《操作系统》

课程讲稿

第4章 第6讲

信号处理机制

软件所制

第四章 第6讲 信号处理机制

**学时：1**学时

**教学目的：**学习信号处理机制，了解进程间如何利用中断机制来进行通信……

**课程时间线：**

章节简介

概述

进程中的信号

信号响应

信号操作

5min

10min

25min

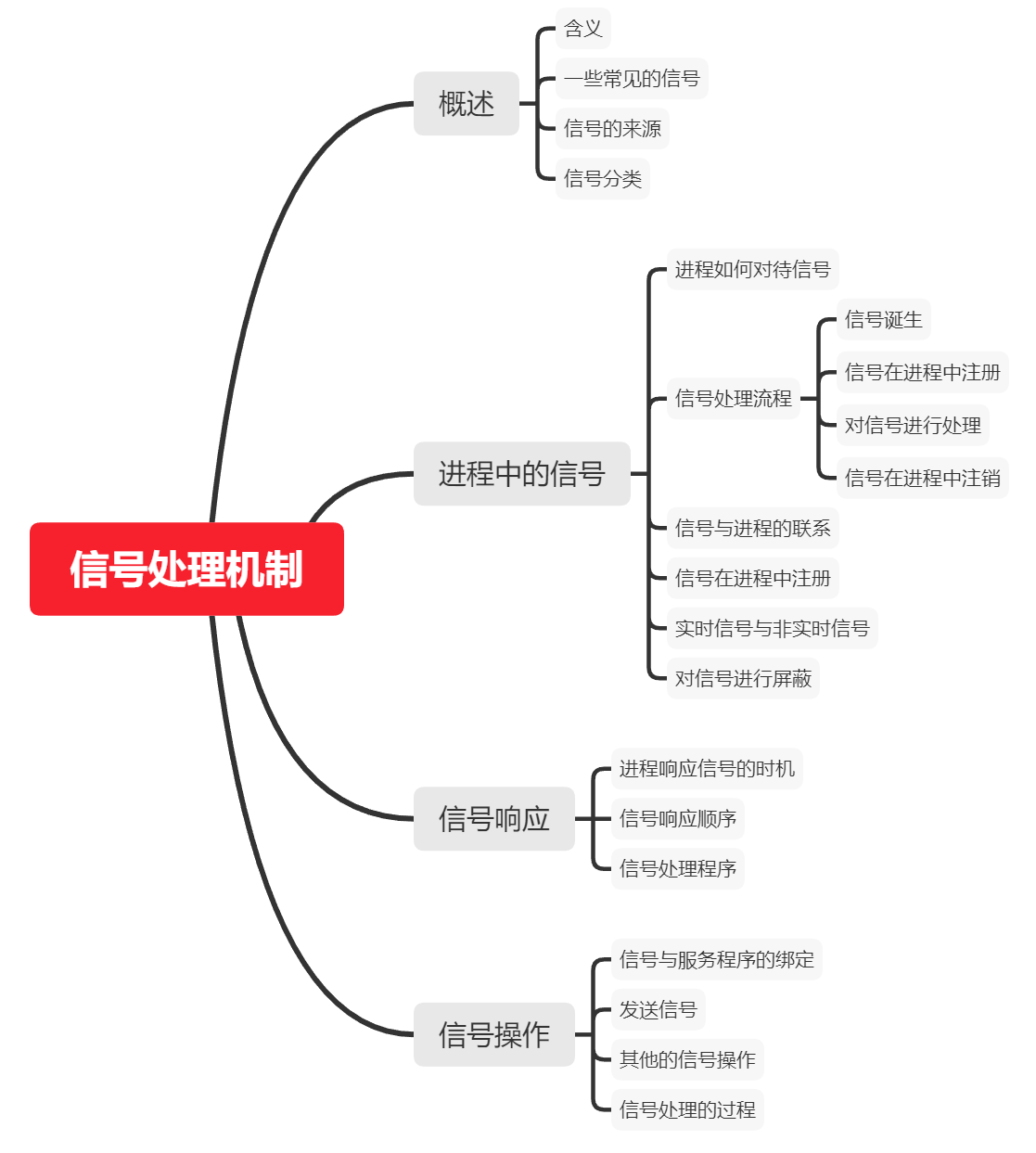
35min

45 min

**课外参考读物：**

《深入理解LINUX内核》

**知识框图：**



**PPT讲稿：**

1. 今天我们就要进入中断机制的最后一节了。
2. 同上一节的系统调用一样，信号机制也是利用中断机制来实现的。
3. 本节中我们将分为四块内容来介绍系统调用，分别是概述、进程中的信号、信号响应和信号操作的处理过程，我们先来看看下概述这一部分。
4. 信号是一种异步机制，其实说得通俗一点，我向你发出信号的意思就是通知你一个消息，至于这个对消息如何反应就是你自己的事了。
5. 下面是Linux中一些常见的信号。值是用于识别信号的，缺省动作是当前进程未定义该信号的操作时默认的操作，可以让当前进程暂停、停止、恢复运行，还可以让当前进程忽略该信号。
6. 信号的来源主要有终端或者外设的输入、硬件或软件产生的异常、或者系统中的其他进程。当然，一个进程要有一定的权限才能向其他进程发送信号。只有具有root权限的super user才可以随意向其他进程发送信号，例如发送杀死其他进程信号。而普通user的进程，只能向属于同一user的进程发送信号。
7. 下面我们按照信号发出的原因来对信号进行一个简单的分类。
   1. 与进程终止相关的信号：当进程退出，或者子进程终止时，它们会向同组线程或者父进程发出这类信号。
