Signals In Depth

CS 241

April 14, 2014

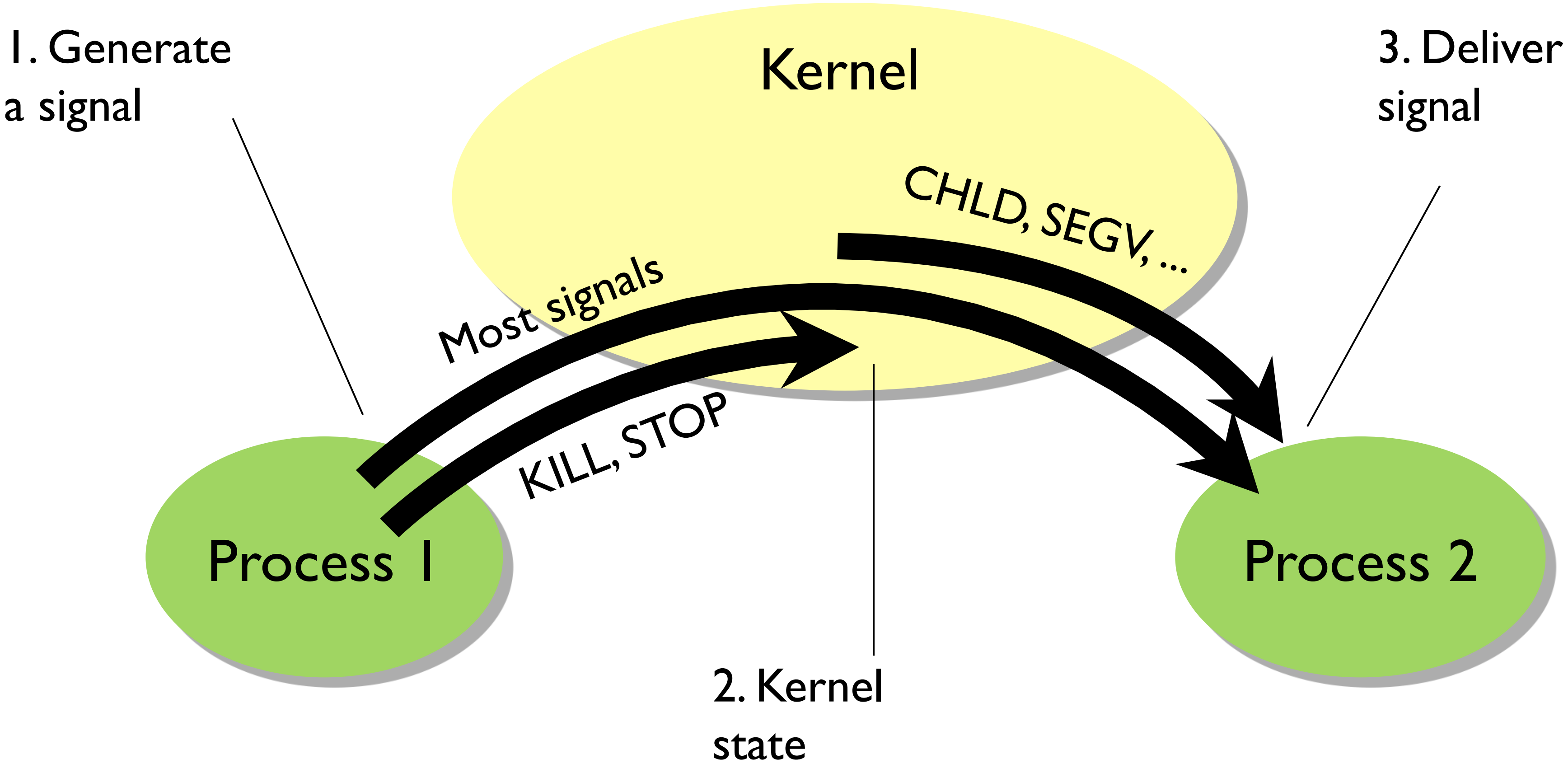
University of Illinois

Announcements

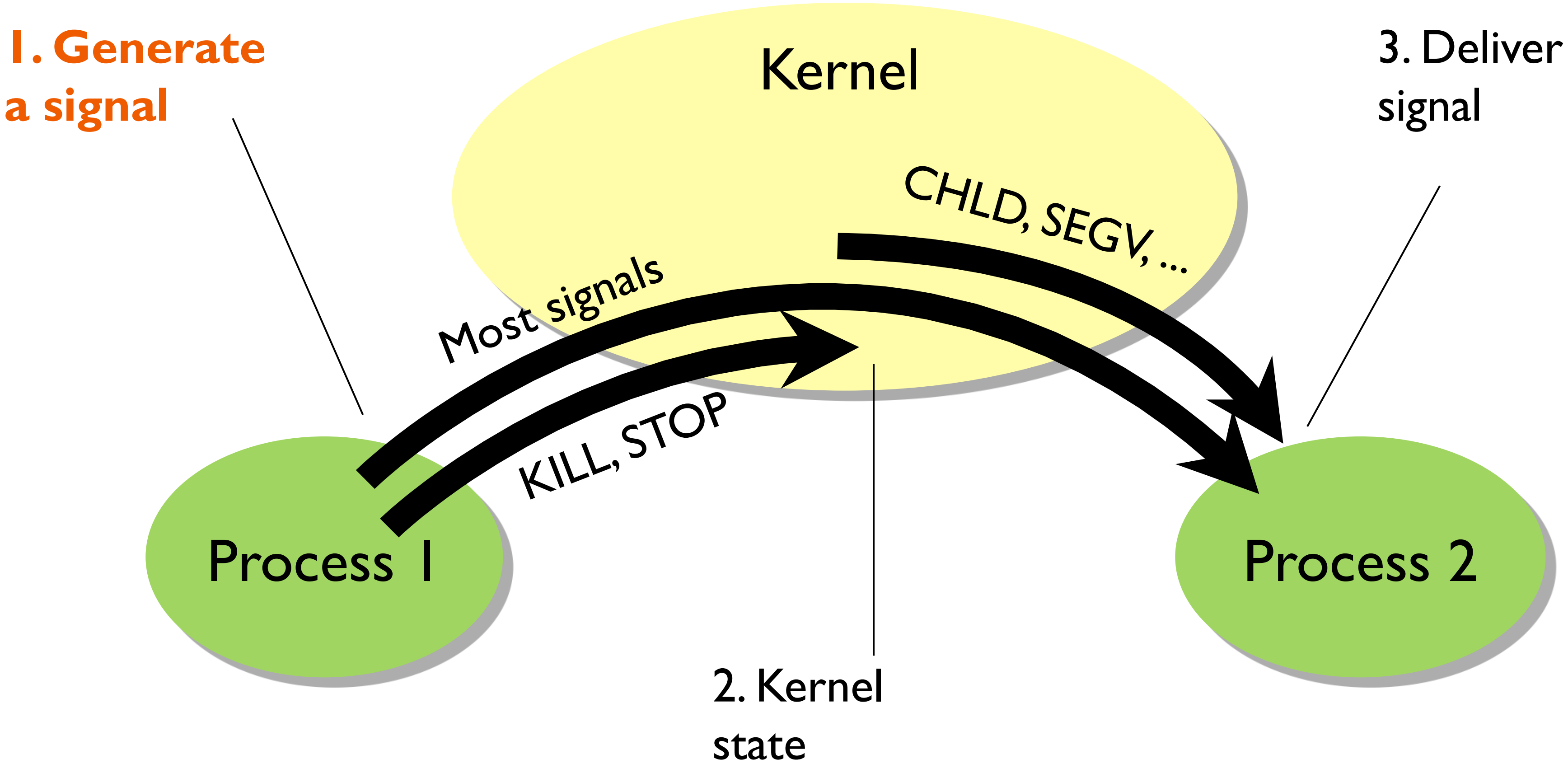
Pebble pickup for those who didn’t already

