Programmation Systèmes
Cours 4 — Signals

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2012–2013

URL http://upsilon.cc/zack/teaching/1213/progsyst/

Sommaire

1. Signal concepts

2. Unreliable signals API

3. Reliable signals API

4. Real-time signals

Outline

1. Signal concepts

2. Unreliable signals API

3. Reliable signals API

4. Real-time signals

Introduction

Definition (Signal)

A signals is a software interrupt. A signal is delivered *to processes* as an asynchronous event wrt the usual execution flow.

Signals are used to represent several kinds of events:

- events generated by (human) users through the terminal
  - e.g. Ctrl-C (SIGINT), Ctrl-Z (SIGSTOP), . . .
- hardware faults
  - e.g. divide-by-0, segment violation (SIGSEGV), invalid memory references (SIGBUS), . . .
- anomalous software conditions (software faults)
  - e.g. writing to a connected IPC facility (SIGPIPE), out of band data notification (SIGURG), time-based reminders (SIGALRM), . . .
- (payload less) process to process signaling via `kill(2)`
- sysadm to process communication via `kill(1)`

Signal processing model

Most of the events a UNIX process deals with are handled according to the pull model:

- when the process is ready/willing to handle an event, it uses syscalls to check whether an event has occurred in the past and to retrieve the associated information

Signals are the most prominent example of asynchronous events under UNIX. They are dealt with according to the push model:[1]

- the process declares its interest in listening for an event (a signal) by registering an handler that will be called as soon as the event occurs
- the handler is passed all information associated to the signal
- normal program execution usually resumes upon handler termination

[1] on Linux there is a desire to support signal management in pull mode, but nothing concrete exists yet

On the (bad) reputation of signals

In the early days of UNIX, the “reputation” of signals was pretty bad, due to a number of unreliability causes in signal delivery. We refer to signal implementation and specification of those days as unreliable signals.[2]

The bad reputation of those days still affects the usage of signals.

. . . even though modern UNIX-es have a much better handling of signals, in terms of reliable specifications and implementations, as well as expressiveness (e.g. POSIX.1-2001 added support for signal payloads).
We refer to modern signals as reliable signals.

[2] we’ll discuss unreliability causes in a bit

Signal names

Each signal is identified by a signal name starting with SIG

- `<signal.h>` associates signal names to *platform-specific* signal numbers
- available signals are standardized by SUS, although some are XSI extensions; each platform might support additional implementation-specific signals
- signal number 0 corresponds to the null signal, which is no signal and has special meaning for `kill`

Available signals on a given platform can be listed with `kill`(1).

Example (Linux x86 signals)

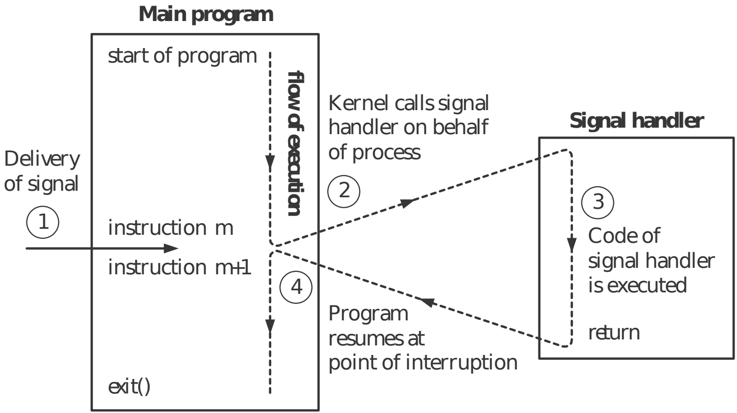
```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP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

Signal disposition

Each process can express his wishes to the kernel about what should happen when a *specific* signal occurs. The signal disposition (or action) is one of the following:

1. ignore the signal: nothings happens, the event is ignored
2. catch the signal: a process-specific handler is executed when the event corresponding to the signal occurs
3. default: every signal is associated to a default action, which is one of:
   - terminate the process (most common)
   - terminate the process and dump core (for further debugging)
   - ignore the signal
   - stop/resume process execution

Signal handler execution


Main program
start of program
flow of execution
Kernel calls signal handler on behalf of process
2
Signal handler
Delivery of signal
1
instruction m
instruction m+1
3
Code of signal handler is executed
4
Program resumes at point of interruption
return
exit()

TLPI, Figure 20-1

Beware: signal handler invocation might interrupt the program at *any* time (!) — instructions in the picture are more fine-grained than your C statements. . .

Signal life cycle

1. A signal is generated *for a process* (or *sent to a process*) when the corresponding event occurs.

2. The signal remain pending between generation and delivery.

3. A process has the option to block signal delivery.
   - each process is associated to a set of signals, called signal mask, that the process is currently blocking
   - if a blocked signal is generated and its disposition is either default or catch, the signal remains pending until either of:
     a. the process unblocks the signal
     b. the process changes signal disposition to ignore the signal

4. A signal is delivered to a process when the action specified by the signal disposition of the receiving process has been taken.
   - note: the decision of what to do with a signal is taken when the signal is delivered, not when it is generated

Notable signals

We briefly go through some of the most notable signals. For a complete list see SUS or the summary table on APUE, Figure 10.1.

We'll go through signals according to the following ad-hoc classification:

- process-related events
- job control
- terminal events
- hardware faults
- software faults
- custom events
- run-time signals

For each signal we mention its number on the *Linux x86* platform.

Notable signals — process-related events

The following signals are used to notify of the occurrence of process-related notifications:

Self notifications

SIGABRT (6, default: terminate+core) generated by calling the abort function

SIGALRM (14, default: terminate) generated at the expiration of a timer set by the alarm function

Other notifications

SIGCHLD (17, default: ignore) sent to the parent process upon the termination of a child

- typical usage: collect termination status with wait

SIGURG (23, default: ignore) notification of some “urgent condition”

- use case: notification of the arrival of out of band data on some input channel

Notable signals — job control

Various signals are delivered to processes for purposes related to UNIX job control:

SIGSTOP (19, default: stop process) sent to a process to suspend its execution

- cannot be ignored or caught

SIGCONT (18, default: resume the process if it was stopped; ignore otherwise) sent to a process *just after* it resume execution

- use case: redraw terminal upon restart

SIGTERM (15, default: terminate process) sent to a process to *ask* for its termination

- it's the signal sent by `kill`(1) by default

SIGKILL (9, default: terminate process) sent to kill a process

- cannot be ignored or caught

Notable signals — terminal events

Many signals can be generated by the terminal driver during interactive usage of the shell:

SIGHUP (1, default: terminate, mnemonic: *Hang UP*) sent to session leader process when the controlling terminal is disconnected

- typical (ab)use: ask daemon processes to reread configuration, based on the observation that daemons do not have a controlling terminal

SIGINT (2, default: terminate) terminal character to request termination of all foreground processes

- usual keyboard shortcut: Ctrl-C

SIGQUIT (3, default: terminate+core) same as SIGINT, but additionally request to dump core

- usual keyboard shortcut: Ctrl-\

Notable signals — terminal events (cont.)

SIGTSTP (20, default: stop process, mnemonic: *Terminal SToP*) interactive stop signal, used to request top of all foreground processes

- usual keyboard shortcut: Ctrl-Z

SIGTTIN (21, default: stop process, mnemonic: *Terminal Try INput*) sent to a background process if it attempts to read from its controlling terminal

SIGTTOU (22, default: stop process, mnemonic: *Terminal Try OUtput*) dual to SIGTTIN, sent to a background process if it attempts to write to its controlling terminal

SIGWINCH (28, default: ignore; *warning*: non-SUS, but supported by most UNIX-es) sent to all foreground processes upon change of the window size associated to the terminal

- use case: redraw the screen
- e.g. top

Notable signals — hardware faults

Some signals are related to (perceived) hardware faults:

SIGBUS (7, default: terminate+core) sent to a process that causes a bus error, e.g.:

- unaligned memory access
- access to a non-existent memory address
- real hardware failure when accessing memory

SIGSEGV (11, default: terminate+core) sent to a process that causes a SEGmentation Violation (or *segmentation fault*), i.e. an attempt to access a memory location it has no right to access

SIGFPE (8, default: terminate+core) invalid arithmetic/floating point operation

- e.g. divide by 0

SIGILL (4, default: terminate+core) sent to a process that attempt to execute an illegal instruction

- e.g. malformed assembly instruction

Notable signals — software faults

Some signals are used to notify of software-related faults:

SIGPIPE (13, default: terminate) sent to a process if it attempts to write to a connected process-to-process IPC facility that has no connected readers

- e.g. pipe, socket

Notable signals — custom events

Two signals are reserved for custom, application-defined use:

- SIGUSR1 (10, default: terminate)
- SIGUSR2 (12, default: terminate)

both signals have no specific meaning other than the handling semantics that custom signal handlers might assign to it

Notable signals — real-time signals

Recent POSIX.1 updates have introduced many APIs for real-time purposes. They include handling of real-time signals.

Real-time signals also introduce a new range of signals. Instead of being a set of signal names, all signals between SIGRTMIN and SIGRTMAX are real-time signals.

- real-time signals can be used as regular signals (and *vice-versa*)
- but, as we'll see, their delivery semantics is different

Example (Linux x86 real-time signals)

