第三十三章

1. 线程的更多细节。
2. 线程栈：

创建线程时，每个线程都有属于线程的线程栈，大小固定，主线程的栈要更大一些。可以使用pthread\_attr\_setstacksize和pthread\_attr\_setstack改变控制线程的大小。通常，创建线程大小的默认值时RLIMIT\_STACK的大小。可以在程序运行前在shell里设置ulimit -s去创建栈的默认值。注意，在程序里使用函数setrlimit有的时候没有作用，因为NPTL在调用main之前就初始化栈大小。

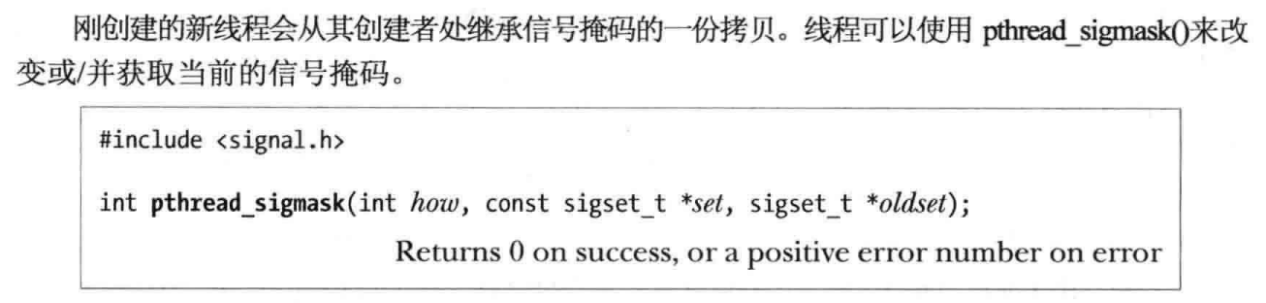
1. 线程与信号：

Unix信号模型是针对Unix进程模型设计的，比pthreads要早问世。因此，与线程模型是有冲突的。

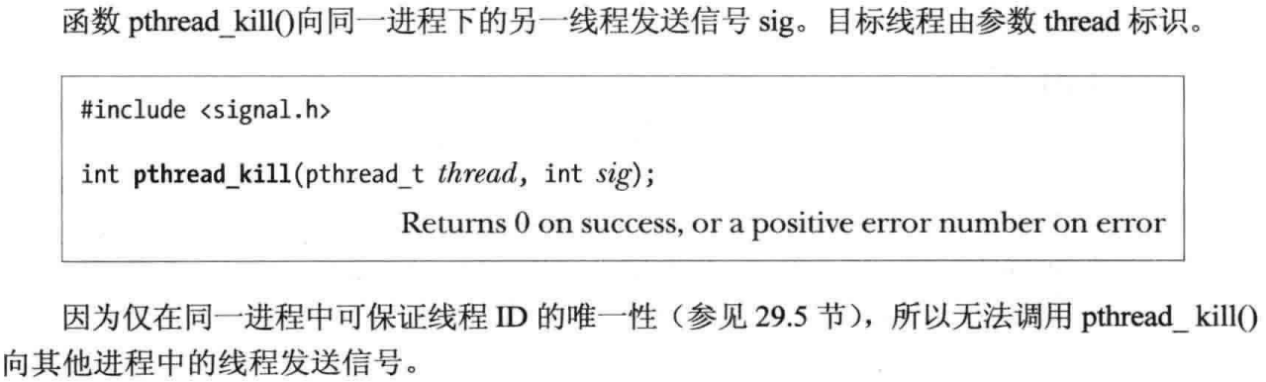
1. Unix信号模型如何映射到线程：

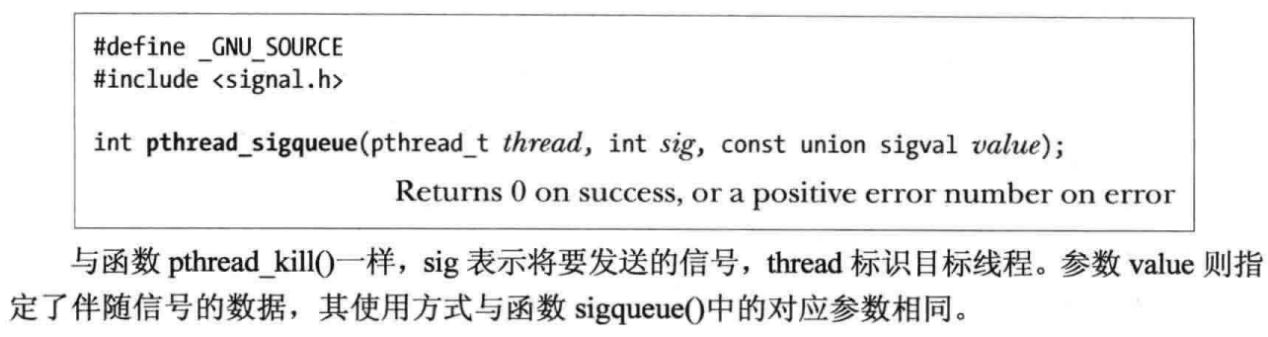
信号模型哪些层面属于进程，哪些属于线程？

1. 信号动作属于进程层面。
2. 对信号的处置属于进程层面。
3. 信号的发送可针对两者。具体查看书本。
4. 多线程程序收到一个信号只由其中一个线程处理一次。
5. 信号掩码针对每个线程。
6. 整个进程和线程所挂起的信号都分别有记录。Sigpeding返回当前线程和进程的交集。
7. 信号备用栈针对线程。
8. 信号中断对pthread\_mutex\_lock和pthread\_cond\_wait的影响。
9. 操作线程信号掩码：



