

Unix系统高级编程

信号和信号捕获

Unit17

信号的基本概念



信号



信号

- 信号是提供异步事件处理机制的软件中断。这些异步事件可能来自硬件设备，如用户同时按下了Ctrl键和C键，也可能来自系统内核，如试图访问尚未映射的虚拟内存，又或者来自用户进程，如尝试计算整数除以0的表达式
- 进程之间可以相互发送信号，这使信号成为一种进程间通信(Inter-Process Communication, IPC)的基本手段
- 信号的异步特性不仅表现为它的产生是异步的，对它的处理同样也是异步的。程序的设计者不可能也不需要精确地预见什么时候触发什么信号，也同样无法预见该信号究竟在什么时候会被处理。一切都在内核的操控下，异步地运行。信号是在软件层面对中断机制的一种模拟



信号处理



信号处理

- 信号有一个非常明确的生命周期
 - 首先，信号被生成，并被发送至系统内核
 - 其次，系统内核存储信号，直到可以处理它
 - 最后，一旦有空闲，内核即按以下三种方式之一处理信号
 - 忽略信号：什么也不做。SIGKILL和SIGSTOP信号不能忽略
 - 捕获信号：内核暂停收到信号的进程正在执行的代码，跳转到事先注册的信号处理函数，执行该函数并返回，跳回到捕获信号的地方继续执行。SIGKILL和SIGSTOP信号不能捕获
 - 默认操作：不同信号有不同的默认操作，通常是终止收到信号的进程，但也有一些信号的默认操作是视而不见，即忽略
- 早期的信号处理函数除了知道来了一个信号之外，对究竟发生了什么可以说是一无所知。现代Unix系统的内核为程序员提供了大量的上下文，甚至用户自定义的数据



信号名称与编号



信号名称与编号

- 每个信号都有唯一的名称和编号
 - 信号的名称是以SIG开头的文本，如SIGHUP、SIGINT等
 - 信号的编号是从1开始连续增加的整数，如1、2、3等
 - 信号的名称与编号之间的对应关系，依赖于具体实现
 - 在<signal.h>头文件中，已经通过宏定义，把每个信号的名称和编号建立了一一映射。无论是程序员还是系统管理员，在任何涉及信号的场合，都应该尽量使用信号的名称(宏)，而不要用它们的编号(字面值)

```
#define SIGHUP      1
#define SIGINT     2
#define SIGQUIT    3
```



信号名称与编号 (续1)

- 执行 “kill -l” 命令，可以查看系统支持的信号列表

\$ kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	64) SIGRTMAX	

- 位于[1, 31]区间的31个信号为非实时信号
- 位于[34, 64]区间的31个信号为实时信号
- 总共62个信号，注意没有32和33信号



可靠信号与不可靠信号



可靠信号与不可靠信号

- 正如前面所述，信号的产生和信号的处理完全是两个异步的操作过程，也就是说这两个动作在时间上并没有确定的先后顺序，甚至可能是同时发生的
- 如果信号的处理速度慢于信号的产生速度，那么就会出现，一个信号正在被处理，多个相同的信号又发送过来
 - 当多个相同的信号被发送到正在处理该信号的进程时，那些还来不及处理的信号会按先后顺序排成队列。等进程处理完手里的信号以后，再依次处理队列中的信号。整个过程中，所有发送给进程的信号一个也不会丢，都能得到处理。这样的信号就叫做可靠信号。实时信号都是可靠信号
 - 反之，不可靠信号不支持排队，可能丢失，在多个相同的信号里进程可能只收到一个。非实时信号都是不可靠信号



常用信号



常用信号

编号	名称	说明	默认操作
1	SIGHUP	进程的控制终端关闭(用户登出)	终止
2	SIGINT	用户产生中断符(Ctrl+C)	终止
3	SIGQUIT	用户产生退出符(Ctrl+\)	终止+转储
4	SIGILL	进程试图执行非法指令	终止+转储
5	SIGTRAP	进入断点	终止+转储
6	SIGABRT	abort函数产生	终止+转储
7	SIGBUS	硬件或对齐错误	终止+转储
8	SIGFPE	算术异常	终止+转储
9	SIGKILL	不能被捕获或忽略的进程终止信号	终止
10	SIGUSR1	进程自定义的信号	终止
11	SIGSEGV	无效内存访问	终止+转储
12	SIGUSR2	进程自定义的信号	终止



