Signals

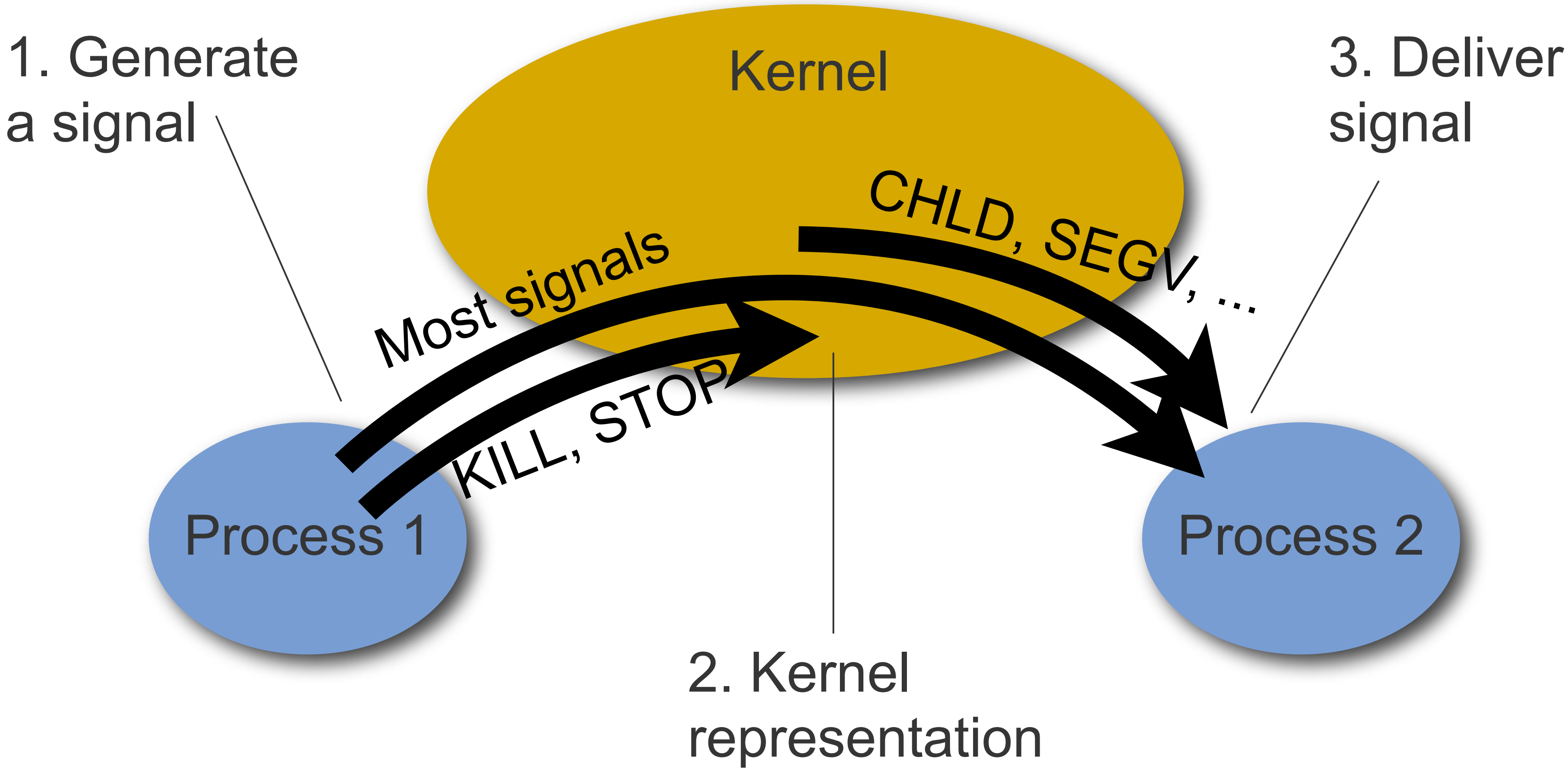
Signals

- Signal: notification to a process of an event
- Asynchronous notification: interrupt whatever I was doing, jumping to signal handler