   2. 与进程例外事件相关的信号：如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误，这个信号由内核发送给越界的进程。
   3. 与执行系统调用时遇到非预测错误条件相关的信号：如执行一个并不存在的系统调用。当进程执行一个并不存在的系统调用时，内核会向这个进程发送这个信号。
   4. 在用户态下的进程发出的信号：如进程调用系统调用kill向其他进程发送信号。例如我们在shell里kill掉某个进程，这个时候shell这个进程就会向目标进程发送这个信号。
   5. 与终端交互相关的信号：如用户关闭一个终端，或按下break键等情况。当我们点击鼠标关掉某个进程时，内核就会像这个进程发送一个信号。
8. 同7
9. 下面我们来看看信号和进程之间的关系
10. 前面我们讲过信号的缺省操作，就是当前进程未定义该信号的操作时由内提供的默认操作。此外，我们还可以自己定义信号的操作方法，这就类似于中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。最后我们还可以偷懒，忽略某个信号。
11. 一个完整的信号处理流程可以分为以下四个阶段：
    1. 信号诞生：也就是内核或者进程有向其他进程发送信号的需求。
    2. 信号在进程中注册：即内核或某个进程向目标进程发送信号，据方法就是在目标进程的相关数据结构中注册这个信号。
    3. 对信号进行处理：调用进程自己注册的信号处理函数或者缺省函数来处理这个信号。
    4. 信号在进程中注销：当信号处理函数完成执行之后就将这个信号在进程的相关数据结构中抹去。
12. 下面来看看信号处理机制中的一些重要的数据结构。首先看看代表单个信号的数据结构sigqueue。其中比较重要的成员有：
    1. list：挂接在一个进程上的sigqueue实例组成的链表
    2. info：此信号的一些数据，包括发送进程的信息，信号本身的信息