常用信号 (续1)

编号	名称	说明	默认操作
13	SIGPIPE	向读端已关闭的管道写入	终止
14	SIGALRM	alarm函数产生/真实定时器到期	终止
15	SIGTERM	可以被捕获或忽略的进程终止信号	终止
16	SIGSTKFLT	协处理器栈错误	终止
17	SIGCHLD	子进程终止	忽略
18	SIGCONT	进程由停止状态恢复运行	忽略
19	SIGSTOP	不能被捕获或忽略的进程停止信号	停止
20	SIGTSTP	用户产生停止符(Ctrl+Z)	停止
21	SIGTTIN	后台进程读控制终端	停止
22	SIGTTOU	后台进程写控制终端	停止
23	SIGURG	紧急I/O未处理	忽略
24	SIGXCPU	进程资源超限	终止+转储



常用信号 (续2)

编号	名称	说明	默认操作
25	SIGXFSZ	文件资源超限	终止+转储
26	SIGVTALRM	虚拟定时器到期	终止
27	SIGPROF	实用定时器到期	终止
28	SIGWINCH	控制终端窗口大小改变	忽略
29	SIGIO	异步I/O事件	终止
30	SIGPWR	断电	终止
31	SIGSYS	进程试图执行无效系统调用	终止+转储

- SIGSTKFLT(16)信号, Linux系统内核已不再产生, 保留该信号仅为向下兼容
- SIGIO(29)信号, 非Linux操纵系统的默认操作为忽略



中断符和退出符

【参见：intquit.c】

- 中断符和退出符



捕获信号

捕获信号

signal

signal

信号捕获流程

重入问题

一次性问题

终止和停止

一专多能

太平间信号

信号处理的继承与恢复

信号处理的继承与恢复

signal



signal

- 设置针对特定信号的处理方式

```
#include <signal.h>
```

```
typedef void (*sighandler_t) (int);
```

```
sighandler_t signal (int signum, sighandler_t handler);
```

成功返回原信号处理方式，失败返回SIG_ERR

- *signum*: 信号编号
- *handler*: 信号处理方式，可取以下值
 - SIG_IGN - 忽略信号
 - SIG_DFL - 默认操作
 - 信号处理函数指针 - 捕获信号



signal (续1)

- 信号处理函数
 - void sighandler (int *signum*) { 信号处理代码 }
 - *signum*: 信号编号
- 例如
 - void sigint (int signum) {
 printf ("%d进程: 收到%d信号\n", getpid (), signum);
}
 - int main (void) {
 if (signal (SIGINT, sigint) == SIG_ERR) {
 perror ("signal"); exit (EXIT_FAILURE); }
 ...
 return 0; }



