Basic POSIX signal concepts

Last modification date: 05.11.2018

Note: The material does not cover real-time signal generation/delivery and thread-specific signal handling

POSIX signal

Signal - a mechanism by which a process or thread may be **notified of**, or **affected by**, an **event** occurring in the system. The term signal is also used to refer to the event itself.

Examples of such events:

- **hardware exceptions** : hardware faults, timer expiration, terminal activity
- **actions by processes**: calls of **`kill()`**, **`alarm()`**, exiting child process and other

Signals in API:

POSIX extension	Header file	Prefixes of API symbols
-	<signal.h>	sa_, uc_, SIG[A-Z], SIG_[A-Z]
XSI		ss_, sv_
RTS		si_, SI_, sigev_, SIGEV_, sival_

Important signals

Nr	Name	Meaning	Action
1	**SIGHUP**	Hangup	Exit
2	**SIGINT**	tty interrupt (typically: ^C)	Exit
9	**SIGKILL**	**Unconditional** process termination	Exit
11	**SIGSEGV**	Segmentation Fault	Core dump + exit
13	**SIGPIPE**	Broken Pipe	Exit
14	**SIGALRM**	Alarm Clock	Exit
15	**SIGTERM**	Software interrupt	Exit
	SIGUSR1,2	Two „user interrupts” (no pre-defining meaning)	Exit
	SIGCHLD	Child Status Changed	Ignore
	SIGCONT	Process to be continued	Continue
	SIGSTOP	**Unconditional** stop for a process	Stop
	SIGTSTP	Stop of a process via tty (typically: ^Z)	Stop
	SIGTTIN	Stopped (tty input)	Stop
	SIGTTOU	Stopped (tty output)	Stop

Programmatic signal generation

`int kill(pid_t pid, int sig)`

the function sends a signal number **sig**>0 .

- `pid > 0` to a process with given PID
- `pid == 0` to all process that belong to the process group of the sender (normally to all children and perhaps some ancestors)
- `pid == -1` to all processes in the system (except `init`)
- `pid < - 1` to all processes that belong to the process group `pgid=-pid`

If `sig==0`, no signal is sent, but normal error checking is performed.

The function returns normally 0 and –1 upon failure (the global variable `errno` , defined in `<errno.h>` is set, to inform about the reason of failure)

Note: there exists a system command of the same name and purpose

Signal targets

POSIX: „At the time of generation, a determination shall be made whether the signal has been generated for the process or for a specific thread within the process.

- Signals which are generated by some action attributable to a particular thread, such as a hardware fault, shall be generated for the thread that caused the signal to be generated.
- Signals that are generated in association with a process ID or process group ID or an asynchronous event, such as terminal activity, shall be generated for the process."

Signals can thus be **synchronously-generated** or they are **asynchronous events**

Actions to be taken by the recipient

- Each **process** has always defined an **action to be taken** in response to **each signal** defined by the system.
- Signal can be
  - **delivered** - when the appropriate action for the process and signal is taken (**ignoring** or calling **signal handlers** - **user defined** or **default** i.e. system defined).
  - **accepted** - when the signal is selected and returned by one of the **`sigwait()`** functions (signal was handled **synchronously**).
- Signal can also be **blocked** – postponing decision on delivery or acceptance.
- Between the generation of a signal and its **delivery** or **acceptance**, the signal is said to be **pending**.

Blocking

- Each process has a **signal mask** that defines the set of signals currently blocked from delivery to it. The signal mask from a process is inherited from its parent.
- `sigset_t mask` – mask of signals.
- Bits of the signal mask can be changed/tested:

```
int sigemptyset(sigset_t *set); // zeroes all mask bits
int sigfillset(sigset_t *set); // sets all mask bits
int sigaddset(sigset_t *set, int signo); // sets bit nr signo
int sigdelset(sigset_t *set, int signo); // clears bit nr signo
int sigismember(sigset_t *set, int signo); // tests bit nr signo
```

- `int sigprocmask(int how, const sigset_t *set, sigset_t *old)`

  The function modifies the set of blocked signals (if `set!=NULL`) and returns previous mask (if `old!=NULL`). Parameter how:

  `SIG_BLOCK` - the specified signals will be blocked by process

  `SIG_UNBLOCK` - the specified signals will be unblocked

  `SIG_SETMASK` - the specified mask becomes the process signal mask

- `int sigpending(int how, const sigset_t *set)`

  Returns information on pending signals. `sigismember(set,nr)` call can be used to determine if a signal of given `nr` is pending.

