

异常控制流 2 回课

唐雯豪

November 21, 2019

Outline

1 壳

2 信号

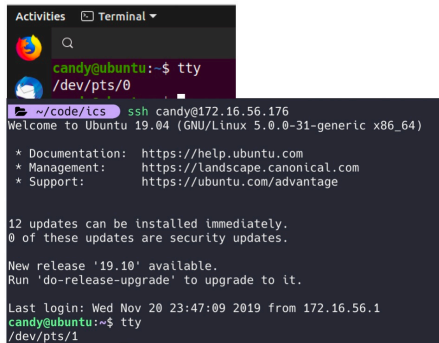
Shell

shell 是一个交互型的**应用程序**，它代表用户运行其他程序。
常见 shell:

- ▶ sh : Original Unix shell
- ▶ csh/tcsh : BSD Unix C shell
- ▶ bash : “Bourne-Again” Shell (default Linux shell)
- ▶ zsh : A powerful shell with high customizability

Shell 与 Terminal 的区别

注意, shell 和 terminal 不同, terminal 是一个 I/O 设备, 详细可见此链接<https://unix.stackexchange.com/questions/4126/what-is-the-exact-difference-between-a-terminal-a-shell-a-tty>
使用 `tty` 命令可以查看当前终端。



```
Activities [Terminal]
candy@ubuntu:~$ tty
/dev/pts/0

~/code/ics ssh candy@172.16.56.176
Welcome to Ubuntu 19.04 (GNU/Linux 5.0.0-31-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

12 updates can be installed immediately.
0 of these updates are security updates.

New release '19.10' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Nov 20 23:47:09 2019 from 172.16.56.1
candy@ubuntu:~$ tty
/dev/pts/1
```

一个简单的 Shell 实现

~~等 shell lab 放出来大家应该就都会了。~~

Outline

1 壳

2 信号

信号

- ▶ 信号是一种高层次的软件形式的异常控制流
- ▶ 用来通知用户进程某个事件的发生
- ▶ Linux 支持 30 种信号（编号 1…30）
- ▶ 信号携带的信息只有这个编号？

Sigation

Posix 标准定义了 `sigaction` 函数，允许用户指定信号处理语义。

```
int sigaction(int signum, const struct sigaction *act,
struct sigaction *oldact);|
```

当 `sigaction` 中 `sa_flags` 的 `SA_SIGINFO`

位被指定后，信号处理程序可以额外接收一个类型为

`siginfo_t*` 的参数，里面包含了更多关于发送信号进程的信息。

```
typedef struct __siginfo {
    int      si_signo;          /* signal number */
    int      si_errno;         /* errno association */
    int      si_code;          /* signal code */
    pid_t    si_pid;           /* sending process */
    uid_t    si_uid;           /* sender's ruid */
    int      si_status;         /* exit value */
    void     *si_addr;         /* faulting instruction */
    union {
        int si_value;          /* signal value */
        long si_band;          /* band event for SIGPOLL */
        unsigned long __pad[7]; /* Reserved for Future Use */
    };
} siginfo_t;
```


信号相关概念

- ▶ 发送信号
- ▶ 接收信号
- ▶ 待处理信号
 - ▶ 已发送但未被接收
 - ▶ 一种类型至多有一个，不排队，后来的被丢弃
 - ▶ 在 pending 位向量中维护
- ▶ 阻塞接收信号
 - ▶ 被阻塞时，仍可以被发送（修改 pending），但不会被接收
 - ▶ 在 blocked 位向量中维护

进程组

进程组

- ▶ 进程组由一个正整数进程组 ID 来标识，每个进程属于一个进程组，默认子进程和父进程在一个进程组。

作业 (job)

- ▶ 表示对一条命令进行求值而创建对进程
- ▶ 每个 job 对应一个独立的进程组，ID 通常取父进程中的一个。
- ▶ 任一时刻有至多一个前台 job 和任意个后台作业

发送信号

原因：

- ▶ 内核检测到某个系统事件，如子进程终止 (SIGCHLD)
- ▶ 进程执行系统调用要求内核发送信号

方法：

- ▶ `/bin/kill -SIGNUM PID` 发送任意信号，PID 为负表示进程组
- ▶ 键盘 Ctrl+C 向前台进程组每个进程发送 SIGINT 信号，Ctrl+Z 发送 SIGTSTP
- ▶ `int kill(pid_t pid, int sig)`

接收信号

何时接收信号？

- ▶ 当内核把进程 `p` 从内核模式切换到用户模式
- ▶ 一定是从 exception handler 返回时（系统调用返回、上下文切换（中断））
- ▶ 像 `sleep` 这样的会阻塞进程的系统调用期间怎么接收信号？

一种可能的解释

linux 的进程有不止三种状态：

- ▶ R (TASK_RUNNING), 可执行状态
- ▶ S (TASK_INTERRUPTIBLE), 可中断的睡眠状态
- ▶ D (TASK_UNINTERRUPTIBLE), 不可中断的睡眠状态
- ▶ T (TASK_STOPPED or TASK_TRACED), 暂停状态 (挂起)
- ▶ ...

