# 浅谈线程


小组15级成员

杜肖孟


1.Linux POSIX 线程的两种实现

2.线程概述

3.进程和线程的区别

4.并发编程


1. 线程实现模型

2. LinuxThreads：最初的Linux线程实现

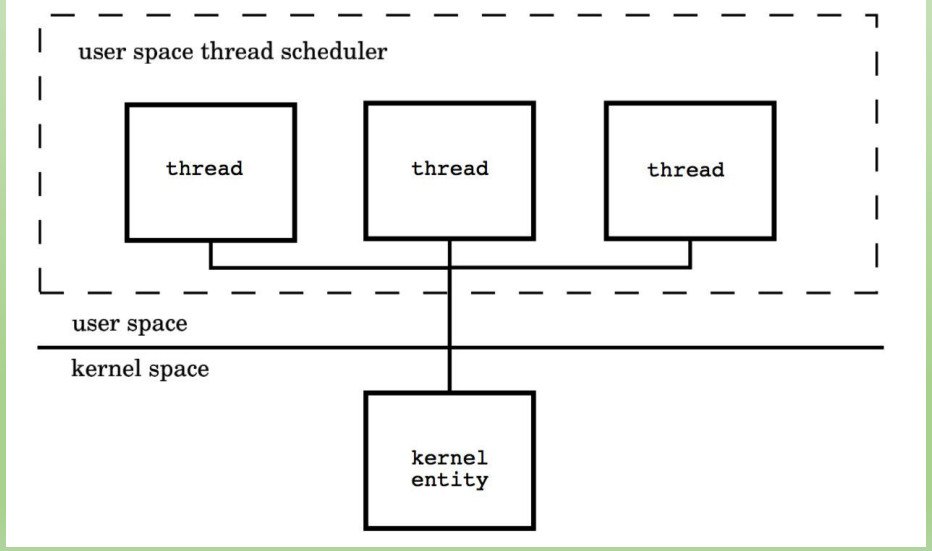
3. NPTL：Linux线程实现的现代版


- 多对一实现（用户级线程）

一个内核线程和多个用户线程组成，线程创建、调度、同步等操作全部有进程内用户空间的线程库来处理，内核完全不知道进程中的多个线程

优点：线程操作不需要切换到内核空间，执行速度很快；线程库不需要内核支持

缺点：某一个线程系统调用阻塞，那么所有的线程都会被阻塞；内核无法调度这些线程，此时调度是以进程为单位

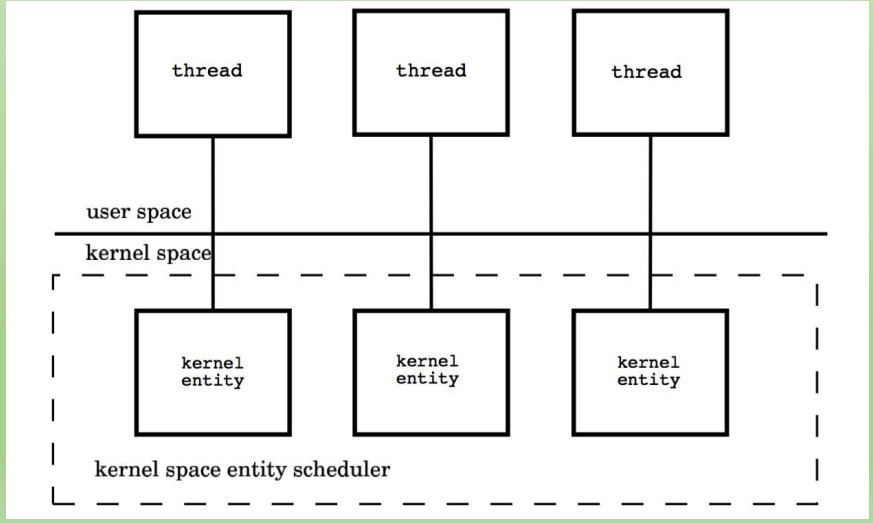



- 一对一实现（内核级线程）

每个用户线程都和一个特定的内核线程交互，内核分别对每个线程做调度处理。

优点：遭阻塞的系统调用不会导致进程中的所有线程被阻塞；多处理器硬件平台，可将多个进程调度到不同CPU

缺点：创建、同步、上下文切换等慢一些；每个线程对应一个内核线程，占用内核资源，尤其是程序有大量线程

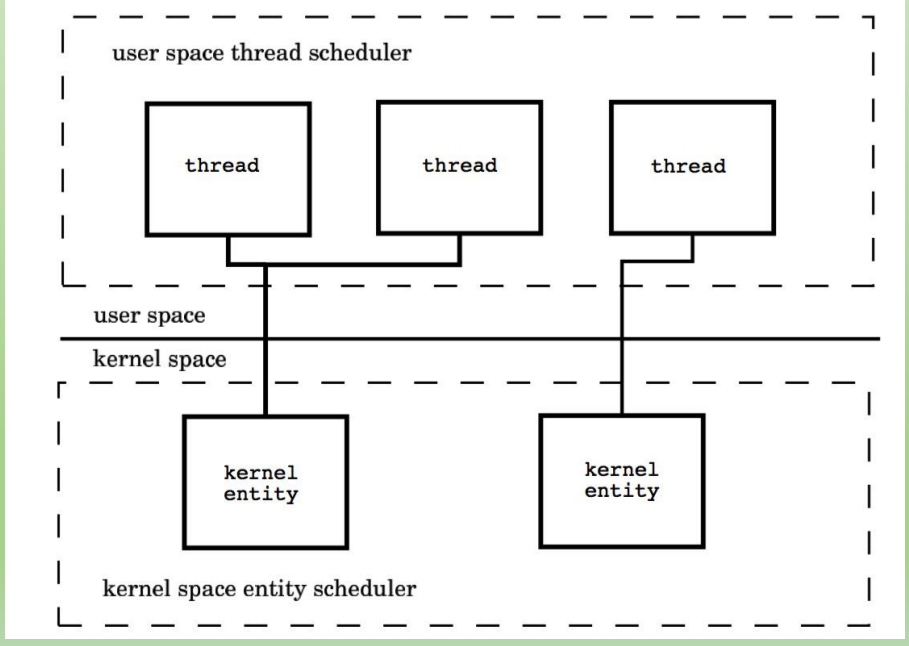



- 多对多实现（两级模型）

每个进程都可拥有多个与之相关的内核线程，并且可以把多个线程映射到一个内核线程。

优点：结合前两者的优点，避免前两者的缺点

缺点：过于复杂




# LinuxThreads

实现要点：

- 创建使用clone()，并指定如下标志：CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
- 除应用程序创建的线程外，还会创建一个附加的管理线程，来协调进程拥有的所有线程
- 利用信号来处理内部的操作，比如同步原语

与标准背离之处：

- 同一进程的不同线程调用getpid()返回不同的值，getppid()返回管理线程的进程ID
- 如果某线程执行了exec()，那么产生进程与调用线程拥有相同的进程ID，而与主线程的进程ID不同
- 采用kill()或sigqueue()向某进程发送信号，只能由指定的线程来接收和处理，即使这一线程阻塞此信号
- ……


# NPTL

实现要点：

- 扩展系统调用clone()，添加更多标志：CLONE_THREAD | CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | …
- 没有使用管理线程，管理线程的一些需求是并不需要的
- 修改并支持与POSIX模型一样的信号处理
- 改进了线程组的实现
- ……

```
