Week 09 1

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

November 14, 2023

Announcements

- A3 is due Wednesday
  - still TA office hours
  - I'll take questions
    - until 10 past 3
- A4 will be released this week
  - and is due Dec 6th

Exam date

- tentatively
  - still not final
- December 15th 2pm

Inter-process Communication

- Thus far, local (to a process)
  - file descriptors
  - system calls
- are the only paths
  - outside the containment
  - of a particular process

Pipes

- Are simple uni-directional
  - FIFO buffers
  - managed by the OS
- Ultimtely,
  - data "moves" through
    - file-descriptors

Does a process have to read a pipe?

- there's no mechanism of IPC
  - that we've seen yet
- that is more direct
  - more *necessary* to engage with

Signals

- are sent **by** the kernel
  - **to** specific processes
- but are requested by (other) processes
  - they **must**
    - be ignored
    - kill the receiving process
    - *suspend* the process
    - **handled manually**

How long does it take?

- From when I send a signal
  - to when it is delivered?
- system calls that generate a signal

    - proceed to the kernel
    - which then sets the signal
        * to pending for the target PID
- so they are received
    - the next time
        * the process gets scheduled
- self-signal is immediate

Why use signals

- it is not just processes
    - that might want to send a signal
- suppose your CPU can detect
    - when it is being asked
    - to divide by 0
        * the kernel can send a signal!
        * raising an error

Most common signals

- Interrupt character
    - typing `ctrl+c`
- Suspend character
    - typing `ctrl+z`
- detectable memory violations
    - Segmentation fault!

Useful new signals

- SIGALRM
  - we can ask the system
  - for a (variable accuracy) timer
    - to wake the program
- SIGFPE
  - floating point exception
  - from the hardware detecting
    - divide by zero, or overflows

We will work with standard signals

- 5 bits worth of them
  - starting at 1
  - therefore 31 signals
    - you can find macros
    - in `signal.h`
- E.g. `ctrl+c`
  - `#define SIGINT 2`

Ok, what are the system calls?

- we're starting
  - to get the hang of this
- the most basic
  - you guessed it is
    - `int signal()`

Receiving: `sigaction`

- as with `dup`
  - which we preferred `dup2`
    - and then there exists `dup3`
- there is an updated
  - and more standard API
    - called `sigaction`

`sigaction` in action

```
struct sigaction {
    void   (*sa_handler)(int);    /* Address of handler */
    sigset_t sa_mask;             /* Signals blocked during handler
                                     invocation */
    int      sa_flags;            /* Flags controlling handler invocation */
    void   (*sa_restorer)(void);  /* Not for application use */
};

int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

`sa_handler`

- `sa_handler` is a function **pointer**
  - run code at this address
  - if signal is encountered
- As a pointer
  - it is effectively a number
- Some special values are also valid
  - `SIG_IGN`: ignore these signals
    - can't do this for kill!
  - `SIG_DFL`: restore defaults

Sending: `kill()`

- this is much more straightforward
- `int err = kill(destinationPID, SIG_NUM);`
  - this can be done *within*
    - the same process
- You can also use
  - a command line utiltiy
    - called `kill`

Some gotchas

- you only know *if*
  - a signal has occurred
    - not how many (just on/off)
- you can't manually handle
  - `SIGKILL`
- but `SIGTERM` and `SIGINT`
  - can have handlers

signals are more under-the-hood

- they are happening all the time
- you just don't perceive them much

WORKSHEET

`signals.pdf`