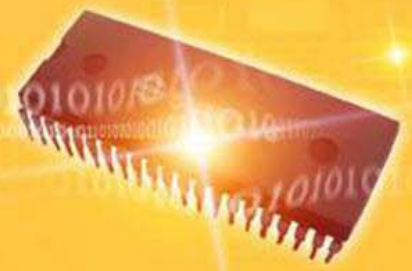
嵌入式系统工程师



线程

大纲

- 线程概述
- 线程的基本操作
- 线程同步互斥

大纲

- 线程概述
- 线程的基本操作
- 线程同步互斥

线程概述

- 每个进程都拥有自己的数据段、代码段和堆栈段，这就造成进程在进行创建、切换、撤销操作时，需要较大的系统开销。
- 为了减少系统开销，从进程中演化出了线程。
- 线程存在于进程中，共享进程的资源。
- 线程是进程中的独立控制流，由环境（包括寄存器组和程序计数器）和一系列的执行指令组成。

线程概述

- 线程的概念

  每个进程有一个地址空间、和一个控制线程。

线程概述

- 线程和进程的比较
  - 传统意义上的进程被称为重量级进程(heavyweight process，HWP)，从现代角度看，就是只拥有一个线程的进程。
  - 线程与进程有很多类似的性质，习惯上也称线程为轻量级进程(lightweight process，LWP)或称为迷你进程。

线程概述

- 调度
  - 线程是CPU调度和分派的基本单位。
  - 进程是系统中程序执行和资源分配的基本单位。
- 拥有资源：
  - 进程是拥有系统资源的一个独立的单位，它可以拥有自己的资源。
  - 线程自己一般不拥有资源（除了必不可少的程序计数器，一组寄存器和栈），但它可以去访问其所属进程的资源，如进程代码段，数据段以及系统资源（已打开的文件，I/O设备等）。

线程概述

- 并发性

  不仅进程间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行。

- 系统开销
  - 同一个进程中的多个线程可共享同一地址空间，因此它们之间的同步和通信的实现也变得比较容易。
  - 在进程切换时候，涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置；而线程切换只需要保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作，从而能更有效地使用系统资源和提高系统的吞吐量。

线程概述

- 使用多线程的目的主要有以下几点：
  - 多任务程序的设计

    一个程序可能要处理不同应用，要处理多种任务，如果开发不同的进程来处理，系统开销很大，数据共享，程序结构都不方便，这时可使用多线程编程方法。
  - 并发程序设计

    一个任务可能分成不同的步骤去完成，这些不同的步骤之间可能是松散耦合，可能通过线程的互斥，同步并发完成。这样可以为不同的任务步骤建立线程。

线程概述

- 网络程序设计

  为提高网络的利用效率，我们可能使用多线程，对每个连接用一个线程去处理。

- 数据共享

  同一个进程中的不同线程共享进程的数据空间，方便不同线程间的数据共享。

- 在多CPU系统中，实现真正的并行。

大纲

- 线程概述
- 线程的基本操作
- 线程同步互斥

线程的基本操作

- 就像每个进程都有一个进程号一样，每个线程也有一个线程号。
- 进程号在整个系统中是唯一的，但线程号不同，线程号只在它所属的进程环境中有效。
- 进程号用pid_t数据类型表示，是一个非负整数。线程号则用pthread_t数据类型来表示。
- 有的系统在实现pthread_t的时候，用一个结构体来表示，所以在可移植的操作系统实现不能把它做为整数处理。

