CSci 4061
Introduction to Operating Systems
(Advanced Control—Signals)
What is a Signal?

- Signals are a form of *asynchronous* IPC
  - Earlier: Non-blocking I/O and check if it has happened => polling
  - Problem with polling?
- Signals are used by the kernel to deal with severe errors
  - `badprog.c` => `char *c; c=0x10; *c= 'a')`
  - Memory-error, core dumped
  - `SIGSEGV` sent to `badprog`; immediate “tap” on the shoulder
What is a signal?

- Think of it as a software interrupt

**Unusual conditions**, rather than “normal IPC”
Often it is the OS communicating to the process!
Signal (cont’d)

Signals can also be sent from process to process

- `ps` (to get a list of pids)
- `kill -9 1235` (sends SIGKILL to the process)
- *weird*: the `-9` is the SIGKILL, the `kill` command sends any signal
Another Example

> ./mybigprog (hit ctrl-C, interrupt)

What happens?

- CPU is running mybigproc and gets keyboard interrupt
- OS receives interrupt (^C)
- OS knows it occurred in the terminal running mybigproc
- OS sends SIGINT to mybigproc
  
- Default action: SIGINT causes process to terminate
Default Signal Options

Nothing, death (w or w/o core image)
Stopped (blocked—SIGSTOP)
Continued (unblocked—SIGCONT)

What is default action for SIGCHLD?

signal sent from child to parent when it exits
Signal Concepts

You can override the default action for *most* signals; for those:

• If you ask to **ignore** a signal, it has no effect
• If you **block** a signal, it remains pending until you unblock it
• You can **catch** a signal by specifying a handler ~ ignore except you do something specific ....

• **SIGKILL, SIGSTOP, SIGCONT** cannot be caught, blocked, or ignored
Some Signals

• Signals carry no other info besides their name, examples:
  SIGIO
    • I/O completion (non-blocking I/O)
  SIGCHLD
    • Child exit (use instead of wait ())
  SIGALRM
    • Timer expired
  SIGFPE, SIGPIPE
  SIGUSR1, SIGUSR2: not used by the kernel
Sending signals

• `ps -a` (to get a list of pids)
  • `kill -s SIGINT 3423` (SIGKILL is the default)
Sending signals

• In code:

```c
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pd, int signal);
```

- `pid > 0` => process
- `pid = 1` => every process except `init`
- `pid = 0` => process group, `getgrp`

Process group = set of processes bound to same terminal

E.g. parent – children in the same process group
Sending Signals (cont’d)

• `kill (7421, SIGSTOP);`
• Limitations?
• `user must own the process (i.e. be the effective user_id)`
Signal Handling

• When process gets a signal: multiple choices
  • Default
  • Ignore/block signals (protect against ^C)
  • Take specific action

• How does a process do any of this?
Signal Details

Steps to dealing with signals

1. Identify signal(s) of interest
   - Signals you wish to deal with (i.e. don’t want default)
   - In some cases, you may want to treat signals as a set
   - Ignore/block signals (protect against ^C)
   - Take specific action

2. Decide how to deal with them
   a) {Un}Block signals
   b) Handle/ignore signals
Signal Sets

1) 
#include <signal.h>
sigset_t set:

int sigemptyset (sigset_t *s);
int sigfillset (sigset_t *s);
int sig(add/del)set (sigset_t *s, int signo);

sigemptyset (&set);
sigaddset (&set, SIGINT);
// or
sigfillset (&set);
sigdelset (&set, SIGCHLD);
Blocking Sets of Signals

2a)

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

sigset_t set;
sigfillset (&set);
sigprocmask (SIG_SETMASK, &set, NULL); // blk signals in set

sigprocmask (SIG_UNBLOCK, &set, NULL); // unblock set

Cannot block SIGKILL, SIGSTOP
```
Signal Action Handling

2b)

```c
int sigaction (int signo, const struct sigaction *act, ...*oact);
```

- `act` contains:
  - The action *signal handler* (function to call): `sa_handler`
  - Signals to *mask/block* while handler is executing (signal set): `sa_mask`
  - Some flags to control behavior

```c
void catchint (int signo) { // no other info!
    fprintf (stderr, "catching signal = %d/n", signo);
    ...
}
```

Sort of like an OS callback!
#include <signal.h>

void main ( ) {
    struct sigaction act;
    act.sa_handler = catchint;
    sigfillset (&act.sa_mask);

    sigaction (SIGINT, &act, NULL);
    //SIGINT will not term. process
    // ^c will be caught (after handler setup)
    ...
}

Cannot catch all signals...
Signal Action Handling (cont’d)

Blocking queues the signals for possibly later delivery
May want to ignore the signal all together

