《openEuler内核编程》

课程讲稿

第四章 第1讲

进程

软件所制

第四章 第1讲 进程

**学时：**2学时

**教学目的：**系统学习进程

**PPT讲稿：**



这一章我们将学习操作系统课程中的进程管理内容，一共包括12个学时。这也是操作系统课程中比较重要的部分



这一章一共有7讲内容，包括进程、线程、内核进程隔离、同步、信号量、IPC，以及操作系统调度。今天的内容是第一讲，进程。



这一节也是分三部分，前面一部分先和大家介绍一下操作系统中进程机制的引入，了解进程这个概念是从何而来的。后面会讲一下操作系统中的进程管理，以及进程上下文状态。其中进程上下文切换是一个我们需要重点学习的部分。



1952-1964年IBM开发的早期的计算机，主要是做一些批处理作业。比如常驻监督程序，主要用来自动装填纸卡。客户为IBM计算机开发的操作系统，包括通用汽车、通用电气在IBM机器上的操作系统，包括一些车控、中控的系统。这类操作系统，一次只处理一个程序，调试的时间也很长，一般以天为单位。

而在IBM 7090计算机上，第一次实现了多个程序同时运行，当时叫做CTSS，Compatible Time Sharing System，进程这个概念才开始登上历史舞台。

1990年Fernando J. Corbato 以此获得了图灵奖

这里，我们区分两个基本概念，单道编程和多道编程。其中单道编程，是早期的微软DOS系统和Mac系统，主要做批处理作业，同一时刻只有一个线程执行。这样做，简化了当时操作系统的设计与实现，去除了并发性。我们现在用的电脑、编程的语言和方式，让我们知道这样是很没有效率和浪费资源的。因此，多道编程，即同一时刻执行多个线程成为操作系统设计的主流方向。包括Multics，Unix系统、Windows NT-8， MAC OSX， android，ios都属于多道编程的系统。多道编程增加了OS设计与实现的难度，确提高了并发性，并发性带来的是性能的提升，和高效的资源利用。

多道编程本身，需要满足四个主要的需求，即virtual machine abstraction 硬件虚拟抽象，每个程序都期望独占机器资源，怎么使用，怎么划分，就需要对底层硬件资源进行抽象，从而保证上层系统和应用的使用。concurrency 并发度，这是多道编程的本质，通过程序间并发提高硬件资源的利用；protection，多个应用执行过程中，需要实现对于共享资源的仲裁和独占资源的保护。Coordination，即程序间需要协作，才能提升性能。