线程的基本操作

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void * arg);
```

功能:

创建一个线程。

线程的基本操作

参数:

- thread: 线程标识符地址。
- attr: 线程属性结构地址。
- start_routine: 线程函数的入口地址。
- arg: 传给新线程执行函数的参数。

返回值:

成功返回0，失败返回非0;

线程的基本操作

- 与fork不同的是pthread_create创建的线程不与父线程在同一点开始运行，而是从指定的函数开始运行，该函数运行完后，该线程也就退出了。
- 线程依赖进程存在的，如果创建线程的进程结束了，线程也就结束了。
- 线程函数的程序在pthread库中，故链接时要加上参数-lpthread。

例：01_pthread_create.c

线程的基本操作

- #include <pthread.h>

  int pthread_join(pthread_t thread, void **value_ptr);

  功能:

  等待子线程结束，并回收子线程资源。

  参数:

  - thread: 被等待的线程号。
  - value_ptr: 指针的地址，调用此函数后，指针指向一个存储线程完整退出状态的静态区域，可用来存储被等待线程的返回值。

  返回值:

  成功返回0，失败返回非0。

例: 02_pthread_join.c

线程的基本操作

- 创建一个线程后应回收其资源，但使用pthread_join函数会使调用者阻塞，故Linux提供了线程分离函数：pthread_detach。

线程的基本操作

- #include <pthread.h>

  int pthread_detach(pthread_t thread);

  功能:

  使调用线程与当前进程分离，使其成为一个独立的线程，该线程终止时，系统将自动回收它的资源。

  参数:

  thread: 线程ID

  返回值:

  成功: 返回 0，失败返回非0。

  例: 03_pthread_detach.c

线程的基本操作

在进程中我们可以调用exit函数或_exit函数来结束进程，在一个线程中我们可以通过以下三种在不终止整个进程的情况下停止它的控制流。

- 线程从执行函数中返回。
- 线程调用pthread_exit退出线程。
- 线程可以被同一进程中的其它线程取消。

线程的基本操作

- #include <pthread.h>

  void pthread_exit(void *value_ptr);

  功能:

  使调用线程退出。

  参数:

  value_ptr: 指向线程退出状态的静态区域，在别的线程中可用pthread_join访问。

  一个进程中的多个线程是共享该进程的数据段，因此，通常线程退出后所占用的资源并不会释放。

例: 04_pthread_exit.c

线程的基本操作

- 取消线程是指取消一个正在执行线程的操作。

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

功能:

取消线程。

参数:

thread: 目标线程ID。

返回值:

成功返回 0，失败返回出错编号。

线程的基本操作

- pthread_cancel函数的实质是发信号给目标线程thread，使目标线程退出。
  - 此函数只是发送终止信号给目标线程，不会等待取消目标线程执行完才返回。
  - 然而发送成功并不意味着目标线程一定就会终止，线程被取消时，线程的取消属性会决定线程能否被取消以及何时被取消。
    - 线程的取消状态
    - 线程取消点
