Contents

Signals

- Objectives:
  - An overview of signals
  - Related function libraries and problems, e.g., reliability & incompatibility.
- What is a signal?
  - Software interrupts
    - A way of handling asynchronous events
    - e.g., SIGABRT, SIGALRM.
  - 15 signals for Version 7, 31 signals for SVR4 & 4.3+BSD – <signal.h> (# > 0)

Signals

- Conditions to generate signals:
  - Terminal-generated signals – DELETE key, ^c → SIGINT
  - Signals from hardware exceptions → SIGFPE ~ divided-by-0, SIGSEGV ~ illegal memory access, etc.
  - Function *kill*
    - Owner or superuser
  - Shell command *kill,* e.g., kill –9 pid
  - Signals because of software conditions → SIGPIPE ~ reader of the pipe terminated, SIGALRM ~ expiration of an alarm clock

Signals

- The disposition of a signal (the action)
  - Ignore signals
    - SIGKILL and SIGSTOP can not be ignored.
    - There could be undefined behaviors for ignoring signals, such as SIGFPE.
  - Catch signals
    - Provide a signal handler
      - e.g., calling waitpid() when a process receives SIGCHLD
  - Apply the default action – Figure 10.1

Signals

- Remark – Figure 10.1
  - Terminate w/core – not POSIX.1
    - No core file: Non-owner setuid process, non-grp-owner setgid process, no access rights at the working dir, file is too big (RLIMIT_CORE)
    - core.*prog*
  - Hardware faults
    - Implementation-defined faults

Signals

- SIGABRT – terminate w/core
  - Call abort()
- SIGALRM – terminate
  - Call setitimer()
- SIGBUS – terminate w/core
  - Implementation-defined HW fault
- SIGCHLD – ignore
  - It was sent whenever a process terminates or stops

Signals

- SIGCONT – continue/ignore
  - Continue a stopped process, e.g., vi
- SIGEMT – terminate w/core
  - Implementation-defined HW fault
- SIGFPE – terminate w/core
  - Divid-by-0, floating point overflow, etc.
- SIGHUP – terminate
  - Disconnection is detected by the terminal interface (no daemons) → controlling process of a terminal
  - Triggering of the rereading of the config files (daemons)

Signals

- SIGILL – terminate w/core
  - Illegal hardware instruction (4.3BSD do it for abort() in the past)
- SIGINFO – ignore (BSD4.3+)
  - Status request for fg processes (^T)
- SIGINT – terminate
  - DELETE key or ^C
- SIGIO – terminate/ignore
  - Indicate an asynchronous I/O event (SIGIO=SIGPOLL, Terminate on SVR4)

Signals

- SIGIOT – terminate w/core
  - Implementation-defined HW fault (System V did it for abort() in the past)
- SIGKILL – terminate
  - Could not be ignored or caught!
- SIGPIPE – terminate
  - reader of the pipe/socket terminated
- SIGPOLL – terminate (SVR4)
  - A specific event happens on a pollable device.

Signals

- SIGPROF – terminate
  - A profiling timer expires (setitimer)
- SIGPWR – ignore (SVR4)
  - System dependent on SVR4
    - UPS → init shutdowns the system
- SIGQUIT – terminate w/core
  - ^\ triggers the terminal driver to send the signal to all foreground processes.
- SIGSEGV – terminate w/core
  - Invalid memory access

Signals

- SIGSTOP – stop process (like SIGTSTP)
  - Can not be caught or ignored
- SIGSYS – terminate w/core
  - Invalid system call
- SIGTERM – terminate
  - Termination signal sent by *kill* command
- SIGTRAP – terminate w/core
  - Implementation-defined HW fault
- SIGTSTP – stop process
  - Terminal stop signal (^Z) to all foreground processes

Signals

- SIGTTIN – stop process
  - Generated when gb processes try to read from the controlling terminal
- SIGTTOU – stop process
  - Generated when gb processes try to write to the controlling terminal
  - Could be generated by terminal operations, e.g., tcflush
- SIGURG – ignore
  - Urgent condition (e.g., receiving of out-of-band data on a network connection)

