Signals

Table of Contents

Introduction

- Signals are Software Interrupts to Handle Asynchronous Events
- `signal(7)`
- Signal Concepts
  - numerous conditions
    - the terminal-generated signals
    - hardware exceptions
    - software conditions
    - the `kill(2)` function
    - the `kill(1)` command
  - handle signal
    - ignore the signal
      - but we cannot ignore `SIGKILL` and `SIGSTOP`
    - catch the signal, the register customized handler
      - but we cannot register `SIGKILL` and `SIGSTOP`
    - let the default action apply
      - every signal has a default action
- List of Linux Signals (e.g. Ubuntu 18.04 x86_64)
  - POSIX.1-1990 Standard

Signal	Value	Action	Comment
`SIGHUP`	1	Term	Hangup detected on controlling terminal or death of controlling process
`SIGINT`	2	Term	Interrupt from keyboard
`SIGQUIT`	3	Core	Quit from keyboard
`SIGILL`	4	Core	Illegal Instruction
`SIGABRT`	6	Core	Abort signal from `abort(3)`

Signal	Value	Action	Comment
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

- SUSv2 and POSIX.1-2001

Signal	Value	Action	Comment
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad system call (SVr4); see also seccomp(2)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD); see setrlimit(2)

- Linux 2.4

Signal	Value	Action	Comment
SIGIOT	6	Core	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7	Term	Emulator trap
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)

Signal	Value	Action	Comment
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for SIGCHLD
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for SIGPWR
SIGLOST	-,-,-	Term	File lock lost (unused)
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Core	Synonymous with SIGSYS

The Signal Function

- `signal(2)`: `void (*signal(int signum, void (*handler)(int)))(int)`
  - `typedef void (*sighandler_t)(int);`
  - `sighandler_t signal(int signum, sighandler_t handler);`
  - return: previous disposition of signal, `SIGERR` on error
  - `signum` is the name of the signal
  - `handler` is the function to be called when the signal occurs
    - can also be `SIG_IGN` or `SIG_DFL`
    - `SIG_IGN` ignore `signum`
    - `SIG_DFL` use default action in above table
  - implementations differs among system
    - better to use `sigaction` function

- Example

```
void sig_usr(int signo) {
  if (signo == SIGUSR1) {
    printf("received SIGUSR1\n");
  } else if (signo == SIGUSR2) {
    printf("received SIGUSR2\n");
  } else {
    printf("receive signal %d\n", signo);
  }
}

int main() {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
    printf("can't catch SIGUSR1");
  }
  if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
    printf("can't catch SIGUSR2");
  }
  while (1) {
    pause();
```

```
  }
}
```

```
$ ./a.out &
[1] 15512
$ kill -USR1 15512
received SIGUSR1
$ kill -USR2 15512
received SIGUSR2
$ kill 15512  # default send SIGTERM
[1]+ Terminated ./a.out
$ kill -l     # check available signals
 1) SIGHUP   2) SIGINT   ...
```

- Signal Setup
  - signal is either default or ignore when program is executed
  - `fork(2)`: child inherits parent's signal dispositions
  - `exec(3)`: child changes the signal dispositions to their default action
  - the shell sets `SIGINT` and `SIGQUIT` in the background process to be ignored

Interrupted System Calls

- A System Call may be Interrupted
  - if process catch a signal while in slow system call
  - then the system call return an error and `errno` was set to `EINTR`
    - `EINTR`: 4, interrupted system call
- Handle Interrupted System Calls

```
again:
  if ((n = read(fd, buf, BUFFSIZE)) < 0) {
    if (errno == EINTR)
      goto again;
  }
```

- Prevent Interrupted System Calls
  - 4.2BSD introduced automatic restarting of certain interrupted system calls
  - `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`
  - the feature can be disabled if you don't like it

Reentrant Functions

- When a Signal is Being Caught

    - instruction is interrupted
    - execute signal handler instructions
    - if the handler returns, then the process continues the interrupted instructions

