Asynchronous Events: Signals

- Signals
  - Concepts
  - Generating Signals
  - Catching Signals
  - Waiting for Signals
  - Loose end: Program start-up
  - Loose end: Signal Handling and Threads
- Reading: Stevens, Ch 10

Signals: Concepts

- **Asynchronous Events**: Appear to occur at random time.
- Polling for asynchronous events?
  - Ask kernel: "Did Event X happen since I last checked?"
- Asynchronous handling of events:
  - Tell kernel: "If and when Event X happens, do the following."

Set and Forget!

Conditions that Generate Signals

Terminal-generated signals: triggered when user presses certain key on terminal. (e.g. **SIGINT** and ^C)

Hardware-exception generated signals: Hardware detects condition and notifies kernel. (e.g. **SIGFPE** divide by 0, **SIGSEGV** invalid memory reference)

`kill(2)` function: Sends any signal to another process.

`kill(1)` command: The command-line interface to `kill(2)`.

Software-condition generated signals: Triggered by software event (e.g. **SIGURG** by out-of-band data on network connection, **SIGPIPE** by broken pipe, **SIGALRM** by timer)

"Disposition" of the Signal

Tell the kernel what to do with a signal:

1. Ignore the signal. Works for most signals.

   Does not work for **SIGKILL** and **SIGSTOP**.

   Unwise to ignore hardware exception signals.

2. Catch the signal. Tell the kernel to invoke a given function whenever signal occurs.

   Example: Write signal handler for **SIGTERM** to clean up after program when it is terminated.

3. Default action. All signals have a default action.

Signals and their Default Actions (Mac OS X)

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

Generating Signals: `kill(2)` and `raise(3)`

```
#include <signal.h>

int kill(pid_t pid, int sig);
   /* send signal 'sig' to process 'pid' */
```

```
/* example: send signal SIGUSR1 to process 1234 */
if (kill(1234, SIGUSR1) == -1)
  perror("Failed to send SIGUSR1 signal");

/* example: kill parent process */
if (kill(getppid(), SIGTERM) == -1)
  perror("Failed to kill parent");
```

```
#include <signal.h>

int raise(int sig);
   /* Sends signal 'sig' to itself.
      Part of ANSI C library! */
```

"Catching" Signals: Signal Handlers

defining signal handlers the old-fashioned way…

```
#include <signal.h>

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

In English: "The function signal takes two arguments: an integer and a pointer to a function that takes an integer and returns nothing. The function signal itself returns a pointer to a function that takes an integer as argument and returns nothing."

The prototype can be simplified through the use of a `typedef` as follows:

```
typedef void Sigfunc(int);

Sigfunc * signal(int, Sigfunc*);
```

```
#define SIG_ERR (void(*)())-1
#define SIG_DFL (void(*)())0
#define SIG_IGN (void(*)())+1
```

Simple Signal Handling: Example

```
static void sig_usr(int); /* one handler for two signals */

int main (void) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    perror("cannot catch signal SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    perror("cannot catch signal SIGUSR2");
  for(;;) pause();
}

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

Modern Signal Handling: `sigaction()`

```
#include <signal.h>

int sigaction (int signo, const struct sigaction * act,
                          struct sigaction * oact);
/* install new signal handler from 'act', return old
   signal handler in 'oact'. */
```

```
struct sigaction {
  void     (*sa_handler)(int); /* SIG_DFL, SIG_IGN
                                  or pointer to function */
  sigset_t sa_mask;            /* signals to block */
  int      sa_flags;           /* flags and options */
  void     (*sa_sigaction)(int, siginfo_t *, void *);
}                              /* real-time handler */
```

```
struct sigaction new_act; /* set sighandler for SIGINT */

new_act.sa_handler = mysighandler; /* set new handler */
new_act.sa_flags   = 0;            /* no special options */
sigemptyset(&new_act.sa_mask);     /* clear mask */
sigaction(SIGINT, &new_act, NULL); /* where is error checking?! */
```

"real-time" Signals: Handling Memory Errors

```
/* -- SET FAULT HANDLER */
struct sigaction act;

act.sa_sigaction = SIGSEGV_handler;
sigemptyset(&act.sa_mask);
act.sa_flags     = SA_SIGINFO;

if (sigaction(SIGSEGV, &act, &oact) < 0)
        perror("sigaction");
```

```
/* -- SEGMENTATION FAULT HANDLER */
static void SIGSEGV_handler(int sig, siginfo_t * info, void * d) {
    if (info->si_signo == SIGSEGV) printf("SIGSEGV\n");
    else printf("*** other ***\n");
    printf("signal code   ");
    if (info->si_code == SEGV_ACCERR) printf("SEGV_ACCERR\n");
    else printf("**** other *****\n");
    printf("address %u\n", (unsigned long)(info->si_addr));
    do_something(info->si_addr);
}
```

Need more Details?!! : ucontext

```
/* -- SEGMENTATION FAULT HANDLER */
static void
SIGSEGV_handler(int sig, siginfo_t * info, ucontext_t * uc){

    [. . . ]

    /* -- IDENTIFY INSTRUCTION THAT CAUSED FAULT */
    unsigned long pc, *pcptr, instruction;
#if defined(SOLARIS)
    pc          = (unsigned long) uc->uc_mcontext.gregs[1];
    pcptr       = (unsigned long *) pc;
    instruction = *pcptr;
#endif
    /* -- READ OR WRITE OPERATION? */
    read_fault  = LOAD_INSTRUCTION(instruction);
    write_fault = STORE_INSTRUCTION(instruction);

    [ . . . ]
}
```

Signals: Terminology

- A signal is **generated** for a process when event that causes the signal occurs. (Hardware exception, software condition, etc.)
- A signal is **delivered** when action for a signal is taken.
- During the time between generation and delivery, signal is **pending**.
- A process has the option of **blocking** the delivery of a signal.
  - Signal remains blocked until process either (a) unblocks the signal, or (b) changes the action to ignore the signal.
- The system determines what to do with a blocked signal when the signal is delivered, not when it is generated.
- What happens when blocked signal is generated more than once? (If system delivers the signal more than once, the signal is **queued**. -- not done in most UNIX systems)
- What happens when more than one signal is ready to be delivered to a process? (POSIX does not specify order, but Rationale suggests that signals related to current state be delivered first)
- **signal mask** to control set of signals that are blocked from delivery.

Blocking Signals

blocking signals vs. ignoring signals

```
#include <signal.h>  /* modify signal mask */

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

```
/* the "how" parameter:
SIG_BLOCK   : add collection of signals
   to those already blocked.
SIG_UNBLOCK : delete a collection of
   signals from those currently blocked.
SIG_SETMASK : set the collection of
   blocked signals to given set. */
```

```
#include <signal.h>  /* manipulate sets of signals */

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

Waiting for Signals

- Typically, signal interrupts process execution to handle asynchronous event.
- What if process has nothing else to do?!

```
#include <signal.h>  /* wait for signal */

int pause(void);
```

How do we wait for **particular** Signal?

```
/* Approach 1, using a global variable (buggy!) */

/* Have the signal handler set quitflag to 1. */
static volatile sig_atomic_t quitflag = 0;

while (quitflag == 0)
        pause();
/* ?! */
```

```
/* Approach 2, using global variable (also buggy!) */

/* Have the sighandler set quitflag to 1. */
static volatile sig_atomic_t quitflag = 0;

int      signum;
sigset_t sigset;

sigemptyset(&sigset); sigaddset(&sigset, signum);
sigprocmask(SIG_BLOCK, &sigset, NULL);

while (quitflag == 0)
        pause();
/* ?! */
```

Waiting for specific Signal(s)

```
#include <signal.h>

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

1. The signal mask of process is set to `sigmask`.
2. Process is suspended until a signal is caught or until a signal occurs that terminates process.
3. If signal is caught and if signal handler returns, then
   1. `sigsuspend` returns
   2. signal mask of process is set to value before the call to `sigsuspend`.

How do we wait for Particular Signal?

```
/* Correct approach */
static volatile sig_atomic_t quitflag = 0;
signal(SIGINT, sig_int); signal(SIGQUIT, sig_int);
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGQUIT);
/* block SIGQUIT and save current signal mask */
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
while (quitflag == 0)
  sigsuspend(&zeromask);

/* SIGQUIT has been caught and is now blocked; do whatever */
quitflag = 0;
/* reset signal mask, which unblocks SIGQUIT */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

```
void sig_int(int signo) {/* signal handler */
  if (signo == SIGINT) printf("\ninterrupt\n");
  else if (signo == SIGQUIT) quitflag = 1;
  return;
}
```

Example: Protect Crit. Section from particular Signal

```
sigset_t newmask, oldmask, zeromask;

signal(SIGINT, sig_int);

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset  (&newmask, SIGINT);

/* block SIGINT and save current signal mask */
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

critical_section();

/* allow all signals and pause */
sigsuspend(&zeromask);

/* reset signal mask, which unblocks SIGINT */
sigprocmask(SIG_SETMASK, &oldmask, NULL);

/* … and continue processing */
```

Signal Disposition on Program Start-up

Process Creation (`fork()`)
- Child inherits parent's disposition.
- Also inherits the parent's signal handlers.

Program Loading (`exec()`)
- Status of all signals is either default or ignore.
- If process calling **`exec`** is ignoring signal, child ignores it as well.
- Example: Interactive shell and background processes.

```
cc main.c &
```

Signal Handling and Threads

- All threads in process share signal handlers.
- Signal delivery:
  - synchronous: delivered to thread that caused it.
  - asynchronous: delivered to some thread that has it unblocked.
  - directed: delivered to specific thread.
- Directed signal delivery:

```
#include <signal.h>
#include <pthread.h>

int pthread_kill(pthread_t thread, int sig);
```

```
if (pthread_kill(pthread_self(), SIGKILL))
  cerr << "Failed to commit suicide\n");
```

Signal Handling and Threads (II)

- Masking signals for threads.
  - Rule of thumb: use `sigprocmask` in main thread, and then use `pthread_sigmask()`.
- General approach to signal handling in multi-threaded programs:
  - Dedicate particular threads to signal handling
    - Simpler to localize
    - Simpler to control the priority and scheduling of signals.