today: signals
ex. "^C" to interrupt program
$ kill -TSTP [pid#] (temporary stop) (also ^Z; stops w/o terminating)
$ kill -CONT [pid #] (continue)
$ kill -INT [pid #] (like ^C, an interrupt)

signal = "very small message"
exceptional control flow
ex. SIGSEGV (signal re: segmentation violation)
from dereferencing a NULL ptr ("no physical memory mapping to this virtual address space")
each one has its own number (#defined)
SIGFPE from dividing int. by 0
SIGPIPE (writing to pipe after read end has been closed)
SIGILL (illegal assembly code instruction)
SIGCHLD (OS sends to parent if child process stops, ends, restarts, etc.)

↳ some handled "synchronously" - have to be handled right then
handler executed immediately
usually the program did something bad to itself (eg. NULL ptr)
"traps"

↳ some handled "asynchronously" - external to process
no clear emergency for the process
at some time slice boundary (coming on/off the CPU) check for asynchronous signals & handle

↳ important one for multiprocessing: SIGCHLD custom
by default ignored - but you can set up signal handler for it
typically use waitpid, which deallocates process control block
PCB contains return info. when process returns from main

example: Disneyland

```
static size_t numDone = 0;

int main(-->) {
    signal(SIGCHLD, reapChildProcesses);
    for(size_t kid = 1; kid <= 5; kid++) {
        pid_t pid = fork();
        if (pid == 0) {
            sleep(3 * kid);
            cout << "child " << kid << " all done." << endl;
            exit(0);
        }
    }
    while (numDone < 5) {
        cout << "Not enough kids." << endl;
        sleep(5);
        cout << "Dad wakes up." << endl;
    }

    return 0;
}
```

signal handler has to take ↓ an int (the # of the signal)

```
static void reapChildProcesses(int unused) {
    waitpid(-1, NULL, 0);
    numDone++;
}
```

↖ CPU maintains SIGCHLD to be handled as bool, not int - so we can't just wait for 1 to finish (don't know how many have triggered)

```
static void reapChildProcesses(int unused) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, WNOHANG);
        if (pid <= 0) break;
        numDone++;
    }
}
```

WNOHANG

<= 0

if returns 0 with WNOHANG, means a child has finished but not all child processes

can't be 0! this function will loop until all child processes are finished

processes send signals via kill system call
int kill(pid_t pid, int sig);

to themselves:

```
int raise(int sig)
```