 如果我们需要用户程序能够实时地了解文件系统上某一个目录的变化情况，从实时这个角度出发，显然，我们需要有一个中断机制。我们都知道，硬件中断能够实时地把系统某一个部件的情况反映给中央处理器，同样的，要想把位于系统内核中的文件系统的情况实时地反映给用户程序，我们也需要一个由操作系统内核到达用户进程的软件中断机制。熟悉 Linux 系统编程的读者朋友们立即就会想到，这个中断机制在 Linux 系统中早已就有了，这就是信号传递 signal。 找到了信号传递这样一个中断用户进程的机制，一切似乎都已齐备，看来可以动手实现这样一个 Linux 文件系统的守护神，来实时地监视文件系统的变化情况，并且及时地把消息通知给用户程序了。不过且慢，让我们搜索一下 Linux Kernel，看看是否有别人也在做同样的工作。哈哈，果不其然，原来这样一个实时地监视文件系统情况的机制早已在 Linux 内核中实现了。下面一段就是取自 Linux Kernel 文档的一段小小例程，说明了 Linux Kernel 中的 dnotify 功能的用法。dnotify 就是指 directory notification，监视文件系统上一个目录中的情况。 #define _GNU_SOURCE /* needed to get the defines */ #include /* in glibc 2.2 this has the needed values defined */ #include #include #include static volatile int event_fd. // 信号处理例程 static void handler(int sig, siginfo_t *si, void *data) { event_fd = si->si_fd. } int main(void) { struct sigaction act. int fd. // 登记信号处理例程 act.sa_sigaction = handler.

sigemptyset(amp.act, NULL). // 需要了解当前目录"."的情况 fd = open(".", O_RDONLY). fcntl(fd, F_SETSIG, SIGRTMIN). fcntl(fd, F_NOTIFY, DN_MODIFY|DN_CREATE|DN_MULTISHOT). /* we will now be notified if any of the files in "." is modified or new files are created */ while (1) { // 收到信号后，就会执行信号处理例程。 // 而 pause() 也就结束了。 pause(). printf("Got event on fd=%d\n", event_fd). } } 上面这一小段例程，对于熟悉 Linux 系统编程的读者朋友们来说，是很容易理解的。程序首先注册一个信号处理例程，然后通知 Kernel，我要观察 fd 上的 DN_MODIFY 和 DN_CREATE 和 DN_MULTISHOT 事件。（关于这些事件的详细定义，请读者朋友们参阅文后所列的参考资料。） Linux Kernel 收到这个请求后，把相应的 fd 的 inode 给做上记号，然后 Linux Kernel 和用户应用程序就自顾自去处理各自的别的事情去了。等到 inode 上发生了相应的事件，Linux Kernel 就把信号发给用户进程，于是开始执行信号处理例程，用户程序对文件系统上的变化也就可以及时的做出反应了。而在这整个过程中，系统以及用户程序的正常运行基本上未受到性能上的影响。这里还需要说明的是，dnotify 并没有通过增加新的系统调用来完成它的功能，而是通过 fcntl 来完成任务的。增加一个系统调用，相对来说是一个很大的手术，而且如果设计不当，处理得不好的话，伤疤会一直留在那里，这是 Linux Kernel 的开发者们所非常不愿意见到的事情。