Week 08 1

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

October 31, 2023

Announcements

- reading week is definitely
  - **next week**
- you have all the concepts
  - that you need to finish A3
  - It is due **Nov. 15th**

Recap

- processes are
  - abstractions that run code
    - on a system
  - with their own memory model
- `fork` allows us
  - to duplicate an existing process
  - and continue execution in both

Recap cont'd

- `exec` provides
  - a way to swap the contents
  - of one process' internals with
    - an executable file('s)
    - i.e. the process becomes
      - a different program
- `wait`
  - simply waits for *child*
    - processes
  - but required for coordination

More `wait`

- last class didn't feature
  - too much with `wait`
  - though you saw it in lab

Wait and return codes

- note that `wait(int *status)`
  - waits for **any** child to finish
    - called within the parent
  - so if you have 3 children
    - you could call this 3 times
- specifically calling
  - `waitpid(pid_t childpid, int *status, int options);`
    - allows you to wait
      - for specific children

Status codes for wait

- these are integrated into an int
  - we use macros to access
    - certain **bit fields**
      - you'll learn more in a few weeks
- for now, you can use:
  - `WIFEXITED(status)`
    - true if exits normally
  - `WEXITSTATUS(status)`
    - for the numeric code
      - returned by child's main

Let's do a worksheet together

`wait.pdf`

Communication between processes

- if processes are separated
  - even if they are duplicates
- how could we ever get them
  - to work together?
- what have we learned so far
  - that might enable this?

Consider using a file

- what if we opened a specific file
- and we write to it
  - with Process A
    - (and close it nicely)

- and then read from it
  - with Process B
    - (without interfering with A)

Inter-process communication

- this would allow Process A
  - to write arbitrary bytes
  - later consumed by Process B
- we could have an additional file
  - for the other direction!
  - or share just one file...
    - (this gets more difficult)

System conveniences

- rather than do this
  - and waste disk space
  - and worry about opening
    - at the right time
    - without interference
- we have an additional system tool
  - called **pipes**
  - that are exactly what we describe

first recall file-descriptors

- this is extensively discussed
  - in the PCRS videos
- but essentially
  - all the open *file-like* constructs
    - are assigned a number

        * a file descriptor
- this allows the system to keep track
    - of what each process opens

standard fd's

- this includes
    - stdin: 0
    - stdout: 1
    - stderr: 2
- *as well as pipes*!
- **Note:** files are slightly different
    - the FILE struct is used
    - but there is an underlying
        * `fileno()`

Piping

- as you might expect
    - we use another system call:
        * `int pipe(int fd[2]);`
    - the return value
        * is for error detection
- otherwise two fd's are written
    - to the argument

I don't see how this connects to inter-process communication

- fd's are interesting
    - because `fork` duplicates them
    - so that child processes
        * **access the same ones**

- i.e. the fd numbers refer to the same
  - file-like abstraction
    - provided by the system

A parent-child pipe example

```
#include <unistd.h>

int pipefds[2];
pipe(pipefds); // no error check

int read_end = pipefds[0];
int write_end = pipefds[1];

pid_t result = fork();
if (result == 0){ //child
    close(read_end);
    write(write_end, ..., ...);
    close(write_end);
} else { //parent
    close(write_end);
    read(read_end, ..., ...);
    close(read_end);
}
```

Two more system calls

- `size_t read(int fd, void *dest_buffer, size_t num_to_read);`
  - much like `fread`, but with fds
    - not with `FILE` structs
- `size_t write(int fd, void *src_buffer, size_t num_to_write)`
  - much like `fwrite`, but with fds