# Signals (continued)
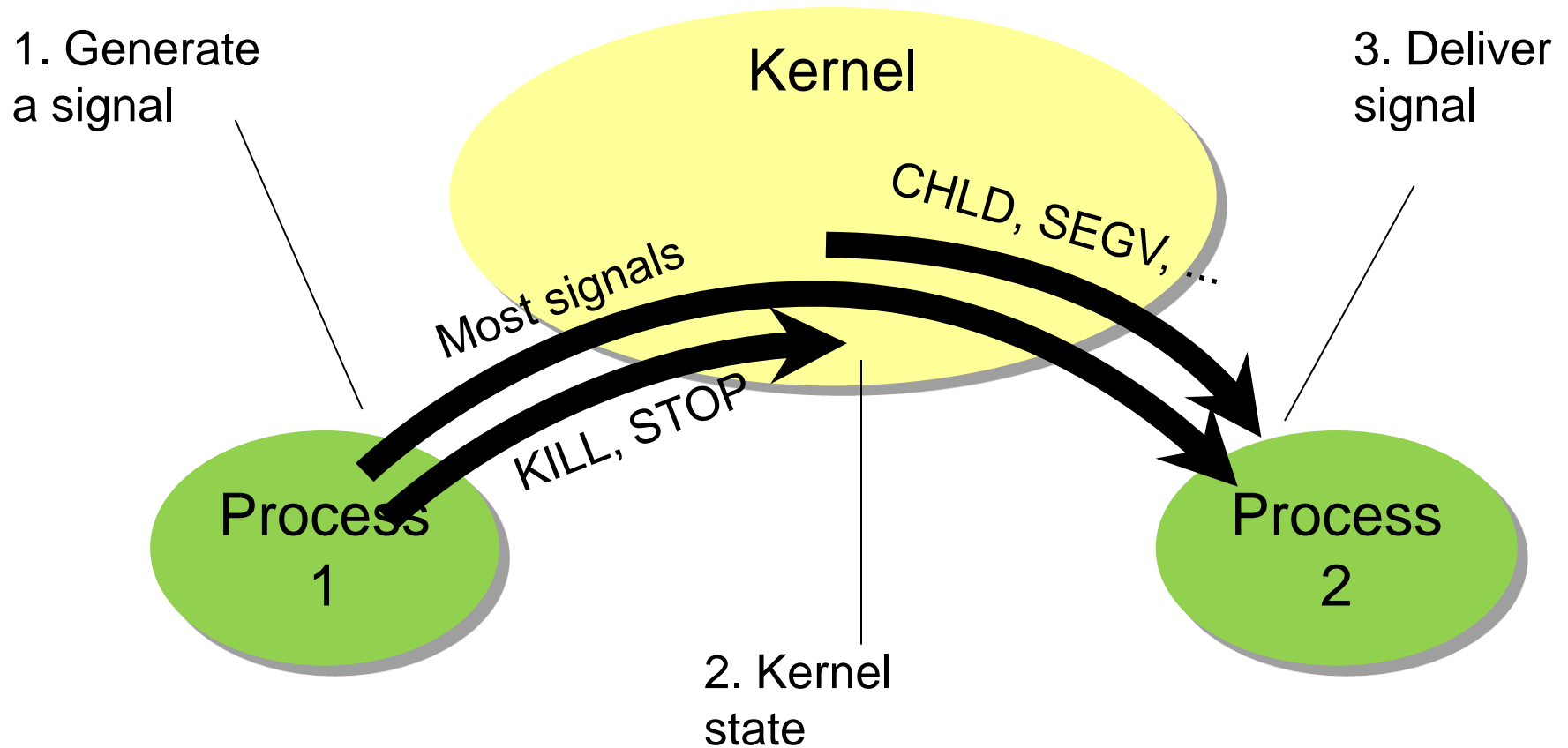
CS 241

April 9, 2012

University of Illinois

# Signals

A signal is an asynchronous notification of an event

- Asynchronous: could occur at any time
- Interrupts receiving process; jumps to signal handler in that process
- A (limited) menu of event types to pick from

