Signals

- Common Signals
- Common Signals Continued
- Signals Continued
- Signal Function
- Sending Signals
- Sending Signals Continued
- An Example Signal Handler
- Example Signal Handler Execution Log
- Signal Gotchas
- Unreliable Signals
- Remembering Occurrence of a Signal
- Interrupted System Calls
- Interrupted System Calls Continued
- Non-Reentrant Functions
- `SIGCHLD` versus `SIGCLD`
- Implement `sleep()` Using `alarm()`: First attempt
- Problems with `sleep()` First Attempt
- Implement `sleep()` Using `alarm()`: Second attempt
- `sleep()` Second Attempt: Review
- Reliable Signal Terminology

- Signal Sets and `sigprocmask()`
- `sigpending()`
- `sigaction()`
- `sigaction()` Flags
- `sigaction()` Flags Continued
- Reliable `signal()`
- `sigsetjmp()` and `siglongjmp()`
- Preventing a Signal in a Critical Region
- `sigsuspend()` Function
- A Final `sleep()`
- A Final `sleep()` Continued
- Real-Time Signals
- Real-Time Signal Details
- Sending Real-Time Signals
- References

Common Signals

`SIGABRT`
  Generated by calling `abort()`.

`SIGALRM`, `SIGPROF`, `SIGVTALRM`
  Generated by calling `alarm()` or `setitimer()`.

`SIGBUS`, `SIGEMT`, `SIGFPE`, `SIGIOT`, `SIGILL`, `SIGPWR`, `SIGSEGV`, `SIGTRAP`
  Indicates hardware conditions.

`SIGCHLD`
  Indicates a child termination.

`SIGCONT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`
  Job control signals.

`SIGHUP`
  Indicates terminal hangup. Sent to session leader. If session leader terminates, then it is sent to each foreground process.

Common Signals Continued

`SIGINFO`

Information requested on foreground processes (`^T`).

`SIGINT`

Signal generated to foreground processes by interrupt char (`^C`).

`SIGIO`, `SIGPOLL`

I/O signals.

`SIGKILL`

Kills a process. Cannot be caught.

`SIGPIPE`

Caused by a write to a pipe not being read. Similarly for sockets.

`SIGQUIT`

Terminates foreground process group with core dump.

Signals Continued

SIGSYS

Bad system call.

SIGTERM

Terminate process.

SIGURG

Urgent condition. Used for out-of-band data on network.

SIGUSR1, SIGUSR2

Available for any purpose by user.

SIGWINCH

Generated if window size changes.

SIGXCPU, SIGXFSZ

Generated if soft limit exceeded.

Signal Function

```
typedef void SigFunc(int);
SigFunc *signal(int signo, SigFunc *handler);
```

or without the `typedef`:

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

- `signo` specifies signal whose disposition is being changed.
- `handler` specifies the *signal handler*.
- Returns previous signal disposition if ok, `SIG_ERR` on error.
- No way to query signal disposition without changing it.
- Special dispositions of `SIG_IGN` (ignore the signal) or `SIG_DFL` (set the default disposition).
- In original Unix API with various problems and incompatibilities; modern programs should use `sigaction()`

Sending Signals

```
int kill(pid_t pid, int signo);

int raise(int signo);
```

- `raise()` allows a process to send a signal to itself (ANSI-C).

- Operation of kill depends on `pid`:

  `pid > 0`

  Signal sent to specified process.

  `pid == 0`

  Signal sent to all process with same group ID as sending process.

  `pid < 0`

  Sent to all processes with group ID equal to absolute value of `pid`.

  `pid == -1`

  Used for broadcasting.

Sending Signals Continued

- Super user can send signal to any process.
- Non super users can only send signals to processes with same real or effective UID (with exception of `SIGCONT` which can be sent by any process in same session).
- `kill(1)` is shell interface to `kill(2)`.
- `signo == 0` can be used to query existence of a process.