Signals

- SIGUSR1 – terminate
  - User-defined
- SIGUSR2 – terminate
  - User-defined
- SIGVTALRM – terminate
  - A virtual timer expires (setitimer)
- SIGWINCH – ignore
  - Changing of a terminal window size

Signals

- SIGXCPU – terminate w/core
  - Exceed the soft CPU time limit
- SIGXFSZ – terminate w/core
  - Exceed the soft file size!

signal

#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);

- signo – Figure 10.1
- func: SIG_ING, SIG_DFL, the address of the signal handler/ signal-catching function
  - SIGKILL & SIGSTOP
- Returned value: the address of the previous handler.

signal

- Remark:
  - SVR4: signal function – unreliable signal semantics
  - 4.3+BSD: defined in terms of sigaction function – reliable signal semantics
  - typedef void Sigfunc(int)
    - Sigfunc *signal(int, sigfunc *);
  - Constants:

    #define SIG_ERR (void (*)())-1

    #define SIG_DFL (void (*)())0

    #define SIG_IGN (void (*)())1

signal

- Program 10.1 – Page 272
  - Program to catch SIGUSR[12]
- Program Start-Up
  - All signals are set to their default actions unless some are ignored.
    - The exec functions change the disposition of any signals that are being caught to their default action.
      - Fork()
  - The shells automatically set the disposition of the interrupt and quit signals of background processes to “ignored”.

signals

```
int sig_int();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
   signal(SIGINT, sig_int);
```

- Not able to determine the current disposition of a signal without changing it.
- fork() lets the child inherits the dispositions of the parent!

Unreliable Signals

```
int sig_int();
…
signal(SIGINT, sig_int);
…
sig_int() {
    signal(SIGINT, sig_int);
    …  }
```

- Def: Unreliable Signals
  - Signals could get lost!
- Why?
  - The action for a signal was reset to its default each time the signal occurred.
  - The process could only ignore signals, instead of turning off the signals.

Unreliable Signals

```
int sig_int_flag;
main() {
    int sig_int();
    …
    signal(SIGINT, sig_int);
    …
    while (sig_int_flag == 0)
        pause();
}
sig_int() {
    signal(SIGINT, sig_int);
    sig_int_flag = 1; }
```

- Example:
  - A process could sleep forever!
  - pause() puts the process to sleep until a signal is caught.

Interrupted System Calls

- Traditional Approach
  - "Slow" system calls could be interrupted → errno = EINTR
- "Slow" System Calls (not disk I/O):
  - Reads from or writes to files that can block the caller forever (e.g., pipes, terminal devices, and network devices)
  - Opens of files (e.g., terminal device) that block until some conditions occurs.
  - pause, wait, certain ioctl operations
  - Some IPC functions

Interrupted System Calls

- A typical code sequence

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

- Restarting of interrupted system calls – since 4.2BSD
  - ioctl,read, readv, write, writev, wait, waitpid
  - 4.3BSD allows a process to disable the restarting on a per-signal basis.

Interrupted System Calls

- Figure 10.2 – Page 277
  - Summary of signal implementations
  - SV & 4.3+BSD: sigaction() with SA_RESTART
  - 4.3+BSD: sigvec or sigaction() with SA_RESTART
- Programs 10.12 and 10.13 are implementations of signals with/without restarting.

Reentrant Functions

- Potential Problem:
  - In the signal handler, we can't tell where the process was executing when the signal was caught!
    - Examples: malloc, getpwnam
    - Occurrence Time: Anytime, e.g., by timer…
- Figure 10.3 – reentrant functions
  - *-marked functions – not in POSIX.1, but in SVR4

Reentrant Functions

- Non-Reentrant Functions
  - Those which use static data structures
  - Those which call malloc or free
  - Those which are part of the standard I/O library – usage of global data structures
- Restoring of *errno* inside a handler
  - wait() and SIGCHLD
- Updating of a data structure – longjmp()
- Program 10.2 – Page 280
  - getpwnam(), SIGALRM, SIGSEGV

SIGCLD Semantics

- SIGCLD (SV) vs SIGCHLD (BSD, POSIX)
  - SIGCHLD
    - Handling is similar to those of other signals.
  - SIGCLD: Use signal() or sigset() to set its disposition →
    - The children of the calling process which sets its disposition to SIG_IGN will not generate zombie processes (not for BSD).
      - wait() returns –1 with errno = ECHILD until all children terminate.