- Right after class today

Signaling overview



Signaling overview



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?

```
int main(void) {
    alarm(5);
    while(1);
}
```

Example of program signaling itself

“Infinite” loop for 5 seconds

Then interrupted by alarm

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

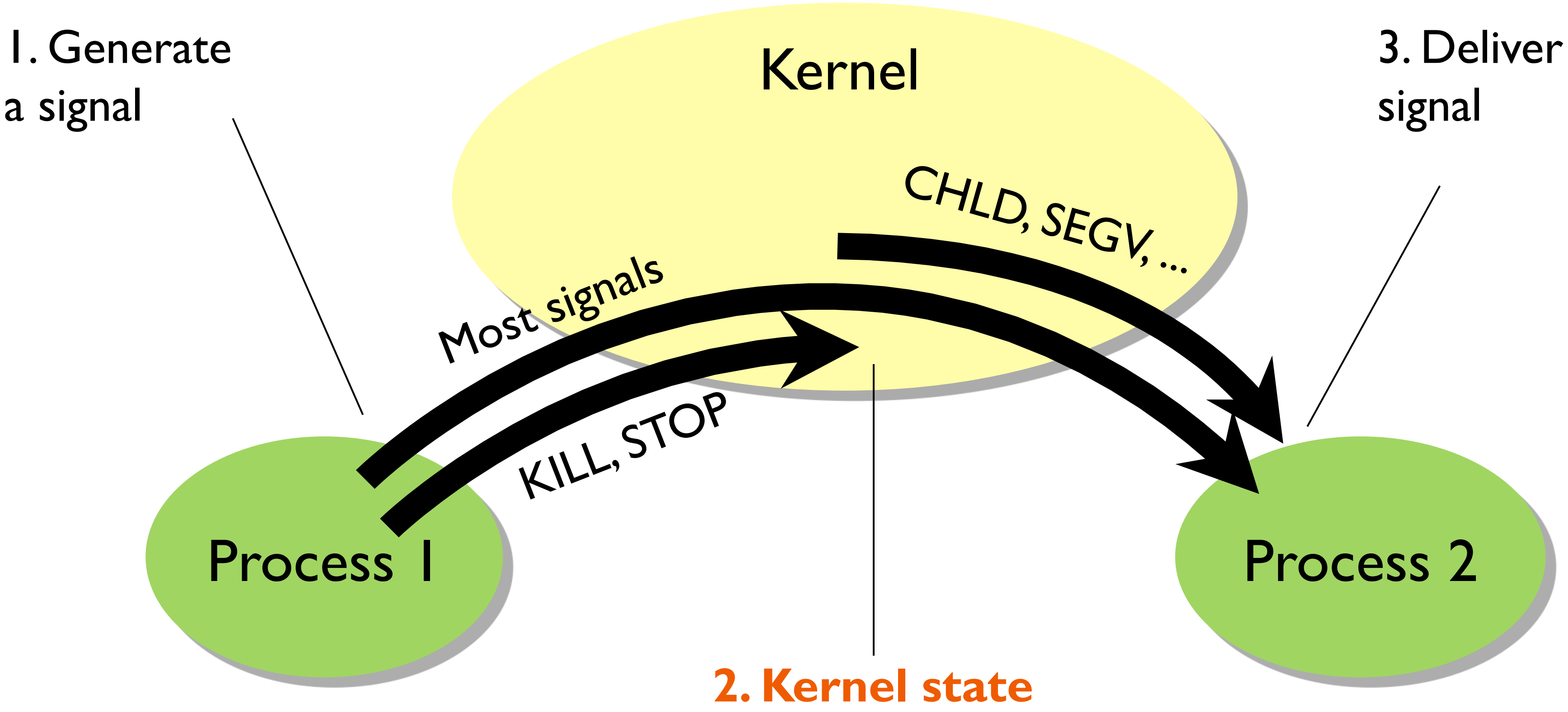
Morbid example

```
#include <stdlib.h>
#include <signal.h>

int main(int argc, char** argv) {
    if (fork())
        sleep(30);
    else
        kill(getppid(), SIGKILL);
}
```

What does this do?

Signaling overview



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
  - How to deliver the signal

Kernel signaling procedure

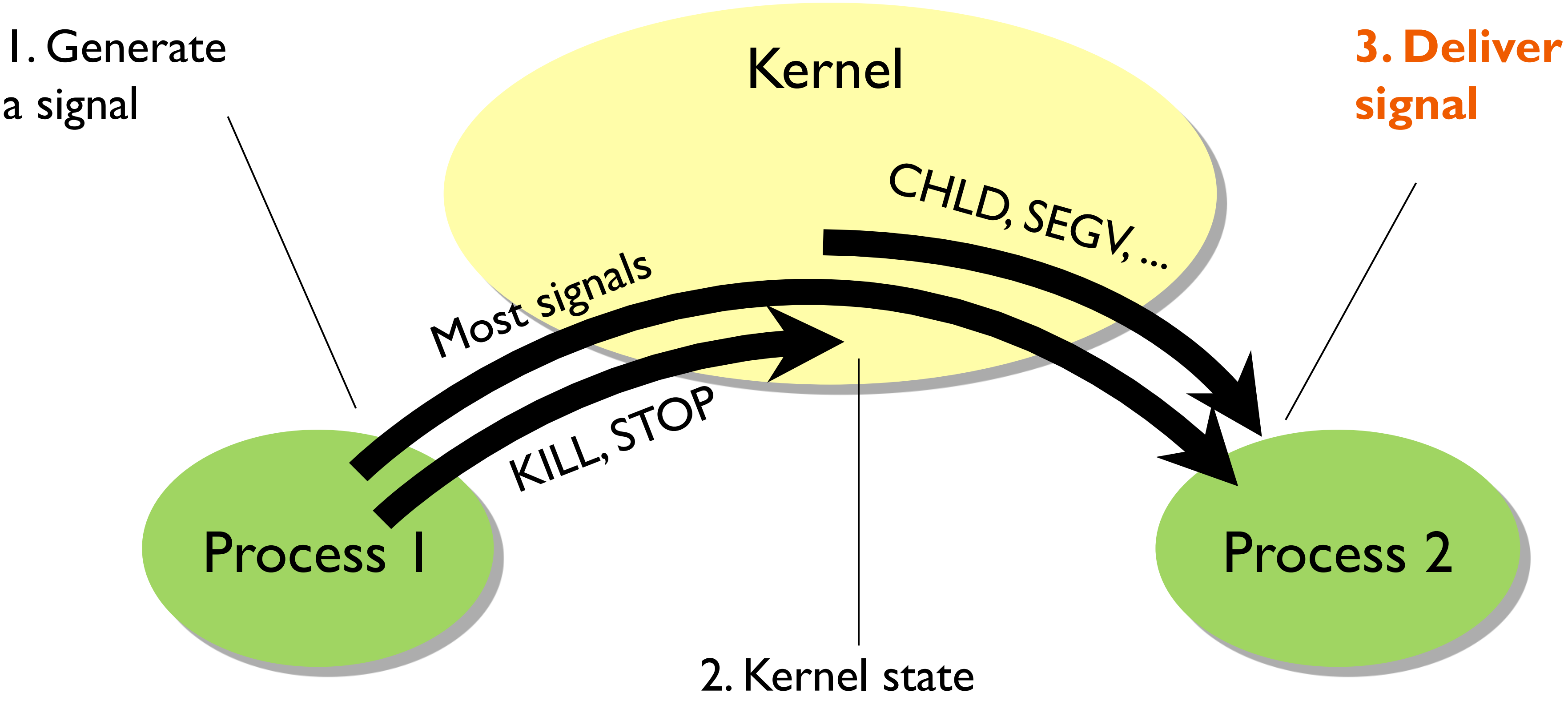
When signal arrives

- Set pending bit for this signal
- Only one bit per signal type!
- Wait until ready to be delivered (not blocked)

When 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



1. Generate a signal

Kernel

3. Deliver signal

CHLD, SEGV, ...

