OS MMXXII

MIT ;)

https://pdos.csail.mit.edu/6.828

Štruktúra prednášky

Úvod do OS

Štart prvého procesu xv6

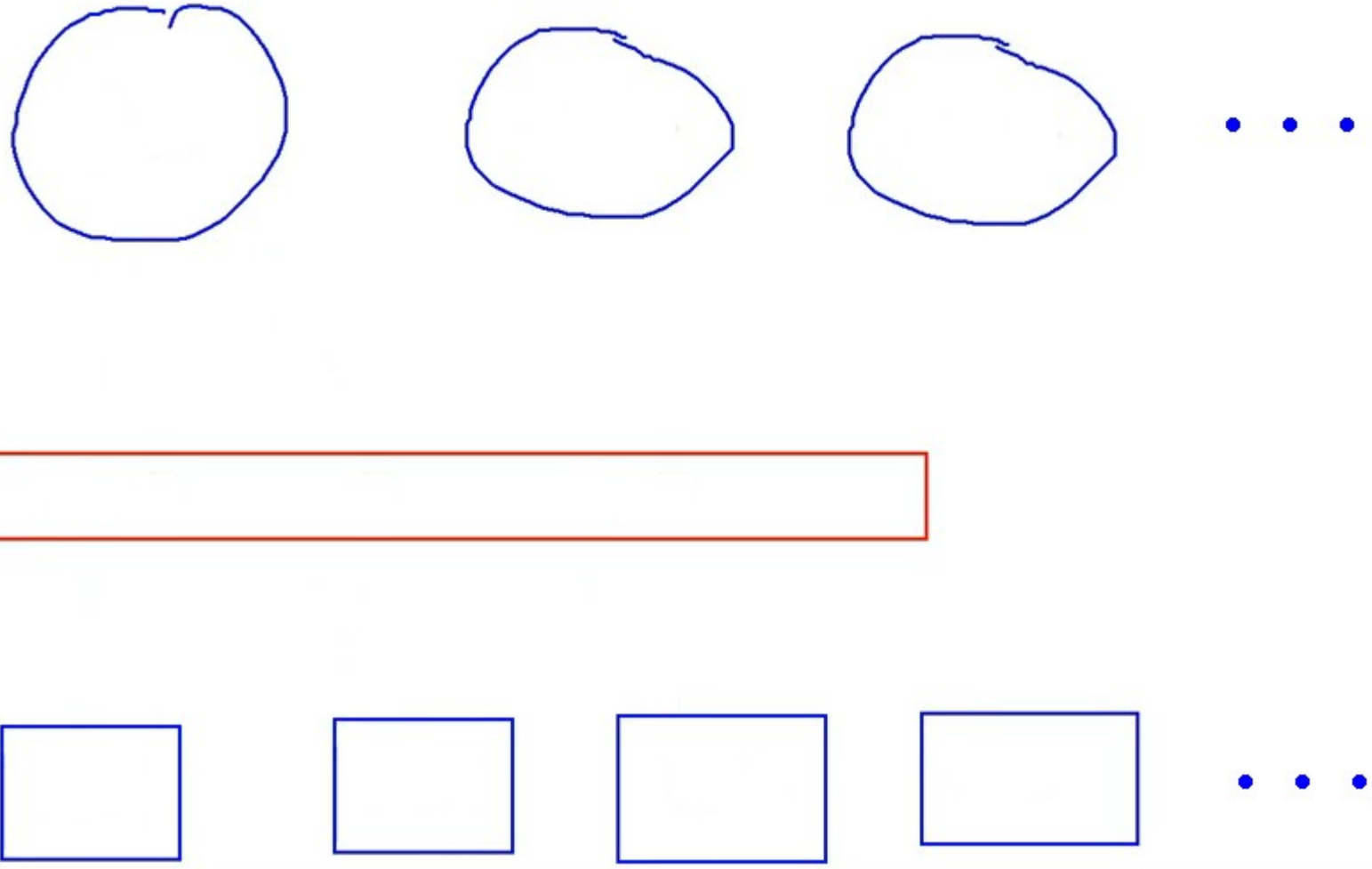
Izolácia procesov

- Systémové volania
- Virtuálna pamäť
  - Segmentácia
  - Stránkovanie

Opakovanie

- Načo slúži OS?