An Example Signal Handler

```
int
main(void)
{
  if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
    perror("SIGUSR1"); exit(1);
  }
  if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
    perror("SIGUSR2"); exit(1);
  }
  for ( ; ; )
    pause();
}

static void
sig_usr(int signo) /* argument is signal number */
{
  if (signo == SIGUSR1) {
    printf("received SIGUSR1\n"); /* unsafe */
  }
  else if (signo == SIGUSR2) {
    printf("received SIGUSR2\n"); /* unsafe */
  }
  else {
    fprintf(stderr, "received signal %d\n", signo);
    exit(1);
  }
  return;
}
```

Example Signal Handler Execution Log

```
$ ./sigusr &
[1] 4348
$ kill -USR1 4348
$ received SIGUSR1

$ kill -USR2 4348
$ received SIGUSR2

$ kill 4348
$
[1]+  Terminated              ./sigusr
$
```

Signal Gotchas

- Unreliable signals.
- Non-reentrant library functions.
- Inconsistent semantics (`SIGCLD` and `SIGCHLD`).
- Slow system calls may or may not be automatically restartable when interrupted by a signal.
- Race conditions.

Unreliable Signals

- Signal disposition reset to default when the signal occurred.
- Usually remedied by reinitializing the handler within the handler:

```
void sigInt(int signo) {
  if (signal(SIGINT, sigInt)) { ... }
  ...
}
```

- However, if signal occurred again before handler was reinitialized, then default action for signal would occur (usually process termination).
- Program would appear to work most of the time.

Remembering Occurrence of a Signal

- Use a flag to remember occurrence of a signal:

```
int sigIntFlag = 0;
...
signal(SIGINT, sigInt);
...

while (sigIntFlag == 0) pause;
...
void sigInt(int signo) {
  signal(SIGINT, sigInt);
  sigIntFlag = 1;
}
```

- What if signal occurs between test of `sigIntFlag` and `pause()`?

Interrupted System Calls

- *Slow* devices are those which can block *forever*: includes pipes, terminal devices, network devices, some IPC and `ioctl()` operations, as well as `pause()`.

- If a signal occurs while a system call is accessing a slow device, then in early Unix systems, the call returned with `errno` set to `EINTR`.

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

Interrupted System Calls Continued

- 4.2 BSD automatically restarted such system calls which were interrupted. Consider

  ```
  alarm(TIME_OUT);
  if ((n = read(fd, buf BUFSIZE)) < 0) {
    if (timeOutExpired) { ... }
    ...
  }
  ...
  ```

  Above code which attempts to put a time-out on a read, will not work with above semantics (it also has a race condition).

- 4.3 BSD allowed control over which signals interrupt systems calls and which signals lead to the call being automatically restarted.

- SysV never automatically restarted interrupted system calls by default.

Non-Reentrant Functions

- Non-reentrant functions cannot be called from a signal handler.

- POSIX specifies that certain functions are *async-signal-safe* (listed in text). Functions not on the list should not be called from signal handler.

- It is usually not safe to call any of the C library functions (like `printf()` or `malloc()`) unless the signal handler will also terminate the program, or the library has been guaranteed to be reentrant.

- Only 1 `errno` per process; hence it is necessary to save and restore `errno` within signal handler.

SIGCHLD versus SIGCLD

- Semantics of BSD SIGCHLD are normal, like that of any other signal.

- System V has unusual semantics for SIGCLD:

  - If disposition explicitly set to SIG_IGN, then no zombies.

  - If disposition set to be handled, then kernel checks to see if there are any terminated children, in which case it immediately calls the hander.

- Consider the need to reestablish the signal handler when the handler is first entered ... leads to a recursive loop!!

Implement `sleep()` Using `alarm()`: First attempt

```
static void
sigAlarm() {
  return;
}

unsigned int
sleep1(unsigned int nSecs) {
  if (signal(SIGALRM, sigAlarm)
      == SIG_ERR) {
    return nSecs;
  }
  alarm(nSecs);
  pause();
  return alarm(0);
}
```

Problems with `sleep()` First Attempt

1. If caller of `sleep1()` has a `alarm()` set, then that `alarm()` is lost. Can be corrected by using return value of `alarm()`.