```
$ kill -l
<snip>           34) SIGRTMIN     35) SIGRTMIN+1   36) SIGRTMIN+2
37) SIGRTMIN+3   38) SIGRTMIN+4   39) SIGRTMIN+5   40) SIGRTMIN+6
41) SIGRTMIN+7   42) SIGRTMIN+8   43) SIGRTMIN+9   44) SIGRTMIN+10
45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13  48) SIGRTMIN+14
49) SIGRTMIN+15  50) SIGRTMAX-14  51) SIGRTMAX-13  52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9   56) SIGRTMAX-8
57) SIGRTMAX-7   58) SIGRTMAX-6   59) SIGRTMAX-5   60) SIGRTMAX-4
61) SIGRTMAX-3   62) SIGRTMAX-2   63) SIGRTMAX-1   64) SIGRTMAX
$
```

Interlude — core dump

Definition (core dump)

A core dump (or *core file*) is an image of the memory of a process, taken at *crash-time* of the corresponding process.

Useful information to debug the crash is stored in core files:

- memory dump at the time of crash
- termination status (usually abnormal)
- copies of processor registries

Interlude — core dump (cont.)

The default disposition of many signals includes code dumps. Nonetheless we rarely see core files around. Why?

Interlude — core dump (cont.)

The default disposition of many signals includes code dumps. Many default setups set to 0 the maximum size limit on core dumps.

Such a limit can be inspected and changed with `ulimit`.

Example (fiddling with core file size limit)

```
$ help ulimit                                  learn how to use ulimit
<snip>

$ ulimit -c
0                                      core file generation is disabled

$ ulimit -c unlimited          enable core file generation, no size limit
$ ulimit -c
unlimited
```

Interlude — core dump example

```
#include <stdlib.h>
#include <unistd.h>

int main(void) {
        sleep(60);
        exit(EXIT_SUCCESS);
}
```

Interlude — core dump example (cont.)

```
$ gcc -Wall -g -o sleep sleep.c
$ ./sleep
Ctrl-C                                   SIGINT, default disposition: terminate (no core)
^C
$ ls -l core
ls: cannot access core: No such file or directory
$ ./sleep
Ctrl-\                                     SIGQUIT, default disposition: terminate+core
^\Quit
$ ls -l core
ls: cannot access core: No such file or directory
$ ulimit -c unlimited
$ ./sleep
Ctrl-\                                                                    SIGQUIT again
^\Quit (core dumped)
$ ls -l core
-rw------- 1 zack zack 237568 ott 19 15:44 core
$ du core
100
$
```

Interlude — core dump example (cont.)

```
$ gdb sleep core
<snip>
Core was generated by ‘./sleep’.
Program terminated with signal 3, Quit.
#0  0x00007f0b7c7731b0 in nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
(gdb)

(gdb) bt
#0  0x00007f0b7c7731b0 in nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x00007f0b7c773040 in sleep () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x0000000000400542 in main () at sleep.c:5
(gdb)
$
```

Outline

1. Signal concepts
2. Unreliable signals API
3. Reliable signals API
4. Real-time signals

signal

The main activity related to signal management is changing signal dispositions for a given process.
The signal function is the simplest interface to that activity.

```
#include <signal.h>

void (*signal(int signo, void (*handler)(int)))(int)
```

Returns: *previous signal disposition if OK, SIG_ERR on error*

signal (cont.)

The main activity related to signal management is changing signal dispositions for a given process.
The signal function is the simplest interface to that activity.