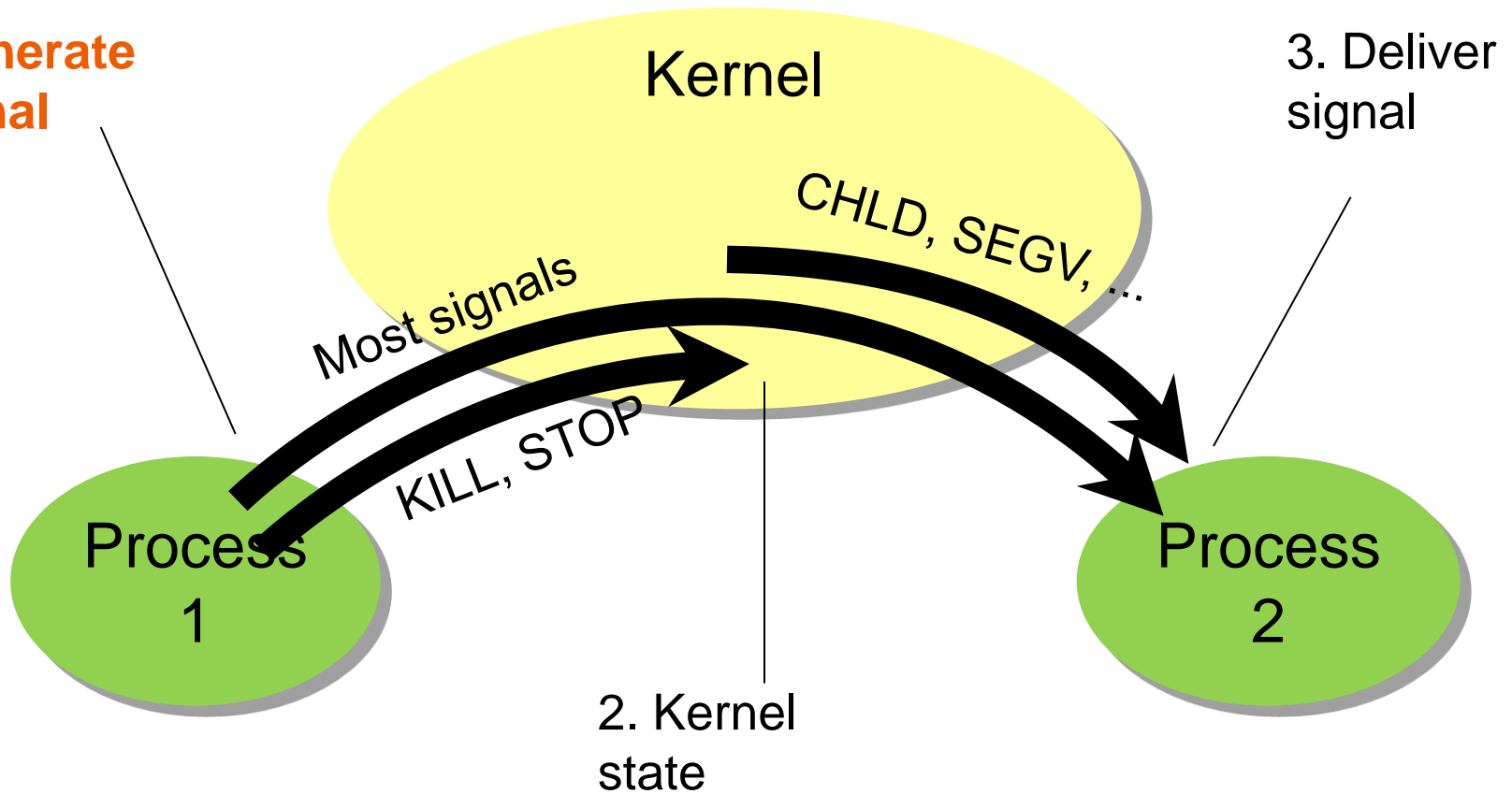What events could be asynchronous?

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

# Signaling overview



1. Generate a signal

Kernel

CHLD, SEGV, ...

Most signals

KILL, STOP

3. Deliver signal

Process 1

Process 2

2. Kernel state

# Signaling overview



1. **Generate a signal**

Kernel

3. Deliver signal

Most signals

CHLD, SEGV, ...

KILL, STOP

Process 1

Process 2

2. Kernel state

# Generating a signal

Generated by a process with syscall 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 is stops (SIGCHLD)
- floating point exception, e.g. div. by zero (SIGFPE)
- bad memory access (SIGSEGV)
- ...