- If Handler Uses the Same System Call as the Interrupted Instructions
    - if works well, they are reentrant functions
    - some functions can not work with reentrant, because
        - they use static data structures
        - they are part of the standard I/O library (buffering)
        - they call `malloc` or `free`
    - reentrant functions example

```
void handler(int signo) {
  tmpnam(NULL);
}
int main() {
  char *s = tmpnam(NULL);
  signal(SIGALRM, handler);
  alarm(3);
  for (int i = 0; i < 6; ++i) {
    printf("tmpnam = %s\n", s);
    sleep(1);
  }
}
```

- `SIGCLD` and `SIGCHLD`
    - the signal disposition is `SIG_DFL`: default ignore
    - zombie avoidance
        - explicitly set disposition to `SIG_IGN`
        - no zombie will be created if children of the calling process terminate

Reliable Signal Model

- Delivery of a Signal
    - delivered: a process received the signal and action is taken
    - pending: a signal is generated, but not delivered
- Blocking the Delivery of a Signal
    - if signal is blocked and handler is `SIG_DFL` or a handler
    - then the signal remains pending until
        - unblock the signal or
        - change handler to `SIG_IGN`
- Block Signal is Generated More Than Once
    - POSIX.1 allows deliver signal once or more
    - most UNIX systems do not queue signals

- Deliver Signals More Than Once
  - POSIX.1 does not specify the order
  - it suggests that signals related to the current state be delivered before other signals
- Signal Mask
  - defines the blocked signals in a process
  - each signal has a corresponding bit

Functions for Handling Signals

- `kill(2)`: `int kill(pid_t pid, int sig);`
- `raise(3)`: `int raise(int sig);`
  - return: 0 OK, -1 error
  - send signal to
    - `pid > 0`: `pid` process
    - `pid == 0`: every process in the process group as same as the caller
    - `pid == -1`: every process the caller has permission to send, except process 1
    - `pid < -1`: every process in the process group `-pid`
  - `raise(sig)` equals `kill(getpid(), sig)`
  - `NULL` signal
    - may be used to check the existence of a process
    - no such process: `kill` returns -1 and `errno` is set to `ESRCH`
      - `ESRCH`: 3, no such process
- `alarm(2)`: `unsigned int alarm(unsigned int seconds);`
  - return: 0 or number of seconds remaining until previously scheduled alarm
  - `SIGALRM` is generated when the timer expires, default terminate the process
  - only one alarm clocks per process
- `pause(2)`: `int pause(void);`
  - return: -1 with `errno` set to `EINTR`
  - suspend a process until it received a signal
- `abort(3)`: `void abort(void);`
  - send `SIGABRT` to the caller
  - the signal can be caught by a handler, but
    - the handler will not return if it calls `exit(3)`, `_exit(2)` or `_Exit(2)`
    - if the handler returns, then `abort` terminates the process
    - if the handler calls `siglongjmp`, the program may continue to execute
- Signal Sets: `sigsetops(3)`
  - operation
    - `sigemptyset(3)`: `int sigemptyset(sigset_t *set);`
      - initializes `set` to empty
    - `sigfillset(3)`: `int sigfillset(sigset_t *set);`

            - initializes `set` to full
        - `sigaddset(3)`: `int sigaddset(sigset_t *set, int signum);`
            - add `signum` to `set`
        - `sigdelset(3)`: `int sigdelset(sigset_t *set, int signum);`
            - delete `signum` from `set`
        - return: 0 OK, -1 error
    - membership test
        - `sigismember(3)`: `int sigismember(const sigset_t *set, int signum);`
        - return: 1 if true, 0 if false, -1 error

- `sigprocmask(2)`: `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
    - return: 0 OK, -1 error
    - block or unblock signals
    - `how`
        - `SIG_BLOCK`: $newblkset = curblkset \cup set$
        - `SIG_UNBLOCK`: $newblkset = curblkset - set$
        - `SIG_SETMASK`: $newblkset = set$
    - if `oldset` is not `NULL`, then previous value of the signal mask is stored `oldset`
    - if `set` is `NULL`, then the mask remains unchanged

- `sigpending(2)`: `int sigpending(sigset_t *set);`
    - return: 0 OK, -1 error
    - get currently pending signals