$ getconf GNU_LIBPTHREAD_VERSION
```


# 线程组

线程组就是共享同一线程组标识(TGID)的一组线程。

- 线程组中首个线程的线程ID与其线程组ID相同，称之为线程组首线程；
- 线程组中的所有线程拥有同一父进程ID，即与线程组首线程ID相同；
- 组内每个线程都有一个唯一的线程标识符(TID)来标识自身；
- 如果组中任一线程调用了exec()，那么除首线程之外的其他线程都会终止，新进程在首线程中执行；
- 如果线程组中某个线程调用fork() || vfork()创建子进程，那么组中的任何线程都可使用wait()或类似函数来监控该子进程


1.Linux POSIX 线程的两种实现

2.线程概述

3.进程和线程的区别

4.并发编程


# 线程概述

1. 虚拟地址空间

2. 一个进程能开多少线程？

3. 如何正确使用多线程？


# 局部性原理

- 空间局部性

大多数程序倾向于访问过的内存地址附近的内存（指令通常是按顺序执行的，数据也一般是向量或者数组等形式存储的）

- 时间局部性

程序倾向于在不久的将来再次访问最近刚访问过的内存地址（某条指令一旦执行，不久以后可能再次执行，数据也是）


Linux/x86-32中典型的进程内存结构

# 一个进程能开多少线程？


3G /（一个线程的空间）≈ 能创建的最大线程数

# 如何正确使用多线程？


我们用多线程目的之一是为了提高程序处理的效率，那么线程是不是开的越多越好呢?

写一段程序，求1--MAX（MAX <= 10 0000 0000）的和，将这些数平分给 N个线程，每个线程计算各自的部分之和，最后加到一块，测出执行时间。改变MAX和N的值作比较。


1.Linux POSIX 线程的两种实现

2.线程概述

3.进程和线程的区别

4.并发编程


1. 两者的相同点

   并发手段、调度实体、状态相似

2. 实现方式的差异

   进程是资源分配的基本单位，线程是调度的基本单位。
   进程的个体间是完全独立的，而线程间是彼此依存的。
   pid_t fork(void);
   int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...

3. 多任务程序设计模式的区别

   资源是否独立

4. 个体辈分关系的不同

   进程辈分关系森严

5. 通信方式、资源管理方式

   ……


1.Linux POSIX 线程的两种实现

2.线程概述

3.进程和线程的区别

4.并发编程


# 并发编程

1. 协程

2. 并发语言Erlang

3. "池"化思想


Xiyou Linux Group
The Art of Linux Kernel
www.xiyoulinux.org

# Thank you!