Determination of the action to be taken

- The determination of which action is to be taken is made **at the time the signal is delivered**, independently of the means by which the signal was originally generated.
- POSIX: If a subsequent occurrence of a pending signal is generated, it is implementation-defined as to whether the signal is **delivered or accepted more than once**. UNIX: typically **only one** pending (non-RT) signal is allowed.
- **The order** in which multiple, simultaneously pending (non-RT) signals are delivered to or accepted by a process **is unspecified**. Programmer can change signal mask to make a signal delivered or use **`sigwait()`** call to have the signal accepted.

Signal actions upon delivery

- Actions upon delivery
  - ignore the signal; symbolically: **`SIG_IGN`**
  - perform signal-specific default (system-handled) action (**ignoring** or **process termination** with possible **core dump**, **stopping** process, process **continuation**); symbolically: **`SIG_DFL`**
  - catch signal using a provided handler function pointer.

  ```
  void handler_name(int signo);
  ```

  where **`signo`** is the signal number that caused invocation of the handler

- Initially all signals shall be set to **`SIG_DFL`** or **`SIG_IGN`** prior to entry to the **`main()`** routine of the process.

Programming asynchronous handling

```
int sigaction ( // defines action upon signal delivery
   int sig,  // signal which handling is to be set
   const struct sigaction *act,  // current disposition
   struct sigaction *oact  // old disposition
);

struct sigaction{  // the structure holding disposition
   void(*sa_handler)(int);  // pointer to a signal handler
                // or SIG_DFL, SIG_IGN
       sigset_t sa_mask;  // mask of blocked signals
       int sa_flags;  // flags that modify signal handling
}
```

Notes:

1. During signal handler execution the next occurrence of the same signal and signals marked by respective `sa_mask` bits are blocked.
2. `if(sa_flags&SA_RESTART)` → returning from a handler resumes the interrupted „long” library function (otherwise the function fails, setting `errno==EINTR`)

Remarks on signal handler

- The handler can recognize signal number that triggered its call because of handler parameter, but not the signal origin.
- Currently handled signal is blocked for the time of the handler's execution. Other signals can be blocked if necessary by setting process signal mask.
- The handler blocks the execution of the main code, thus it must be as short as possible. Time consuming functions (like sleep or blocking I/O) should not be used.
- The handler should interact with remaining code with global atomic variables of type

```
volatile sig_atomic_t
```

Correct asynchronous signal handling

```
volatile sig_atomic_t usr_interrupt; // interrupt-safe flag

void handler(int signr){    // signal handler
   if(signr==SIGUSR1) usr_interrupt++; // safely increment flag
}

int main(int argc, char *argv[]){
sigset_t mask, oldmask;
struct sigaction sa;
. . . . . . . . . . . . . . . . . . . . . .
   sigemptyset(&mask);
   sigaddset(&mask, SIGUSR1);
   sigprocmask(SIG_BLOCK, &mask, &oldmask); // block SIGUSR1 saving old mask in
oldmask
   memset(sa,0,sizeof(struct sigaction));     // preparation of struct sigaction
   sa.sa_handler = handler;          // for new disposition
   if(sigaction(SIGUSR1,&sa,NULL)){ // catching SIGUSR1 with handler
requested
         . . . // error handling
   } else {
      while(!usr_interrupt)                // check SIGUSR1 delivery flag
         sigsuspend(&oldmask);             // suspend process if not
          . . .
      sigprocmask(SIG_UNBLOCK, &mask, NULL);// retrieve old signal mask into oldmask
   }
. . . . . . . . . . . . . . . . . . . . . .
}
```

Warning: incorrect use of global flags

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
struct two_words {
#ifdef _LONG
   long a, b;
#else
   int  a, b;
#endif
} mem; // global structure
////////////////////////////////
void handler(int signum){
// print-out the global structure
#ifdef _LONG
   printf("%ld,%ld\n",
#else
   printf("%d,%d\n",
#endif
      mem.a, mem.b);
// schedule next SIGALRM signal
   alarm(1);
}
```

```
int main(void){
struct sigaction sa;
static struct two_words
      zeros = { 0, 0 },
      ones = {1, 1};
   mem = zeros;
   memset(sa,0,sizeof(sa));
   sa.sa_handler = handler;
   if(sigaction(SIGUSR1,&sa,NULL){
         alarm(1);//schedule alarm
         while (1){// spinning
                 mem = zeros;
                 mem = ones;
         }
         return EXIT_SUCCESS;
   }
   return EXIT_FAILURE;
}
```

// NOTE: for 64b architecture define _LONG,

// to see effects of non-atomic updates of

// mem structure.

Side-effects of asynchronous signal handling

If a signal is delivered during execution of some blocking („slow”, interruptible) system functions, the functions are terminated prematurely with –1 return code and with **errno** set to **EINTR** (unless **`sa_flags&SA_RESTART is`** set when defining signal handling with **`sigaction()`**)

Example. Implementation of 5 second time-out while copying standard input to standard output.