# **Signals from the command line: kill**

kill -l

- Lists the signals the system understands

kill [-signal] pid

- Sends signal to the process with ID pid
- Optional argument signal may be a name or a number (default is SIGTERM)

kill -9 pid or kill -KILL pid or kill -SIGKILL pid

- Unconditionally terminates process pid

# Signals in the interactive terminal

Control-C is SIGINT

- Interactive attention signal

Control-Z is SIGSTOP

- Execution stopped – cannot be ignored

Control-Y is SIGCONT

- Execution continued if stopped

Control-\ is SIGQUIT

- Interactive termination: core dump

# A program can signal itself

Similar to raising an exception
- ○ `raise(signal)` or
- ○ `kill(getpid(), signal)`

Or can signal after a delay
- ○ `unsigned alarm(unsigned seconds);`
- ○ Calls are not stacked
  - ■ any previously set `alarm()` is cancelled
- ○ `alarm(20)`
  - ■ Send `SIGALRM` to calling process after 20 seconds
- ○ `alarm(0)`
  - ■ cancels current alarm

# Example: What does this do?

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

Example of program signaling itself

"Infinite" loop for 10 seconds

Then interrupted by alarm

- Doesn't matter that while loop is still looping
- No signal handler set by program; default action: terminate

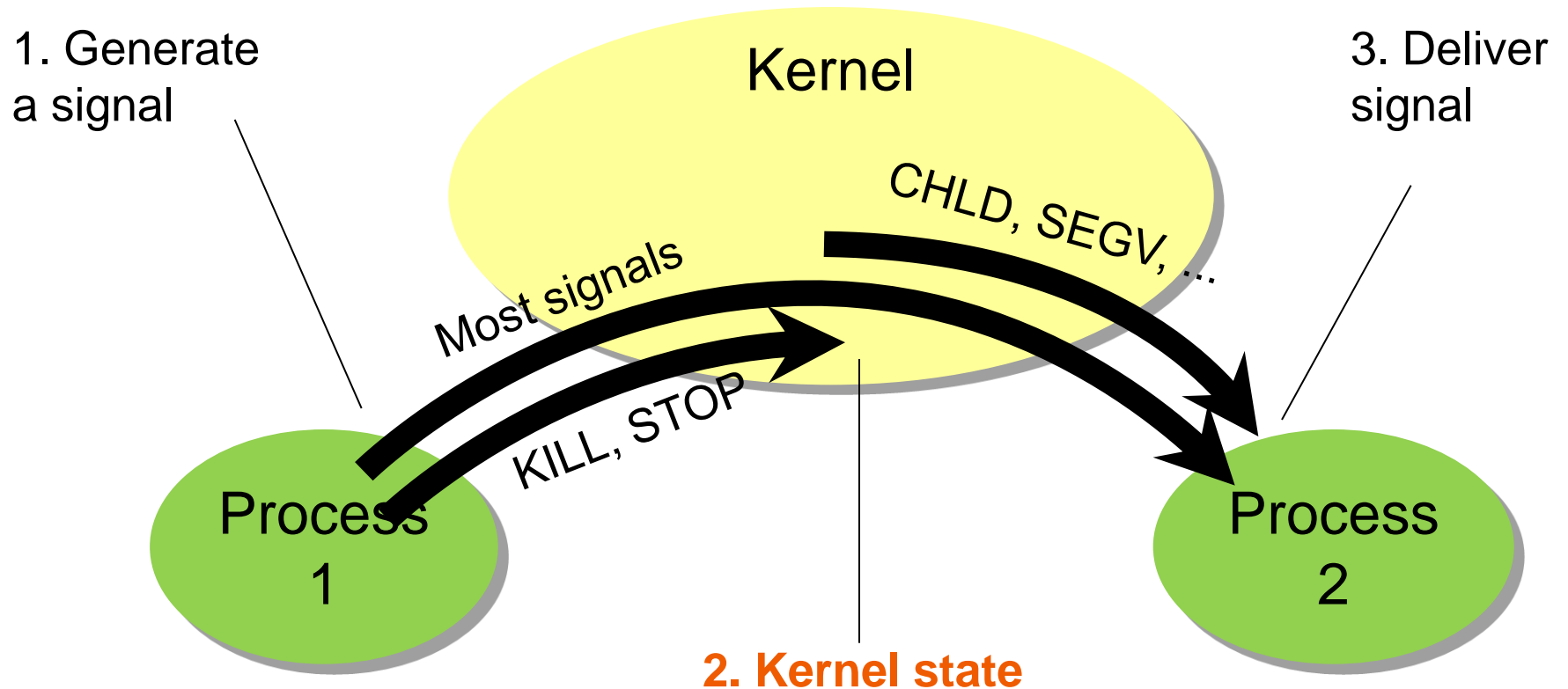# Morbid example

```
#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 overview



1. Generate a signal

Kernel

3. Deliver signal

CHLD, SEGV, …

Most signals

KILL, STOP

Process 1

Process 2

**2. Kernel state**

# Kernel state

A signal is related to a specific process

In the process's PCB (process control block), 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
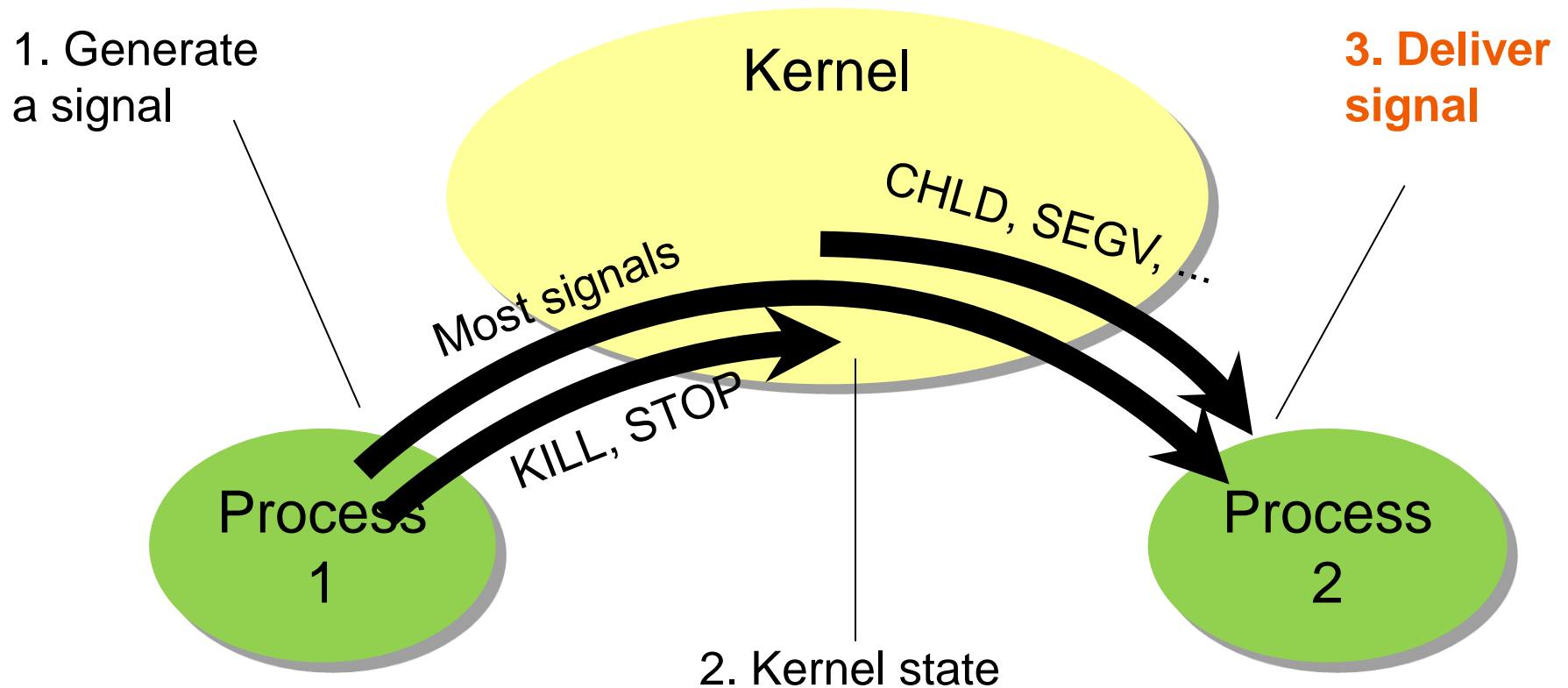  - What to do to deliver the signal

# 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
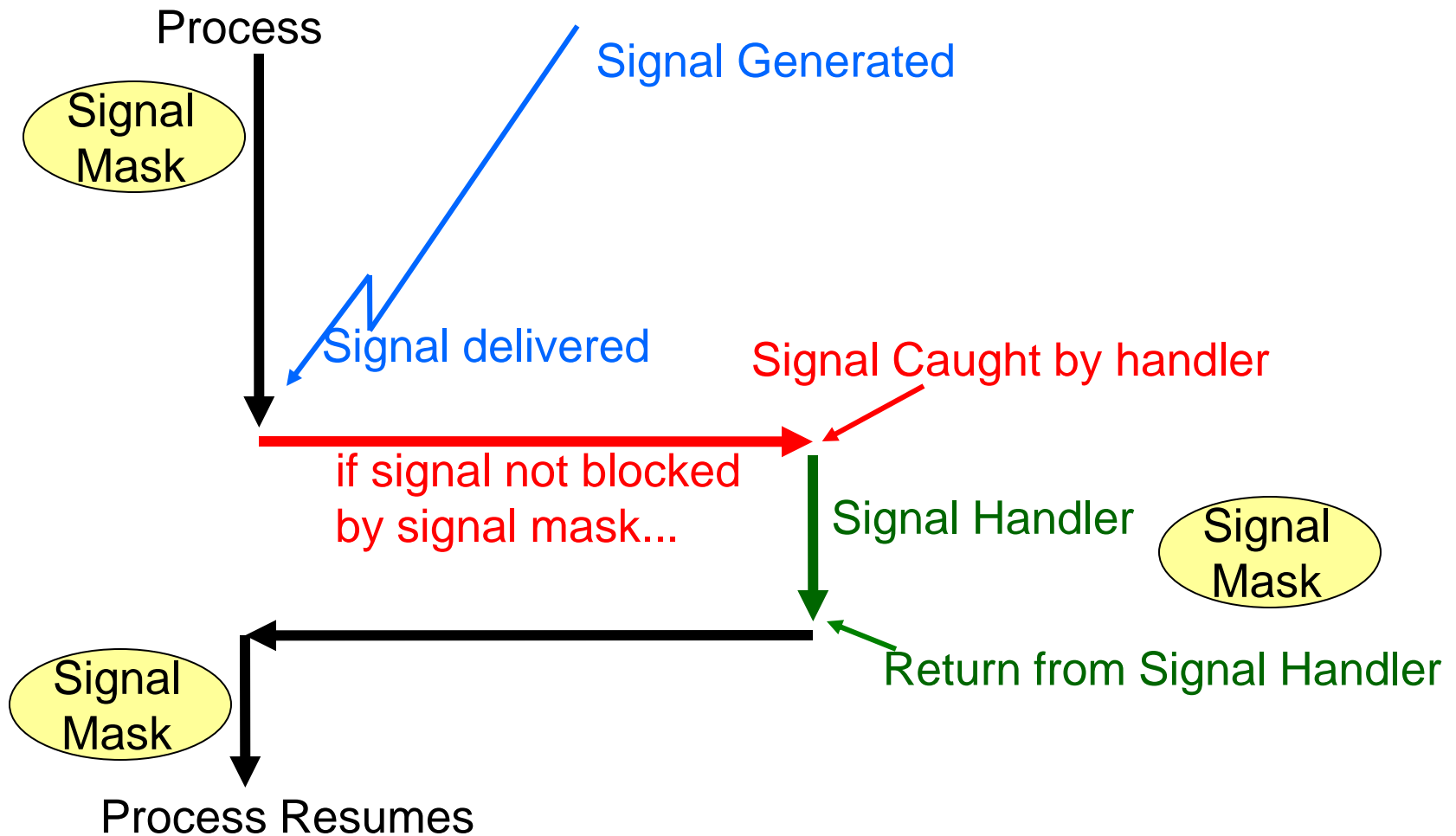
# Signaling overview

# Delivering a signal

Kernel may handle it

- Not delivered to target program at all!
- SIGSTOP, SIGKILL
- Target process can't handle these
- They are really messages to the kernel about a process, rather than messages to a process

But 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

| SigInt | SigQuit | SigKill | … | SigCont | SigAbrt |
|--------|---------|---------|-----|---------|---------|
| 1 | 0 | 1 | … | 1 | 0 |

# Signal mask example

Block all signals:

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

Instead of sigfillset, you might try:

- 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()

# sigaction

**#include <signal.h>**

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

Changes the action taken by a process when it receives a specific signal

Notes

- signum is 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

# Example: Catch SIGINT

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

void handle(int sig) {
    char handmsg[] = "Ha! Handled!!!\n";
    int msglen = sizeof(handmsg);
    write(2, handmsg, msglen);
}

int main(int argc, char** argv) {
    struct sigaction sa;
    sa.sa_handler = handle;   /* the handler function!! */
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask); /* block all signals during handler */

    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

Inside kernel, only one pending signal of each type at a time

- If another arrives while first one still pending, second 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, 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?
  - cond_wait() unlocks mutex and waits till condition occurs

# Puzzle:
# Using signals to send
# a stream of data