并不只有停止（挂起），还有可中断睡眠和不可中断睡眠！

对于慢速系统调用（P540，如 sleep, pause, write），会潜在的阻塞进程较长时间（期间可能上下文切换），进程状态会被设置为 TASK_INTERRUPTIBLE；在信号发送的时候，就会将其状态改为 TASK_RUNNING，这样他就会被调度到（效果上就是慢速系统调用还未等到就被唤醒了），**系统调用被中断**，设置 errno 并切换回用户态，过程中发现有信号需要接收并接收；（在这之后还会判断是否可以自动重启，比如 sleep, write 都是可以自动重启的）。

接收信号的行为

每种信号都有一个默认行为：

- ▶ 进程终止（+ 并转储内存）
- ▶ 进程停止（挂起）直到被 SIGCONT 信号重启
- ▶ 忽略该信号

可以使用 `signal(signum, handler)` 函数修改默认行为，但是 **SIGSTOP** 和 **SIGKILL** 不能被修改。

handler 可以是

- ▶ SIG_IGN，忽略
- ▶ SIG_DEL，恢复默认行为
- ▶ 某个用户级函数的地址，被称为**信号处理程序**。
调用信号处理程序称为**捕获信号**，执行称为**处理信号**

接收信号的过程

当内核把进程 p 从内核模式切换到用户模式时，会检查进程 p 的待处理且未阻塞信号的集合，若为空则返回给 p 的下一条指令（故障应该还是返回给当前指令），否则选择一个最小的信号 p ，接收并重复，**直到集合为空**，再返回给 p 。

阻塞和解除阻塞信号

隐式的信号阻塞

- ▶ 内核默认阻塞当前信号处理程序正在处理的信号类型
- ▶ 为什么要阻塞？处理过程中新来的同种信号一并接收不好吗？
- ▶ 更安全，新的信号可能在“处理信号的循环已经结束，清空 pending 的语句还没开始的时候”发送到（书上那个回收子进程的例子）

显式的信号阻塞

- ▶ 使用 `sigpromask(how, set, oldset)` 和它的辅助函数

信号处理程序

特点：

- ▶ 与主程序**并发运行，共享全局变量**
 - ▶ 这里的并发只是信号处理程序会在主程序运行的**任一时刻**突然运行直到其结束
 - ▶ 与 fork 出的子进程不同，这里的全局变量共享
- ▶ 信号处理程序可以被其他信号处理程序/异常中断
 - ▶ **异常也许不会被异常中断？**（系统调用可以被上下文切换中断）

安全的信号处理

一些保守的原则：

0 尽可能简单

1 使用异步信号安全的函数

- ▶ 可重入

- ▶ 不会被中断（在处理程序内部设置阻塞即可）

`printf` 不安全（使用了全局 buffer），`exit` 不安全（会刷新 buffer）

许多系统调用是内部信号安全的 `write`, `waitpid`, `sleep`, `kill`, `exit`

2 保存和恢复 `errno`（比如之前提到的慢速系统调用出错会设置 `errno`）

3 访问全局数据结构时阻塞所有信号

4 用 `volatile` 声明全局变量

5 使用 `sig_atomic_t` 声明标志。（保证单个的读和写不会被中断）

安全的信号处理需要注意的问题

信号不会排队 → 信号处理程序要处理完所有事件。
并发错误（竞争）。

- ▶ 主程序和信号处理程序都会修改同一个全局数据结构，虽然已经有那些原则了，但还是有可能发生：“要求先执行主程序的部分再执行信号处理程序的部分，但是只保护了主程序修改全局数据结构附近的代码，前面的没有保护可能被信号处理程序抢先”。
- ▶ 就是说，主程序里的代码只用了原则 3，但原则 3 比可重入弱。

显式地等待信号

例如，等待某个子进程终止。
直接使用 `waitpid` 就行了。
一般的，

- ▶ `while(!pid)` ; 浪费资源
- ▶ `while(!pid) sleep(1);` 太慢
- ▶ `while(!pid) pause();` 信号接收发生在 `while` 和 `pause` 之间
- ▶ `while(!pid) sigsuspend(prev);`
`sigsuspend(*mask)` 用 `mask` 暂时替换当前阻塞集合，并挂起进程，直到接收到一个行为或者是运行一个信号处理程序或者是终止的信号。并且**异步信号安全**！

Thanks!