- Example

```c
int main() {
  sigset_t newmask, oldmask, pendmask;
  /* sig_quit print caught SIGQUIT and then reset SIGQUIT handler */
  if (signal(SIGQUIT, sig_quit) == SIG_ERR) {
    printf("err signal register\n");
  }
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGQUIT);
  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
    printf("err sigprocmask\n");
  }
  printf("SIGQUIT blocked\n");
  sleep(5);
  if (sigpending(&pendmask) < 0) {
    printf("err sigpending\n");
  }
  if (sigismember(&pendmask, SIGQUIT)) {
    printf("SIGQUIT in pending\n");
  }
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
    printf("err sigprocmask\n");
```

```
    }
    printf("SIGQUIT unblocked\n");
    sleep(5);
}
```

```
$ ./a.out
^\^\^\
SIGQUIT in pending
caught SIGQUIT
SIGQUIT unblocked
```

- `sigaction(2)`: `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - return: 0 OK, -1 error
  - examine or modify (or both) the signal action
  - `sigaction`

```
struct sigaction {
  void (*sa_handler)(int);  // default handler
  void (*sa_sigaction)(int, siginfo_t *, void *);  // alternative
handler when SA_SIGINFO is enabled
  sigset_t sa_mask;  // blocked signals when executing the
handler
  int sa_flags;      // various options
  void (*sa_restorer)(void);
};
```

  - `sa_flags`

`sa_flags`	**Description**
`SA_INTERRUPT`	obsoleted after Linux 2.6.19
`SA_NOCLDSTOP`	when `signum` is `SIGCHLD`, do not receive notification when child processes stop
`SA_NOCLDWAIT`	when `signum` is `SIGCHLD`, do not transform children into zombies
`SA_RESETHAND`	when handler is established, restore the action to the default upon entry to the handler
`SA_RESTART`	when handler is established, restart some system calls instead of generating `EINTR`
`SA_NODEFER`	when handler is established, do not prevent the signal from being received from within its own handler

sa_flags	Description
SA_SIGINFO	when handler is established, use sa_sigaction handler
SA_ONSTACK	when handler is established, call the handler on an alternate signal stack provided by sigaltstack(2)
SA_RESTORER	used in C libraries, sa_restorer field contains the address of a "signal trampoline", see sigreturn(2)

- sigsuspend(2): int sigsuspend(const sigset_t *mask);
  - return: -1 with errno set to EINTR
  - replace signal mask and then suspends the process until
    - a signal whose action is to invoke a handler or
    - a signal to terminate a process, sigsuspend(2) does not return in this condition
  - it is not possible to block SIGKILL or SIGSTOP
  - catching signal between sigprocmask(2) and pause(2)
    - after handler is done, the program is suspended
    - use sigsuspend(2) to prevent this condition

- Example: Wait for a Global Variable to be Set

```
volatile sig_atomic_t quitflag;

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) printf("signal(SIGINT)
error");
  if (signal(SIGQUIT, sig_int) == SIG_ERR) printf("signal(SIGQUIT)
error");
  sigemptyset(&zeromask);
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGQUIT);
  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
printf("SIG_BLOCK error");
  while (quitflag == 0) sigsuspend(&zeromask);
  quitflag = 0;
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
printf("SIG_SETMASK error");
}
```

- Example: Process Synchronization

```
// library
volatile sig_atomic_t sigflag;
sigset_t newmask, oldmask, zeromask;
void sig_usr(int signo) { sigflag = 1; }

void TELL_WAIT(void) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR) printf("signal(SIGUSR1)
error");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR) printf("signal(SIGUSR2)
error");
  sigemptyset(&zeromask);
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGUSR1);
  sigaddset(&newmask, SIGUSR2);
  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
printf("SIG_BLOCK error");
}

void TELL_PARENT(pid_t pid) { kill(pid, SIGUSR2); }

void TELL_CHILD(pid_t pid) { kill(pid, SIGUSR1); }

void WAIT_PARENT(void) {
  while (sigflag == 0) sigsuspend(&zeromask);
  sigflag = 0;
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
printf("SIG_SETMASK error");
}

void WAIT_CHILD(void) {
  while (sigflag == 0) sigsuspend(&zeromask);
  sigflag = 0;
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
printf("SIG_SETMASK error");
}

// application
int main(void) {
  pid_t pid;
  TELL_WAIT();
  if ((pid = fork()) < 0) {
    printf("fork error");
  } else if (pid == 0) {
    WAIT_PARENT(); /* parent goes first */
    charatatime("output from child\n");
  } else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);
  }
}
```