```c
void main () {
    struct sigaction act;
    act.sa_handler = SIGIGN;
    sigaction (SIGINT, &act, NULL);
    //SIGINT will not term. Process
    ...
    // ^c will be ignored
}
```
Why Block Signals?

- Usually want handler to run without interruption
- <example>
Race conditions

• Execution outcome depends on timing of events
  • => some timings cause program to break
• <example>
What can I do in a hander?

- Ideally: limited
- Cleanup: remove memory, files, before shutting down
- Can do some actually programming
  - If you change global variable values (CAREFUL)
Interrupt-driven Programs

Suppose I want my programs to wait for certain signals \texttt{SIGIO} (I/O or network packets), \texttt{SIGINT}, etc.

\texttt{SIGIO} has to be enabled

\texttt{fcntl (fd, F_SETFL, flags | O_ASYNC);}
Interrupt-driven Programs

I want to block until the signals come (no busy waiting)

```c
#include <unistd.h>
int pause(); //block until I receive a signal

//set up signal masks/handlers
sigaction (...);
while (...){
    ...
    pause ();
    ...
```
Interrupted System Calls

Signals can safely interrupt most system calls

- The system calls return **EINTR** (error code)

- Just re-run the call: some systems do this automatically (Linux) but not all

- Slow blocking calls like `read`, `write` can be interrupted

```c
while (retval = read (fd, buf, size),
     retval == -1 && errno = EINTR);
```
Signals as application exceptions

Sending a signal to yourself

```c
int raise (int sig);
void SIGhandler (int sig)
{
    printf("\n OVERFLOW. Closest answer is %ld! = %ld\n", i-1, prev_fact);
    exit(0);
}
void main ()
{
    long  fact;
    printf("Factorial Computation:\n\n");
signal(SIGUSR1, SIGhandler); /* install SIGUSR1 handler */
for (prev_fact = i = 1; ; i++, prev_fact = fact) {
    fact = prev_fact * i;      /* computing factorial */
    if (fact < 0)              /* if the results wraps around */
        raise(SIGUSR1);       /* we have overflow, print it */
    else if (i % 3 == 0)       /* otherwise, print the value */
        printf("     %ld! = %ld\n", i, fact);
}
}       
<example: raise>
```
Alarms

Sending a signal (SIGALRM) to yourself in the future

```c
int alarm (int secs);  // does not block
```

// must define a handler for this to work
```c
void alarm_handler (int sig) {
    fprintf (stderr, “IN ALARM HANDLER\n”);
}
```

```c
static struct sigaction act;
act.sa_handler = alarm_handler;
sigfillset (&act.sa_mask);
sigaction (SIGALRM, &act, NULL);
alarm (60);
...
```
Alarms (cont’d)

• Suppose we want an alarm—EVERY k time units
  • Could keep calling `alarm (k)`?
    • `<run alarm.c>`
  • Expensive: lots of system calls and
    • won’t work if k<< secs

• Instead we can set up a timer
Alarms

Sending a signal `SIGALRM` in the future

```c
int alarm (int secs); // does not block
```

If we want an alarm, EVERY k time units

- Could keep calling `alarm (k)`
- Expensive: lots of system calls and
  - won’t work if k<< secs
Alarms (cont’d)

```c
void main (){ 
    struct itimerval interval;
    struct sigactin act;
    act.sa_handler = alarm_handler;

    sigaction (SIGALRM, &act, NULL);
    interval.it_interval.tv_sec = 1; // time to first int
    interval.it_interval.tv_usec = 0;
    interval.it_value.tv_sec = 0;     // value to reload
    interval.it_value.tv_usec = 100;
    setitimer (ITIMER_REAL, &intervals, NULL);}
while (1);}

//will send SIGALRMS every 100 usec
```
Signal Concerns

- Talked about races
- Talk about a related issue ...
Reentrancy
Signal Wrap-Up

• Concerns (similar concerns for threads):
  • **race conditions**: if handler accesses global variables (also accessed in the program) can lead to errors
    • sharing \( x \ldots (x=0; \ldots y = 2/x); \)
  • **non-reentrancy**: multiple invocations of the handler can cause problems

• **when/why does this happen?**
  • **non-local state**: globals, statics, file ptrs
Signal Advice

Signal handlers
  Keep ‘em short

Be aware of potential race conditions

Be aware of non-reentrancy