而进程就成了满足这些需求的关键。进程本身，提供了多道并发、隔离保护、资源共享和调度协作。

那什么是进程呢？

进程是程序执行的一个实例。

我们通过top或者htop都可以看到系统下执行的进程。有应用进程，也有系统进程。

为了保护用户之间的安全，用户在系统中有独立私有的账户。

那用户加载程序的时候，多个用户可以加载相同程序吗？一个用户可以加载相同程序的多个实例吗？

答案是可以，这也是进程这个机制引入的原因

进程是程序执行的一个实例。那两个实例之间的关系是？

相互独立，又相互依存的。不同应用的进程相互独立，但是存在父子进程，进程间在内核又存在功效的资源。

我们来看一个具体的例子。

这个程序里，myval这个静态变量。

你在执行过程中，会发现，静态变量的地址总是相同的，但其值确实不同的。

因为，这里的地址不是绝对的物理地址，而是相对的虚拟地址。每个进程都有自己独立的内存地址空间。

那这样做的优势，就是让编译器不需要关心具体的地址。而且进程的地址空间，可以远大于实际内存。

我们再来区分一下程序和进程的概念，

程序不是进程。程序是静态的，是由代码和数据组成的。而进程是动态的，它包括代码的执行，数据的处理，输入和输出数据、操作系统的资源，以及其他

程序和进程也不是1对1的映射关系。

通过进程的机制，我们可以实现的两种虚拟机制。包括CPU虚拟和内存虚拟。操作系统通过进程上下文环境来实现对CPU的虚拟。通过地址空间机制来实现对内存系统的虚拟。

两种虚拟机制共同构筑了操作系统对底层硬件资源的虚拟和抽象。从而实现多到并发、隔离保护、资源共享和调度协作。

我们看一个例子。Chrome浏览器，你们会在top的时候看到多个进程，不止包括他的标签页，还包括他运行时的插件。

如果一个页面崩溃，不会使得整个浏览器崩溃。

好，我们对进程的概念做一个总结。

进程是一种操作系统对程序执行的抽象。是一种执行的实体。是系统调度的基本单位，是一个程序的动态执行上下文

程序执行的实体，包括了程序执行状态，主要包括寄存器状态，和内存数据状态

同时，进程也是资源分配的载体。他拥有独立的地址空间:memory (address space), file descriptors, file system context, ... 。也拥有独立I/O 状态:file descriptor table, network sockets

程序协调的机制：进程间通信IPC、pipe、socket



介绍完了进程的概念，我们下面介绍一下操作系统中的进程管理

进程在内核中的表示，是由PCB也即进程控制块，和进程队列来表示的。

进程管理，主要是进程生命周期的管理和进程状态的变迁维护，即创建，销毁、调度等。



进程的组成。

一个进程包括程序执行过程中的所有状态，包括内存地址空间，执行程序的代码、数据，执行stack：包含所有调用的状态，Program counter （PC）：指向下一个指令，通用寄存器值。操作系统资源集合：打开的文件、网络链接等



在内核中，每个进程的创建与销毁都由内核执行，每个进程都需要在内核登记信息

内核用进程控制块（Process Control Block, PCB）来保存进程的信息

PCB是进程在内核中的表示，也是一种索引

19:

PCB包括了进程状态，机器状态，调度与统计信息、内存管理信息、I/O状态信息等

将CPU硬件状态从一个进程切换到另一个进程——叫做一次进程上下文切换（Context Switch）

正常一秒钟可以发生100到1000次

调度时：进程放弃CPU给其他进程

同一时刻只有一个进程在运行

给重要进程更多的运行时间

当操作系统销毁一个运行的进程，会讲当前寄存器的值保存到该进程的PCB，而当OS开始唤醒一个新的进程，他会将保存在PCB的值加载进硬件寄存器。

问：这时候运行中的代码会发生什么？

20:

Linux中，以task\_struct这个结构体表征PCB，记录了我们上面提到的所有进程状态信息。

Linux 5.5.10中大概有670行的代码

21:

在linux 中每一个进程都由task\_struct 数据结构来定义. task\_struct就是我们通常所说的PCB.它是对进程控制的唯一手段也是最有效的手段. 当我们调用fork() 时, 系统会为我们产生一个task\_struct结构。然后从父进程,那里继承一些数据, 并把新的进程插入到进程树中, 以待进行进程管理。因此了解task\_struct的结构是我们理解任务调度的关键。

22:

为什么要这么定义进程状态？

通过位操作，可以迅速确定。一个指令操作，

23:

Task\_struct里通过一个双向链表，讲所有进程PCB链接起来，方便查找进程。

那怎么访问所有的进程呢？

Linux内核提供了两个函数，一个foreach轮训，一个按照链表链接，寻找下一个。

24:

我们可以看到，pid表示进程标示符，tgid则表示整个线程组中领头的线程id。

系统调用 getpid() 返回什么？

Linux系统允许用户使用一个叫做进程标识符的PID来标识

进程，PID顺序编号，新创建进程是前一个进程的PID加1，

不过PID值有一个上限，达到上限之后再开始循环使用闲置

的小PID。

25:

进程PCB中同时记录了其父子进程、兄弟进程关系。

\*parent是该进程现在的父进程，有可能是”继父“\*. \*这里children指的是该进程孩子的链表，可以得到所有孩子的进程描述符\*. \*sibling该进程兄弟的链表，也就是其父亲的所有孩子的链表.