```
#include <signal.h>

void (*signal(int signo, void (*handler)(int)))(int)
```

Returns: *previous signal disposition if OK, SIG_ERR on error*

It can be made easier to the eyes by applying appropriate typedef substitutions:

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

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

Returns: *previous signal disposition if OK, SIG_ERR on error*

signal (cont.)

---

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

sighandler_t signal(**int** signo, sighandler_t handler);

Returns: *previous signal disposition if OK, SIG_ERR on error*

- *signo* is the name (or number) of the signal whose disposition we want to change
- *handler* is one of:

SIG_IGN	to request ignoring of *signo*
SIG_DFL	to reset signal disposition of *signo* to the default
pointer	to the handler—a function accepting a *int* parameter and returning void—that will be invoked to complete signal delivery passing the number of the signal being delivered

- signal returns SIG_ERR if the request fails; (a pointer to) the previous signal disposition if it succeeds

signal — example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

/* one handler for both signals */
static void sig_usr(int signo) {
        if (signo == SIGUSR1)
                printf("received SIGUSR1\n");
        else if (signo == SIGUSR2)
                printf("received SIGUSR2\n");
}
int main(void) {
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR1");
        if (signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR2");
        for ( ; ; )
                sleep(60);
} /* based on APUE, Figure 10.2 */
```

signal — example (cont.)

Demo

signal inheritance

Upon fork

- child inherits parent’s signal disposition
  - ignored and default signals remain the same in child
  - caught signals will continue to be caught by the same handlers

Upon exec

can we do the same?

signal inheritance

Upon fork

- child inherits parent's signal disposition
  - ignored and default signals remain the same in child
  - caught signals will continue to be caught by the same handlers

Upon exec

> while *ignore and default* dispositions could remain the same, *catch* dispositions could not: function pointers would be meaningless in the address space of a new (different) program

- exec resets to the default actions all catch disposition
- whereas ignore and default dispositions are inherited by the new program

Sending signals

Signals can be sent to arbitrary processes using `kill`[3] and to the current process using `raise`:

```
#include <signal.h>

int  kill (pid_t pid, int signo);

int raise(int signo);
```

Returns: *0 if OK, -1 on error*

The meaning of *pid* depends on its value:

$pid > 0$	signal sent to process with PID *pid*
$pid == 0$	signal sent to all processes in the same process group of the sender
$pid < 0$	signal sent to all processes of process group $abs(pid)$
$pid == -1$	signal sent to all processes

[3] historic misnomer

Sending signals — permissions

As signal can have important consequences (e.g. program termination, but also polluting the file system with core dumps), appropriate permissions are required to *send* a signal.

General `kill` permission rules

- a superuser process can `kill` arbitrary processes
- a normal user process can `kill` processes whose real or saved set-user-ID are equal to the sender process real or effective uid

Notes:

- permission to `kill` a specific process can be checked by `kill`-ing with the null signal
- group `kill`-ing sends signals only to processes allowed by permissions (without failing)

pause

pause blocks a process until a signal is caught (i.e. activating a signal handler):

```
#include <unistd.h>

int pause(void);
```

Returns: *-1 with errno set to EINTR*

- note: ignored signals do not trigger pause return
- remember: *delivered* != *sent*

alarm

#include <unistd.h>

unsigned int alarm(**unsigned int** seconds);

Returns: *0 or n. of seconds until previously set alarm*

Using alarm, a process can set a timer that will expire *seconds* seconds in the future. After timer expiration, the signal SIGALRM will be sent to the calling process.
Note: default action for SIGALRM is process termination.

There is only one timer per process.

- alarm(0) cancels the timer
- the n. of seconds left before previous timer expiration is returned at each invocation

alarm — example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include "helpers.h"

void clock_tick(int signo) {
        printf("\r%ld", time(NULL));    /* overwrite prev. time with new */
        alarm(1);                       /* re-set alarm */
}

int main(void) {        /* a (UNIX time) clock */
        setvbuf(stdout, NULL, _IONBF, BUFSIZ);  /* avoid buffering */
        printf("\e[2J\e[H");    /* home and clear screen w/ ANSI ESC seqs */

        if (signal(SIGALRM, clock_tick) == SIG_ERR)
                err_sys("can't catch SIGALRM");
        clock_tick(-1);         /* print current time */
        alarm(1);
        for ( ; ; )             /* wait/catch loop */
                pause();
        exit(EXIT_SUCCESS);
} /* clock.c */
```

alarm — example (cont.)

Demo

sleep

sleep—which we have used often—is a timeout-powered version of pause:[4]

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns: *0 or number of unslept seconds*

The calling process will suspend until either:

1. the given number of seconds elapses, or
2. a signal is caught by the process and its signal handler returns
   - the return value tells us the remaining time, in seconds, until termination condition (1)

[4] depending on whether sleep is implemented using alarm or not, there might be nasty interactions among the two. . .

Unreliable-signal semantics

In early UNIX-es, signal were unreliable and hard to control:

1. signals could get lost and never be delivered to target processes
2. there was no way for a process to temporarily block a signal:
    - either the process catches a signal (accepting the possibility of being interrupted at any time),
    - or the process ignores it (losing the possibility of ever knowing that someone send him a signal while he was ignoring it)

Even if *implementations* of modern UNIX-es are past these issues, by using the unreliable signal API there is no way of knowing *for sure* that these issues are gone.
Whether it is the case or not is implementation-dependent.

Unreliability I — reset to default

Action dispositions used to be reset to the default action at each delivery. Code like the following was (and still is) common place:

```
/* signal handler */
void my_handler(int signo) {
    signal(SIGINT, my_handler);  /* re-establish handler */
    ...                          /* process signal */
}

int main(void) {
    ...
    signal(SIGINT, my_handler);  /* establish handler */
    ...
}
```

what’s wrong with this code?

Unreliability I — reset to default

Action dispositions used to be reset to the default action at each delivery. Code like the following was (and still is) common place:

```
/* signal handler */
void my_handler(int signo) {
    signal(SIGINT, my_handler);  /* re-establish handler */
    ...                          /* process signal */
}

int main(void) {
    ...
    signal(SIGINT, my_handler);  /* establish handler */
    ...
}
```

- problem: there is race condition between the start of handler execution and the re-establishment of the signal handler
- a signal delivered in between will trigger default action
    - potentially terminating the process

Unreliability II — snoozing signals

Context: alternate program phases where we can’t be interrupted (“critical regions”), with phases where we can, without losing relevant signals delivered during critical regions
Building block: wait for a signal to occur, then proceed

Let’s try with a global flag:

```
int sig_int_flag = 0;                  /* global flag */
void my_handler(int signo) {
    signal(SIGINT, my_handler);
    sig_int_flag = 1;                  /* caught signal, set flag */
}
int main(void) {
    ...
    signal(SIGINT, my_handler);        /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();                       /* wait for signal */
    ...                                /* caught signal, proceed */
}
```

Unreliability II — snoozing signals (cont.)

code idea:

1. let's wait for a signal
2. when the signal handler returns the program will be awakened (thanks to pause) and the flag will tell us if a specific handler has been executed
3. if the signal is not relevant, go to (1)

Unreliability II — snoozing signals (cont.)

code idea:

1. let's wait for a signal
2. when the signal handler returns the program will be awakened (thanks to pause) and the flag will tell us if a specific handler has been executed
3. if the signal is not relevant, go to (1)

- problem: race condition between the test `sig_int_flag == 0` and `pause`
- if a signal gets delivered in that window (and if it's delivered only once): the program will block forever because nobody will (re)check the flag before blocking (forever)
- problem mitigation: using `sleep` instead of `pause`
  - it induces timeout and/or polling problems, depending on `sleep` argument

Interrupted system calls

System calls invocations can be (intuitively) classified in two classes:

1. "slow" invocations that might block indefinitely, e.g.:
    - read, write, and ioctl when called on "slow" devices that could in turn block indefinitely (e.g. terminal, pipe, socket)
        - note: disk I/O fails the above definition
    - blocking open (e.g. on a FIFO)
    - wait and friends
    - socket interfaces
    - file locking interfaces
    - IPC synchronization primitives (message queues, semaphores, Linux futexes, etc.)
2. "fast" invocations: every other invocation

If a signal gets caught during a slow syscall invocation, the syscall might—after execution of the handler—return an error and set errno to EINTR.

In early UNIX-es that was the only possible behavior.

Interrupted system calls (cont.)

The pro of interrupt-able system calls is that they allow to have a way out of situations that could block forever.

- e.g. blocking reads from a terminal with an away user

The cons of interrupt-able system calls is that the code needs to deal with the EINTR error condition explicitly and restart the interrupted syscall invocation with code like:

```
while ((n = read(fd, buf, BUFFSIZE)) != 0) {
    if (n == -1) {
        if (errno == EINTR)
            continue;
        else
            // handle other error cases
    }
    // handle success cases
}
// handle end of file
```

Unreliability III — EINTR uncertainty

Some of the early UNIX-es (most notably BSDs) introduced automatic restart of the system calls: `ioctl`, `read`, `readv`, `writev`, `wait`, `waitpid`.

The drawback of automatic restart is obvious: it throws away the advantages of interrupt-able syscalls.

POSIX.1 *allows* implementations to restart system calls but do not *require* it:

> Using the unreliable signal API there is no way of knowing whether slow syscall invocations will be restarted or not.

Unreliability IV — signal queuing

What happens if the same signal is generated twice, before the target process has a chance to deliver it?

POSIX.1 allows for two possibilities:

signal queuing the kernel keeps track of the number of signals generated and performs an equal number of deliveries

- note: (non real-time) signals are indistinguishable from one another; therefore the order is irrelevant

no signal queuing the kernel only keeps a bitmask of pending signals and performs a single delivery

Using the unreliable signal API there is no way of being sure if queuing is in effect or not.

Unreliability IV — signal queuing example

```
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

void sig_usr(int signo) {
        printf("caught signal %d\n", signo); }
int main(void) {
        sigset_t newmask, oldmask;
        if ((signal(SIGUSR1, sig_usr) == SIG_ERR)
            || signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("signal error");
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGUSR1); sigaddset(&newmask, SIGUSR2);
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) /* block SIGUSR*
                err_sys("SIG_BLOCK error");
        sleep(10);      /* SIGUSR* here will remain pending */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) /* unblocks SIGUSR
                err_sys("sigprocmask error");
        if (signal(SIGUSR1, SIG_DFL) == SIG_ERR) err_sys("signal error");
        if (signal(SIGUSR2, SIG_DFL) == SIG_ERR) err_sys("signal error");
        printf("SIGUSR* unblocked\n");
        while (1)  pause();
        exit(EXIT_SUCCESS);
} /* no-queue.c */
```

Unreliability IV — signal queuing example (cont.)

Demo

Unreliability V — causality

Whereas homonymous signals are indistinguishable, different signals are.

Let's assume different signals are generated for the same target process p in the order $s_1, \dots s_n$.
What would be the delivery order of signals to p?

For regular signals POSIX.1 gives no guarantee about the preservation of any order between generation and delivery.

Unreliability V — causality example