捕获信号

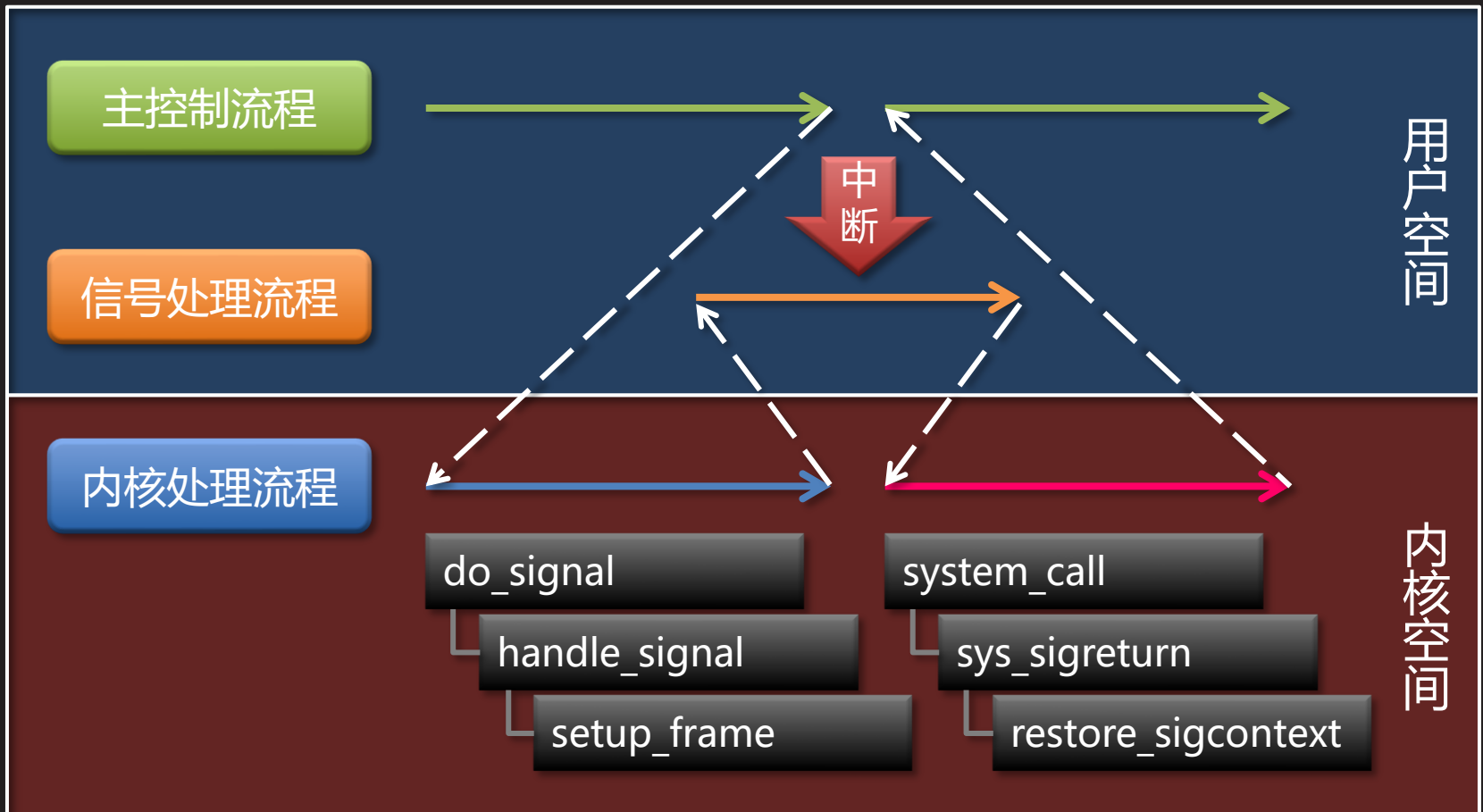
【参见：signal.c】

- 捕获信号



信号捕获流程

- 主控制流程、信号处理流程和内核处理流程



重入问题

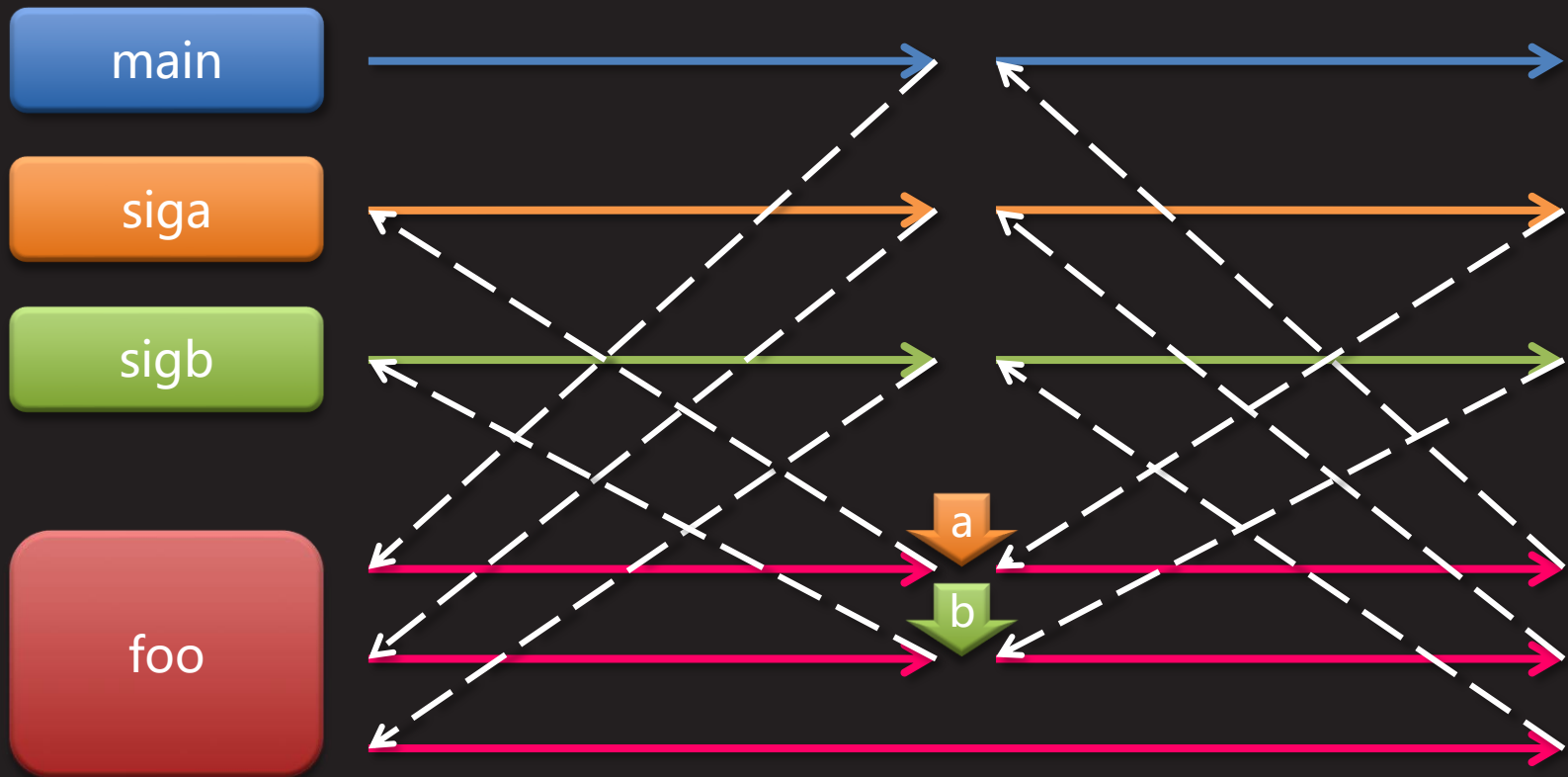
- 参与信号处理的函数必须是可重入函数
 - 何为重入
 - 假设进程的主控制流程此刻正在调用foo函数
 - 就在foo函数刚执行到一半的时候，内核向进程递送了信号a
 - 假设进程对信号a做了捕获，那么此时流程将转入信号a的处理函数siga，而siga函数在执行过程中也调用了foo函数
 - 于是foo函数中的代码又被执行，刚执行到一半，内核又递送了信号b
 - 假设进程对信号b也做了捕获，并用sigb函数来处理，而在sigb函数中同样调用了foo函数
 - 注意此时的foo函数已被重入3次，假设该函数中包含了对全局变量、静态变量、磁盘文件等共享型资源的访问，其结果将会如何？数据的一致性和安全性如何得到保证？



重入问题 (续1)

- 参与信号处理的函数必须是可重入函数
 - 何为重入
 - 重入绝非多线程应用的专利，包含类似信号这样的异步操作的应用，即便是单线程的，同样会有重入问题

知识讲解



重入问题 (续2)

- 参与信号处理的函数必须是可重入函数

- 编写可重入函数

- 编写参与信号处理的函数(比如前例中的foo函数)时必须要注意, 不要对进程在中断时所做的事情做任何假设。尤其是在对全局变量、静态局部变量、磁盘文件这些带有共享特性的对象做写操作的时候, 必须要谨慎。当然, 如果能够不碰或者只读这些全局对象, 那当然再好也没有
- 可重入函数是指可以安全地调用自身(从信号处理中或从其它线程中)的函数。为了使函数可重入, 函数绝不能操作静态数据, 只访问在栈里分配的数据和调用者提供的数据, 同时也不得调用任何不可重入的函数

- 调用可重入函数

- 在参与信号处理的函数中只调用那些在POSIX.1-2003和Unix信号规范中明确为信号安全的函数, 不失为明智之举。例如: abort()、accept()、access()、bind(), 等等