26:

这个字段将task对应的三种PID都存在一个数组中，因此，要取PID，则只需取PIDTYPE\_PID对应的实例即可，同理也可以取得PGID和SID。

27:

Current宏是内核中利用很多的一个宏。

他是一个全局指针，指向当前进程的struct task\_struct结构体，即表示当前进程。

例如current->pid就能得到当前进程的pid，current-comm就能得到当前进程的名称。

通过esp的值获得current宏的值

从上图可以看出，linux把两个不同的数据结构紧凑地存放在单独为进程分配的存储区域内：进程堆栈和进程描述符thread\_info。且进程描述符起源与该内存区的开始，堆栈起源于内存末端。esp寄存器是CPU栈指针，用来存放栈顶的指针。从用户态刚切换到内核态的时候，堆栈是空的，esp指向0x015fbfff,随着数值的写入，它的值开始递减，内存共8KB，由于进程描述符不可以超过1000个字节，所以堆栈可以写入约7200字节的内容，即esp的有效位，是13位，所以屏蔽掉该13位的值，就可以获得进程描述符的起始地址，就是current指针指向的地址。

28:

我们再来看进程的生命周期

包括new、ready、running、waiting、terminated

问：同时可以有多少个进程在同时运行

Waiting状态通常是在等待事件触发，一般是发起I/O请求，等待I/O完成之后进行

29:

Linux进程状态包括这五种。

30:

我们也可以通过top命令，查看当前进程的状态。

31:

这张图给出了5这种状态变换的流图。

32:

进程队列，用于维护管理进程状态的变迁。

主要包括三种，运行队列，等待队列，阻塞队列。

而每一种会有很多等待队列，每种类型的等待队列都有一个，包括磁盘的，输入的，时钟中断的，网络的等等。

33:

PCB只是存储在系统内存中的数据结构

当一个进程被创建，OS为其分配一个PCB，初始化，并将其放置到ready 队列中

当进程执行过程中，如I/O操作，他的PCB被从一个队列放置到另一个队列

当进程结束，PCB被释放

34:

进程被另一个进程创建。父进程是创建者，子进程被创建 （unix: PPID）

父进程定义或贡献资源和权限给自己的子进程

UNIX: process user id被继承

什么时候会创建进程？

系统初始化时

用户请求创建一个新进程

正在运行的进程执行了创建进程的系统调用

问题，编译的时候，谁创建了gcc? Gcc内部包括什么？谁创建了cpp和cc1？

谁创建了shell？

谁创建了第一个进程 （PID 0 or 1？）

35:

这是Unix的系统进程树

从进程0，即root进程，创建内核守护进程，和init进程，再由init进程创建其他进程。

36:

内核选择一个ready的进程，为其分配一个处理器的时间片，并开始执行（时间片倒计时）

如何选择？

进程调度算法

什么时候选择？？

37:

进程进入等待（阻塞）的情况

请求并等待系统服务，无法立即完成

启动某种操作，无法立即完成

需要的数据没有到达

只有进程自身才能知道何时需要等待某种事件的发生

38:

进程会被抢占的情况，主要包括两种：高优先级进程就绪、进程的时间片用完

39:

唤醒进程的主要情况，包括：被阻塞进程需要的资源可被满足、被阻塞进程等待的事件到达

进程只能被别的进程或系统进程唤醒

40:

进程结束的情况：

正常退出（自愿）

错误退出（自愿）

致命错误（强制性的）

被其他进程kill （强制性的）

41:

我们来看linux下进程管理的主要函数

Fork系统调用会创建当前进程的副本。

包括函数，堆栈、寄存器状态等。

主要的好处是什么？

怎么确定哪个是父进程 哪个是子进程？

42:

Fork()创建和初始化一个新的PCB；创建一个新的地址空间（但，复制父进程内容）；初始化地址空间：复制父进程的整个地址空间内容；初始化内核资源（指向父进程使用的资源，如open files）；将PCB放于ready队列

Fork return twice

将子进程的pid返回父进程，返回“0”给子进程

43:

这段代码会返回什么？

44:

这个输出顺序是一定的吗？

45:

复制了父进程的地址空间和PC寄存器指向

46:

执行过程中会有差异

他们已经是两个进程

47:

父子进程可以同时运行，或者在单核上被os调度

48:

为什么要用fork（）

复制父进程大部分内容呢

父子进程协同工作

依赖父进程的数据完成任务

最简单的，web server，需要启动多个子进程服务接收到的请求。

子进程需要获得父进程得到的请求信息。

49:

我们发现Fork复制大量数据——有很大开销

因此，出现了写时复制copy-on-write

有相同的指标指向相同的资源

直到子进程尝试修改资源时，系统会真正复制一个副本给子进程

避免被修改的资源被直接察觉，对其他进程透明

如果没有修改资源，就不会有副本建立，减少开销

50:

Linux中的相关函数

Exec（）系统调用，在fork()之后使用，以新的程序来替换进程的代码段或地址空间。即， 父子进程运行不同的程序

注意：代码段和数据段，即整个地址空间被替换

Exec（）系统调用包括很多变种，大家可以通过man函数或者查询Linux代码，查看他们之间的区别。

51:

执行了exec（）后，进程的什么没有改变？

进程的id没有改变。

除此之外，执行新程序的进程还保持了原进程的进程ID和父进程ID、实际用户ID、组ID、会话ID、控制中断、时钟时间、当前工作目录、文件所、进程信号屏蔽、未处理信号等等。

52:

Exec()

停止当前进程

加载程序“prog”到进程的地址空间

为新程序初始化硬件上下文和参数

将PCB放于ready队列

Note: exec()并没有创建一个新的进程

Exec()成功调用——没有返回

返回意味着？

意味着执行出错。

53:

Wait系统调用。

父进程需要等待子进程执行完成

变体：waitpid()

子进程执行完毕，wait()才返回父进程

54:

Exit系统调用：结束进程执行。主要是为了释放资源

为什么需要OS作这些操作？

如果不执行，浪费物理资源！

55:

Linux下，一个父进程先于子进程消失会发生什么？

该子进程成为孤儿进程

是所有子进程被杀死，还是所有子进程将init作为父进程？

56:

Linux所有孤儿进程讲进程1作为父进程

从Linux3.4开始，进程可以触发prctl()系统调用，Orphaned children proces将会以最近的ancestor process为父进程，不一定为process #1

作出改变，是因为linux内核中的很多内核监控进程，kernel daemon process，需要监控这些子进程的运行。而不是简单的把orphaned children交给进程1

57:

这部分，我们来讲解进程上下文状态

58:

进程执行环境，叫做进程上下文状态。每个进程独有，实现了处理器的虚拟

上下文组成，包括CPU寄存器、地址空间、进程所分配的OS资源

这个context，还可以分为内核上下文和用户态上下文。

59:

为什么要上下文切换。

需要对硬件执行资源，也即CPU进行分时复用。

每个虚拟CPU由这些结构容纳表示

如何从一个虚拟CPU切换到另一个虚拟CPU呢？

何时触发切换呢？

定时器中断，自愿释放资源，放弃，时间片耗尽、I/O中断等

60:

如图所示，程序1 睡眠的时候，OS通过schedule函数调度，保存线程，并恢复其他进程执行。

61:

进程切换过程中的地址空间变化，

我们可以看到地址空间内，firefox恢复和被调度的时候，其地址空间内容的变化。

62:

进程切换本质是对执行上下文环境(Context)的切换

地址空间，CPU寄存器状态

内核切换代码是系统开销(overhead )

切换开销即最小的切换时间

63:

问题1:如何获得控制

问题2:如何进行上下文切换

64:

Linux须从用户态“陷入”内核态

协同式 :进程自愿放弃控制权给OS。执行yield等调用；OS唤醒进程

抢占式: OS 通过定时器剥夺processes执行权。进程被分配相应时间片配额；随着时间定时终端，时间片配额递减；当时间片用光时，进程被抢占

65:

如何实现进程切换。

进程上下文切换与机器体系架构直接相关

OS 必须在保存状态时不更改状态 。运行代码去保存寄存器?而保存代码自身会更改寄存器

Solution: 软硬协同

CISC机器拥有特出的指令来保存和恢复所有寄存器

RISC:为内核预留寄存器或者(have way to carefully save one and then continue)

66:

切换时所有的寄存器都需要保存吗？

只需要保存“跨函数调用需要保存”的寄存器

下表给出了x86和mips架构下，切换时需要保存的寄存器

67:

Linux关于上下文切换的代码在内核的sched.c文件下。

内核代码主要在swtich\_to这个宏中，主要是架构相关的汇编指令

进程切换点在于栈指针的切换。当栈指针改变时，所有的局部变量随之改变