《openEuler内核编程》

课程讲稿

第四章 第2讲

线程

软件所制

第四章 第2讲 线程

**学时：**1学时

**教学目的：**系统学习线程

**PPT讲稿：**



这一节我们讲学习线程的内容。



略



这一节分四部分内容，前面一部分先和大家介绍一下操作系统中线程机制的引入，了解线程这个概念是从何而来的。后面会讲一下操作系统中的线程管理，线程的设计模型，以及Linux中的线程。



我们先来回顾一下上一节我们对于进程的学习。

进程是程序执行的一个实例。

进程在操作系统中用PCB和进程队列表示。

而PCB主要包括了地址空间、操作系统资源、统计信息、执行状态（PC、SP、regs等）等信息。

我们回顾所学，就可以看出，进程的开销，主要包括创建、通信、切换和隔离保护的开销。

进程创建开销，包括要初始化Task\_struct，初始化所有的数据结构。而因为进程间地址空间的隔离，进程间通信都需要操作系统介入，通常至少一次数据拷贝。而对于一些非宏内核的操作系统，其通过IPC进行通信，至少两次或四次的消息传递和数据拷贝。开销更大。

而对于进程上下文切换，因为要保存和恢复CPU上下文，以及地址空间，进程执行的局部性会被破坏，从而造成更大的时间开销。

进程间通信是操作系统中主要的通信机制。主要包括三种通信的情况。

进程创建时，父进程在fork()子进程时将所有数据和状态传递给子进程

操作系统提供通信机制

共享内存：多个进程同时读写同一块内存区域；隐蔽信道

系统调用来声明共享区域；共享内存映射成功后，不需要OS介入

消息传递：通过send()/receive()系统调用建立的明确通信

IPC

开销较大——系统调用和消息传递

Lightweight IPC

多道编程最典型的例子，web server。

Fork多个进程副本来处理并发请求；每个进程映射到相同的内存地址；轮换调度这些进程

开销包括时间上的开销和空间上的开销。

空间：PCB、页表等

时间：创建数据结构时间、fork，复制地址空间时间等

而线程概念的引入，就是考虑到进程存在的诸多开销问题。

他的核心设计思想是：分离进程和他的执行状态。

进程是资源分配的主体：基于进程分配地址空间、内存、I/O等。传统进程只包含一个线程

而线程是指令执行的主体：主要基于线程管理CPU机器状态、栈执行环境等

多线程进程包含多个线程

线程在同一进程的地址空间，可共享code、data、same privileges and resources (files, sockets, etc.)

一个线程对应到一个进程，而进程可以拥有多个线程

线程是最基本的执行单元，而进程成为线程执行的容器

线程的引入最早可以追溯到IBM System/360时代。当时叫做MVT

相较于进程，线程有切换开销低、创建开销低、共享通信开销低的优势。

因为线程切换，只切换CPU寄存器状态，地址空间和资源不变，即不需要切换Cr4只向的页表，因此改变的局部性不多。因为线程的大量数据可以共享，有时还不需要陷入内核，因此创建开销很低。

