Signals and Timers
Introduction to Signals

Signal

- Notification to a process of an event
  - Interrupt whatever I was doing, and jump to signal handler

- Enables Coordination of asynchronous events
  - Email message arrives on my machine
    - Mailing agent (user) process should retrieve it
  - Invalid memory access happens
    - OS should inform scheduler to remove process from the processor
  - Alarm clock goes off
    - Process which sets the alarm should catch it
Basic Signal Concepts

- **Generation**
  - The time the event that causes the signal occurs

- **Delivery**
  - The time when a process receives the signal

- **Lifetime**
  - The interval between generation and delivery

- **Pending**
  - A signal that is generated but not delivered

- **Catch**
  - A process catches a signal if it executes a signal handler when the signal is delivered
  - Alternatively, a process can ignore a signal when it is delivered

- **Block**
  - A process can temporarily prevent a signal from being delivered by blocking it

- **Signal Mask**
  - The set of signals currently blocked
Generating Signals

- **Symbolic name**
  - Starting with SIG
  - Signal names are defined in `<signal.h>`

- **Users generated signals**
  - e.g., `SIGUSR1`

- **OS generated signals**
  - e.g., `SIGSEGV` – invalid memory reference

- **System call generated signals**
  - e.g., `SIGALRM` – alarm
## Some POSIX Required Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>Default action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>abort process</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>alarm clock</td>
<td>abnormal termination</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>access undefined part of memory</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>child terminated, stopped or continued</td>
<td>ignore</td>
</tr>
<tr>
<td>SIGILL</td>
<td>invalid hardware instruction</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGINT</td>
<td>interactive attention signal (usually ctrl-C)</td>
<td>abnormal termination</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>terminated (cannot be caught or ignored)</td>
<td>abnormal termination</td>
</tr>
</tbody>
</table>
### Some POSIX Required Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>Default action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGSEGV</td>
<td>Invalid memory reference</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>Execution stopped</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>termination</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>Terminal stop</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>Background process attempting read</td>
<td>stop</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>Background process attempting write</td>
<td>stop</td>
</tr>
<tr>
<td>SIGURG</td>
<td>High bandwidth data available on socket</td>
<td>ignore</td>
</tr>
</tbody>
</table>
How Signals Work

Signal Mask → Signal Generated → Signal delivered
if signal not blocked by signal mask...
Signal Caught by handler → Signal Handler
Return from Signal Handler
Signal Mask → Process Resumes

Copyright ©: University of Illinois CS 241 Staff
A little puzzle

- Signals can be seen as a kind of interprocess communication
- What’s the difference between signals and, say, pipes or shared memory?
  - Asynchronous notification
  - Doesn’t send a “message” as such; just a signal number
  - Puzzle: Then how could I do this?
Signaling

1. Generate a signal
2. Kernel representation
3. Deliver signal

Most signals
KILL, STOP

CHLD, SEGV, ...

Process 1

Process 2

2. Kernel representation
Signaling

1. Generate a signal
2. Kernel representation
3. Deliver signal

Process 1

Process 2

Kernel

Most signals
KILL, STOP

CHLD, SEGV, ...

1. Generate a signal
2. Kernel representation
3. Deliver signal
Generating a signal

- Generated by a process
  - System call `kill(pid, signal)`
    - Sends `signal` to process `pid`
    - Poorly named: sends any signal, not just `SIGKILL`

- Generated by the kernel, when...
  - a child process exits or stops (`SIGCHLD`)
  - floating point exception, e.g. div. by zero (`SIGFPE`)
  - bad memory access (`SIGSEGV`)
  - ...

Copyright ©: University of Illinois CS 241 Staff
Generating signals from the command line

- Signal a process from the command line
  - Use `kill`
  - `kill -l`
    - List the signals the system understands
  - `kill [-signal] pid`
    - Send signal to the process with ID `pid`
    - Optional argument may be a name or a number (default is `SIGTERM`).

