实验八 线程

1、 实验目的

1) 加深对线程的理解，线程与进程的区别与联系；

2) 掌握多线程基本编程方法，多线程之间同步与互斥的方法；

2、 实验准备

拷贝 lab8 中文件（课堂上提供）到~/raspberry/prj_os/lab8，

执行命令
make
编译；

3、 实验内容

3.1 线程基本知识

为进一步减少处理机的空转时间，支持多处理器以及减少上下文切换开销，进程在演化中出现了另一个概念——线程。它是进程内独立的一条运行路线，**处理器调度的最小单元**，也可以称为轻量级进程。由于线程的高效性和可操作性，在嵌入式系统开发中运用得非常广泛。

进程的特点：

- 每个进程认为自己有独立的地址空间(32 位的 CPU 的里面，虚拟的地址空间是 4GB)，独占 CPU（通过内核欺骗应用程序：内核与硬件的 MMU 即内存管理单元狼狈为奸）；
- 进程之间要交互数据，必须通过 IPC 机制（管道、共享内存、消息队列， 信号用于异步通信，信号量一般用于同步和共同数据的保护）；

线程的特点：

- 从属于进程；
- 进程中可以有一个线程也可以有多个线程；
- 同一个进程中的线程共享进程的资源；
- 子线程共享父线程的地址空间（数据共享很容易，关注资源保护）；
- 在程序中线程以函数方式实现；

进程与线程的共同点：

- 线程和进程都会被 LINUX 操作系统作为一个任务被调度，在内核中都通过一个结构体 task_struct 来表示；
- 每个任务（应用空间的进程、线程、内核线程）的内存管理是通过 task_struct 里面的成员 struct mm_struct *mm 来进行管理；

线程与进程的不同点：

- 在同一进程的不同线程，他们是共享同一进程的地址空间（进程的 4GB 的虚拟地址空间，地址空间在内核中是通过 struct mm_struct *mm 来指向）；
- 多线程中，数据共享很容易（可以通过全局变量[慎用]，malloc()分配的地址指针），但数据比较容

易出现资源竞争（多线程编程中要特别关注资源的保护：信号量、互斥锁，条件变量）；

- 多线程是作为一个库来运行（没有提供系统调用）；
- 多线程的效率一般比多进程要高一些（为什么？）；

Attention：

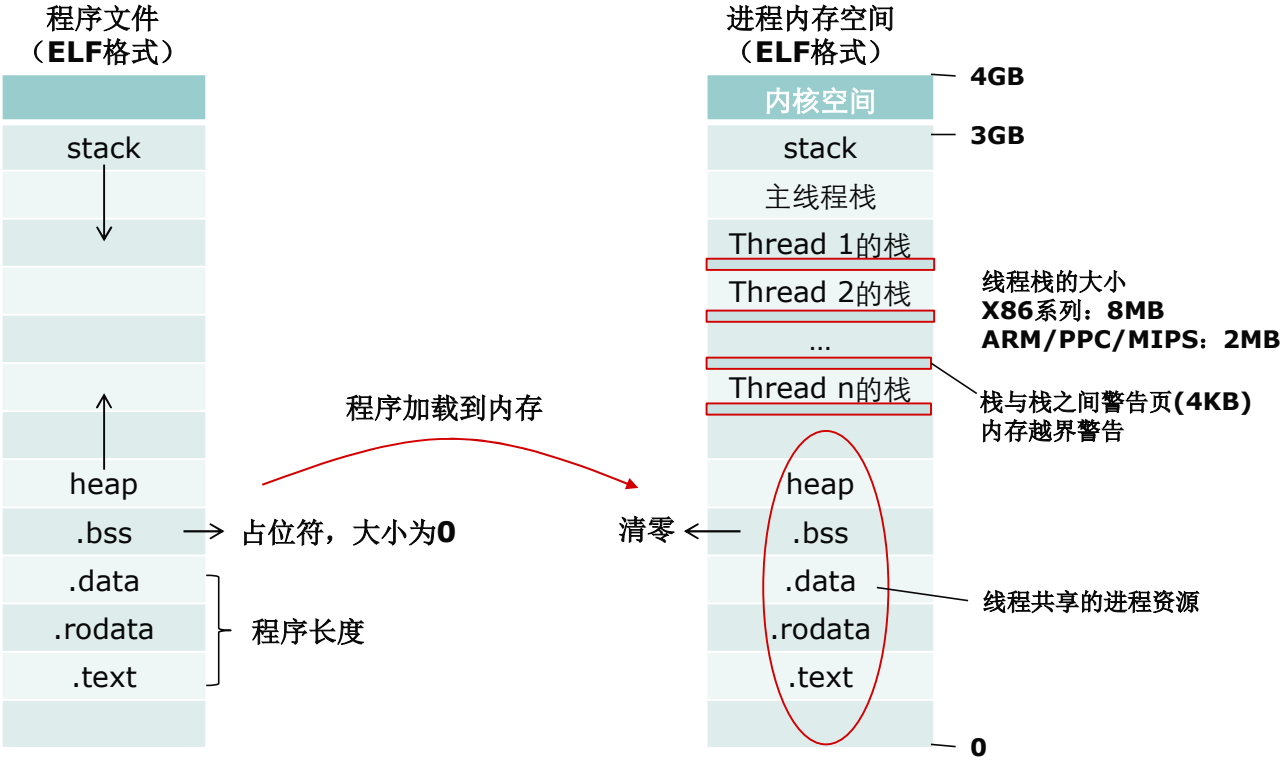
- ✧ **进程是系统分配资源的最小单位；**
- ✧ **线程是 CPU 调度的最小单位，CPU 按一定的时间片调度线程；**

