6.828 2014 L4: Shell & OS organization --

Homework solution ==

- Let's review solution for sh.c
    - exec
    - redirect
    - pipe
- The process graph for a complicated pipeline
    - Who waits for whom? (draw a tree of processes)
    - Why close read-end and write-end? ensure that every process starts with 3 file descriptors, and that reading from the pipe returns end of file after the first command exits.

Exploring system calls with more shell features ==

- You can run the shell, redirect its stdin/stdout, etc.
    - I'll run this shell script with `sh < script`:

      ```
      echo one
      echo two
      ```

    - What will this shell command do? `$ sh < script > out`
        - the script itself didn't redirect the echo output, but it did inherit a fd 1 that was redirected to out.
        - to make this work it is important that offset is implicit (maintained by kernel)
        - this is why read and write don't take an offset argument
    - Is the following the same as above?

      ```
      $ echo one > out
      $ echo two > out
      ```

    - How to implement sequencing/lists (;,&&,||)

      ```
      $ gcc x.c ; ./a.out

      $ gcc x.c && ./a.out
      ```

    - How to implement nesting?

      ```
      $ (echo one; echo two) > out
      ```

- How does the shell implement "&"? `$ sleep 2 &`
    - Q: What if a background process exits while sh waits for a foreground process?

System call observations ==

- The fork/exec split looks wasteful; why is it useful? (A: exercise 2)

- System call interface simple, just ints and char buffers. why not have open() return a pointer reference to a kernel file object?

- Linux has a nice representation of a process and its FDs, under /proc/PID/

    - maps: VA range, perms (p=private, s=shared), offset, dev, inode, pathname
    - fd: symlinks to files pointed to by each fd.
- The file descriptor design:

    - nice interaction with fork
    - FDs help make programs more general purpose: don't need special cases for files vs console vs pipe
    - shell pipelines only work for programs w/ common formats (lines of text)

OS organization ==

- Now we have a feel for what Unix system call interface provides, how to implement the interface?

- Why have an o/s at all? why not just a library? then apps are free to use it, or not -- flexible apps can directly interact with hardware some tiny O/Ss for embedded processors work this way

- Key requirement: isolation multiplexing interaction

- helpful approach: abstract machine resources rather than raw hardware File system, not raw disk TCP, not a raw ethernet Processes, not raw CPU/memory abstractions often ease multiplexing and interaction also more convenient and portable

- Goals: apps must use OS interface, cannot directly interact with hardware apps cannot harm operating system

- Hardware support for isolation

    - Processors support user mode and kernel mode
        - some instructions can only be executed in kernel mode e.g., instructions to directly interact with hardware
    - If an application executes a privileged instruction hardware doesn't allow it
        - instead switches to kernel mode
        - kernel can clean up
- Leverage hardware support

    - Operating systems runs in kernel mode

        - kernel is a big program services: processes, file system, net low-level: devices, virtual memory all of kernel runs with full hardware privilege (convenient)
    - Applications run in user mode

        - isolated from kernel
        - systems calls switch between user and kernel mode application call instructions to enter kernel instruction enters kernel at an entry point specified by kernel
- What to run in kernel mode?

    - xv6 follows a traditional design: all of the OS runs in kernel mode

        - this design is called a monolithic kernel

        - kernel interface ~= system call interface
        - good: easy for subsystems to cooperate one cache shared by file system and virtual memory
        - bad: interactions are complex leads to bugs no isolation within kernel
    - alternative: microkernel design

        - many OS services run as ordinary user programs file system in a file server
        - kernel implements minimal mechanism to run services in user space IPC virtual memory threads
        - kernel interface != system call interface
        - good: more isolation
    - jos: doesn't abstract hardware resources, but still provide isolation

- Can one have process isolation WITHOUT h/w-supported kernel/user mode? yes! see Singularity O/S, later in semester but h/w user/kernel mode is the most popular plan

xv6 kernel address space ==

- boundary between apps and kernel kernel/user bit kernel address space ...

- start w. kernel address space machine has booted code runs without virtual memory set up up kernel address space walk through kvmalloc()

Edit By MaHua