A little puzzle

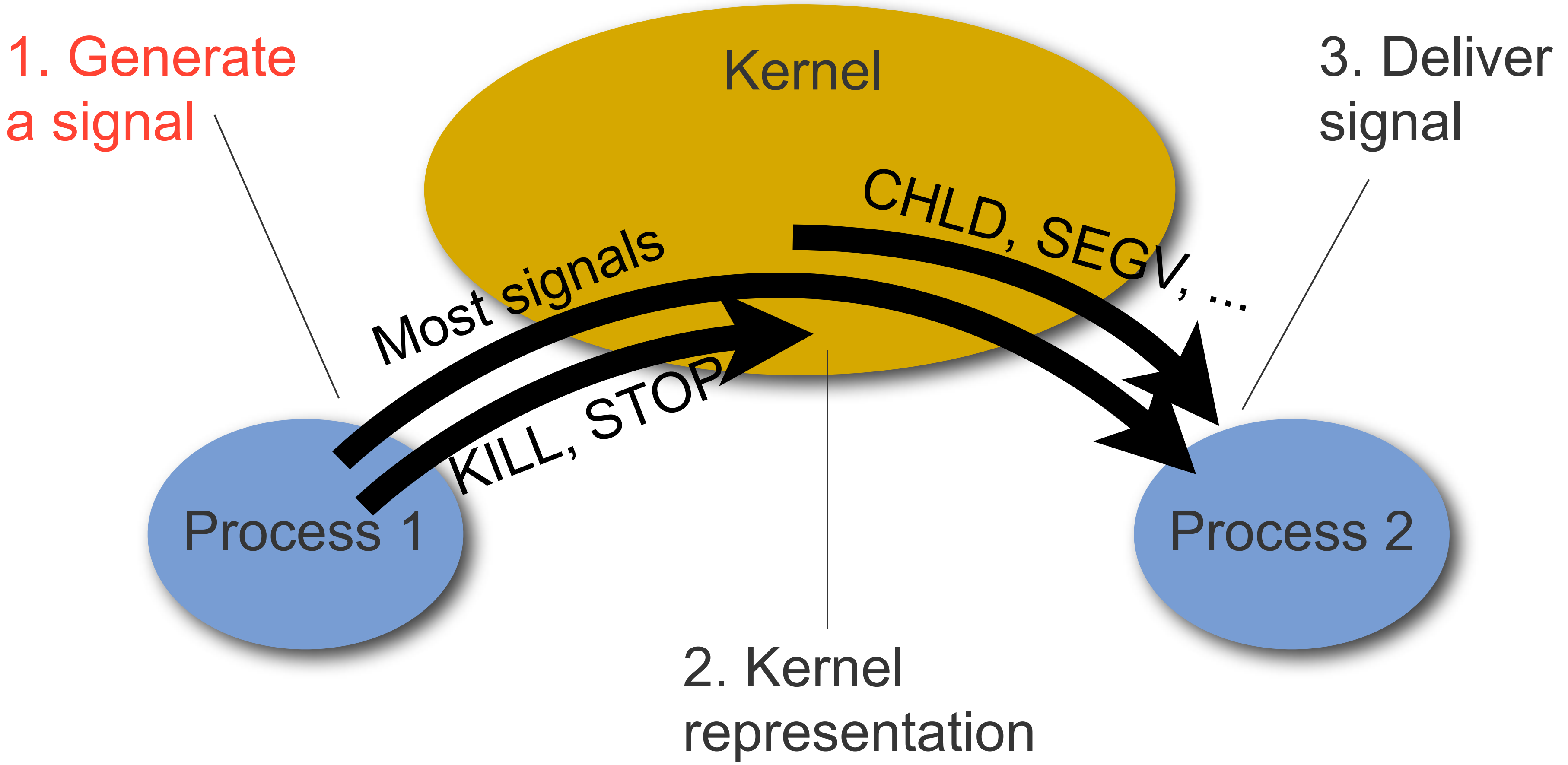
- Signals can be seen as a kind of interprocess communication
- What’s the difference between signals and, say, pipes or shared memory?
  - Asynchronous notification
  - Doesn’t send a “message” as such; just signal number
  - Puzzle: Then how could I do *this*? [DEMO]

Signaling Overview



Copyright ©: University of Illinois CS 241 Staff

Signaling Overview



Copyright ©: University of Illinois CS 241 Staff

Generating a signal

- Generated by a process
  - ...via system call `kill(pid, signal)` to send `signal` to process `pid`
  - `kill` is 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)
  - ...