一个进程中的多个线程共享以下资源：

- ◆ 可执行的指令
- ◆ 静态数据
- ◆ 进程中打开的文件描述符
- ◆ 信号处理函数
- ◆ 当前工作目录
- ◆ 用户 ID
- ◆ 用户组 ID

每个线程私有的资源如下：

- ◆ 线程的 ID
- ◆ PC（程序计数器）和相关寄存器
- ◆ 栈（局部变量、返回地址）
- ◆ 错误号（errno）
- ◆ 信号掩码和优先级
- ◆ 执行状态和属性



3.2 线程基础编程

实现多线程编程首先要创建线程。创建线程实际上就是确定调用该线程函数的入口点，这里通常使用的函数是 pthread_create()。在线程创建之后，就开始运行相关的线程函数，在该函数运行完之后，该线程也就退出了，这是线程退出的一种方法。另一种退出线程的方法是使用函数 pthread_exit()，这是线程的主动退出行为。这里要注意的是，在使用线程函数时，不能随意使用 exit()退出函数进行出错处理，由于 exit()的作用是使调用进程终止，往往一个进程包含多个线程，因此，在使用 exit()之后，该进程中的所有线程都终止了。因此，在线程中就可以使用 pthread_exit()来代替进程中的 exit()。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以用 wait()系统调用来同步终止并释放资源一样，线程之间也有类似机制，那就是 pthread_join()函数。pthread_join()可用于将当前线程挂起来等待线程的结束。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。

结束线程方法：

- 调用 return，使线程结束；
- 调用函数 pthread_exit()，使线程结束；
- 调用函数 exit()，使线程结束；

主线程与子线程：

- 进程结束时，进程中的所有线程结束；
- 主线程结束时，其它子线程也终止；
- 如果子线程是分离属性，结束时由系统释放资源；
- 如果子线程是接合属性，结束时必须由主线程调用 pthread_join()释放资源；

3.1.1 函数说明

pthread_create()函数语法要点

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，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

pthread_exit()函数语法要点

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()函数语法要点

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()函数语法要点

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

pthread_detatch()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_detach(pthread_t thread)
函数传入值	th：要置为分离属性的线程的标识符
函数返回值	成功：0
	出错：返回错误码

The pthread_detach() function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