2. Disposition of SIGALRM changed. Can be fixed by saving return value of `signal()` and restoring it before returning.

3. Race condition between `alarm()` and `pause()`.

Implement `sleep()` Using `alarm()`: Second attempt

```
static jmp_buf alarmEnv;

static void
sigAlarm(int signo)
{
  longjmp(alarmEnv, 1);
}

unsigned int
sleep2(unsigned int nSecs)
{
  if (signal(SIGALRM, sigAlarm) == SIG_ERR) {
    return nSecs;
  }
  if (setjmp(alarmEnv) == 0) {
    alarm(nSecs);
    pause();
  }
  return alarm(0);

}
```

`sleep()` Second Attempt: Review

- Fixes problem (3).
- Assume that (1) and (2) can be taken care of.
- If alarm interrupts another signal handler, then that signal handler is aborted!!

Reliable Signal Terminology

- A signal is *delivered* to a process when the action for a signal is taken.
- A signal is *pending* during the time it is generated and the time it is delivered.
- With reliable signals, a process has the option of *blocking* a signal. If a blocked signal occurs and its disposition is default or catch, then the signal remains pending until the process unblocks it or sets its disposition to ignore.
- The delivery of a blocked signal depends on its disposition at the time it is delivered, not the time at which it was generated.
- If more than 1 occurrence of the same signal can be pending, then the system may *queue* the signals. Most systems do not queue signals and only a single signal will be delivered.

Signal Sets and `sigprocmask()`

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sigNum);
int sigdelset(sigset_t *set, int sigNum);
int sigismember(const sigset_t *set, int sigNum);

int sigprocmask(int how, const sigset_t *set1,
                sigset_t *oset);
```

- `how` is used when `set1` is non-null: `SIG_BLOCK` blocks `set1` signals; `SIG_UNBLOCK` unblocks `set1` signals; `SIG_SETMASK` sets signal mask to `set1`.

- If `oset` non-`NULL`, then previous signal mask is returned via `oset`.

sigpending()

```
int sigpending(sigset_t *set);
```

- `sigpending()` returns all pending signals. Use `sigismember()` to process return value.

sigaction()

```
struct sigaction {
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;
} SigAction;
int sigaction(int signo,
              const SigAction *act,
              SigAction *oact);
```

- `sa_handler` field of SigAction allows specifying a handler just as for `signal()`.

- `sa_mask` indicates signals which should be masked out during handler execution. When handler returns, old mask is automatically restored.

- When handler is invoked, `signo` is always added to the mask which is installed before the handler is entered.

`sigaction()` Flags

`sa_flags` member of SigAction allows `sigaction()` to emulate different behaviors of `signal()`:

`SA_NOCHLDSTOP`

When `signo` is SIGCHLD, do not generate signal when a child stops (still generated when a child terminates).

`SA_RESTART`

Systems calls interrupted by this signal are automatically restarted.

`SA_ONSTACK`

If an alternate stack defined using `sigaltstack()` use alternate stack for delivery of this signal.

`SA_NOCLDWAIT`

Emulate Sys V behavior of not creating zombies for terminated children.

`sigaction()` Flags Continued

`SA_NODEFER`

Signal not automatically blocked when handler is entered.

`SA_RESETHAND`

Reset the handler to `SIG_DFL` before handler is entered.

`SA_SIGINFO`

Pass additional information (2 additional arguments) to signal handler.

Simulate unreliable signals using `SA_NODEFER|SA_RESETHAND`.

Reliable signal()

```
typedef void (SigFunc)(int sigNum);

SigFunc *
signal(int sigNum, SigFunc *func)
{
  struct sigaction act, oact;

  act.sa_handler = func;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT; /*Sun OS */
#endif
  }
  else {
#ifdef SA_RESTART
    act.sa_flags |= SA_RESTART;
#endif
  }
  if (sigaction(sigNum, &act, &oact) < 0) {
    return SIG_ERR;
  }
  return oact.sa_handler;
}
```

sigsetjmp() and siglongjmp()