相较于进程，线程存在CPU上的隔离保护，但在内存上不存在完全的隔离保护

因为其天生共享部分地址空间，所以其通信开销更低。

从总体上看，线程共享的部分，是指内存状态、I/O状态，这在同一进程内，所有线程是可见的。

其私有部分，主要是指线程私有的机器状态。包括程序执行指针（Program counter， EIP on x86）

；其他CPU相关寄存器register；线程执行环境栈：函数调用参数、临时变量、返回PC

线程的资源回收，相较于进程，线程只回收栈，而进程需要回收所有资源。

相较于进程，线程的并发性更高，更容易实现I/O overlapping

如果想要查看线程的内存足迹。

我们可以通过暂停一个程序的执行，并通过GDB检查状态

主要的GDB命令，包括attach，info 等。希望大家下去可以自行查阅相关资料，并进行尝试使用。

典型的线程库包括pthread，和windows下的 win32 thread

常用操作包括创建/销毁；暂停/继续；同步：barrier, lock

常用命令工具包括：PS命令， GDB的相关命令

第二个部分，我们来查看线程的管理操作

类似于PCB，线程的管理控制块叫做TCB，我们可以看到，他是进程PCB的一部分子集。

包括进程控制信息的部分内容，和私有的执行栈。

而进程PCB则还包括一些CPU的信息、内存地址空间的信息和I/O、文件管理信息等



典型的线程编程接口。

如我们讲过的Fork。

互斥的操作如acquire、release，条件变量相关的wait，signal，broadcast，以及一些警告的api。

这些我们会在后面的课程详细讲解。

当线程调度时，也需要线程的上下文切换。

那他需要保存什么样的数据信息呢？

需要保存栈吗？Cache和TLB该怎么办？

可能触发进程的上下文切换

什么时候触发？



过程调用，就是一次简单的函数调用。

他需要调用者或被调用者保存部分上下文。

线程和过程调用相比。

线程可以异步、乱序执行，且不能用栈保存状态。因为线程是私有栈

线程可以并行的运行在多个CPU上，而过程调用是顺序的。



好，了解了线程机制的引入和线程管理的基本概念，我们总结一下线程的设计模型

19:

看到图上，线程在不同操作系统中的表现方式。

最早的DOS系统，是one thread per process，每个进程只有一个线程，且只有一个进程地址空间。

早期的unix系统，虽然也是每个进程只有一个线程，但是是多地址空间的系统，即多进程系统。

我们知道JVM，java是一个每个进程可以有多个线程的模型。

而现在的操作系统，则基本上都是多进程系统，且每个系统有任意线程的模型。

这样设计的原因是什么？

20:

为什么线程会出现。

就是线程出现后，不再以进程为调度单位，而是以线程为调度单位。

那问题来了，内核怎么选择PCB呢？

有两种基本的方法：

1、内核知道进程中的所有线程，并去调度线程

2、内核不知道线程存在，将线程事做普通进程管理调度，利用用户态的调度器来调度每个进程中的多个线程

举个栗子，一家租车公司，把一辆车分别组给家庭a和b，a有三个成员，b有四个成员。

要求在两个家庭中round-robin，每个家庭成员每人2个小时然后轮换

两种方式：全局调度和本地调度

这两种方式有什么优劣呢？

想象不同方式下线程切换的开销；谁决定线程的调度顺序；当一个线程调用阻塞的系统调用呢？

21:

而线程的实现主要包括两种：用户态线程和内核态线程。

用户态线程就是我们上面提到过的用户态线程库进行管理的，包括pthread和win32 thread。

用户态线程在执行时，内核是不知道线程存在的。

内核太线程：

内核态直接支持线程。

22:

由一组用户态的线程库函数来完成线程的管理（run-time system）

包括创建、终止、同步、调度；没有内核的介入，内核只看到一个执行实体

对内核是透明的

线程由进程中的PC、寄存器、栈和TCB表示；所有线程操作通过用户态的过程调用完成；用户态线程操作速度是内核态线程操作速度的100X量级。 为什么？

23:

用户态线程调度分两种情况

如果线程属于同一个进程，由用户态线程库进行管理，只需要对TCB信息进行load/store，操作系统不需要介入

如果线程属于不同进程，正常的进程切换方式，由OS进行（trap in/out of kernel），操作系统需要load/store PCB 和TCB info

24:

从另一个维度，又可以分为抢占和非抢占式调度。

非抢占式调度，没有timer机制强制让线程让出CPU，线程必须自愿放弃CPU给其他线程 e.g. pthread\_yield，调度器不考虑线程历史，执行是协作式（co-operative），而非强迫式competitive

抢占式调度，通过信号模拟中断，触发强制调度 e.g. alarm，会带来许多实现问题，如对应的信号量无法作他用

25:

那什么时候会发生阻塞呢？

线程执行read()读取磁盘发生阻塞，线程等待数据从磁盘读回，整个进程阻塞直到硬盘I/O完成

解决方案：使用wrapper functions

即重载系统调用（提供同名另一个版本的系统调用），有时候也叫hook

在进入内核时检查是否会发生阻塞，E.g. select() before read()

如果调用阻塞，调度另外一个线程执行

但是这种方法过于复杂：需要处理所有的阻塞调用

好处：批处理方式执行，将同步转化为异步

26:

当然用户态线程也存在问题。

如果一个线程调用系统调用，阻塞的系统调用会阻塞整个进程

用户态线程对内核透明，不能很好的与OS进行交互和集成

操作系统可能会成为性能瓶颈，调度有idle线程的进程；阻塞一个发起了I/O请求的进程，即使该进程的其他线程还可以继续执行；当一个线程hold锁时，不调度该线程所在进程

27:

那什么是内核太线程呢？

操作系统管理所有的线程和进程；线程管理操作全部由内核完成，包括创建、销毁、调度、同步等；线程是调度基本单元，而非进程

线程调度管理；PCB不再被调度；如果一个线程阻塞，内核可以调度同一进程中的其他线程执行

这是用户态线程和内核态线程的主要区别。

28:

相较于用户态线程，内核态线程创建和管理较耗时

因为需要陷入内核，内核维护更多的数据结构。调度的粒度也更细

与内核集成较好，一个阻塞的系统调用不会阻塞整个进程

29:

好，那么这两种实现机制的区别是？请位同学回答一下

同时使用用户态和内核态线程，将一个内核态线程和一个用户态线程绑定；或者将多个用户态线程在多个内核态线程上复用

典型的如JVM

Java线程是用户态线程

在以前版本，每个进程只有一个内核态线程

需要将全部java线程在这一个内核态线程上复用

现在的操作系统中

可以在多个内核态线程上复用

Java线程可以远多于内核态线程

30:

我们再来讨论三种典型的线程映射模型。

映射，是指线程哭将用户态线程映射到内核态的方式

而在设计之初，主要有什么设计权衡呢？

对于内核态线程：物理上并行度，多少个处理器核或硬件线程？

对于用户态线程：应用级并行度，有多少并发任务需求？

不同的映射模型，代表不同的设计权衡

N:1 （many-to-one）多个用户态线程映射至1个内核态线程

1:1 （one-to-one）1个用户态线程映射到1个内核态线程

M:N （many-to-many）M个用户态线程映射到N个内核态线程

31:

N：1模型

常用语用户线程库的实现。

优势：无需系统调用——开销小；不依赖系统设施——可移植性高

不足：线程不能并行化执行；一个线程等大爱会阻塞整个进程

32:

1:1模型

常用于内核态线程的实现

优势：更高的并发度

1个线程阻塞，其他线程可以继续执行

不足：开销

导致过多的内核态线程，线程操作都涉及到内核，线程需要占用内核资源

33:

M：N模型

在BSD和windows的一些版本中实现

将任意用户态线程，随机绑定调度到任意个内核态线程之上。

优势：灵活性

充分利用多核并行，OS可以创建内核态线程

实现应用级并行，应用程序自己创建线程

不足：过于复杂

大多数应用采用1:1映射

34:

基本上所有的现代操作系统都采用了内核态线程模型

35:

我们将一下线程池的概念。这也是设计开发里经常用到的一种技术。

主要解决的问题，是面向服务器应用的优化。

为每个服务请求创建线程开销大，因为被创建线程在服务完毕后退出

请求的增多-->带来更多的线程数量，更大的服务器开销

线程池（Thread pool）预先创建一定数量的线程，等待服务；用户请求到达时，唤醒线程 （唤醒线程开销远低于创建开销）；请求结束时，线程并不直接退出，而是释放回线程池；确定线程池的容量

36:

好，我们将一下线程在Linux下的实现

37:

我们上一节学过，在Linux中，最常用的结构体，task\_struct表征的是Linux的进程描述符，包括的进程的所有上下文信息。

38:

而在Linux中，用thread\_info，来表征线程的数据结构。

按理来说，thread\_info应该是task\_struct的子集。

但是，在实现上，重复的数据结构会造成资源的浪费，以及更新上的延迟和一致性问题。

所以，在Linux实现里，这两个数据结构是互补的内容。一起构成了Linux线程的上下文状态。

还记得我们上面讲N-1的方式，不依赖系统设施

Task\_struct是体系结构无关的进程信息，这个结构和其他内核数据结构一样是在堆里面分配的。之前这个结构是放在内核栈上的，但是随着这个数据结构不断变大，开始不适合放在内核栈中。