多进程和多线程编程的基本函数可做如下类比：

进程函数	线程函数	说明
Fork()	Pthread_create()	创建
Exit()	Pthread_exit()	退出
Wait()/waitpid()	Pthread_join()	资源回收
Get_pid()	Pthread_self()	获取 ID
Abort()	Pthread_cancel()	取消进程/线程

3.1.2 实验操作

1） 查看 thread_base.c 源代码，编译并运行，查看结果，体会 pthread_create(), pthread_join(), pthread_exit()等函数的功能及用法；

2） 查看 thread_detach.c 源代码，编译并运行，查看结果，体会 pthread_detach()函数的功能及用法；

3） 查看 thread_params.c 源代码，编译并运行，查看结果，体会父子线程之间的参数传递功能；

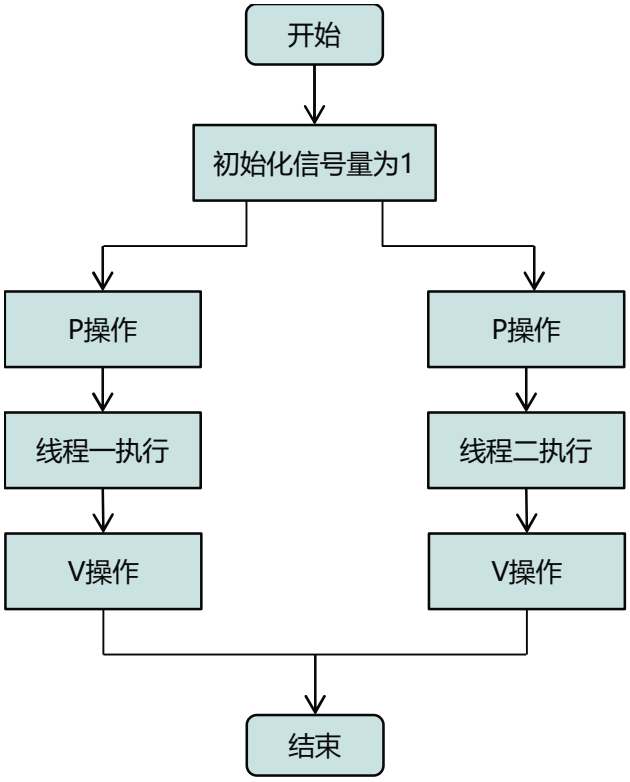
3.3 多线程资源保护机制-信号量

3.3.1 线程间信号量通信原理及主要函数

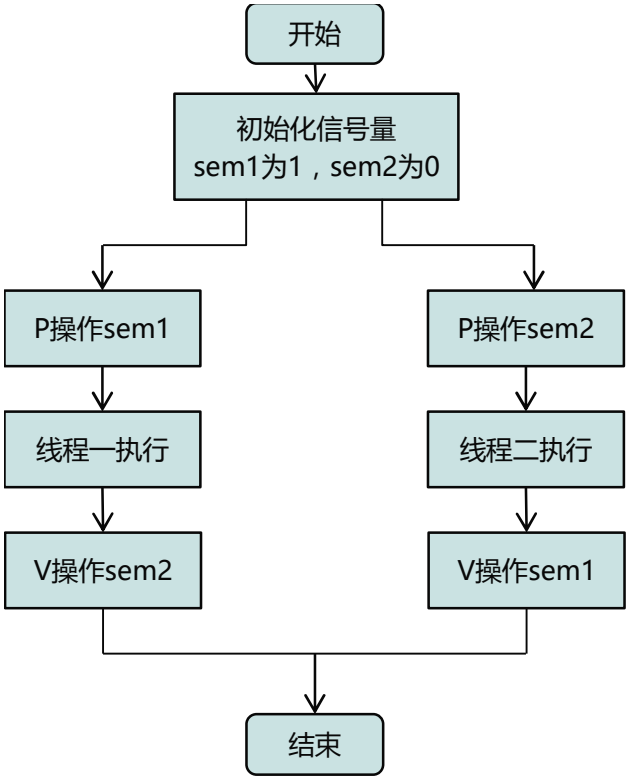
信号量广泛用于进程或线程间的同步与互斥，信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。