一次性问题

- 在某些Unix系统上，通过signal函数注册的信号处理函数只能一次性有效
 - 系统内核在每次调用信号处理函数之前，会先将对该信号的处理恢复为默认操作
 - 为了获得持久有效的信号处理，可以在信号处理函数中调用signal函数重新注册
 - ```
void sigint (int signum) {
 ...
 signal (SIGINT, sigint); }

int main (void) {
 signal (SIGINT, sigint);
 ... }

```



# 终止和停止

- SIGKILL(9)和SIGSTOP(19)信号即不能被忽略，也不能被捕获，只能按缺省方式终止或停止接收到信号的进程
  - if (signal (SIGKILL, SIG\_IGN) == SIG\_ERR) {  
    perror ("signal"); exit (EXIT\_FAILURE); }
  - if (signal (SIGKILL, SIG\_DFL) == SIG\_ERR) {  
    perror ("signal"); exit (EXIT\_FAILURE); }
  - if (signal (SIGSTOP, sigstop) == SIG\_ERR) {  
    perror ("signal"); exit (EXIT\_FAILURE); }
  - 以上三个对signal函数的调用都会失败，错误信息都是“Invalid argument (无效参数)”，这说明signal函数会检查传给它的第一个参数，如果是这两个消息则直接返回失败，并置errno为EINVAL



# 一专多能

- 信号处理函数带有一个信号编号参数 *signum* 是有意义的，这意味着可以让一个信号处理函数处理多个不同的信号

```
– void sighandler (int signum) {
 switch (signum) {
 case SIGINT:
 处理SIGINT信号; break;
 case SIGQUIT:
 处理SIGQUIT信号; break;
 case ...
 }
}
```

- 但一般情况下，还是更倾向于为每个需要处理的信号定义专门的信号处理函数，毕竟这更符合分而治之的设计理念



# 太平间信号

- 如前所述，无论一个进程是正常终止还是异常终止，都会通过系统内核向其父进程发送SIGCHLD(17)信号。父进程完全可以在针对SIGCHLD(17)信号的信号处理函数中，异步地回收子进程的僵尸，简洁而又高效

```
– void sigchld (int signum) {
 pid_t pid = wait (NULL);
 if (pid == -1) {
 perror ("wait");
 exit (EXIT_FAILURE); }
 printf ("%d子进程终止\n", pid);
}
```



```
– if (signal (SIGCHLD, sigchld) == SIG_ERR) {
 perror ("signal"); exit (EXIT_FAILURE); }
```

# 太平间信号 (续1)

- 但这样处理存在一个潜在的风险，就是在sigchld信号处理函数执行过程中，又有多个子进程终止，由于SIGCHLD(17)信号不可靠，可能会丢失，形成漏网僵尸，因此有必要在一个循环过程中回收尽可能多的僵尸

```
– void sigchld (int signum) {
 for (;;) {
 pid_t pid = wait (NULL);
 if (pid == -1) {
 if (errno != ECHILD) {
 perror ("wait"); exit (EXIT_FAILURE); }
 printf ("子进程都死光了\n"); break; }
 printf ("%d子进程终止\n", pid); }
 }
```



# 太平间信号 (续2)

- 但上面的代码同样存在问题，因为当所有的子进程都处于运行状态时wait函数会阻塞，这时sigchld信号处理函数就不会返回，被信号中断的操作也就无法继续。为此，可以考虑用具有非阻塞特性的waitpid函数取代wait函数

```
– void sigchld (int signum) {
 for (;;) {
 pid_t pid = waitpid (-1, NULL, WNOHANG);
 if (pid == -1) {
 if (errno != ECHILD) {
 perror ("wait"); exit (EXIT_FAILURE); }
 printf ("子进程都死光了\n"); break; }
 if (! pid) break;
 printf ("%d子进程终止\n", pid); } }
```



# 信号处理的继承与恢复





# 信号处理的继承与恢复

- fork/vfork函数创建的子进程会继承父进程的信号处理方式，直到子进程调用exec函数创建新进程替代其自身
- exec函数创建的新进程会将原进程中被设置为捕获的信号会还原为默认操作
  - 毕竟位于原进程地址空间中的信号处理函数，此时已经被新进程的地址空间完全取代了



# 信号处理的继承

【参见：inherit.c】

- 信号处理的继承



# 信号处理的恢复

【参见：recover.c】

课堂练习

- 信号处理的恢复



# 总结和答疑