`setjmp(3)` and `longjmp(3)`

- `setjmp(3)`: `int setjmp(jmp_buf env);`
- `longjmp(3)`: `void longjmp(jmp_buf env, int val);`
- `setjmp(3)` and `longjmp(3)` can Jump Accross Functions
  - handler perform `longjmp(3)` to main loop
  - but mask of a signal is set automatically when that signal is caught
  - `longjmp(3)` exits the handler with the mask, and it will cause problems
- Solutions
  - `setjmp(3)` should save signal mask
  - the mask is restored when `longjmp(3)` is called
- POSIX.1 does not define how do `setjmp(3)` and `longjmp(3)` handle masks
  - FreeBSD and MAC OS X save and restore the mask automatically
  - however, Linux must work with `sigsetjmp(3)` and `siglongjmp(3)`
- `sigsetjmp(3)`: `int sigsetjmp(sigjmp_buf env, int savesigs);`
  - return: 0 if called directly, nonzero if returning from a call to `siglongjmp(3)`
  - if `savesigs` is nonzero
    - usually set `savesig` to nonzero
    - the process's current signal mask is saved in `env`
    - the mask will be restored if a `siglongjmp(3)` is later performed with this `env`
- `siglongjmp(3)`: `void siglongjmp(sigjmp_buf env, int val);`
- Example

```
sigjmp_buf jmpbuf;
volatile sig_atomic_t canjump;

void sig_usr1(int signo) {
  time_t starttime;
  if (canjump == 0) return;
  pr_mask("starting sig_usr1: ");
  alarm(3);
  starttime = time(NULL);
  for (;;)
    if (time(NULL) > starttime + 5) break;
  pr_mask("finishing sig_usr1: ");
  canjump = 0;
  siglongjmp(jmpbuf, 1);
}

void sig_alrm(int signo) { pr_mask("in sig_alrm: "); }
```

```
int main(void) {
  if (signal(SIGUSR1, sig_usr1) == SIG_ERR) printf("signal(SIGUSR1)
error");
  if (signal(SIGALRM, sig_alrm) == SIG_ERR) printf("signal(SIGALRM)
error");
  pr_mask("starting main: ");
  if (sigsetjmp(jmpbuf, 1)) {
    pr_mask("ending main: ");
    exit(0);
  }
  canjump = 1;
  for (;;) pause();
}
```

```
$ ./a.out &
[1] 8874
starting main:
$ kill -SIGUSR1 8874
starting sig_usr1: SIGUSR1
in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:   # SIGUSR1 if savesigs is 0 (i.e. mask in the sig_usr1
function)
```

The System Function

- In Ch06
  - a simple system implementation is introduced
  - but it does not handle signals
- POSIX.1 requires the system function
  - ignore `SIGINT`/`SIGQUIT`
    - these signals should be sent only to child process
  - block `SIGCHLD`
    - parent will not confuse between the termination of child and other child
- System Implementation
  - preserve original signal action for `SIGINT` and `SIGQUIT`
  - block `SIGCHLD`
  - child
    - restore handler
    - unblock `SIGCHLD`
    - execute command
  - parent
    - wait for child
  - restore handler and unblock `SIGCHLD`

```c
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;
  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);
  sigaddset(&chldmask, SIGCHLD);
  if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0) return -1;
  if ((pid = fork()) < 0) {
    status = -1;
  } else if (pid == 0) {
    sigaction(SIGINT, &saveintr, NULL);
    sigaction(SIGQUIT, &savequit, NULL);
    sigprocmask(SIG_SETMASK, &savemask, NULL);
    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127);
  } else {
    while (waitpid(pid, &status, 0) < 0) {
      if (errno != EINTR) {
        status = -1;
        break;
      }
    }
  }
  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;
}
```

- Job Control Signals
  - except `SIGCHLD`, most applications don't handle these signals
  - vi editor will save/restore terminal state when the process is stopped/continued

Signal	Description
SIGCHLD	Child stopped or terminated
SIGCONT	Continue if stopped
SIGSTOP	Stop signal
SIGTSTP	Stop typed at terminal
SIGTTIN	Terminal input for background process

Signal	Description
SIGTTOU	Terminal output for background process