1. 向线程发送信号：

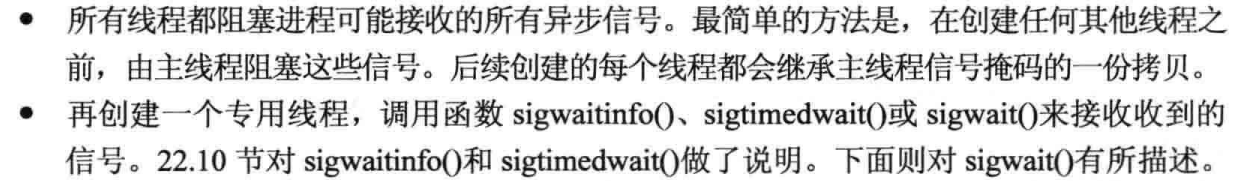




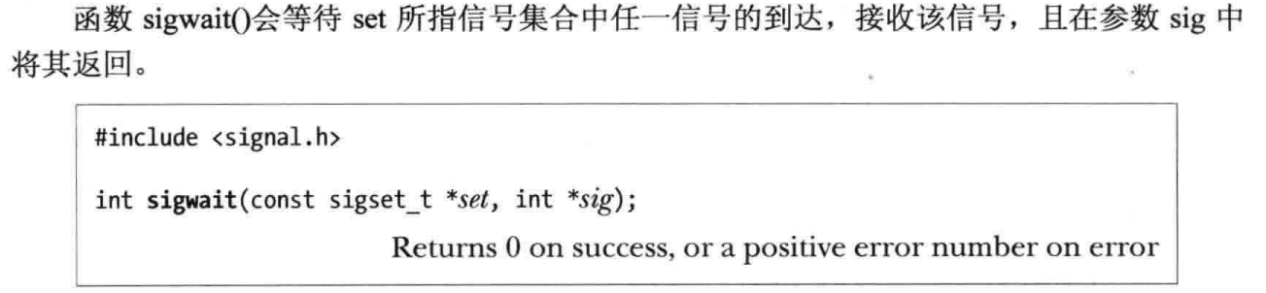
1. 妥善处理异步信号：

由于所有pthreads API都不是异步信号安全函数，所以多线程应用程序必须处理产生异步信号时，通常不应该使用信号处理函数。

推荐方法：



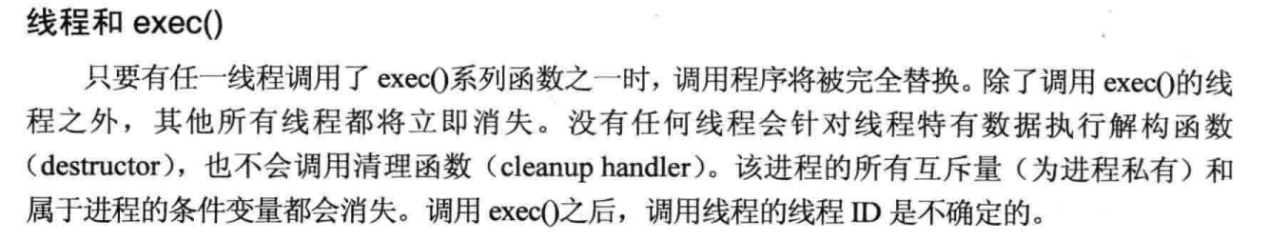
这样可以使用非异步信号安全函数。



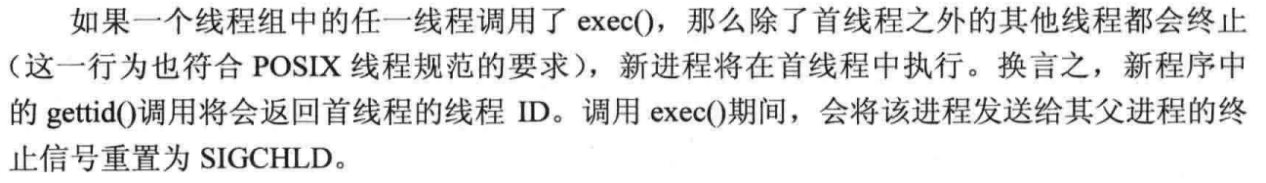
1. 线程和进程控制：

线程和exec：