```
void hand(int sig){ // Normally no long operations are performed in handlers
   fprintf(stderr,"hand(%d)\n".signr); // This line is provided for
                 // (improper) demonstration of handler activity
   return;
}
int main(int argc, char *argv[]){
char buf[20];
int n;
static struct sigaction sa; // Note: static variables are 0 initialized
  sa.sa_handler=hand;
//sa.sa_flags=SA_RESTART; // Activation of automatic restart. What if
  uncommented.?
  if(sigaction(SIGALRM,&sa,NULL)) return EXIT_FAILURE 1;
  alarm(5);
  while((n=read(0,buf,sizeof(buf)))>0)     write(1,buf,n);//
  fprintf(stderr,"n=%d, errno=%d\n",n,errno);
  if(errno) perror("readsig");
  return 0;
}
```

Side-effects – cont.

Signal delivery affects also functions which put a process asleep, e.g. **`sleep()`** and **`nanosleep()`**. The functions return prematurely after signal is handled by a handler. To sleep for a predefined amount of type, despite signal handling, the following tricks can be used.

For sleep function, typical construct is:

```
int tt, t = 5;// 5 second sleep
  for(tt = t; tt > 0; tt = sleep(tt));
```

For nanosleep function

```
struct timespec tt, t = {5, 0};
for(tt=t;nanosleep(&tt,&tt);)
  if(EINTR!=errno) {
  perror("nanosleep:");
  ...
}
```

In GNU programming environment the macro **`TEMP_FAILURE_RETRY`** is defined.

Pattern of use:

```
#define _GNU_SOURCE
#include <unistd.h>
      ...
TEMP_FAILURE_RETRY(fun_call)
```

```
while( (n = TEMP_FAILURE_RETRY(
                read(0,buf,sizeof(buf))
        )) >0)
                 write(1,buf,n);
```

The macro can be used to wait for blocking system function call, ignoring intermediate returns due to signal handling

Handling SIGCHLD signal

How to eliminate zombies ?

1. Create **SIGCHLD** handler ➔

```
void SIGCHLD_handler(int sig){
  pid_t pid;
  for (;;) {
       pid = waitpid(0, NULL, WNOHANG);
       if (0 == pid) return;
       if (0 >= pid) {
         if (ECHILD == errno) return;
          perror("waitpid:");
       }
}
```

2. Activate the handler (**sigaction**() call) ➔

```
struct sigaction sa;
memset(sa,0,sizeof(struct sigaction));
sa.sa_handler=SIGCHLD_handler;
if(sigaction(SIGCHLD,&sa,NULL)) {
   // error handling
}
```

3. Call **wait**() before exiting

```
while (TEMP_FAILURE_RETRY(wait(NULL))>0);
```

Async-signal-safe functions (POSIX Std 1003.1-2001)

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

All **async-signal-safe functions** shall behave as defined when called from or interrupted by a signal-catching function. When a signal interrupts an unsafe function or the signal-catching function calls an unsafe function, the behavior is **undefined**.

Synchronous signal handling

`int sigsuspend(const sigset_t *mask);` – waiting for delivery of signals other than specified with the mask (which are temporarily blocked)

`int sigwait(const sigset_t *mask, int *signr);` – a blocked signal, specified with the mask, signal is removed from the list of blocked signals and its number returned via **`*signr`**.

`int pause(void);` - blocks the calling process until any signal is delivered to the process (i.e. signal is properly handled by a signal handler).

```
sigset_t mask, oldmask;
int signr;
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &mask, &oldmask); // block SIGUSR1, saving
                                           // old signal mask in oldmask
    while(! sigwait(&mask,&signr){// retrieve pending signal nr into signr
          . .. ..          // handle the signal number signr
          printf(”signal nr %d accepted\n”,signr);
    }
```

Note: **`sigwait()`** is **blocking** if there is no pending signal, suspending execution of the caller.

Terminal generated signals

stty utility shall set or report on terminal I/O characteristics for the device that is its standard input. Example use cases:

- **stty –a** Writes to standard output all the current settings for the terminal.
- **stty operands** Sets terminal I/O characteristics, e.g.:

  sane Reset all modes to some reasonable, unspecified, values.

  tostop (**−tostop**) Send **`SIGTTOU`** for background output.

  <control> ***string*** Sets <*control*> to ***string***.

Typically*:*

control	**Char. *string***	**Meaning**
intr	^C	SIGINT generation
quit	^\	SIGQUIT generation
susp	^Z	SIGTSTP generation