Generating signals from the command line

- You can send a signal to a process from the command line using **kill**
- `kill -l` lists the signals the system understands
- `kill [-signal] pid` will send **signal** to the process with ID **pid**.
  - The optional argument may be a name or a number (default is SIGTERM).
- To unconditionally kill a process, use:
  - `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
- `raise(signal)` or `kill(getpid(), signal)`
- Or can signal after a delay:
  - `unsigned alarm(unsigned seconds);`
  - `alarm(20)` sends `SIGALRM` to calling process after 20 real time seconds.
  - Calls are not stacked
  - `alarm(0)` cancels alarm

A program can signal itself

- Example: infinite loop ... for 10 seconds

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

Morbid example

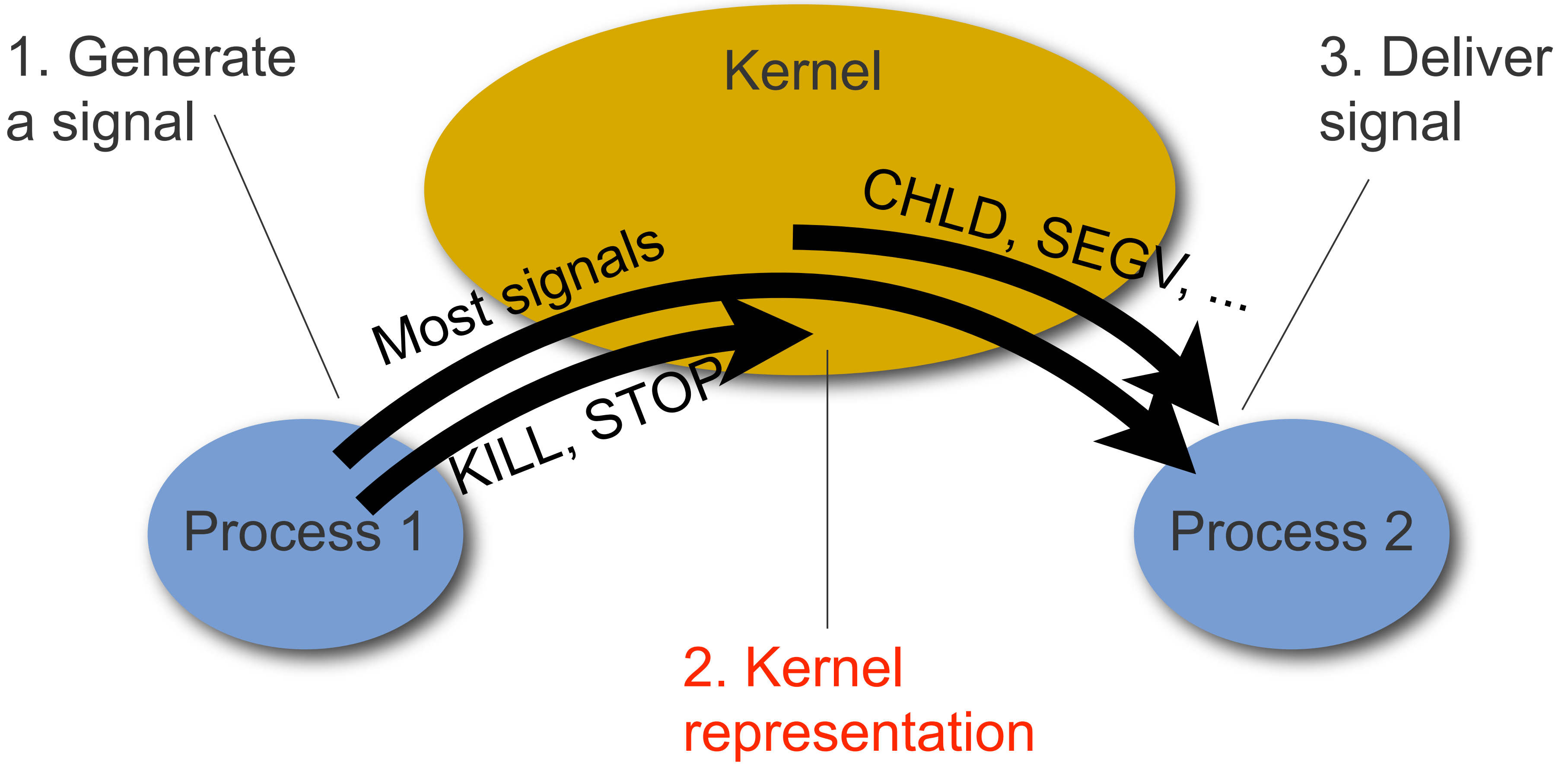
- What does this do?