SIGCLD Semantics

- The kernel checks if there is any child ready to be waited when SIGCLD is set to be caught → call SIGCLD handler!

- Program 10.3 – Page 282
  - The SIGCLD handler which does not work under SVR2!
  - Be aware of any “#define SIGCHLD SIGCLD”

Reliable Signals

- A signal is *generated* when
  - the event that causes the signal occurs!
  - A flag is set in the process table.
- A signal is *delivered* when
  - the action for the signal is taken.
- A signal is *pending* during
  - the time between its delivery and generation.

Reliable Signals

- A signal is *blocked* until
  - the process unblock the signal, or
  - The corresponding action become "ignore".
  - (if the action is either default or a handler)
  - *A signal mask* for each process – sigpromask()
- The system determines which signals are blocked and pending!
  - sigpending()

Reliable Signals

- Signals are *queued* when
  - a blocked signal is generated more than once.
  - POSIX.1 (but not over many Unix)
- Delivery order of signals
  - No order under POSIX.1, but its Rationale states that signals related to the current process state, e.g., SIGSEGV, should be delivered first.

kill and raise

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

- *pid* > 0 → to the process
- *pid* == 0 → to "all" processes with the same gid of the sender (excluding proc 0, 1, 2)
- *pid* < 0 → to "all" processes with gid == |pid|
- *pid* == -1 → broadcast signals under SVR4 and 4.3+BSD

kill and raise

- Right permissions must be applied!
  - Superuser is mighty!
  - Real or effective uid of the sender == that of the receiver
    - _POSIX_SAVED_IDS → receiver's saved set-uid is checked up, instead of effective uid
    - SIGCONT → member of the session
- signo ==0 ~ a null signal
  - Normal error checking is performed by kill() to see if a specific process exists.
    - kill() returns –1, and *errno* == ESRCH

alarm & pause

#include <unistd.h>

unsigned int alarm(unsigned int secs);

- There could be a delay because of processor scheduling delays.
- A previously registered alarm is replaced by the new value – the left seconds is returned!
- alarm(0) resets the alarm.
- Default: termination

alarm & pause

#include <unistd.h>

int pause(void);

- Return if a signal handler is executed.
  - Returns –1 with *errno* = EINTR
- Program 10.4 – Page 286
  - Potential problems:
    - Any previous alarm?
    - The lost of the previous SIGALRM handler
    - A race condition (between alarm() & pause())

alarm & pause

- Program 10.5 – Page 287
  - setjmp() inside sleep()
  - When a future SIGALRM occurs, the control goes to the right point in sleep2().
- Program 10.6 – Page 288
  - SIGALRM interrupts SIGINT handling
  - How if SIGALRM interrupts other signal handlers → they are aborted!

alarm & pause

- Program 10.7 – Page 289
  - Timeout a read (on a “slow” device)!
    - A race condition: (between alarm() & read())
    - Automatic restarting of read()?
      - No portable way to specifically interrupt a slow system call under POSIX.1.
- Program 10.8 – Page 290
  - Timeout & restart a read by longjmp!
    - Problems with other signal handlers!

Signal Sets

- Why?
  - The number of different signals could exceed the number of bits in an integer!

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


Signal Sets

- A macro Implementation if # of signals <= bits in an integer:

```
#define sigemptyset(ptr) ( *(ptr) = 0)
#define sigfillset(ptr) ( *(ptr) =
  ~(sigset_t)0, 0)
```

  - Program 10.9
    - Not one-line macros because of the checking requirements for validity and the setting of *errno* under POSIX.1.

sigprocmask

#include <signal.h>

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

- If *set* is not null, check *how* SIG_BLOCK, SIG_UNBLOCK, SIGMASK (Figure 10.4.); otherwise,…
- At least one of the pending, at least one of unblocking signals will be delivered when the sigprocmsk() returns.

