链滴

Linux 异步通知——信号

作者：zhang-ke-wei

原文链接：https://ld246.com/article/1598176562941

来源网站：链滴

在介绍信号之前先回忆一下中断，中断是指计算机运行过程中，出现某些意外情况需主机干预时机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运。中断是处理器所提供的一种异步机制。我们配置好中断以后就可以让处理器去处理其他的事情了，中断发生以后会触发我们事先设置好的中断服务函数，在中断服务函数中做具体的处理。
<h2 id="一-异步通知简介">一、异步通知简介</h2>
信号类似于硬件上使用的“中断”，只不过信号是软件层次上的。算是在软件层次上对中断的一模拟，驱动可以通过主动向应用程序发送信号的方式来报告自己可以访问了，应用程序获取到信号以就可以从驱动设备中读取或者写入数据了。整个过程就相当于应用程序收到了驱动发送过来了的一个断，然后应用程序去响应这个中断，在整个处理过程中应用程序并没有去查询驱动设备是否可以访问一切都是由驱动设备自己告诉给应用程序的。
ps:阻塞、非阻塞、异步通知，这三种是针对不同的场合提出来的不同的解决方法，没有优劣之，在实际的应用中，根据自己的实际需求选择合适的处理方法即可。
异步通知的核心就是信号，在 arch/xtensa/include/uapi/asm/signal.h 文件中定义了 Linux 所持的所有信号，这些信号如下所示：
<pre><code class="highlight-chroma">#define SIGHUP 1 /* 终端挂起或控制进程终止 */
#define SIGINT 2 /* 终端中断(Ctrl+C 组合键) */
#define SIGQUIT 3 /* 终端退出(Ctrl+\组合键) */
#define SIGILL 4 /* 非法指令 */
#define SIGTRAP 5 /* debug 使用，有断点指令产生 */
#define SIGABRT 6 /* 由 abort(3)发出的退出指令 */
#define SIGIOT 6 /* IOT 指令 */
#define SIGBUS 7 /* 总线错误 */
#define SIGFPE 8 /* 浮点运算错误 */
#define SIGKILL 9 /* 杀死、终止进程 */
#define SIGUSR1 10 /* 用户自定义信号 1 */
#define SIGSEGV 11 /* 段违例(无效的内存段) */
#define SIGUSR2 12 /* 用户自定义信号 2 */
#define SIGPIPE 13 /* 向非读管道写入数据 */
#define SIGALRM 14 /* 闹钟 */
#define SIGTERM 15 /* 软件终止 */
#define SIGSTKFLT 16 /* 栈异常 */
#define SIGCHLD 17 /* 子进程结束 */
#define SIGCONT 18 /* 进程继续 */
#define SIGSTOP 19 /* 停止进程的执行，只是暂停 */</code></pre>

#define SIGTSTP 0 /* 停止进程的运行(Ctrl+Z 组合键) */
#define SIGTTIN 2 /* 后台进程需要从终端读取数据 */
#define SIGTTOU 2 /* 后台进程需要向终端写数据 */
#define SIGURG 2 /* 有"紧急"数据 */
#define SIGXCPU 4 /* 超过 CPU 资源限制 */
#define SIGXFSZ 5 /* 文件大小超额 */
#define SIGVTALM 26 /* 虚拟时钟信号 */
#define SIGPROF 7 /* 时钟信号描述 */
#define SIGWINC 28 /* 窗口大小改变 */
#define SIGIO 29 * 可以进行输入/输出操作 */
#define SIGPOLL SGIO
/* #define SIGLOS 29 */
#define SIGPWR 0 /* 断点重启 */
#define SIGSYS 31 /* 非法的系统调用 */
#define SIGUNUSD 31 /* 未使用信号 */

</code></pre>
除了 SIGKILL(9)和 SIGSTOP(19)这两个信号不能被忽略外，其他的信号都可以忽略。这些信号相当于中断号。不同的信号对了相应的处理函数。如果要在应用程序中使用信号，那么就必须设置信所使用的信号处理函数，在应用程序中使用 signal 函数来设置指定信号的处理函数， signal 函数原如下所示：
<pre><code class="highlight-chroma">sighandler_t signal(int signum, sighandler_t handler)
</code></pre>
signum：要设置处理函数的信号。
handler： 信号的处理函数。
返回值： 设置成功的话返回信号的前一个处理函数，设置失败的话返回 SIG_ER 。信号处理函数原型如下所示：
<pre><code class="highlight-chroma">typedef void (*sighandler_t)(int)
</code></pre>
kill -9 PID” 杀死指定进程的方法就是向指定的进程(PID)发送 SIGKILL 这个信号。当按下键盘上 CTRL+C 组合键以后会向当前正在占用终端的应用程序发出 SIGINT 信号， SIGINT 信号默认的动作关闭当前应用程序。这里我们修改一下 SIGINT 信号的默认处理函数，当按下 CTRL+C 组合键以后先终端上打印出 “SIGINT signal！” 这行字符串，然后再关闭当前应用程序。
<pre><code class="highlight-chroma">#include "stdlib.h"
#include "stdio.h"
#include "signal.h