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

int main(int argc, char** argv) {
    while (1) {
        if (fork())
            sleep(30);
        else
            kill(getppid(), SIGKILL);
    }
}
```

- Child kills parent in its sleep

Signaling Overview



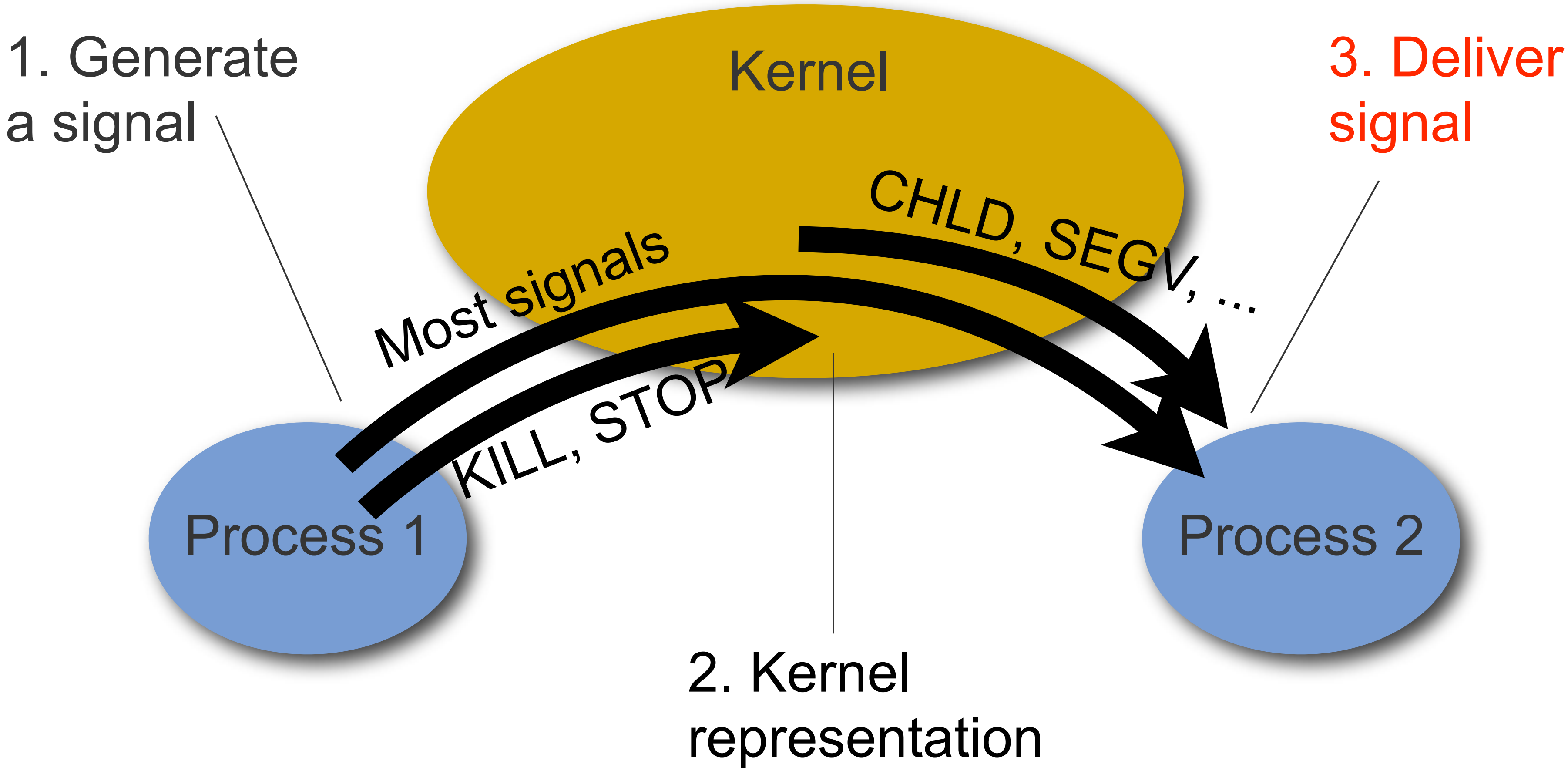
Kernel representation

- 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

- When signal arrives, set pending bit for this signal (N.B.: one bit per signal type!)
- When signal 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 proc.

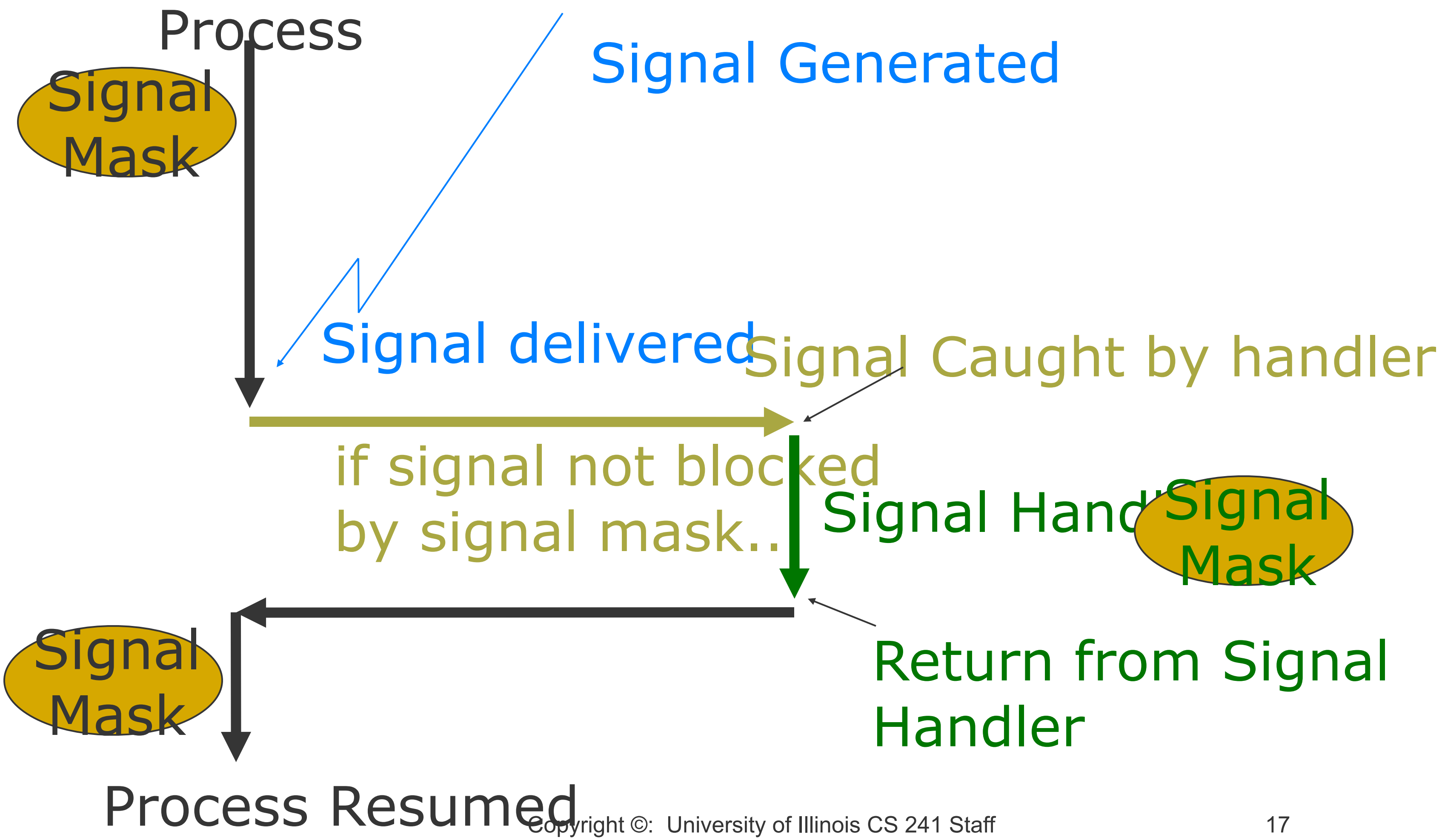
Signaling Overview



Delivering a signal

- Kernel may handle it
  - SIGSTOP, SIGKILL
  - Target process can’t handle these
  - They’re 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...



Process

Signal Mask

Signal Generated

Signal delivered

Signal Caught by handler

if signal not blocked by signal mask..

Signal Hand

Signal Mask

Return from Signal Handler

Signal Mask

Process Resumed

Signal mask

- Temporarily prevents select types of signals from being delivered
- Signal mask implemented as bit array, just like 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);
```