    - 线程的取消类型

线程的基本操作

- 线程的取消状态

  在Linux系统下，线程默认可以被取消。编程时可以通过pthread_setcancelstate函数设置线程是否可以被取消。

  pthread_setcancelstate(int state,
  int *old_state);

  - state:

    PTHREAD_CANCEL_DISABLE: 不可以被取消、

    PTHREAD_CANCEL_ENABLE: 可以被取消。

  - old_state:

    保存调用线程原来的可取消状态的内存地址。

例: 05_pthread_setcancelstate.c

线程的基本操作

- 线程的取消点

  线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过pthread_testcancel函数设置线程的取消点。

  void pthread_testcancel(void);

  - 当别的线程取消调用此函数的线程时候，被取消的线程执行到此函数时结束。
  - POSIX.1保证线程在调用表1、表2中的任何函数时，取消点都会出现。

例：05_pthread_testcancel.c

线程的基本操作

- 线程的取消类型

  线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过pthread_setcanceltype函数设置线程是否可以立即被取消。

  pthread_setcanceltype(int type,int *oldtype);

  - type:

    PTHREAD_CANCEL_ASYNCHRONOUS: 立即取消、

    PTHREAD_CANCEL_DEFERRED: 不立即被取消

  - oldtype:

    保存调用线程原来的可取消类型的内存地址。

例: 05_pthread_setcanceltype.c

线程的基本操作

和进程的退出清理一样，线程也可以注册它退出时要调用的函数，这样的函数称为线程清理处理程序(thread cleanup handler)。

- 注意:
  - 线程可以建立多个清理处理程序。
  - 处理程序在栈中，故它们的执行顺序与它们注册时的顺序相反。

线程的基本操作

- #include <pthread.h>

  void pthread_cleanup_push(

  void (*routine)(void*), void *arg);

  功能:

  将清除函数压栈。即注册清理函数。

  参数:

  - routine: 线程清理函数的指针。
  - arg: 传给线程清理函数的参数。

线程的基本操作

- #include <pthread.h>

  void pthread_cleanup_pop(int execute);

  功能:

  将清除函数弹栈，即删除清理函数。

  参数:

  execute: 线程清理函数执行标志位。

  - 非0，弹出清理函数，执行清理函数。
  - 0，弹出清理函数，不执行清理函数

线程的基本操作

- 当线程执行以下动作时会调用清理函数:
  - 调用pthread_exit退出线程。
  - 响应其它线程的取消请求。
  - 用非零execute调用pthread_cleanup_pop。

  无论哪种情况pthread_cleanup_pop都将删除上一次pthread_cleanup_push调用注册的清理处理函数。

注意:

pthread_cleanup_pop、pthread_cleanup_push必须配对使用。