Most signals

KILL, STOP

Process 1

Process 2

2. Kernel state

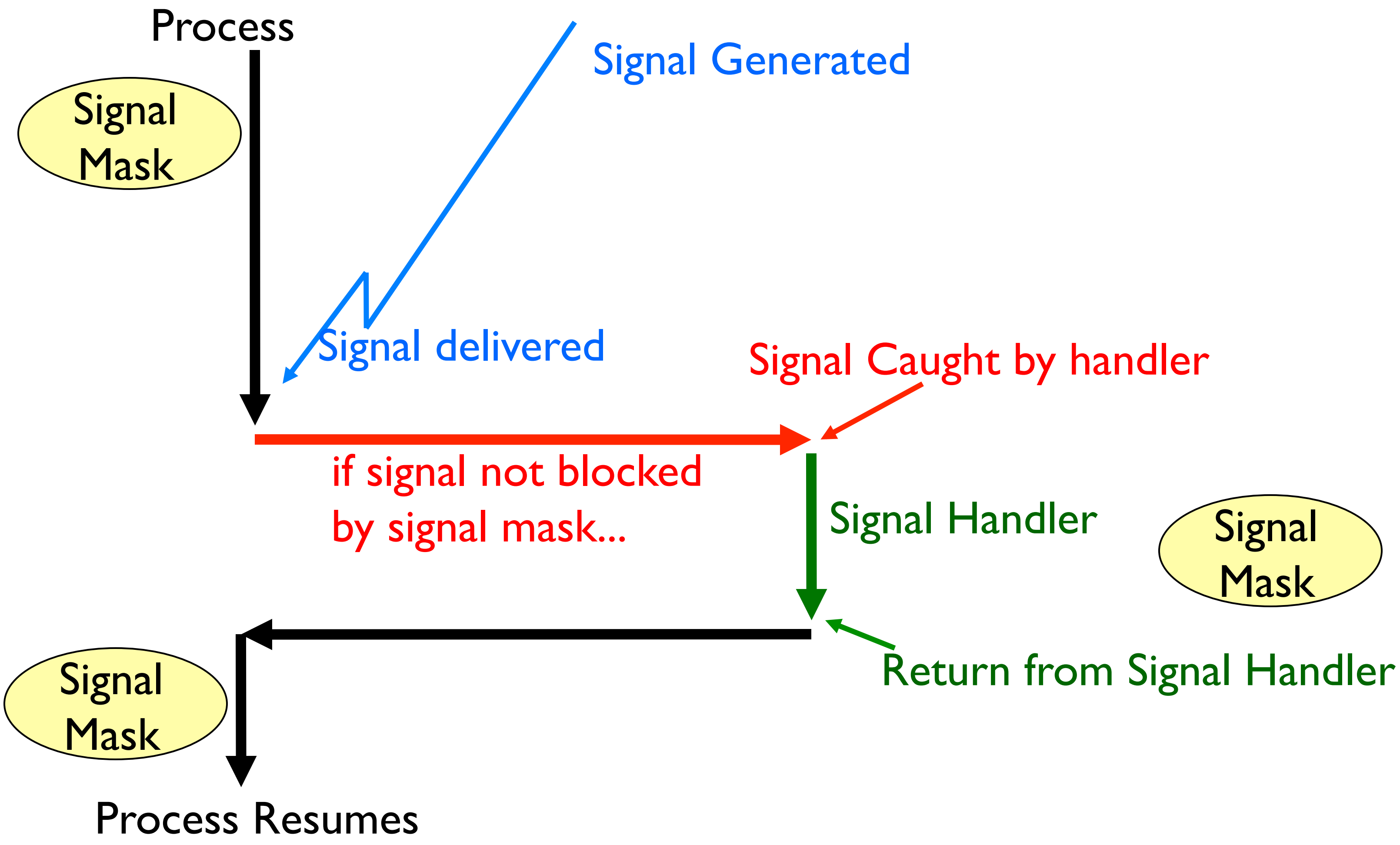
Delivering a signal

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`

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

Or, How To Completely Abuse Signaling Functionality In Order To
illustrate potentially unexpected behavior in signals,
illustrate that in the end, everything's just bits, and
pull off epic systems hackery

Puzzle

Can we support arbitrary communication between processes using only signals?

How would we transmit one bit of information using signals?

How can we build from there into a stream of data?

Puzzle solution attempt

```c
int main(int argc, char** argv) {
    pid_t    friend;
    sigset_t signals_to_mask;

    printf("I'm process %d.  Who should I talk to? ", getpid());
    scanf("%d", &friend);

    if (!strcmp(argv[1], "read")) {
        sigfillset(&signals_to_mask);
        sigprocmask(SIG_SETMASK, &signals_to_mask, NULL);
        while (1) {
            putchar(recv_char());
            fflush(stdout);
        }
    }

    else
        while (1)
            send_char(friend, getchar());
}
```

Reader

Writer

Puzzle solution attempt

```
int main(int argc, char** argv) {
    pid_t    friend;
    sigset_t signals_to_mask;

    printf("I'm process %d.  Who should I talk to? ", getpid());
    scanf("%d", &friend);

    if (!strcmp(argv[1], "read")) {
        sigfillset(&signals_to_mask);
        sigprocmask(SIG_SETMASK, &signals_to_mask, NULL);
        while (1) {
            putchar(recv_char());
            fflush(stdout);
        }
    }

    else
        while (1)
            send_char(friend, getchar());
}
```

Block signals so we can use sigwait()