我发现书本的表达出现矛盾，到底哪个是正确的？

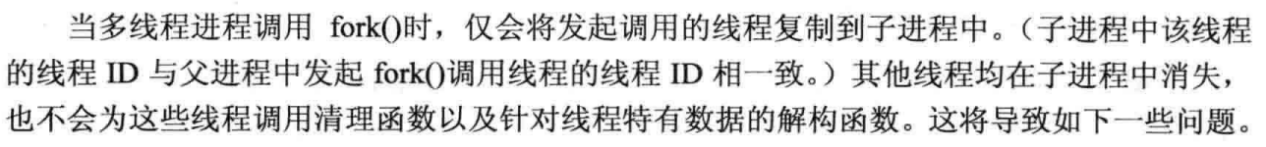


下图为clone那一章（二十八）表述。



自己验证了一下，下图的表述似乎是正确的。

线程和fork：

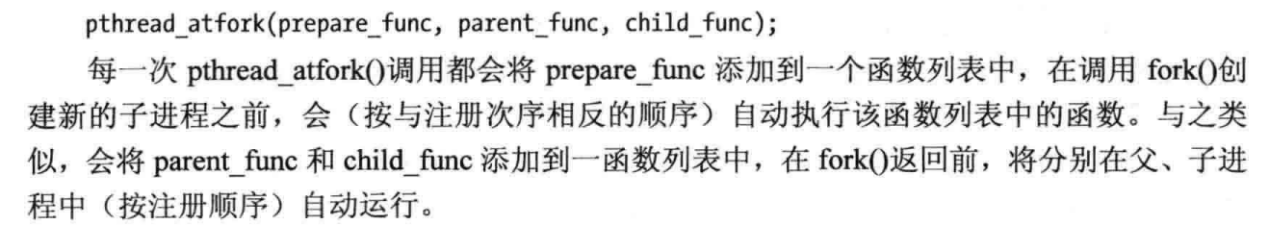


这里有一句话表述是子进程与父进程中的线程ID一致，我试了一下，是不一致的。

多线程程序调用fork会导致问题：

1. 全局数据不一致。
2. 内存泄漏。

解决方法：要么在调用fork后调用exec，要么使用fork处理函数。



线程与exit：

所有线程消失，解构函数和清理函数不会执行。