Linux 2.6前是没有thread\_info这个结构的，

而thread\_info是体系结构相关的信息，位于内核栈的底层。

放在内核栈里的优势，只需要对stack pointer作几个位操作，就可以快速指向他，不需要用稀缺的寄存器来存储其位置。隐翠，task\_struct虽然移出了内核栈，但一些进程使用的重要的数据结构仍在内核栈中，都放在了thread\_info中。

随着per-cpu结构的盛行，调度器会缓存很多进程相关信息在per-CPU area, 这会比从内核栈栈底去查询更快。因此linux社区一直在寻求get rid of thread\_info。

With the cache, Andy says, the 1.5µs performance regression becomes a 0.5–1µs performance gain.

39:

这是thread\_info在地址空间中的位置。Thread\_info中，包含一个指针，指向这个线程所在进程的进程描述符结构。

40:

Linux线程是一个相对特殊的设计模型。

Linux线程的独特实现：将所有的线程作为标准进程实现，线程仅仅是一个与其它进程共享资源的进程，每个线程有一个唯一的task\_struct数据结构。

这也是我为什么说task\_struct和thread\_info互补的原因。

创建线程的调用，clone

普通进程fork，也可以用clone来实现

Vfork，可以通过不同参数的clone来实现

41:

这是clone系统调用的各种flags标记，

包括指定是否和父进程共享文件、文件系统信息、是否新建一个新的地址空间、等等。

42:

而Linux的内核线程。这个我们和说的设计模型不一样。

主要是指在后台进行特定操作的线程，通常为守护线程 kernel daemon；可正常被调度和抢占

与普通线程区别

内核线程没有自己的地址空间，只运行在内核态

请注意与内核态线程模型的区别！

例子：

软中断线程ksoftirqd、flushd

43:

Linux内核线程相关的编程接口，包括创建，执行、结束等内核函数调用

44:

我们在来讲一下posix线程

IEEE定义的标准，在不同OS维护应用的兼容性

定义了应用程序编程接口（API）：包括命令行shell、基础工具集；在类Unix系统和其他系统间维护兼容性

POSIX Threads，Pthreads

线程的POSIX标准；定义了创建和管理线程的API

45:

Pthreads，定义了一组C语言的数据结构类型、函数和约束；头文件在操作系统定义的pthread.h；库为的libpthread.so

分类

约100个Pthread相关API函数,都以pthread开头；线程管理- creating, joining threads etc.；互斥-Mutexes；条件变量-Condition variables ；同步-线程间同步锁和barrier

46:

让我们看一下Linux线程的演化。

LinuxThreads （2.4版以前）

对POSIX线程标准的部分实现,未能实现完善线程支持 ；当时的内核并没有为进程内线程设定相同的进程ID，信号处理机制存在问题；占用SIGUSR1和SIGUSR2来实现进程间协同机制，限制了系统了灵活性。

NTPL（Native POSIX Thread Library, 2.6及以后）

Linuxthreads被NPTL线程库所取代 ；出现:Red Hat Enterprise Linux第3版,Linux内核2.6版 ；NPTL目前是GNU C的完整组成部分

常用工具

PTT:为NPTL设计的踪迹跟踪工具

OPTS(Open POSIX Test Suite):检验是否满足POSIX标注

47:

NTPL采用了和Linuxthreads相似的途径：内核感知到的主要的抽象实体还是进程 ；线程通过clone系统调用被创建(在NPTL库中).

内核对线程原语的支持

Futex:实现线程睡眠、唤醒及竞争同步等功能

Gettid:获取线程ID，若是单线程进程，TID等于PID

Getpid:同一进程的线程返回相同的进程ID

NTPL时一种1:1模型的线程库

用户通过pthread\_create创建线程；线程作为调度实体

48:

Fork系统调用的语义

Fork只复制调用者线程还是所有线程?？

答案：Linux系统中fork()只复制调用者线程

信号处理时，

将信号传递给哪个线程?

异常出现(Segment fault)时是杀掉进程还是线程?

使用线程时

线程的函数需要可以重入(re-entrant) ；线程函数使用栈上变量，访问全局变量会有访问冲突问题

若需要使用全局变量 ：采用thread-local存储 (pthread\_key\_create) ；或定义数组，为每个线程准备专属元素

49: