno。



### 8.3.6 线程的互斥

线程间的互斥是为了避免对共享资源或临界资源的同时使用，从而避免因而产生的不可预料的后果。临界资源一次只能被一个线程使用。线程互斥关系是由于对共享资源的竞争而产生的间接制约。

#### 1. 互斥锁

假设各个线程向同一个文件顺序写入数据，最后得到的结果一定是灾难性的。互斥锁用来保证一段时间内只有一个线程在执行一段代码，实现了对一个共享资源的访问进行排队等候。互斥锁是通过互斥锁变量来对访问共享资源排队访问。

## 2.互斥量

是pthread\_mutex\_t类型的变量。互斥量有两种状态:lock(上锁)和unlock(解锁), 当对一个互斥量加锁后, 任何其它试图访问互斥量的线程都会被阻塞, 直到当前线程释放互斥量上的锁。如果释放互斥量上的锁后, 有多个阻塞线程, 这些线程只能按一定的顺序有一个得到互斥量的访问权限完成对共享资源的访问后, 要对互斥量进行解锁, 否则其它线程将一直处于阻塞状态。



### 3.操作函数:

使用互斥锁实现线程同步时主要包括四个步骤：初始化互斥锁、上锁、解锁、销毁互斥锁。Linux系统中提供了一组与互斥锁相关的系统调用，分别为：pthread\_mutex\_init()、pthread\_mutex\_lock()、pthread\_mutex\_unlock()、pthread\_mutex\_destroy()。

### (1) 互斥锁的初始化

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

参数：mutex为互斥量，由pthread\_mutex\_init调用后填写默认值；attr为属性，通常默认NULL。

### (2) 上锁

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

参数：mutex为待锁定的互斥量。

### (3) 解锁

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

参数：mutex为待解锁的互斥量。若函数调用成功则返回0，否则返回errno，其中常见的errno有两个，分别为EBUSY和EAGAIN，它们代表的含义如下：

- (1) EBUSY：参数mutex指向的互斥锁已锁定；
- (2) EAGAIN：超过互斥锁递归锁定的最大次数。

原则上，已上锁则能再上锁，上了锁必须解锁，否则称为死锁。

#### (4) 判断是否上锁

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

参数: mutex表示待锁定的互斥量。

返回值: 0代表已上锁, 非零表示未上锁。

#### (5) 销毁互斥锁

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

参数: mutex表示待销毁的互斥量。

**Example 8.3.6-1:** 在两个线程函数中使用互斥锁

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <unistd.h>
4.      char str[1024];
5.      pthread_mutex_t mutex; //定义互斥锁
6.      void *run1(void *buf){
7.          while(1){
8.              pthread_mutex_lock(&mutex); //加锁
9.              sprintf(str,"run-----1");
10.             printf("%s\n",str);
11.             sleep(5);
12.             pthread_mutex_unlock(&mutex); //解锁
13.             usleep(1);
14.         }
15.     }
16.     void *run2(void *buf){
17.     while(1){
18.         pthread_mutex_lock(&mutex); //加锁
19.         sprintf(str,"run-----2");
20.         printf("%s\n",str);
21.         sleep(2);
```

```
22.     pthread_mutex_unlock(&mutex); //解锁
23.         usleep(1);
24.     }
25. }
26. int main(int argc,char **argv){
27.     pthread_mutex_init(&mutex,NULL); //初始化mutex
28.     pthread_t tid1,tid2;
29.     pthread_create(&tid1,NULL,run1,NULL);
30.     pthread_create(&tid2,NULL,run2,NULL);
31.     pthread_join(tid1,NULL);
32.     pthread_join(tid2,NULL);
33.     pthread_mutex_destroy(&mutex); //销毁锁
34. }
```

### 8.3.7线程的同步

#### 1.条件变量

条件变量就是一个变量，用于线程等待某件事情的发生，当等待事件发生时，被等待的线程和事件一起继续执行。等待的线程处于休眠状态，直到另一个线程给它唤醒，才开始活动，条件变量用于唤醒线程。

使用条件变量控制线程同步时，线程访问共享资源的前提是程序中设置的条件变量得到满足。条件变量不会对共享资源加锁，但会使线程阻塞，若线程不满足条件变量规定的条件，就会进入阻塞状态直到条件满足。



条件变量的使用分为以下四个步骤：

- (1) 初始化条件变量；
- (2) 等待条件变量满足；
- (3) 唤醒阻塞线程；
- (4) 条件变量销毁。

## 2.操作函数

### (1) 条件变量初始化函数

```
int pthread_cond_init(pthread_cond_t *restrict cond,const pthread_condattr_t  
*restrict attr);
```

参数：cond是条件变量指针，通过该函数实现条件变量赋初值；attr代表条件变量的属性，通常默认NULL，表示使用默认属性初始化条件变量，其默认值为PTHREAD\_PROCESS\_PRIVATE，表示当前进程中的线程共用此条件变量；也可将attr设置为PTHREAD\_PROCESS\_SHARED，表示多个进程间的线程共用条件变量。

## (2) 线程同步等待函数(睡眠函数)

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
```

参数: cond为条件变量; mutex为互斥锁。

说明: 哪一个线程执行pthread\_cond\_wait, 哪一个线程就开始睡眠, 在睡眠时同时先解开互斥锁, 好让其它线程可以继续执行。



### (3) 发送条件信号（唤醒函数）

```
int pthread_cond_signal(pthread_cond_t *cond);
```

参数：cond为条件变量。

说明：在另一个线程中使用，当某线程符合某种条件时，用于唤醒其它线程，让其它线程同步运行。其它线程被唤醒后，马上开始加锁，如果此时锁处于锁定状态，则等待被解锁后向下执行代码。

#### (4) 条件变量销毁

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

需要注意的是，只有当没有线程在等待参数cond指定的条件变量时，才可以销毁条件变量，否则该函数会返回EBUSY。

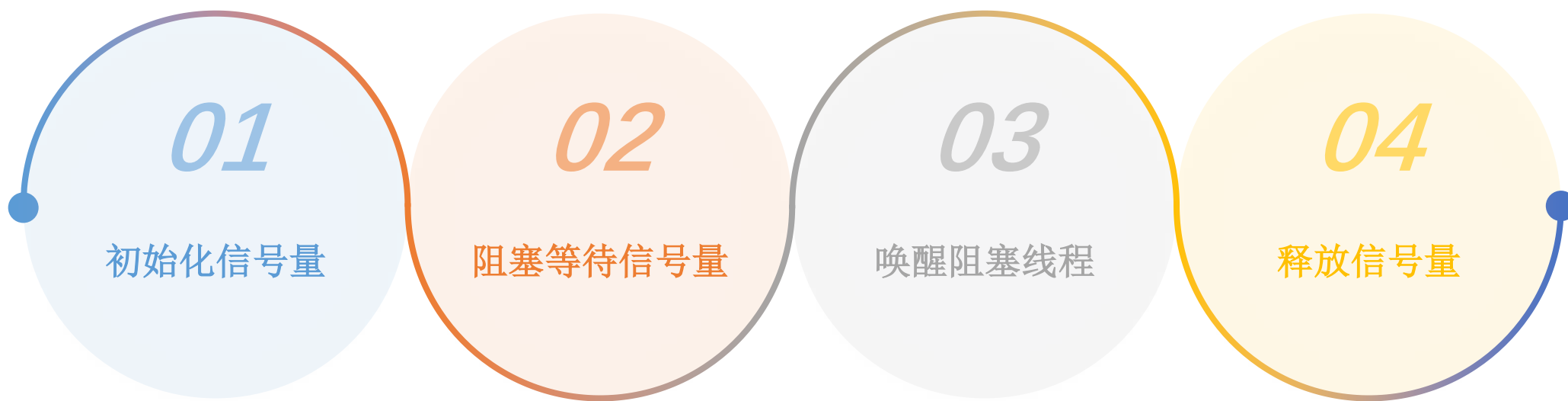
### 8.3.8 信号量

信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。也被称为PV原子操作。

PV原子操作，广泛用于进程或线程之间通信的同步和互斥。其中P代表通过的意思，V代表释放的意思。是指不可中断的过程，由操作系统来保证P操作和V操作。PV操作是针对信号量的操作，就是对信号量的加减过程。

P操作，信号量sem减1的过程，如果sem小于等于0，P操作被阻塞，直到sem变为大于0为止。即加锁过程。V操作，信号量sem加1的过程。即解锁过程。

相对互斥锁而言，信号量既能保证同步，防止数据混乱，又能避免影响线程的并发性。信号量的使用也分四个步骤：



针对以上步骤，Linux系统中提供了一组与线程同步机制中信号量操作相关的函数，这些函数都存在于函数库semaphore.h中。

### (1) 信号量初始化

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

参数: `sem` 为指向信号量变量的指针; `pshared` 共享方式 用于控制信号量的作用范围, 其取值通常为0与非0, 当`pshared`被设置为0时, 信号量将会被放在进程中所有线程可见的地址内, 由进程中的线程共享; 当`pshared`被设置为非0值时, 信号量将会被放置在共享内存区域, 由所有进程共享。 `value` 信号量的初始值, 通常被置为1。



## (2) p操作,减少信号量

```
int sem_wait(sem_t *sem);
```

参数: sem为指向信号量变量的指针。sem\_wait()函数对应P操作,若调用成功,则会使信号量sem的值减一,并返回0;若调用失败,则返回-1,并设置errno。

## (3) V操作,增加信号量

```
int sem_post(sem_t *sem);
```

参数: sem为指向信号量变量的指针; sem\_post ()函数对应V操作,若调用成功,则会使信号量sem的值加一,并返回0;若调用失败,则返回-1,并设置errno。

#### (4) 销毁信号量

```
int sem_destroy(sem_t *sem);
```

参数：sem为指向信号量变量的指针。

#### (5) 获取信号量的值

```
int sem_getvalue(sem_t *sem, int *sval);
```

参数：sem为指向信号量变量的指针，参数sval为一个传入指针，用于获取信号量的值，在程序中调用该函数后，信号量sem的值会被存储在参数sval中。

## 作业5:

1. 基于命名管道的进程双向通信。参考例题7.2.2-1进行修改，使用两个命名管道，实现两个进程（客户端和服务端）之间的全双工通信功能。
  - 1) 客户端和服务端交替对话。
  - 2) 解释程序中每一行的作用。
  - 3) 按下q或其它命令退出程序。
  - 4) 请按照以下三个步骤来分别提交程序和结果：a 先实现只发送功能（如 server → client 单向，参考例7.2.2-1）；b 再实现接收；c 实现其它功能：例如退出程序等。
2. 参考例题8.2.2-1进行修改并讨论不同的线程退出方法并进行讨论。
  - 1) pthread\_exit()。
  - 2) return。
  - 3) exit()。
  - 4) 注释主函数和子函数中的sleep()命令，会有什么不同的结果？请分析。

重庆理工大学/电气学院

CHONGQING UNIVERSITY OF TECHNOLOGY

# 嵌入式Linux系统开发教程

—基于ARM处理器通用平台 (arm9-  
arm11-cortexA系列)