1. 线程的实现模型：

3种实现模型的差异主要集中在线程如何与内核调度实体（KSE）相映射。

多对一实现（用户级线程）：

优点：切换速度快，容易移植。

缺点：一个线程阻塞，所有线程阻塞。内核无法调度线程。

一对一实现（内核级线程）：

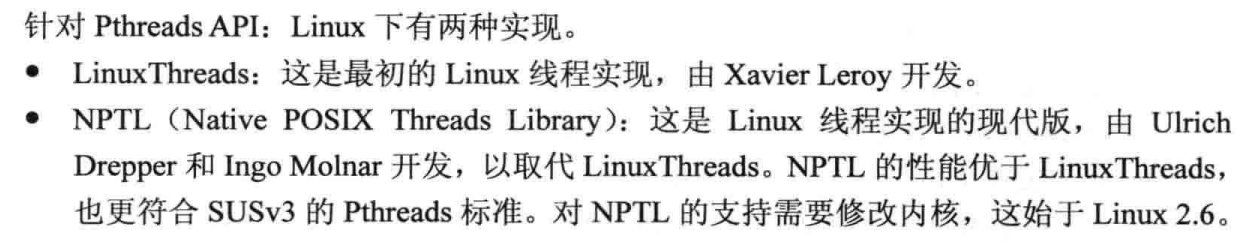
切换慢，创建KSE需要资源。

更多实现采用这种方式。

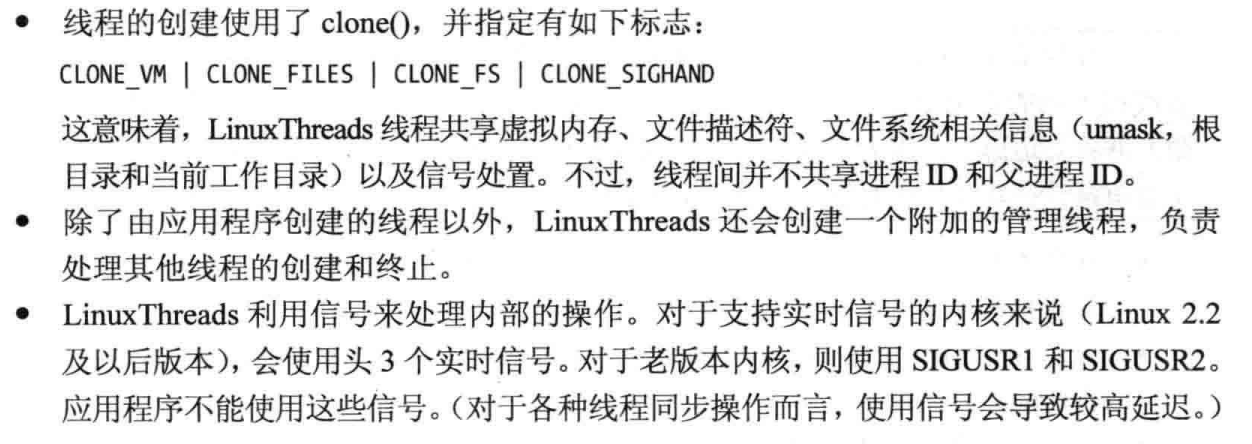
多对多实现（两级模型）：

结合上述两种优点，但是实现难度极大。

1. Linux Posix线程的实现：

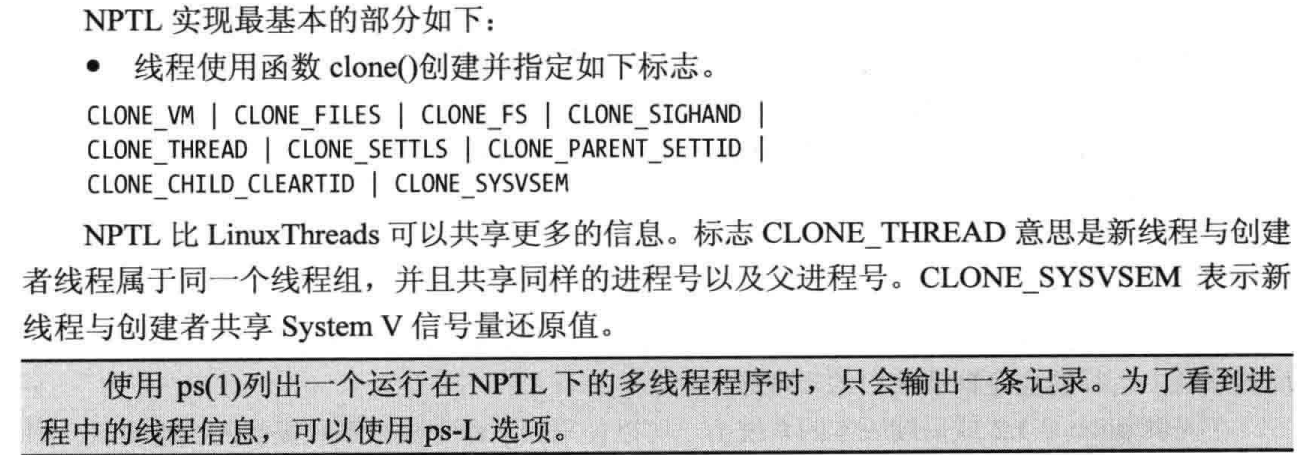