```
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

void sig_usr(int signo) {
        printf("caught signal %d\n", signo); }
int main(void) {
        sigset_t newmask, oldmask;
        if ((signal(SIGUSR1, sig_usr) == SIG_ERR)
            || signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("signal error");
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGUSR1); sigaddset(&newmask, SIGUSR2);
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) /* block SIGUSR*
                err_sys("SIG_BLOCK error");
        sleep(10);      /* SIGUSR* here will remain pending */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) /* unblocks SIGUSR
                err_sys("sigprocmask error");
        if (signal(SIGUSR1, SIG_DFL) == SIG_ERR) err_sys("signal error");
        if (signal(SIGUSR2, SIG_DFL) == SIG_ERR) err_sys("signal error");
        printf("SIGUSR* unblocked\n");
        while (1)  pause();
        exit(EXIT_SUCCESS);
} /* no-queue.c */
```

Unreliability V — causality example

Demo

Signal handler vs static memory

Unrelated to the unreliability of the old API, another reliability concern should be taken into account when programming signals:

> we don't know what the process was doing when the signal was delivered

What if the process:

1. was in the middle of `malloc` or `free`?
2. was in the middle of a function that uses static memory to return a value?

What if we call the same function in the signal handler?

Signal handler vs static memory

Unrelated to the unreliability of the old API, another reliability concern should be taken into account when programming signals:

> we don't know what the process was doing when the signal was delivered

What if the process:

1. was in the middle of `malloc` or `free`?
   - *`malloc` maintains a linked list of allocated blocks and might be in the process of updating it...*
2. was in the middle of a function that uses static memory to return a value?
   - *the return value of the first call will be overwritten by the return value of the handler call...*

What if we call the same function in the signal handler?

FAIL.

Reentrant functions

In general, we cannot exclude that the program was doing anything like the previous examples at signal delivery time.
Therefore, the only safeguard is avoid using the same functionalities *from* signal handlers.

The functions that can safely be invoked from signal handlers are called reentrant functions (or *async-safe functions*). The full list is prescribed by POSIX.

Note: functions of the standard I/O library are not reentrant, due to the usage of global data structures (e.g. buffers).

Yes, we've been lazy and used `printf` within signal handlers.
You can't.

The big list of reentrant functions

accept access aio_error aio_return aio_suspend alarm bind cfgetispeed cfgetospeed cfsetispeed cfsetospeed chdir chmod chown clock_gettime close connect creat dup dup2 execle execve _Exit _exit fchmod fchown fcntl fdatasync fork fpathconf fstat fsync ftruncate getegid geteuid getgid getgroups getpeername getpgrp getpid getppid getsockname getsockopt getuid kill link listen lseek lstat mkdir mkfifo open pathconf pause pipe poll posix_trace_event pselect raise read readlink recv recvfrom recvmsg rename rmdir select sem_post send sendmsg sendto setgid setpgid setsid setsockopt setuid shutdown sigaction sigaddset sigdelset sigemptyset sigfillset sigismember signal sigpause sigpending sigprocmask sigqueue sigset sigsuspend sleep socket socketpair stat symlink sysconf tcdrain tcflow tcflush tcgetattr tcgetpgrp tcsendbreak tcsetattr tcsetpgrp time timer_getoverrun timer_gettime timer_settime times umask uname unlink utime wait waitpid write

— reference: SUS; APUE, Chapter 10

Outline

1. Signal concepts
2. Unreliable signals API
3. Reliable signals API
4. Real-time signals

Signal sets

Several features of the reliable signal API manipulate sets of signals.

- e.g. a process willing to block a set of signals
- e.g. retrieving the current set of pending signals

The first (trivial) part of the reliable signals API is used to manipulate sets of signals and offers basic set operations.
The data type `sigset_t` is defined by POSIX.1 to represent a signal set.

sigset_t manipulation

POSIX.1 functions

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int  sigfillset (sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

Returns: *0 if OK, -1 on error*

```
#include <signal.h>
int sigismember(const sigset_t *set, int signo);
```

Returns: *1 if true, 0 if false, -1 on error*

sigset_t manipulation (cont.)

Glibc functions

(only if _GNU_SOURCE is defined)

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

int sigisemptyset(sigset_t *set);
```

Returns: *1 if set contains no signals, 0 otherwise*

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

int sigorset(sigset_t *dest, sigset_t *left,  sigset_t *right);

int sigandset(sigset_t *dest, sigset_t *left, sigset_t *right);
```

Returns: *0 if OK, -1 on error*

sigset_t manipulation (cont.)

Let $\mathcal{S}$ be the set of all available signals, S, S_1, S_2 sets of signals being manipulated, and $n \in \mathcal{S}$ a specific signal.

The following straightforward analogies with set operations hold:

sisget_t	set operation
sigemptyset(S)	$S \leftarrow \emptyset$
sigfillset(S)	$S \leftarrow \mathcal{S}$
sigaddset(S, n)	$S \leftarrow S \cup \{n\}$
sigdelset(S, n)	$S \leftarrow S \setminus \{n\}$
sigorset(S, S_1, S_2)	$S \leftarrow S_1 \cup S_2$
sigandset(S, S_1, S_2)	$S \leftarrow S_1 \cap S_2$
sigismember(S, n)	$n \in S$
sigisemptyset(S)	$S = \emptyset$

Signal masks

Definition (Signal mask)

A signal mask is a set of signals that are currently blocked from delivery to a process.

Every UNIX process is associated to a signal mask.
Using the `sigprocmask` syscall a process can:

1. retrieve the signal mask
2. change the signal mask
3. do both (1) and (2) in a single *atomic action*

sigprocmask

#include <signal.h>

int sigprocmask(**int** how, **const** sigset_t ∗restrict set, sigset_t ∗restrict oset);

Returns: *0 if OK, -1 on error*

Retrieving the signal mask

- if *oset* is non-NULL, it will be filled with the current signal mask

Changing the signal mask

- if *set* is non-NULL, the current signal mask *M* will be changed according to the value of *how*:

how	effect
SIG_BLOCK	$M \leftarrow M \cup set$
SIG_UNBLOCK	$M \leftarrow M \setminus set$
SIG_SETMASK	$M \leftarrow set$

- before `sigprocmask` returns, at least one of the unblocked and pending signals (if any) get delivered

sigpending

Reminder: blocked signals are raised but not delivered.

During the interim, signals are pending and can be retrieved by the target process using:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Returns: *0 if OK, -1 on error*

The interface is the same of `sigprocmask`'s *oset* parameter: `sigpending`'s *set* will be filled with the set of currently pending signals.

`sigpending` does not *change* the set of pending signals.

sigprocmask & sigpending — example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

void sig_quit(int signo) {
        printf("caught SIGQUIT\n");
        if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
                err_sys("can't reset SIGQUIT");
}
int main(void) {
        sigset_t newmask, oldmask, pendmask;
        if (signal(SIGQUIT, sig_quit) == SIG_ERR)
                err_sys("can't catch SIGQUIT");

        sigemptyset(&newmask);
        sigaddset(&newmask, SIGQUIT);   /* block SIGQUIT */
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
                err_sys("SIG_BLOCK error");
        sleep(5);       /* SIGQUIT here will remain pending */
        if (sigpending(&pendmask) < 0)
                err_sys("sigpending error");
        if (sigismember(&pendmask, SIGQUIT))
                printf("\nSIGQUIT pending\n");

        /* Reset signal mask which unblocks SIGQUIT */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
        printf("SIGQUIT unblocked\n");
        sleep(5);       /* SIGQUIT here will core dump */
        exit(EXIT_SUCCESS);
}
```

sigprocmask & sigpending — example (cont.)

Demo

sigprocmask & sigpending — example (cont.)

Demo

- the pending signal is indeed delivered *before* sigprocmask returns
- we use SIG_SETMASK on the old mask to unblock instead of SIG_UNBLOCK and SIGQUIT
  - in this example the difference is irrelevant, but it does matter when interacting with other code, if we want to be modular wrt signal handling

Welcome, sigaction!

