Week 07 1

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

October 26, 2023

Announcements

- Reading week next week
- A3 is up
- office hours after this lecture

Processes

- what is a process?
- a data-structure
  - a particular abstraction
- that represents an **individual**
  - program that can run
  - on a system

As a datastructure

- variables needed to execute
  - the actual machine instructions
- as well as indicators for
  - where you currently are
- and also the memory model
  - of the current program

Running processes

- processes ultimately
  - feature machine instructions
  - that need to run on HW
- But there are too many
  - so they share the HW
- The OS facilitates this

What does this mean?

- essentially, that
  - at worst
  - you don't know when a process
  - might be interrupted
    - (often after system calls)
- It will continue again
  - exactly where it left off
- but order is unknown

Process IDs (PID)

- most coordination of processes
  - is handled automatically
  - by the OS
- but PID allows us
  - as developers
    - and by proxy, our apps
  - to monitor other processes

Processes in a linux system

- suppose you boot linux
- this means the computer boots some code
  - called the kernel
  - which prepares the filesystem
  - and enables the system calls
- Then, it creates PID 1
  - a.k.a init
  - which sits around to adopt orphans

After init

- init is somehow duplicated
  - PID 2 is just another copy of init
  - with the same variables
    - with it's own memory locations for
      - each of the variables
- this new copy
  - get's cannabalized into the next
    - step in booting the system
- and this model underlies
  - all process creation

Consider if PID 2 was a shell

- what if the exact next program
- was a shell program
  - connected to screen and keyboard
- we could use it to start more!
  - other programs!
  - or maybe a window manager
    - to contain more programs

Fork

- the first system call
- that we need to learn about
- to be process masters

What is fork?

- essentially it duplicates
  - the current process' datastructure
    - memory model, and current code position
  - creating a **child**
  - as if there was a big struct (PCB)
  - somewhere and you just copied it
- this includes
  - the memory model for this process
    - **not the literal locations**

Why is this important

- because all of the variables
  - are duplicated
  - by they are in different places!
- but *some things*
  - like file descriptors/names
  - or other identifiers
    - e.g. PID
  - still access the same entities
    - because they're an ID

So it's the same program?

- not anymore
- once `fork` is executed
- the variables are the same
  - but they are **independent**
  - they just have the same values
  - now there is a place in
    - literal memory, that is different
    - for the original and copies
- but the memory model for both is unchanged

Why would the child be any different?

- We use the return value of fork!
- child processes get a value of 0
  - while parents get the PID of the child
- if non-zero, this is the parent
- if zero, this is the child

Other system calls

- there is an ecosystem
  - of system calls
- for process management

`wait(pid)`

- without a pid
  - it just waits for *children*
  - of the current process
    - to completely finish

- with a pid
  - it can retrieve info
  - most importantly
    - the **exit status**

`exec`

- this is the crucial companion
  - to `fork`
- while `wait` allows us to
  - make sure our processes
    - work together as intended
- `exec` allows us to run
  - **new** programs
- not just duplicates

`exec`

- this is ultimately how
  - an executable
  - gets converted into a process
- give `exec` an executable
  - as an argument
- and it modifies the current process
  - to execute the executable's values

Roughly speaking

- `exec` fills in the process datastructure
  - with new values
  - that it picks up from an executable file
- If the file is a proper
  - executable type (and perm.)
- the values
  - in the current model's addresses
  - are replaced by the executable' values
    - e.g. code, RO-strings, etc.

These are actually system call families

- there are many system calls
- of the `wait` or moreso, `exec` variety
- so what is different
  - not much, besides changes in argument
  - or argument usage
  - the fundamentals are the same

man pages

Let's start an example

WORKSHEET

`fork.pdf` and `fork_fruits.c`