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">void sigint_handle
(int num)
</span></span><span class="highlight-line"><span class="highlight-cl">{
</span></span><span class="highlight-line"><span class="highlight-cl">    printf("\r\nSIGI
T signal!\r\n");
</span></span><span class="highlight-line"><span class="highlight-cl">    exit(0);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">int main(void)
</span></span><span class="highlight-line"><span class="highlight-cl">{
</span></span><span class="highlight-line"><span class="highlight-cl">    signal(SIGINT, s
gint_handler);
</span></span><span class="highlight-line"><span class="highlight-cl">    while(1);
</span></span><span class="highlight-line"><span class="highlight-cl">    return 0;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/image-4d0bf36f.png?imageView2/2/interlace/1/format/jpg"></
>
<h2 id="二-驱动中的信号处理">二、驱动中的信号处理</h2>
<h3 id="1-fasync-struct-结构体">1.fasync_struct 结构体</h3>
<p>首先我们需要在驱动程序中定义一个 fasync_struct 结构体指针变量， fasync_struct 结构体内
如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">struct fasync_struct {
</span></span><span class="highlight-line"><span class="highlight-cl">    spinlock_t fa_lo
k;
</span></span><span class="highlight-line"><span class="highlight-cl">    int magic;
</span></span><span class="highlight-line"><span class="highlight-cl">    int fa_fd;
</span></span><span class="highlight-line"><span class="highlight-cl">    struct fasync_st
uct *fa_next;
</span></span><span class="highlight-line"><span class="highlight-cl">    struct file *fa_fil
;
</span></span><span class="highlight-line"><span class="highlight-cl">    struct rcu_head
a_rcu;
</span></span><span class="highlight-line"><span class="highlight-cl">};
</span></span></code></pre>
<p>一般将 fasync_struct 结构体指针变量定义到设备结构体中。</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">struct my_dev {
</span></span><span class="highlight-line"><span class="highlight-cl">    struct device *d
v;
</span></span><span class="highlight-line"><span class="highlight-cl">    struct class *cls;
</span></span><span class="highlight-line"><span class="highlight-cl">    struct cdev cdev

</span></span><span class="highlight-line"><span class="highlight-cl">    ......
</span></span><span class="highlight-line"><span class="highlight-cl">    struct fasync_st
uct *async_queue; /* 异步相关结构体 */
</span></span><span class="highlight-line"><span class="highlight-cl">    ......
</span></span><span class="highlight-line"><span class="highlight-cl">};
</span></span></code></pre>
<h3 id="2-fasync-函数">2.fasync 函数</h3>
```

<p>如果要使用异步通知，需要在设备驱动中实现 file_operations（参考 <a href="https://ld246.c
m/forward?goto=http%3A%2F%2F124.70.134.76%2FIntroduction-to-character-device-drivers
 target="_blank" rel="nofollow ugc">Linux 字符设备驱动） 操作集中的 fasync 函数，此函
原型如下所示（定义在 include/linux/fs.h 的 file_operations 结构体中）：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">int (*fasync) (int, struct file *, int); /* int (*fasync) (int fd, struct file *filp, int on) */
</code></pre>
<p>fasync 函数里面一般通过调用 fasync_helper 函数来初始化前面定义的 fasync_struct 结构体指
， fasync_helper 函数原型如下</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
</code></pre>
<p>fasync_helper 函数的前三个参数就是 fasync 函数的那三个参数，第四个参数就是要初始化的 fa
ync_struct 结构体指针变量。当应用程序通过 “fcntl(fd, F_SETFL, flags | FASYNC)” 改变 fasync 标
的时候，驱动程序 file_operations 操作集中的 fasync 函数就会执行。</p>
<p>example:</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">struct xxx_dev {
   ......
   struct fasync_st
uct *async_queue; /* 异步相关结构体 */
};

static int xxx_fasy
c(int fd, struct file *filp, int on)
{
   struct xxx_dev *
ev = (xxx_dev)filp->private_data;

   if (fasync_helper
fd, filp, on, &dev->async_queue) < 0)
   return -EIO;
   return 0;
}

static struct file_o
erations xxx_ops = {
   ......
   .fasync = xxx_fa
ync,
   ......
};
</code></pre>
<p>在关闭驱动文件的时候需要在 file_operations 操作集中的 release 函数中释放 fasync_struct，
br>
fasync_struct 的释放函数同样为 fasync_helper， release 函数参数参考实例如下：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">static int xxx_release(struct inode *inode, struct file *filp)
{
   return xxx_fasy
c(-1, filp, 0); /* 删除异步通知 */
}

<span class="highlight-line"><span class="highlight-cl">

static struct file_operations xxx_ops = {
    ......
    .release = xxx_release,
}
</code></pre>
<h3 id="3-kill-fasync-函数">3.kill_fasync 函数</h3>
当设备可以访问的时候，驱动程序需要向应用程序发出信号，相当于产生“中断”。 kill_fasync
函数负责发送指定的信号， kill_fasync 函数原型如下所示：
<pre><code class="highlight-chroma">void kill_fasync(struct fasync_struct **fp, int sig, int band)
</code></pre>
函数参数和返回值含义如下：
fp：要操作的 fasync_struct。
sig： 要发送的信号。
band： 可读时设置为 POLL_IN，可写时设置为 POLL_OUT。
返回值： 无。
<h2 id="三-应用程序对异步通知的处理">三、应用程序对异步通知的处理</h2>
<h3 id="1-注册信号处理函数">1.注册信号处理函数</h3>
应用程序根据驱动程序所使用的信号来设置信号的处理函数，应用程序使用 signal 函数来设置信号的处理函数。前面已经详细的讲过了，这里就不细讲了。
<h3 id="2-将本应用程序的进程号告诉给内核">2.将本应用程序的进程号告诉给内核</h3>
使用 fcntl(fd, F_SETOWN, getpid()) 将本应用程序的进程号告诉给内核。
<h3 id="3-开启异步通知">3.开启异步通知</h3>
使用如下两行程序开启异步通知：
<pre><code class="highlight-chroma">flags = fcntl(fd, F_GETFL); /* 获取当前的进程状态 */
fcntl(fd, F_SETFL, flags | FASYNC); /* 开启当前进程异步通知功能 */
</code></pre>
重点就是通过 fcntl 函数设置进程状态为 FASYNC，经过这一步，驱动程序中的 fasync 函数就执行