- Program 10.10 – Page 294
  - Names of signals!

sigpenging

#include <signal.h>

int sigpending(sigset_t *set);

- Returns the set of pending, blocked signals

- Program 10.11 – Page 295
  - SIGQUIT is delivered until the signal is blocked and before sigprocmask() returns.
  - No queuing of signals.

sigaction

```
struct sigaction {
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;
};
```

```
#include <signal.h>
int sigaction(int signo,
    const struct sigaction *act,
    struct sigaction *oact);
```

- sa_mask: sigemptyset(), etc.
- Figure 10.5 – sa_flags
  - No queuing of signals
- Unlike signal(), signal handlers remain!

(including the delivered signal)

sigaction

- Program 10.12 – Page 298
  - 4.3+BSD: implement signal using sigaction
  - SVR4:
    - signal() provides the older, unreliable signal semantics
- Program 10.13 – Page 299
  - Prevent any interrupted system calls from being restarted.

sigsetjmp & siglongjmp

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int
   savemask);
void siglongjmp(sigjmp_buf env, int val);
```

- sigsetjmp() saves the current signal mask of the process in *env* if *savemask* !=0.
- setjmp & longjmp save/restore the signal mask:
  - 4.3+BSD, but not SVR4

sigsetjmp & siglongjmp

- Program 10.14 – Pages 300-301
  - sigsetjmp & siglongjmp
    - Restoring of the signal mask
    - _setjmp and _longjmp (4.3+BSD)
  - sig_atomic_t – variables of this type could be accessed with a single instruction
    - no extension across the page boundary

sigsuspend

```
if (sigprocmask(…) < 0)
  err_sys(…);

pause();
```

CPU Scheduling could occur!

```
#include <signal.h>
int sigsuspend(const
  sigset_t *sigmask);
```

- Set the signal mask to *sigmsk* and suspend until a signal is caught or until a signal occurs that terminates the process.
- Return –1. errno = EINTR

sigsuspend

- Program 10.15 – Page 304
  - When sigsuspend returns, the signal mask is restored.
- Program 10.16 – Page 306
  - Setting of a global variable
- Program 10.17 – Pages 307-308
  - SIGUSR1: parent → child
  - SIGUSR2: child → parent

sigsuspend

- How to call other system calls while waiting for a signal to occur?

interrupted →

```
while (select(…) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else …
}
```

Lost signals →

Block SIGINT & SIGALRM
Test flags
Call select+unblock as an atomic action

abort

#include <stdlib.h>

void abort(void);

- Sends SIGABRT to the process!
  - The SIGABRT won't return if its handler calls exit, _exit, longjmp, siglongjmp.
- ANSI C requires that if the signal is caught, and the signal handler returns, then abort still doesn't return to its caller.
- Program 10.18 – Page 11
  - POSIX.1 implementation of abort()

abort

- ANSI C
  - The Implementation determines whether output streams are flushed and whether temporary files are deleted.
- POSIX.1
  - POSIX.1 specifies that abort overrides the blocking or ignoring of the signal by the process.
  - If abort() terminates a process, all open standard I/O streams are closed (by fclose()); otherwise, nothing happens.

system

- Program 8.12 – Page 223
  - The implementation of system()
  - Ignoring of SIGINT and SIGCHLD & blocking of SIGCHLD (POSIX.2)
  - Program 10.19 – Page 312
    - Interactive command executed by system()
- Program 10.20 – Pages 314-315
  - Setting of a proper signal mask before fork()
  - Termination status of the shell

sleep

#include <unistd.h>

unsigned int sleep(unsigned int secs);

- Suspend until (1) the specified time elapsed, or (2) a signal is caught and the handler returns - returns the unslept seconds.

- Problems:
  - alarm(10), 3 secs, sleep(5)?
    - Another SIGALRM in 2 seconds? Both SVR4 & 4.3BSD – not POSIX.1 requirements
  - Alarm() & sleep() both use SIGALRM.

sleep

- Under SVR4, alarm(6), 3 secs, sleep(5) → sleep() returns in 3 secs!
- Program 10.21 – Page 318
  - Implementation of sleep()
  - Program 10.4 – Page 286
    - Unreliable signals!
  - No handling of previously set alarms.