Puzzle solution attempt

```
int main(int argc, char** argv) {
    pid_t    friend;
    sigset_t signals_to_mask;

    printf("I'm process %d.  Who should I talk to? ", getpid());
    scanf("%d", &friend);

    if (!strcmp(argv[1], "read")) {
        sigfillset(&signals_to_mask);
        sigprocmask(SIG_SETMASK, &signals_to_mask, NULL);
        while (1) {
            putchar(recv_char());
            fflush(stdout);
        }
    }

    else
        while (1)
            send_char(friend, getchar());
}
```

All the magic
happens in here

Solution attempt: sending

```
void send_bit(pid_t friend, int bit) {
    int signal = bit ? SIGUSR2 : SIGUSR1;
    kill(friend, signal);
}

void send_char(pid_t friend, char c) {
    int i;

    for (i = 0; i < 8; i++)
        send_bit(friend, c & (1 << i));
}
```

If bit is zero, send SIGUSR1

If bit is one, send SIGUSR2

Solution attempt: receiving

```
int recv_bit() {
    int sig;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    sigaddset(&set, SIGUSR2);

    sigwait(&set, &sig);

    return (sig == SIGUSR2) ? 1 : 0;
}

char recv_char() {
    int i;
    char c = 0;
    for (i = 0; i < 8; i++)
        c |= recv_bit() << i;
    return c;
}
```

Construct the set of signals to wait for. Too bad it takes 4 lines of code just to say “SIGUSR1 or SIGUSR2”!

Wait for either signal

Interpret received signal
SIGUSR2 as a 1
SIGUSR1 as a 0

Demo!

What happened?!

- Need to type multiple characters to receive just one
- Receiver is getting garbage

Why did this happen?

- Kernel does not queue all signals: just keeps latest one of each type
- No guarantee that signals received in order sent

How would you fix this?

- See signal-v2