例: 06_pthread_cleanup_exit.c

例: 06_pthread_cleanup_cancel.c

例: 06_pthread_cleanup_pop.c

大纲

- 线程概述
- 线程的基本操作
- 线程同步互斥

线程同步与互斥

- 同步:

  两个或两个以上的线程在运行过程中协同步调，按预定的先后次序运行。

- 互斥:

  一个公共资源同一时刻只能被一个线程使用，多个线程不能同时使用公共资源。POSIX标准中线程同步和互斥的方法,主要有信号量和互斥锁两种方式。

线程同步与互斥

- 互斥锁(mutex)

  mutex是一种简单的加锁的方法来控制对共享资源的访问，mutex只有两种状态,即上锁(lock)和解锁(unlock)。

  - 在访问该资源前，首先应申请mutex，如果mutex处于unlock状态，则会申请到mutex并立即lock；如果mutex处于lock状态，则默认阻塞申请者。
  - unlock操作应该由lock者进行。

线程同步与互斥

- mutex用pthread_mutex_t数据类型表示，在使用互斥锁前，必须先对它进行初始化。
- 静态分配的互斥锁：

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

- 动态分配互斥锁：

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

在所有使用过此互斥锁的线程都不再需要使用时候，应调用pthread_mutex_destroy销毁互斥锁。

线程同步与互斥

- #include <pthread.h>

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

功能:

初始化一个互斥锁。

参数:

mutex: 互斥锁地址。

attr: 互斥锁的属性，NULL为默认的属性。

返回值:

成功返回0，失败返回非0。

线程同步与互斥

- #include <pthread.h>

  int pthread_mutex_lock(pthread_mutex_t *mutex);

  功能:

  对互斥锁上锁，若已经上锁，则调用者一直阻塞到互斥锁解锁。

  参数:

  mutex: 互斥锁地址。

  返回值:

  成功返回0，失败返回非0。

线程同步与互斥

- #include <pthread.h>
  int pthread_mutex_trylock(
  pthread_mutex_t *mutex);

  功能:

  对互斥锁上锁，若已经上锁，则上锁失败，函数立即返回。

  参数:

  mutex: 互斥锁地址。

  返回值:

  成功返回0，失败返回非0。

线程同步与互斥

- #include <pthread.h>

  int pthread_mutex_unlock(
  pthread_mutex_t * mutex);

  功能:

  对指定的互斥锁解锁。

  参数:

  mutex: 互斥锁地址。

  返回值:

  成功返回0,失败返回非0。

线程同步与互斥

- #include <pthread.h>

  int pthread_mutex_destroy(

  pthread_mutex_t *mutex);

  功能:

  销毁指定的一个互斥锁。

  参数:

  mutex: 互斥锁地址。

  返回值:

  成功返回0，失败返回非0。

  例: 07_pthread_mutex.c

线程同步与互斥

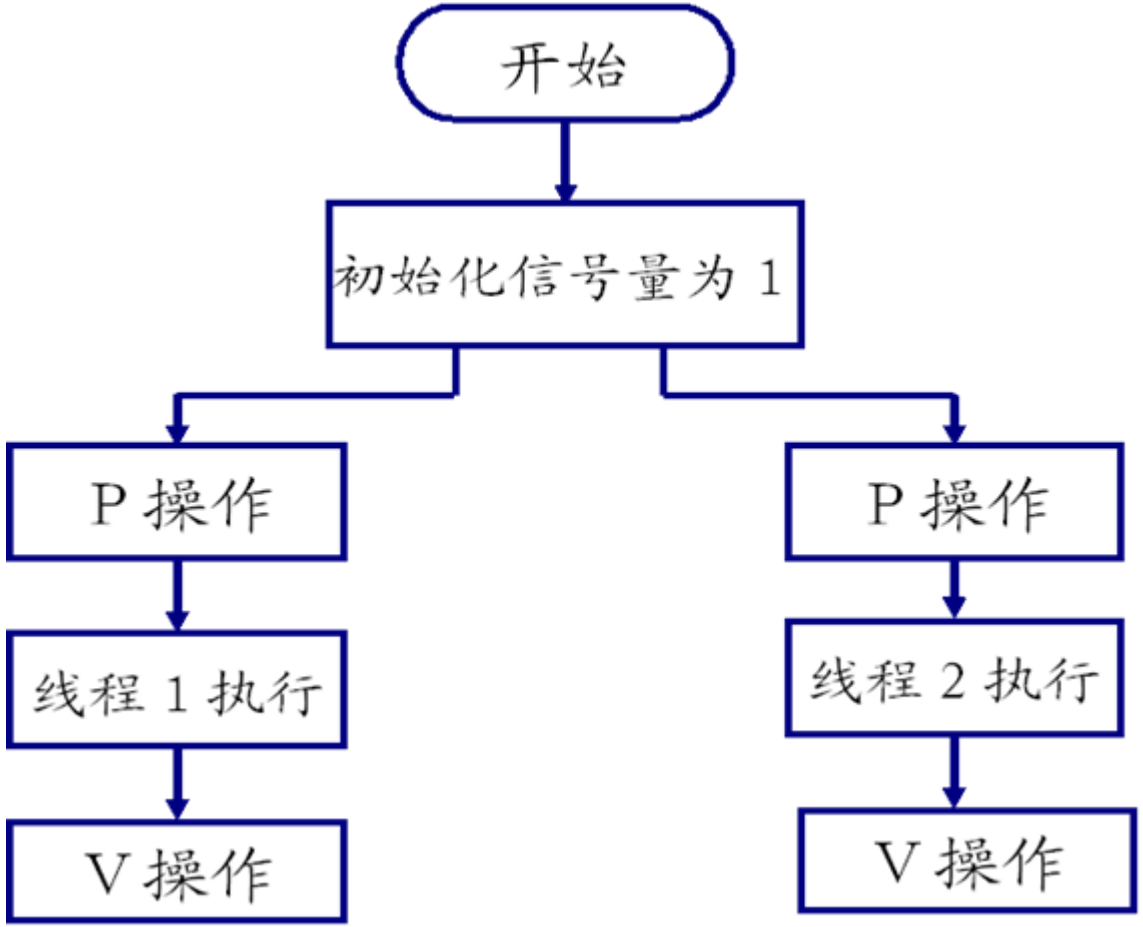