The main ingredient of the reliable signal API is sigaction, used to modify and/or inspect signal dispositions.

sigaction completely subsumes signal and extends it with several extra features:

- expressive signal handlers, which get passed information about the signal raise (and delivery) context
- the ability to block signals during handler execution
- control over restart (EINTR) behavior

sigaction, not ~~signal~~, should be used in all *new* code that deals with signals.

sigaction

Let's look at sigaction prototype:

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Returns: *0 if OK, -1 on error*

- *signo* is the signal whose disposition we want to act upon
- *act*, if non-NULL, is the new signal disposition we want to set for *signo*
- *oact*, if non-NULL, is where we want the current signal disposition (before change, if any) to be put

struct sigaction

The sigaction struct encodes signal dispositions:

```
struct sigaction {
  void      (*sa_handler)(int);   /* old-style handler */
  sigset_t sa_mask;               /* extra signals to block */
  int       sa_flags;             /* signal options */
  /* alternate, new-style handler */
  void      (*sa_sigaction)(int, siginfo_t *, void *);
}
```

The handler can be specified in two *alternative* ways:

1. by default, sa_handler is used as in signal
2. if sa_flags contains the SA_SIGINFO flag, sa_sigaction is used instead.

Example of how to increase prototype expressivity, without giving up backward compatibility.

struct sigaction (cont.)

When using sa_sigaction, signal handlers will be invoked via a richer prototype:

```
void handler(int signo, siginfo_t *info, void *context);
```

- *signo* is as before;
- *info* is a rich structure giving information about the event that caused the signal;
- *context* can be cast to a ucontext_t structure that identifies the context of the process at signal delivery time
  - see getcontext (2)

struct siginfo_t — example

As an example, here is a typical siginfo_t on Linux x86:

```
typedef struct siginfo_t {
  int si_signo;           /* signal number */
  int si_errno;           /* errno value */
  int si_code;            /* signal code (depend on signal) */
  pid_t si_pid;           /* sending process's PID */
  uid_t si_uid;           /* sending process's real UID */
  int si_status;          /* exit value or signal */
  clock_t si_utime;       /* user time consumed */
  clock_t si_stime;       /* system time consumed */
  sigval_t si_value;      /* signal payload value */
  int si_int;             /* POSIX.1b signal */
  void *si_ptr;           /* POSIX.1b signal */
  void *si_addr;          /* memory location that caused fault */
  int si_band;            /* band event */
  int si_fd;              /* file descriptor */
}
```

Note that POSIX.1 only mandates si_signo and si_code.

struct siginfo_t — use cases

- for SIGCHLD, si_pid, si_status, si_uid, si_utime, and si_stime will be set, easing collecting termination information from children
- for SIGILL, SIGSEGV, SIGBUS, SIGFPE, and SIGTRAP si_addr contains the address responsible for the fault, easing debugging
- if a signal is generated by some error condition, si_errno contains the corresponding errno, easing error recovery
- ...

The wonderful world of si_code (cit.)

si_code deserves special mention as it explains how (for user-generated signals) or why (for kernel-generated signals) the signal has been sent. Admissible values of si_code depend on the signal. Some examples:

Signal	si_code	Reason
Any	SI_USER	signal sent by kill
	SI_ASYNCIO	completion of async I/O request
	SI_MESGQ	message arrival on a message queue
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	termination (no core)
	CLD_STOPPED	child has stopped
SIGSEGV	SEGV_MAPERR	address not mapped
	SEGV_ACCERR	invalid permission
SIGFPE	FPE_INTDIV	division by zero
	FPE_INTOVF	integer overflow
...	...	...

By inspecting si_code we can discriminate among very precise signal causes, and hence piggyback more logic into signal handlers.

More on sigaction — persistence

A signal disposition installed with sigaction is granted by POSIX.1 to persist across signal delivery.

The need of code like the following is gone! (and the behavior is no longer implementation-dependent)

```c
/* signal handler */
void my_handler(int signo) {
    signal(SIGINT, my_handler);  /* re-establish handler */
    ...                          /* process signal */
}

int main(void) {
    ...
    signal(SIGINT, my_handler);  /* establish handler */
    ...
}
```

More on sigaction — signal masks

sigaction guarantees that during signal handler execution a temporary signal mask is in effect.

- the kernel guarantees that the temporary mask is in effect only during signal handler execution and that the original mask will be restored as soon as the handler returns

The temporary signal mask composition is as follows:

- the signal being handled, *signo*, is included by default in the temporary signal mask
- addition signals are passed using the sigset_t sa_mask field of struct sigaction

More on `sigaction` — options

`sa_flags` field of `struct sigaction` allows for further control of `sigaction` behavior. It is the bitwise OR of 0 (no options), 1, or more of a set of flags that includes:

SA_NODEFER	do not include *signo* in temporary mask by default
SA_NOCLDWAIT	*automatic child reaping*, for SIGCHLD
SA_ONSTACK	request signal handling on an alternate stack
SA_RESTART	request automatic restart of syscalls interrupted by *signo*
SA_RESETHAND	request *one-shot* mode, reset disposition of *signo* after delivery
SA_SIGINFO	use `sa_sigaction` (already seen)

With `sa_flags` we can obtain back behaviors of unreliable signals *on demand* (but why???).

sigaction — example

```c
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

#define SIG_WHY(i)      ((i)->si_code==SI_USER ? "(kill)" : "")

void sig_dispatch(int signo, siginfo_t *info, void *ctxt) {
        if (info->si_signo == SIGUSR1)
                printf("received SIGUSR1 %s\n", SIG_WHY(info));
        else if (info->si_signo == SIGSEGV)
                printf("received SIGSEGV %s\n", SIG_WHY(info));
}
int main(void) {
        struct sigaction act;
        sigemptyset(&act.sa_mask);
        act.sa_flags = SA_SIGINFO;
        act.sa_sigaction = sig_dispatch;

        if (sigaction(SIGUSR1, &act, NULL) == -1)
                err_sys("can't catch SIGUSR1");
        if (sigaction(SIGSEGV, &act, NULL) == -1)
                err_sys("can't catch SIGSEGV");
        for ( ; ; )
                sleep(60);
} /* sigaction.c */
```

sigaction — example (cont.)

Demo

A (reliable) implementation of signal

```
#include <signal.h>

sighandler_t signal(int signo, sighandler_t func) {
        struct sigaction act, oact;

        act.sa_handler = func;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        if (signo != SIGALRM) { /* or not ... */
#ifdef  SA_RESTART
                act.sa_flags |= SA_RESTART;
#endif
        }
        if (sigaction(signo, &act, &oact) < 0)
                return(SIG_ERR);
        return(oact.sa_handler);
}
```

Snoozing signals — redux

- with `sigprocmask` we can avoid signal interference within critical code region
- if the signal is raised while blocked, it will be delivered as soon as we unblock it

how can we explicitly wait for a signal, no matter if it happens while blocked or after?

```
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region */

if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

/* signal window open again */
pause();    /* wait for signal */
```

sigsuspend

We're back at square one: if the signal is raised either while blocked, or shortly after unblocking and before pause, it will be lost and the process might block forever (race condition!).

