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

Signals are a form of *asynchronous* IPC

- Thus far asynchrony has been passive: check if something happened
- Signals are used by the kernel to deal with severe errors
- **Badprog** *(badprog.c=>char*c; c=0x10; *c= ‘a’)*
- Memory-error, core dumped
- **SIGSEGV** sent to badprog; immediate “tap” on the shoulder

Signals can also be sent from process to process

- **ps** *(to get a list of pids)*
- **kill** −9 1235 *(sends SIGKILL to the process)*
Another Example

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

What happens?

- CPU is running mybigproc and get hardware (keybd) interrupt
- OS receives interrupt (^C)
- Knows it occurred in the terminal running mybigproc
- OS sends SIGINT to mybigproc

• Default action: SIGINT causes process to terminate
Signals

• Unusual events not for standard IPC

• Also handy for timed events

• Default action: death or nothing
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 ...

- *SIGKILL, SIGSTOP* cannot be caught, blocked or ignored
Signal Options

Default action:
Nothing, death (w or w/o core image)
Stopped (blocked - SIGSTOP)
Continued (unblocked - SIGCONT)
Signal Information

Do signals always mean death?
No! A process can:

1) Choose to “handle” the signal
2) Choose to “block” the signal
3) Choose to “ignore” the signal
4) Some signals are ignored by default

Yes, if SIGKILL

SIGSTOP is unique: stops (does not kill) the process by default (cannot be handled or blocked)

Unusual conditions, rather than “normal IPC”
Often it is the OS communicating to the process!
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 1235` (sends default signal `SIGKILL` to the process)
  - `kill -s SIGINT 3423`

- In code:
  ```c
  #include <sys/types.h>
  #include <signal.h>
  int kill (pid_t pd, int signal);
  ```
  - `pid > 0` => process
  - `pid = 0` => process group, `getgrp`
  - `pid = 1` => every process except `init`

- `kill (7421,SIGSTOP);`
- Limitations?
- User must own the process

---

Process group = set of processes bound to same terminal

Parent – child in the same process group
Signal Handling

• When process gets a signal: 3 choices
  • Default action (most cases will cause termination)
  • Ignore/block signals (protect against ^C)
  • Take specific action

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

Steps to using 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);
```

```c
sigprocmask (SIG_SETMASK, &set NULL); // blk signals in set
```

```c
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)
- Signals to *mask/block* while handler is executing (signal set)
- Some flags to control behavior

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

Cannot catch all signals...
<flow of control>
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
}
```

<run example>
Interrupt-driven Programs

Suppose I want my programs to wait for certain signals SIGIO (network packets), SIGINT, etc.

But I want to block until the signals come (no busy waiting)
#include <unistd.h>
int pause(); //block until I receive a signal

//set up signal masks/handlers
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
Alarms

Sending a signal to yourself

```c
int raise (int sig);
```

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 \texttt{alarm} \((k)\)?
  • won’t work if \( k \ll \text{secs} \)

• Instead we can set up a timer
Void main (){
    struct itimerval interval;
    struct sigactin act;
    act.sa_handler = alarm_handler;

    sigaction (SIGALRM, &act, NULL);
    interval.it_value.tv_sec = 0; // time to next exp
    interval.it_value.tv_usec = 100;
    interval.it_value.tv_sec = 0; // valuer to reload
    interval.it_value.tv_usec = 100;
    setitimer (ITIMER_REAL, &intervals, NULL);}
while (1);}

//send SIGALRMS
Signal Advice

Signal handlers
  Keep ‘em short

Be aware of potential conditions
Re-entrancy