- 信号量广泛用于进程或线程间的同步和互斥，信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。
- 进程或线程根据信号量操作结果的值判断是否对公共资源具有访问的权限，当信号量值大于 0 时，则可以访问，否则将阻塞。

线程同步与互斥

- ＰＶ原语是对信号量的操作，一次Ｐ操作使信号量sem减１，一次Ｖ操作使信号量sem加１。
- ＰＶ原语主要用于进程或线程间的同步和互斥这两种典型情况。
  - 若用于互斥，几个进程（或线程）往往只设置一个信号量。
  - 若用于同步操作，往往会设置多个信号量，并且安排不同的初始值，来实现它们之间的执行顺序。

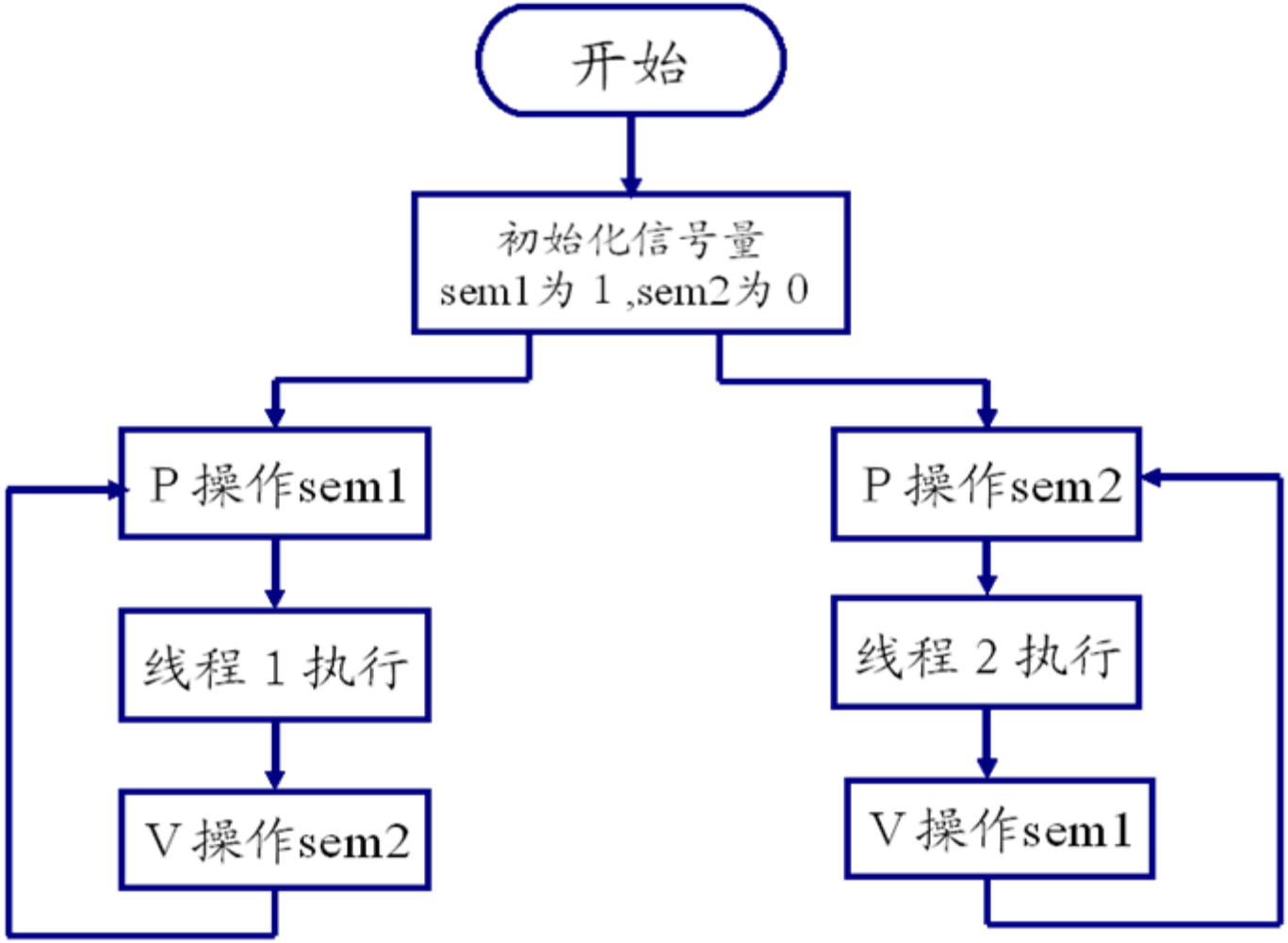
线程同步与互斥

- 信号量用于互斥



线程同步与互斥

- 信号量用于同步



线程同步与互斥

- #include <semaphore.h>

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

  功能:

  创建一个信号量并初始化它的值。

  参数:

  - sem: 信号量地址。
  - pshared: 决定信号量能否在几个进程间共享，目前Linux没有实现进程间共享信号量，故此值只能为0。
  - value: 信号量的初始值。

  返回值:

  成功返回0，失败返回-1。

线程同步与互斥

- #include <semaphore.h>

  int sem_wait(sem_t *sem);

  功能:

  将信号量的值减1，若信号量的值小于0，此函数会引起调用者阻塞。

  参数:

  sem: 信号量地址。

  返回值:

  成功返回0，失败返回-1。

线程同步与互斥

- #include <semaphore.h>

  int sem_trywait(sem_t *sem);

  功能:

  将信号量的值减1，若信号量的值小于0，则对信号量的操作失败，函数立即返回。

  参数:

  sem: 信号量地址。

  返回值:

  成功返回0,失败返回-1。

线程同步与互斥

- #include <semaphore.h>

  int sem_post(sem_t *sem);

  功能:

  将信号量的值加1并发出信号唤醒等待线程。

  参数:

  sem: 信号量地址。

  返回值:

  成功返回0，失败返回-1。

线程同步与互斥

- #include <semaphore.h>

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

  功能:

  获取sem标识的信号量的值，保存在sval中。

  参数:

  - sem: 信号量地址。
  - sval: 保存信号量值的地址。

  返回值:

  成功返回0，失败返回-1。

线程同步与互斥

- #include <semaphore.h>

  int sem_destroy(sem_t *sem);

  功能:

  删除sem标识的信号量。

  参数:

  sem: 信号量地址。

  返回值:

  成功返回0，失败返回-1。

例: 08_semphore_1.c

08_semphore_2.c

线程同步与互斥

- 练习

  生产者消费者:

  - 有一个仓库，生产者负责生产产品，并放入仓库，消费者会从仓库中拿走产品(消费)。
  - 要求:
    - 仓库中每次只能入一人(生产者或消费者)。
    - 仓库中可存放产品的数量最多10个，当仓库放满时，生产者不能再放入产品。
    - 当仓库空时，消费者不能从中取出产品。
    - 生产、消费速度不同。