- To unconditionally kill a process
  - `kill -9 pid` which is the same as
  - `kill -SIGKILL pid`
Generating signals in interactive terminal applications

- **CTRL-C** is **SIGINT**
  - Interactive attention signal
- **CTRL-Z** is **SIGSTOP**
  - Execution stopped – cannot be ignored
- **CTRL-Y** is **SIGCONT**
  - Execution continued if stopped
- **CTRL-\** is **SIGQUIT**
  - Interactive termination: core dump
A program can signal itself

- Similar to raising an exception
  - \texttt{raise(signal)} or
  - \texttt{kill(getpid(), signal)}
- Or can signal after a delay
  - \texttt{unsigned alarm(unsigned seconds)};
  - Calls are not stacked
    - any previously set \texttt{alarm()} is cancelled
  - \texttt{alarm(20)}
    - Send \texttt{SIGALRM} to calling process after 20 seconds
  - \texttt{alarm(0)}
    - cancels current alarm
A program can signal itself

Example: infinite loop ... for 10 seconds

```c
int main(void) {
    alarm(10);
    while(1);
}
```
Morbid example

```c
#include <stdlib.h>
#include <signal.h>
int main(int argc, char** argv) {
    while (1) {
        if (fork())
            sleep(30);
        else
            kill(getppid(), SIGKILL);
    }
}
```

- What does this do?
Signaling

1. Generate a signal
2. Kernel representation
   - KILL, STOP
   - Most signals
   - CHLD, SEGV, ...
3. Deliver signal

Process 1

Process 2

2. Kernel representation

Copyright ©: University of Illinois CS 241 Staff
Kernel representation

- A signal is related to a specific process
- In the process’s PCB, kernel stores
  - Set of pending signals
    - Generated but not yet delivered
  - Set of blocked signals
    - Will stay pending
    - Delivered after unblocked (if ever)
  - An action for each signal type
    - What to do to deliver the signal
Kernel signaling procedure

- Signal arrives
  - Set pending bit for this signal
    - Only one bit per signal type!
- Ready to be delivered
  - Pick a pending, non-blocked signal and execute the associated action—one of:
    - Ignore
    - Kill process
    - Execute signal handler specified by process
Signaling

1. Generate a signal
2. Kernel representation
3. Deliver signal

Process 1

Kernel

Most signals

CHLD, SEGV, ...

KILL, STOP

Process 2

Copyright ©: University of Illinois CS 241 Staff
Delivering a signal

- Kernel may handle it
  - `SIGSTOP`, `SIGKILL`
  - Target process can’t handle these
  - They are really messages to the kernel about a process, rather than to a process

- For most signals, target process handles it (if it wants)
If process handles the signal...
Signal mask

- Temporarily prevents select types of signals from being delivered
  - Implemented as a bit array
  - Same as kernel’s representation of pending and blocked signals

<table>
<thead>
<tr>
<th></th>
<th>SigInt</th>
<th>SigQuit</th>
<th>SigKill</th>
<th>…</th>
<th>SigCont</th>
<th>SigAbrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>…</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Signal mask example

- Block all signals

```c
sigset_t sigs;
sigfillset(&sigs);
sigprocmask(SIG_SETMASK, &sigs, NULL);
```

- See also
  - `sigemptyset`, `sigaddset`, `sigdelset`, `sigismember`
If it’s not masked, we handle it

- Three ways to handle
  - Ignore it
    - Different than blocking!
  - Kill process
  - Run specified signal handler function

- One of these is the default
  - Depends on signal type

- Tell the kernel what we want to do: `signal()` or `sigaction()`
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

- Change the action taken by a process on receipt of a specific signal
- Notes
  - Any valid signal except SIGKILL and SIGSTOP
  - If act is non-null, new action is installed from act
  - If oldact is non-null, previous action is saved in oldact
  - Any
#include <stdio.h>
#include <signal.h>

void handle(int sig) {
    char handmsg[] = "Ha! Blocked!\n";
    int msglen = sizeof(handmsg);
    write(2, handmsg, msglen);
}
Example: Catch control-c

```c
int main(int argc, char** argv) {
    struct sigaction sa;
    sa.sa_handler = handle;
    sa.sa_flags = 0;

    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);
    while (1) {
        printf("Fish.\n");
        sleep(1);
    }
}
```

Note: Need to check for error conditions in all these system & library calls!
Potentially unexpected behavior

- Only one pending signal of each type at a time
  - If another arrives, it is lost
- What’s an interesting thing that could happen during a signal handler?
  - Another signal arrives!
  - Need to either
    - Write code that does not assume mutual exclusion (man `sigaction`), or
    - Block signals during signal handler (`signal()` and `sigaction()` can do this for you)
How to catch without catching

- Can wait for a signal
  - No longer an asynchronous event, so no handler!
- First block all signals
- Then call `sigsuspend()` or `sigwait()`
  - Atomically unblocks signals and waits until signal occurs
  - Looks a lot like condition variables, eh?
And now back to the puzzle...

- Can we support arbitrary communication between processes using only signals?

**Idea**
- Even with two signals, we can get 1 bit of information from receipt of a signal....