Asynchronous Events: Signals

- Signals
  - Concepts
  - Generating Signals
  - Catching Signals
  - Waiting for Signals
  - Loose end: Program start-up
  - Loose end: Signal Handling and Threads

- Reading: R&R, Ch 8 and 13.5
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)**

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>terminate process</td>
<td>terminal line hangup</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>terminate process</td>
<td>interrupt program</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>create core image</td>
<td>create core image</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>create core image</td>
<td>illegal instruction</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>create core image</td>
<td>trace trap</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>create core image</td>
<td>abort program (formerly SIGIOT)</td>
</tr>
<tr>
<td>7</td>
<td>SIGTERM</td>
<td>create core image</td>
<td>emulate instruction</td>
</tr>
<tr>
<td>8</td>
<td>SIGFPE</td>
<td>create core image</td>
<td>floating-point exception</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>terminate process</td>
<td>kill program</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>create core image</td>
<td>bus error</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>create core image</td>
<td>segmentation violation</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>create core image</td>
<td>non-existent system call invoked</td>
</tr>
<tr>
<td>13</td>
<td>SIGPWR</td>
<td>terminate process</td>
<td>write on a pipe with no reader</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>terminate process</td>
<td>real-time timer expired</td>
</tr>
<tr>
<td>15</td>
<td>SIGTERM</td>
<td>terminate process</td>
<td>software termination signal</td>
</tr>
<tr>
<td>16</td>
<td>SIGURG</td>
<td>discard signal</td>
<td>urgent condition present on socket</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>SIGTSTP</td>
<td>stop process</td>
<td>stop signal generated from keyboard</td>
</tr>
<tr>
<td>18</td>
<td>SIGCONT</td>
<td>discard signal</td>
<td>continue after stop child status has changed</td>
</tr>
<tr>
<td>19</td>
<td>SIGSTOP</td>
<td>stop process</td>
<td>stop (cannot be caught or ignored)</td>
</tr>
<tr>
<td>20</td>
<td>SIGTTOU</td>
<td>stop process</td>
<td>background real write attempted to control terminal</td>
</tr>
<tr>
<td>21</td>
<td>SIGIO</td>
<td>discard signal</td>
<td>I/O is possible on a descriptor (see fcntl(2))</td>
</tr>
<tr>
<td>22</td>
<td>SIGXCPU</td>
<td>terminate process</td>
<td>cpu time limit exceeded (see setrlimit(2))</td>
</tr>
<tr>
<td>23</td>
<td>SIGXFSZ</td>
<td>terminate process</td>
<td>file size limit exceeded (see setrlimit(2))</td>
</tr>
<tr>
<td>24</td>
<td>SIGVTALRM</td>
<td>terminate process</td>
<td>virtual time alarm (see setitimer(2))</td>
</tr>
<tr>
<td>25</td>
<td>SIGWINCH</td>
<td>terminate process</td>
<td>profiling timer alarm (see setitimer(2))</td>
</tr>
<tr>
<td>26</td>
<td>SIGPROF</td>
<td>terminate process</td>
<td>status request from keyboard user defined signal 1</td>
</tr>
<tr>
<td>27</td>
<td>SIGURG</td>
<td>terminate process</td>
<td>user defined signal 2</td>
</tr>
<tr>
<td>28</td>
<td>SIGALRM</td>
<td>discard signal</td>
<td>user defined signal</td>
</tr>
<tr>
<td>29</td>
<td>SIGURG</td>
<td>terminate process</td>
<td>user defined signal</td>
</tr>
</tbody>
</table>
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

```
#include <signal.h>
void (*signal)(int signo, void (*)(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:

```c
typedef void Sigfunc(int);
Sigfunc * signal(int, Sigfunc*);
```
Simple Signal Handling: Example

```c
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()`

```c
#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 */

new_act.sa_handler = mysighandler; /* set new handler */
new_act.sa_flags   = 0; /* no special options */
if ((sigemptyset(&new_act.sa_mask) == 1) /* clear mask */
    || (sigaction(SIGINT, &new_act, NULL) == -1))
    perror("Failed to install SIGINT signal handler.");
```
"real-time" Signals: Handling Memory Errors

```c
/* -- 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 %lu", (unsigned long)(info->si_addr));
  do_something(info->si_addr);
}
```

Need more Details?!! : ucontext

```c
/* -- 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 an event that causes the signal occurs. (Hardware exception, software condition, etc.)
- A signal is **delivered** when an action for a signal is taken.
- During the time between generation and delivery, the signal is **pending**.
- A process has the option of **blocking** the delivery of a signal.
  - Signal remains blocked until the 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 a 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

```c
#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?!

```c
#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)

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 reset to value before the call to `sigsuspend`.

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

How do we wait for Particular Signal? (2)

```
/* 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("interrupt
");
    else if (signo == SIGQUIT) quitflag = 1;
    return;
}
```
Example: Protect Crit. Section from particular Signal

```c
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 */
```

Wait for Particular Signal: Use sigwait()

```c
#include <signal.h>
int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
```

1. The `sigwait()` function selects a pending signal from set, atomically clears it from the system's set of pending signals, and returns that signal number in the location referenced by `signo`.
2. It returns 0 on success and -1 on error with `errno` set.
Wait for Particular Signal: Use sigwait()

- This function is much cleaner and avoids races and errors

1. First block all signals
2. Put the signals you want to wait for in sigset_t
3. Call sigwait
4. sigwait blocks the process until at least one of these signals is pending.
5. It removes one of the pending signals and gives you the corresponding signal number in the second parameter.
6. Do what you want

Sigwait Example

```c
int main(void) {
    int signalcount = 0;
    int signo;
    int signum = SIGUSR1;
    sigset_t sigset;

    if ((sigemptyset(&sigset) == -1) ||
        (sigaddset(&sigset, signum) == -1) ||
        (sigprocmask(SIG_BLOCK, &sigset, NULL) == -1))
        perror("Failed to block signals before sigwait");
    fprintf(stderr, "This process has ID %ld\n", (long)getpid());
    for (; ; ) {
        if (sigwait(&sigset, &signo) == -1) {
            perror("Failed to wait using sigwait");
            return 1;
        }
        signalcount++;
        fprintf(stderr, "Number of signals so far: %d\n", signalcount);
    }
}
```
Signal Disposition on Program Start-up

- When program starts ("is execed") the 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 &`

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

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:

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

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

```c
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 multithreaded programs:
  - Dedicate particular threads to signal handling
    - Simpler to localize
    - Simpler to control the priority and scheduling of signals.
    - etc.