- See also `sigemptyset`, `sigaddset`, `sigdelset`, `sigismember`

If it’s not masked, we handle it

- Three ways to handle:
  - Ignore it (Note: different than blocking!)
  - Kill process
  - Run specified signal handler function
- One of these is the default (depends on which signal type)
- Tell the kernel what we want to do: `signal()` or `sigaction()`

Example: Catch control-c

```
#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

```
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 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....

Solution (p.1)

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

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

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

Solution (p.2)

Reader

```
  if (!strcmp(argv[1], "read")) {
    sigfillset(&signals_to_mask);
    sigprocmask(SIG_SETMASK, &signals_to_mask,
                NULL);
    while (1) {
      c = 0;
      for (i = 0; i < 8; i++)
        c |= recv_bit() << i;
      putchar(c); fflush(stdout);
    }
```

Writer

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

All the magic happens in the **`recv_bit()`** and **`send_char()`** functions. How do we implement those?

Solution (p.3)

```
int recv_bit() {
  int sig;
  sigset_t set;
  sigemptyset(&set);
  sigaddset(&set, SIGUSR1);
  sigaddset(&set, SIGUSR2);

  sigwait(&set, &sig);
  return (sig == SIGUSR2) ? 1 : 0;
}
```

These 4 lines construct the set of signals that we want to wait for. It's unfortunate that it takes 4 lines of code just to say "SIGUSR1 or SIGUSR2"!

Wait for either of those signals

Interpret received signal as either a 1 or a 0

Solution (p.4)

```
void send_char(pid_t friend, char c) {
  int i, signal;
  for (i = 0; i < 8; i++) {
    signal = (c & (1 << i)) ? SIGUSR2 : SIGUSR1;
    kill(friend, signal);
  }
}
```

What’s wrong with this “solution”?

1. **Lost signals** (kernel only stores 1 of each type)
2. **Reordered signals** (delivery order is arbitrary)

How can we fix this? **(Solution: see course web site)**

Announcements

- Survey: tinyurl.com/cs241survey
- Have a great break!