The solution is a system call to **atomically** reset the signal mask and put the process to sleep. Enter sigsuspend:

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Returns: *-1 with errno set to EINTR*

- set the signal mask to *sigmask* and put the process into sleep
- sigsuspend will return after the handler of a caught signal returns
  - note: always return a EINTR failure
- the signal mask is returned to its previous value before returning

Helper — pr_mask

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

void pr_mask(const char *label) {
        sigset_t sigs;
        int errno_save;

        errno_save = errno;
        if (sigprocmask(0, NULL, &sigs) < 0)
                err_sys("sigprocmask error");

        printf("%s: ", label);
        if (sigismember(&sigs, SIGINT))   printf("SIGINT ");
        if (sigismember(&sigs, SIGQUIT))  printf("SIGQUIT ");
        if (sigismember(&sigs, SIGUSR1))  printf("SIGUSR1 ");
        if (sigismember(&sigs, SIGUSR2))  printf("SIGUSR2 ");
        if (sigismember(&sigs, SIGALRM))  printf("SIGALRM ");
        /* etc... */
        printf("\n");
        errno = errno_save;
}
```

sigsuspend — example

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "helpers.h"

void sig_int(int signo) {
        pr_mask("\nin sig_int");
}
int main(void) {
        sigset_t newmask, oldmask, waitmask;

        pr_mask("program start");
        if (signal(SIGINT, sig_int) == SIG_ERR)  err_sys("signal(SIGINT) error");
        sigemptyset(&waitmask);
        sigaddset(&waitmask, SIGUSR1);
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGINT);

        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
                err_sys("SIG_BLOCK error");
        pr_mask("<critical region>");
        sleep(4);
        pr_mask("</critical region>");

        if (sigsuspend(&waitmask) != -1)  err_sys("sigsuspend error");
        pr_mask("after return from sigsuspend");

        /* Reset signal mask which unblocks SIGINT. */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
        pr_mask("done");
        exit(EXIT_SUCCESS);
} /* sigsuspend.c */
```

sigsuspend — example (cont.)

```
$ ./sigsuspend
program start:
<critical region>: SIGINT
^C
</critical region>: SIGINT

in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
done:
$
```

- after sigsuspend returns the signal mask is back to its previous value
- why is SIGINT blocked inside sig_int?
- we block SIGUSR1 only to show that the temporary mask set by sigsuspend is in effect

sigsuspend — critical region loop

```
#include "helpers.h"
int quitflag = 0;       /* global flag */
void sig_int(int signo) {
        if (signo == SIGINT)            printf("\ninterrupt\n");
        else if (signo == SIGQUIT)      quitflag = 1;
}
int main(void) {
        sigset_t        newmask, oldmask, zeromask;
        if (signal(SIGINT, sig_int) == SIG_ERR
            || signal(SIGQUIT, sig_int) == SIG_ERR)
                err_sys("signal() error");
        sigemptyset(&zeromask);
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGQUIT);
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
                err_sys("SIG_BLOCK error");
        while (quitflag == 0)
                /* can do critical region work here */
                sigsuspend(&zeromask);
        /* SIGQUIT got caught */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
        exit(EXIT_SUCCESS);
} /* sigsuspend2.c */
```

sigsuspend — critical region loop (cont.)

```
$ ./sigsuspend2
^C
interrupt
^C
interrupt
^C
interrupt
^\
$
```

The example shows the typical...

critical region design pattern for signals

1. block undesired signals by default
2. loop
   a. critical region work
   b. sigsuspend with old, more liberal mask to handle accumulated signals

Reminder — parent/child synchronization

We need synchronization primitives that processes can use to synchronize and avoid race conditions.

As a proof of concept we will consider the following primitives:[5]

WAIT_PARENT child blocks waiting for (a “signal” from) parent

WAIT_CHILD parent blocks waiting for (a “signal” from) children

TELL_PARENT(pid) child “signals” parent

TELL_CHILD(pid) parent “signals” child

[5] we’ll also have TELL_WAIT in both processes, for initialization

Reminder — TELL/WAIT intended usage

```
int main(void) {
    pid_t pid;

    TELL_WAIT();

    if ((pid = fork()) < 0)  err_sys("fork error");
    else if (pid == 0) {
        WAIT_PARENT();     /* parent first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(EXIT_SUCCESS);
}
```

Signal-based TELL/WAIT

We can now give an implementation of TELL/WAIT, based on (reliable) signals:

```c
static int sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;
static void sig_usr(int signo) { /* will catch SIGUSR{1,2} */
    sigflag = 1;
}
void TELL_WAIT(void) {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);
    /* Block SIGUSR1 and SIGUSR2 */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}
```

Signal-based TELL/WAIT (cont.)

```
void TELL_PARENT(pid_t pid) {
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void WAIT_PARENT(void) {
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* wait for parent */
    sigflag = 0;

    /* Reset signal mask to original value. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Signal-based TELL/WAIT (cont.)

```
void TELL_CHILD(pid_t pid) {
    kill(pid, SIGUSR1);     /* tell child we're done */
}

void WAIT_CHILD(void) {
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* wait for child */
    sigflag = 0;

    /* Reset signal mask to original value. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

system vs signals

We've already given an implementation of system based on fork, exec, and wait. What would happen to system in a signal setting?

In particular:

1. what if the process calling system is catching SIGCHLD?
2. what if the user hits Ctrl-C while system's child is running?

system vs signals

We've already given an implementation of system based on fork, exec, and wait. What would happen to system in a signal setting?

In particular:

1. what if the process calling system is catching SIGCHLD?
   - the process is probably doing so to be notified of termination of *his own* children, not of system's specific children
2. what if the user hits Ctrl-C while system's child is running?
   - signals are delivered to all foreground processes in the terminal

A proper system implementation should shield the caller from delivery of signals that are specific to system's child.
More generally, you should think at which signals to mask in the parent when fork-ing.