LinuxThreads实现要点：



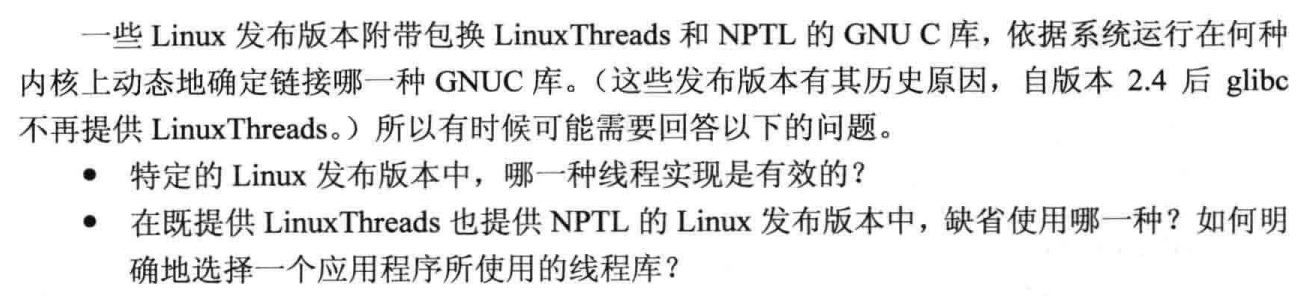
此实现有许多对标准的背离，具体查看书本。

NPTL实现：



NPTL比LinuxThreads要更接近标准。但也有些不一致的地方。具体查看书本。

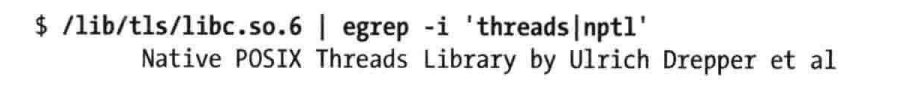
1. 哪一种线程的实现：

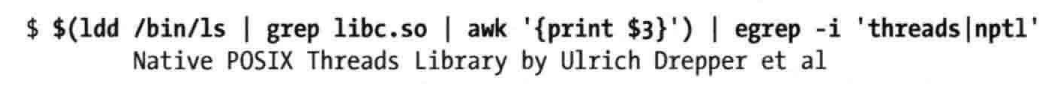


下图命令显示系统提供的线程实现或缺省使用。



或者如下：





如何选择线程的实现？

设置环境变量：

