Why do we use signals?

- asynchronous I/O with aread()
  - returns right away and kernel keeps going with handling the data
  - get a SIGIO signal later
- error in your code (SIGFPE)
  - divide by zero
  - floating point overflow
  - invalid instruction
- impatient user of infinite loop (SIGINT)
  - ˆC to end program
- impending power outage (SIGPWR)
  - So we can do any saving before shutdown
- to check for dying children (SIGCHLD)
  - p = waitpid(-1, &status, WNOHANG)
  - now we don't have to call this method every 100 milliseconds
- user went away (SIGHUP)
- alarms
  - alarms are not inherited by fork() but by execvp()
- suspending a process
  - $ kill -STOP 29; kill -CONT 29
- kill the program SIGKILL
  - cannot be caught or ignored

We can kill a process with:

```
while (fork()) continue;
$ kill -KILL 29316       # does not kill children
# however this does not kill the shell bomb
$ kill -KILL -29316      # kills all children as well
```

Code must often be developed specifically to be able to handle signals; take the code

```
fd = open("foo", O_RDONLY);
fo = open("foo.gz", O_WRONLY);
while (compress(fd, fo)) continue;
close(fo);
unlink("foo");
// THIS CODE IS NOT ATOMIC AND CAN BE INTERRUPTED BY A SIGNAL
```

We can attempt to avoid these errors by implementing a signal handler

```
static void cleanup (int sig) {
  unlink("foo.gz");
  _exit(1);
}
int main(){...
  fd = open("foo", O_RDONLY);
  signal(SIGINT, cleanup);*
  fo = open("foo.gz", O_WRONLY);
  while (compress(fd, fo)) continue;
  close(fo);
  signal(SIGINT, SIG_DFL);*
  unlink("foo");
...}
```

but this still leaves race conditions .

In our current implementation, all threads are affected by the sign. Should all threads handle the signal? Would all threads handle it the same? NO; threads have their own signal mask to ignore signals. This is why pthread_sigmask() affects only current thread!

So how do we handle them? By default threads have their signals blocked. We use pthread_sigmask() to unblock the signal if we want a thread to handle it. Linux picks one random thread to deliver the signal. We make a mask with

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
// how = SIG_BLOCK, SIG_SCIMAKS, SIG_UNBLOCK
```

this allows the signal to arrive even before function returns. With this, we can build critical sections such as

```
sigset_t ss;
sigemptyset(&ss);
sigaddset(&ss, SIGINT);
pthread_sigmask(SIG_BLOCK, &ss, 0);
// critical section here .......
pthread_sigmask(SIG_UNBLOCK, &ss, 0);
```

But how can a signal handler manage memory access?

```
void handle_interrupt(int sig) {
  fprintf(stderr, "Interrupted\n");
  unlink (...);
}
fprintf(...) { malloc(...); } // interrupt
malloc(...) { // operating on heap }
// if we interrupt malloc and fprintf will call malloc again
// the second malloc may corrupt the heap, thus the first malloc call
```

Only some system calls can safely be used in handlers! We can call most system calls, such as:

- _exit()
- write()

But there are exceptions:

- exit() (calls malloc, flushes I/O buffer)
- fprintf()
- malloc()

We can perform all system calls in a single handler with:

```
void handle_interrupt(int sig) {
if (pthread_self() == stgmgr) really_handle_interrupt();
else pthread_kill(SIGINT, stgmgr); // forward signal to stage manager
}

# a more conservative approach is to set the variable and handle outside
sig_atomic_t volatile globv;
void handle_interrupt(int sig) {
  global = 1;
}
// always memory access, no cache!
```

But even with our scrupulous effort, interrupts can still cause difficulty:

```
read("/dev/tty", buf, 100);
// SIGHUP signal arrives
// run SIGHUP handler
// returns and continue reading
```

This means we have to complicate our code:

```
while (read("/dev/tty", buf, 100) == -1 && errno == EINTR) continue;
```

These types of errors are common with long system calls; clearly scheduling concurrent threads properly is important.