先简单回顾一下 PV 原子操作的工作原理。PV 原子操作是对整数计数器信号量 sem 的操作。一次 P 操作使 sem 减一，而一次 V 操作使 sem 加一。进程（或线程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量 sem 的值大于等于零时，该进程（或线程）具有公共资源的访问权限；相反，当信号量 sem 的值小于零时，该进程（或线程）就将阻塞直到信号量 sem 的值大于等于 0 为止。

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。若用于互斥，几个进程（或线程）往往只设置一个信号量 sem，它们的操作流程下图 所示。



当信号量用于同步操作时，往往会设置多个信号量，并安排不同的初始值来实现它们之间的顺序执行，它们的操作流程如下图所示。



Linux 信号量用于线程间的互斥与同步，几个常见函数如下：

- 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。

sem_init()函数语法要点

所需头文件	#include <semaphore.h>
函数原型	int sem_init(sem_t *sem,int pshared,unsigned int value)
函数传入值	sem：信号量指针
	pshared：决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量，所以这个值只能够取 0，就表示这个信号量是当前进程的局部信号量。
	value：信号量所保护的资源个数
函数返回值	成功：0
	出错：-1

sem_wait()等函数语法要点

所需头文件	#include <pthread.h>
函数原型	int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_destroy(sem_t *sem)
函数传入值	sem：信号量指针
函数返回值	成功：0
	出错：-1

sem_getvalue()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int sem_getvalue(sem_t *sem, int *sval)
函数传入值	sem：信号量指针
	sval: 地址指针，用于保存从 sem 中获取的信号量值
函数返回值	成功：0
	出错：-1

sem_getvalue() places the current value of the semaphore pointed to sem into the integer pointed to by sval.

3.3.2 实验操作

查看 thread_huchi.c 源代码，编译并运行，查看结果，体会 sem_init(), sem_wait(), sem_poset()等函数的功能及用法；

作业：

1） 按照对 thread_huchi.c 的源代码分析，write 线程和 read 线程是否应该轮流打印，各线程每次打印一行，实际运行结果是否是这样，如果不是，请给出分析原因；

2） 请修改 thread_huchi.c, 实现 write 线程和 read 线程的同步，从而实现轮流打印一行的效果；

3.4 多线程资源保护机制-互斥锁

3.4.1 线程间互斥锁通信原理及主要函数

互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。这个互斥锁只有两种状态，也就是上锁和解锁，可以把互斥锁看作某种意义上的全局变量。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。可以说，这把互斥锁保证让每个线程对共享资源按顺序进行原子操作。

互斥锁机制的基本函数及操作步骤如下：

- 互斥锁初始化：pthread_mutex_init()
- 互斥锁上锁：pthread_mutex_lock() //加锁的阻塞方式
- 互斥锁判断上锁：pthread_mutex_trylock() //加锁的非阻塞方式
- 互斥锁解锁：pthread_mutex_unlock()
- 消除互斥锁：pthread_mutex_destroy()

常用的互斥锁分为快速互斥锁和检错互斥锁。这 2 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，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()函数语法要点

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	
	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()等函数语法要点

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

3.4.2 实验操作

查看 mutex.c 源代码，编译并运行，查看结果，体会线程间互斥锁通信的方法，pthread_mutex_init (),pthread_mutex_lock (),pthread_mutex_unlock (),pthread_mutex_destroy()等函数的功能及用法；请思考如下问题：

按照程序分析，write 线程和 read 线程是否应该轮流打印，各线程每次打印一行，实际运行结果是否是这样，如果不是，请分析原因；

4、 作业

1） 按照 3.3.2 对 thread_huchi.c 的源代码分析，write 线程和 read 线程是否应该轮流打印，各线程每次打印一行，实际运行结果是否是这样，如果不是，请给出分析原因；

2） 请修改 thread_huchi.c, 实现 write 线程和 read 线程的同步，从而实现轮流打印一行的效果；