```
int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, inv value);
```

- With reliable signal semantics, old signal mask is restored when a signal handler returns.
- If signal handler exits using `longjmp()`, then POSIX does not define whether or not old signal mask should be restored.
- If `sigsetjmp()` called with `savemask` non-zero, then a **corresponding** `siglongjmp()` out of a signal handler will restore the old signal mask.

Preventing a Signal in a Critical Region

```
sigset_t newMask, oldMask;

sigemptyset(&newMask);
sigaddset(&newMask, SIGINT);
if (sigprocmask(SIG_BLOCK, &newMask, &oldMask) < 0) {
  ...
}
/* critical region */
if (sigprocmask(SIG_SETMASK, &oldMask, NULL) < 0) {
  ...
}
pause(); /* wait for signal */
```

Race condition between second `sigprocmask()` and `pause()`.

`sigsuspend()` Function

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

- Resets signal mask to `mask` and goes to sleep **atomically**.

- If a signal is caught and the signal handler returns then returns (with `errno` set to EINTR) and the signal mask set to its previous value.

A Final `sleep()`

```
static void
sigAlarm(int sigNum)
{
  return;
}

unsigned int
sleep(unsigned int nSecs)
{
  struct sigaction newAct, oldAct;
  sigset_t newMask, oldMask, suspMask;
  unsigned int unslept;

  newAct.sa_handler = sigAlarm;
  sigemptyset(&newAct.sa_mask);
  newAct.sa_flags = 0;
  sigaction(SIGALRM, &newAct, &oldAct);
```

A Final `sleep()` Continued

```
  sigemptyset(&newMask);
  sigaddset(&newMask, SIGALRM);
  sigprocmask(SIG_BLOCK, &newMask, &oldMask);

  alarm(nSecs);

  suspMask = oldMask;
  sigdelset(&suspMask, SIGALRM);
  sigsuspend(&suspMask);

  unslept = alarm(0);
  sigaction(SIGALRM, oldAct, NULL);
  sigprocmask(SIG_SETMASK, &oldMask, NULL);

  return unslept;

}
```

Real-Time Signals

- Added by Posix.1b.
- Supported if `_POSIX_REALTIME_SIGNALS` is defined.
- Allows passing information (`int` value or `void *` pointer) to signal handler.
- Allows prioritizing of signals (lower signal numbers have higher priority).
- Allows queuing of signals.

Real-Time Signal Details

- Real-time signal numbers between SIGRTMIN to SIGRTMAX with `RTSIG_MAX` real-time signals in between (min. 8).

- Added a additional handler to `struct sigaction`:

```
struct sigaction {
  void (*sa_handler)();         /* SIG_DFL, SIG_IGN, or
                                 * pointer to function */
  void (*sa_sigaction)          /* Real-time signal */
    (int, siginfo_t *, void *); /* handler function */
  sigset_t sa_mask;             /* additinal signals to be blocked
                                 * during execution of handler */
  int sa_flags;                 /* special flags and options */
};
```

- New handler has prototype:

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

`context` is currently undefined.

- `siginfo_t` contains at least the following members:

```
int si_signo;             /* signal #.  Same as signo */
int si_code;              /* one of SI_USER, SI_QUEUE, SI_TIMER,
                           * SI_ASYNCHIO, SI_MESGQ */
union sigval si_value;    /* union { int sival_int;
                           *         void *sival_ptr;
                           *       };
```

Sending Real-Time Signals

```
int sigqueue(pid_t pid, int signo, const union sigval value);
```

- Additional parameter specifies information sent to the handler via the `info` argument.
- To guarantee queuing, `SA_SIGINFO` must be set in `sa_flags` in `struct sigaction`.
- Multiple signals with the same number generated by `sigqueue()` are queued upto a max of `SIGQUEUE_MAX` (typically 32).

References

Text: Chs 20 - 22.

APUE, Ch. 10.

Jim Frost, *UNIX Signals and Process Groups*, at http://www.cs.ucsb.edu/~almeroth/classes/W99.276/assignment1/signals.html.

FSF, *The GNU C Library*, at http://www.gnu.org/software/libc/manual/.