每个进程都可以挂接多个待处理的信号，因此每个进程都有一个sigqueue实例组成的链表。

1. 在sigqueue中的info是个siginfo类型的成员，保存信号的一些信息，有信号的编号、信号是否由错误引发的标志、信号来源的详细信息以及一些附加的信息。通过sigqueue访问成员info就可以得到这些信息了。
2. 前面介绍的时单个的信号，接下来看看进程和信号是怎么联系到一起的。我们知道进程在内核中的实体就是task\_struct，要和进程相联系肯定就要在task\_struct中有所体现。在进程的task\_struct中有sigpending类型的成员pending，sigpending的定义如下。其中的list用于指向挂在此进程上的信号组成的链表，也就是刚才我们讲的sigqueue实例组成的链表。另外一个signal成员是此进程的待处理信号集位图，每一位的取值代表当前进程是否挂接了对应的信号，即signal的每一位就是对应信号是否在当前进程注册的指示灯。
3. signal是一个sigset\_t类型的数据，是一个位掩码，所包含的比特位数必须大于等于所支持的信号类型数量。因此内核使用了一个unsigned long数组，数组的长度根据宏\_NSIG和\_NSIG\_BPW计算。计算的方法如ppt下面的这个宏所示。可以看到在32位系统和64位系统下的\_NSIG\_BPW有所不同。最后的sigset\_t被定义成了一个无符号长整型数组，数组的位长即为可以在该架构下可以定义的信号个数。
4. 下面的这个数据结构就与信号的处理函数相关了。其中重要的成员有如下两个：
   1. count：共享此sighand\_struct实例的进程数目（clone的父子进程共享），这意味着一个sighand\_struct实例可以由多个进程共享，也就是多个进程可以对同一个信号使用同一个处理函数。
   2. action数组：元素为k\_sigaction类型，保存了\_NSIG个不同的信号处理程序。
5. k\_sigaction是内核中用于管理信号处理函数的数据结构，它的作用主要体现在内部的一个sigaction类型的成员上，看看sigaction的定义，用于描述信号处理程序。里面有指向信号处理程序的函数指针、对于信号处理方式的约束标志和一个位掩码，这个位掩码的每个比特位对应于一个信号，用于在信号处理程序处理信号期间阻塞其他信号。
6. 这张图就反映了进程是如何与信号进行关联的，在task\_struct中的pending指向了待处理的信号链表，右边这些空的方块就是信号的sigqueue实例，sighand则关联到了一些信号处理函数。
7. 那么为什么要屏蔽某些信号呢？看看下面这个例子我们或许就能明白了。
8. 接下来看看信号是如何在进程中注册的：
   1. 当有一个新的信号要挂接到进程上时，内核会为其分配一个新的sigqueue实例，将关于此信号的各项信息填入sigqueue实例的info成员内。
   2. 根据信号的类型，将进程的sigpending实例的signal成员（信号位图）的对应位置1
   3. 将创建的sigqueue实例链入sigpending实例的list链表。

这里对信号还有一种分类方式，即分为实时信号与非实时信号。它们在进程中 注册的方式也有所不同，我们先来看看实时信号与非实时信号到底是怎么回事。

1. 实时信号与非实时信号之间的区别就体现在注册的时候。
   1. 当一个实时信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，故该信号不会丢失，因此实时信号又叫做"可靠信号"。这意味着同一个实时信号可以在同一个进程的未决信号链中占有多个sigqueue结构
   2. 但是当一个非实时信号发送给一个进程时，如果该信号已经在进程中注册（通过sigset\_t signal指示），则该信号将被丢弃。因此非实时信号又叫做"不可靠信号"。这意味着同一个非实时信号在进程的待处理信号链中，至多占有一个sigqueue结构。

这是因为实时信号的重要性较高，因此每个信号都要进行处理，而非实时信号的重要性较低，在合适的情况下可以忽略。

1. 进程可以根据自身情况对各种信号进行屏蔽。刚才我们讲过在sigaction中可以通过位掩码来屏蔽当前信号处理函数在执行时到来的其它信号，但是这个屏蔽也只是在信号处理过程中，当这个信号处理过程结束之后依然无法根据需要屏蔽有的信号。因此在task\_struct也使用了一个sigset\_t类型的成员blocked来直接屏蔽当前进程所不关心的信号。
2. 知道进程和信号之前的关系后，我们再来看看进程是如何响应信号的。
3. 和响应中断一样，进程响应信号也有一定的时机：
   1. 如果接收进程在睡眠，把进程从睡眠队列移到就绪队列，设置eip至信号处理函数，这意味着进程一开始运行就直接处理这个信号。
   2. 如果进程在运行，每轮调度周期都要检查，也就是当进程被调度进入cpu运行时都要检查是否有待处理的信号，如果有则处理，如果没有则继续。
   3. 每次使用系统调用之后，都要先检查进程是否有待处理信号