Signal-aware system implementation

```
#include        <sys/wait.h>
#include        <errno.h>
#include        <signal.h>
#include        <unistd.h>

int system(const char *cmdstring) {
    pid_t pid;
    int status;
    struct sigaction ignore, saveintr, savequit;
    sigset_t chldmask, savemask;

    if (cmdstring == NULL)  return(1);

    ignore.sa_handler = SIG_IGN;     /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);                       /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);
```

Signal-aware system implementation (cont.)

```
    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {                  /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);                 /* exec error */
    } else {                                                /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    } /* now restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);
    return(status);
} // APUE, Figure 10.28
```

Outline

1. Signal concepts

2. Unreliable signals API

3. Reliable signals API

4. Real-time signals

Real-time signals

We have seen that real-time signals are the subset of available signals comprised in between RTMIN and RTMAX.

Their delivery semantics differs substantially from that of ordinary signals:

1. real-time signals are granted to be queued
   - i.e. for each raise of a real-time signal there will be a corresponding delivery
2. real-time signals are delivered according to their priority (ordered from RTMIN to RTMAX)
3. real-time signals can carry payloads

Signal queuing example — redux

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "helpers.h"

void sig_rt(int signo) {
        printf("caught signal %d\n", signo);
}

int main(void) {
        sigset_t newmask, oldmask;

        if (signal(SIGRTMIN, sig_rt) == SIG_ERR
            || signal(SIGRTMIN+1, sig_rt) == SIG_ERR)
                err_sys("signal error");
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGRTMIN);
        sigaddset(&newmask, SIGRTMIN+1);
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) /* block SIGRT* */
                err_sys("SIG_BLOCK error");
        sleep(5);
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) /* unblocks SIGRT* */
                err_sys("sigprocmask error");
        if (signal(SIGRTMIN, SIG_DFL) == SIG_ERR) err_sys("signal error");
        if (signal(SIGRTMIN+1, SIG_DFL) == SIG_ERR) err_sys("signal error");
        printf("SIGRT* unblocked\n");
        while (1) {
                sleep(5);
        }
        exit(EXIT_SUCCESS);
} /* queue.c */
```

Signal queuing example — redux

Demo

siqgueue

The interface for sending signals with payloads is provided by:

#include <signal.h>

int sigqueue(pid_t pid, **int** sig, **const union** sigval value);

Returns: *0 if OK, -1 on error*

The first two arguments are as in `kill`. The 3rd argument allow to send signals with payloads that can be either integers or pointers:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
}
```

Target process can retrieve the payload via the `si_value` field of `struct siginfo_t` (only for `sigaction` new style handlers).

siqqueue — example

With sigqueue, signals can be used as very expressive communication mechanisms (although not necessarily handy. . .).

As an extreme example, we show how to transfer a file across processes using signal payloads and relying on real-time signal queuing.

siqgueue — example (cont.)

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include "helpers.h"

int main(int argc, char **argv) {
        int pid, c;
        union sigval val;
        if (argc < 2) {
                printf("Usage: sigqueue-send PID\n");
                exit(EXIT_FAILURE);
        }
        pid = atoi(argv[1]);
        while ((c = getchar()) != EOF) {
                val.sival_int = c;
                if (sigqueue(pid, SIGRTMIN, val) < 0)
                        err_sys("sigqueue error");
        }
        exit(EXIT_SUCCESS);
} /* sigqueue-send.c */
```

sigqueue — example (cont.)

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "helpers.h"

void receive_char(int signo, siginfo_t *info, void *ctxt) {
        if (putchar(info->si_value.sival_int) == EOF)
                err_sys("putchar error");
        fflush(stdout);
}

int main(void) { /* sigqueue-recv */
        struct sigaction act;
        sigemptyset(&act.sa_mask);
        act.sa_flags = SA_SIGINFO;
        act.sa_sigaction = receive_char;
        if (sigaction(SIGRTMIN, &act, NULL) == -1)
                err_sys("sigaction error");
        for ( ; ; )
                pause();
        exit(EXIT_SUCCESS);
} /* sigqueue-recv.c */
```

siqgueue — example (cont.)

```
$ ./sigqueue-recv &
[1] 11546
$ ./sigqueue-send 11546 < /etc/issue
Debian GNU/Linux wheezy/sid \n \l
$
```

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

- ~~Unreliability I — reset to default~~
  - `sigaction`

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

- ~~Unreliability I — reset to default~~
  - `sigaction`
- ~~Unreliability II — snoozing signals~~
  - `sigaction` (for signal handlers)
  - `sigprocmask` + `sigsuspend` (for arbitrary critical regions)

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

- ~~Unreliability I — reset to default~~
  - `sigaction`
- ~~Unreliability II — snoozing signals~~
  - `sigaction` (for signal handlers)
  - `sigprocmask` + `sigsuspend` (for arbitrary critical regions)
- ~~Unreliability III — EINTR uncertainty~~
  - `sigaction` + SA_RESTART

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

- ~~Unreliability I — reset to default~~
  - sigaction
- ~~Unreliability II — snoozing signals~~
  - sigaction (for signal handlers)
  - sigprocmask + sigsuspend (for arbitrary critical regions)
- ~~Unreliability III — EINTR uncertainty~~
  - sigaction + SA_RESTART
- ~~Unreliability IV — signal queuing~~
  - sigqueue + real-time signals

Signal (un)reliability

We have improved over most of the causes of signal unreliability:

- ~~Unreliability I — reset to default~~
  - sigaction
- ~~Unreliability II — snoozing signals~~
  - sigaction (for signal handlers)
  - sigprocmask + sigsuspend (for arbitrary critical regions)
- ~~Unreliability III — EINTR uncertainty~~
  - sigaction + SA_RESTART
- ~~Unreliability IV — signal queuing~~
  - sigqueue + real-time signals
- ~~Unreliability V — causality~~
  - real-time signal priorities (only partially addressed)

Signal reliability on Linux

- the Linux kernel version of `signal` provides System V semantics (i.e. signal disposition reset by default, no signal blocking during handler execution)
- starting from version 2 of glibc (i.e. all GNU `libc6` and above), the `signal` wrapper does not call kernel's `signal` but uses `sigaction`, offering reliable semantics
- the default behavior with respect to EINTR is a bit complicated, but well documented in `signal(7)`. The main principles are:
    - common I/O syscalls respect the presence (or absence) of SA_RESTART (read/write, open, wait, socket interfaces, flock, etc.)
    - syscalls where interruptions are part of their semantics ignore SA_RESTART, are always interrupted (pause/sigsuspend/sleep, timeout-powered socket interfaces, select/poll, etc.)