可见信号处理都是从内核态切换到用户态的时候发生。由于用户态和内核态的切换是很频繁的，因而信号通常能很快地得到目标进程的响应，看起来就跟中断的效果一样。

1. 接下来看看进程时如何响应信号的。前面我们总结了信号响应的时机，都是从内核态切换到用户态的时候发生。因此我们直接看从都是从内核态切换到用户态相关的函数。在内核模式下会使用exit\_to\_usermode\_loop函数返回用户模式，exit\_to\_usermode\_loop里面调用了 do\_signal函数，再通过do\_signal里面的get\_signal函数将挂接在信号处理队列上且未被屏蔽的信号按照位图给定的顺序依次进行处理。对于实时信号与非实时信号的处理也不一样：
   1. 不可靠信号：内核将把这个信号从进程对应的位图以及pending中移除；
   2. 可靠信号：内核会把该信号从进程对应的pending队列中移除，如果pending队列中不再有此信号，再将位图中的对应位清零。
2. 信号响应的顺序是严格按照信号位图来的，如果一个进程有多个待处理信号，则对于同一个待处理的实时信号，内核将按照发送的顺序来处理信号。如果存在多个不同的待处理信号，则值（或者说编号）越小的越先被处理。如果即存在不可靠信号，又存在可靠信号（实时信号），将优先处理不可靠信号。那么思考一下为什么要优先处理不可靠信号？

当有自己的同类在待处理队列中时，不可靠信号会被丢弃。

有点同情弱者的意思。

1. 一个进程可以处理多种信号，同一重信号在不同进程中的处理方式也可以不同，因此我们可以自己设计进程对于信号的处理方式。前面讲过，信号处理函数由进程的sigaction实例中的sa\_handler指向，实际上就是一个回调函数。我们以及知道信号处理函数是在内核态返回用户态的时候被调用的，并且由于每个进程可以定义自己的信号处理程序，故存在安全风险。因此信号处理程序在用户空间实现，也在用户空间运行。
2. 内核为用户程序提供了三个默认的信号处理程序，分别为：
   1. SIG\_DFL：采取默认操作，也就是缺省操作。
   2. SIG\_IGN：忽略该信号
   3. SIG\_ERR：返回产生这个信号的错误码，
3. 有了前面知识的铺垫，我们就可以学习最后一节了。
4. 下面看看如何讲信号和我们自己定义的信号处理程序进行绑定。早期Linux主要用signal系统调用来实现信号与服务例程的绑定，但是由于这个系统调用功能比较单一，所以现在已经较少使用了。
5. 当前的Linux主要采用sigaction系统调用来实现信号与服务例程的绑定，参数里面涉及的数据结构我们刚才已经讲过。sigaction相较于signal引入了sigaction结构体，支持了一些信息的传递，并且有修改信号处理函数的功能，这一点是老的signal系统调用所不具备的。
6. 信号发明出来就是为了通信的，为了通信内核就必须提供一系列功能函数，接下就来看看关于信号的一些其他的操作。系统调用kill看起来非常吓人，好像要杀死某个进程一样，但是它只是用作向进程发送信号，这个系统调用非常简单，就是向pid代表的进程发送sig这个信号。
7. 前面的kill只能向进程发送信号，当信号处理函数需要参数的时候就不再适用了。因此出现了sigqueue，sigqueue这个函数的功能比kill更加强大，它能在发送信号的同时传递信号处理函数所需的参数。
8. 其他的操作还有alarm延时发送操作、abort终止操作，以及强制睡眠操作。
9. 最后看看信号的完整处理过程。在每个处理信号的时机（从内核态返回用户态）到来时，内核都会检查当前进程对应的TIF\_SIGPENDING标志位（若为1则有，否则没有），若没有待处理的信号，则跳过信号处理阶段直接进入用户态运行。这里的TIF\_SIGPENDING标志位每个进程都会对应一个，有待处理信号时则为1，否则为0。思考一下，TIF\_SIGPENDING标志位是什么时候被置1的？（信号被发送到进程的待处理信号链上时）；那又是什么时候被置0呢？（当前进程的待处理信号都处理完毕时）

若TIF\_SIGPENDING为1 ，则从位图中按照信号的响应次序来选择首先要响应的信号

1. 如我们前面所讲，当内核发现当前进程有某个待处理信号时，就会在当前进程注册的信号处理函数中查找是否有对应的处理函数，如果有则执行之；若没有则执行缺省操作。
2. 当信号处理函数执行完毕后，内核会再次查询该进程是否还有待处理的信号，如果有则再次进入信号处理程序，若没有则将TIF\_SIGPENDING标志位置0，返回到信号处理前的程序状态继续执行。