Opakovanie

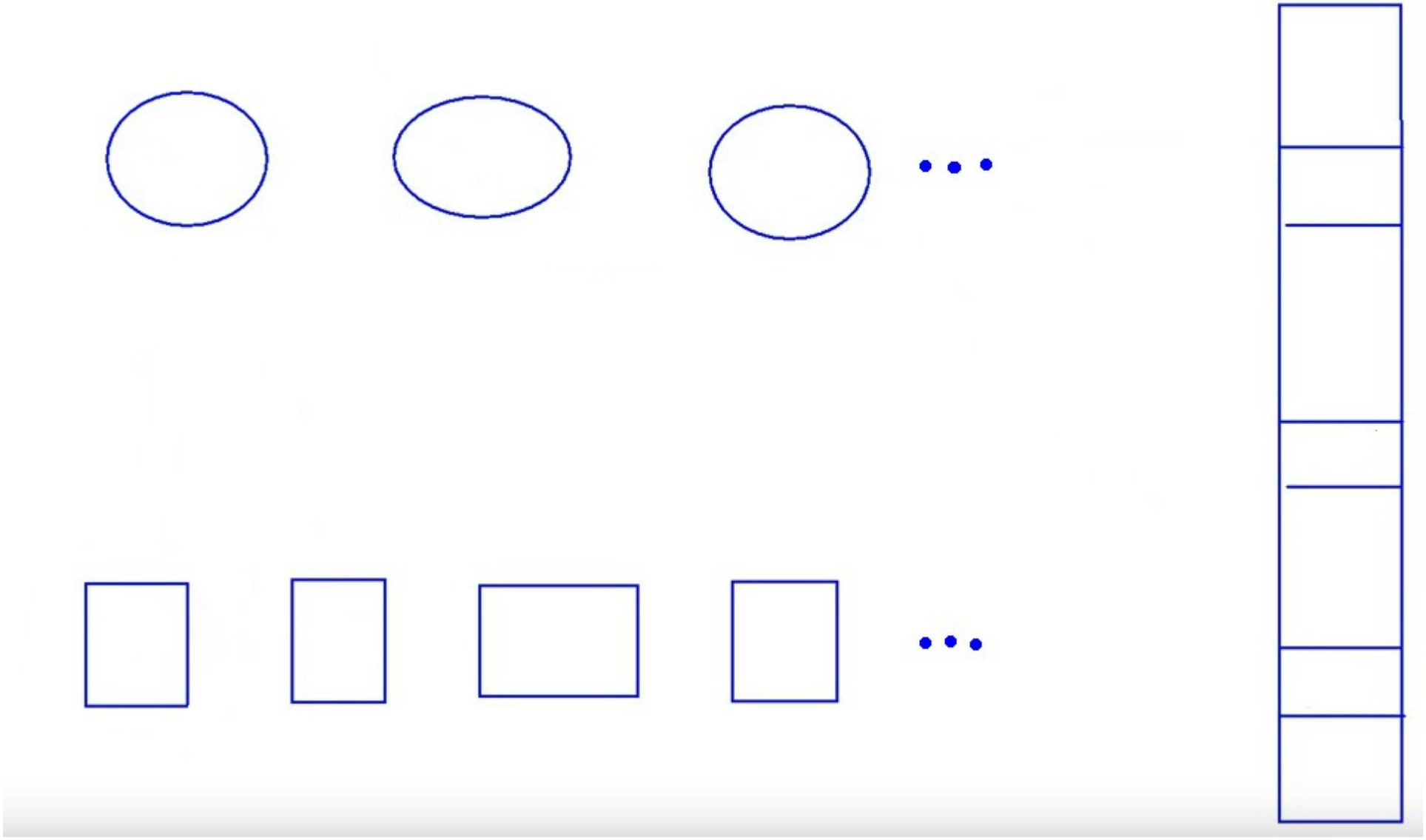


Opakovanie

- Ciele OS

1. Umožniť beh viacerým aplikáciám (súčasne)
2. Izolovať aplikácie
3. Umožniť aplikáciám komunikovať
4. Efektívne využívať hardvér

Môžu bežať app bez OS?



Môžu bežať app bez OS?

- Obrázok 2

- Aplikácie priamo interagujú s hw
  - CPU
  - RAM
  - HDD
  - …

Môžu bežať app bez OS?

- Problém multiplexingu

- Aplikácia sa sama musí vzdať CPU
  - Ak to programátor zabudne urobiť, iná app sa k CPU nedostane
  - Ak sa app dostane do nekonečného cyklu, iná app sa už k CPU nedostane
  - Nie je možné ukončiť beh inej aplikácie (`kill`)
- Tento prístup sa používa pri OS reálneho času (*kooperatívne* plánovanie procesov)

Môžu bežať app bez OS?

- Problém vzájomného prístupu do pamäte

- Nejestvuje izolácia, všetky aplikácie majú priamy prístup do pamäte
- Aplikácia môže prepísať údaje inej aplikácie
- Aplikácia môže prepísať kód zdieľanej knižnice pre všetky aplikácie
- Každá aplikácia má prístup ku všetkým údajom iných aplikácií

Rozhranie UNIX-like OS

- OS vytvára abstrakciu hardvéru

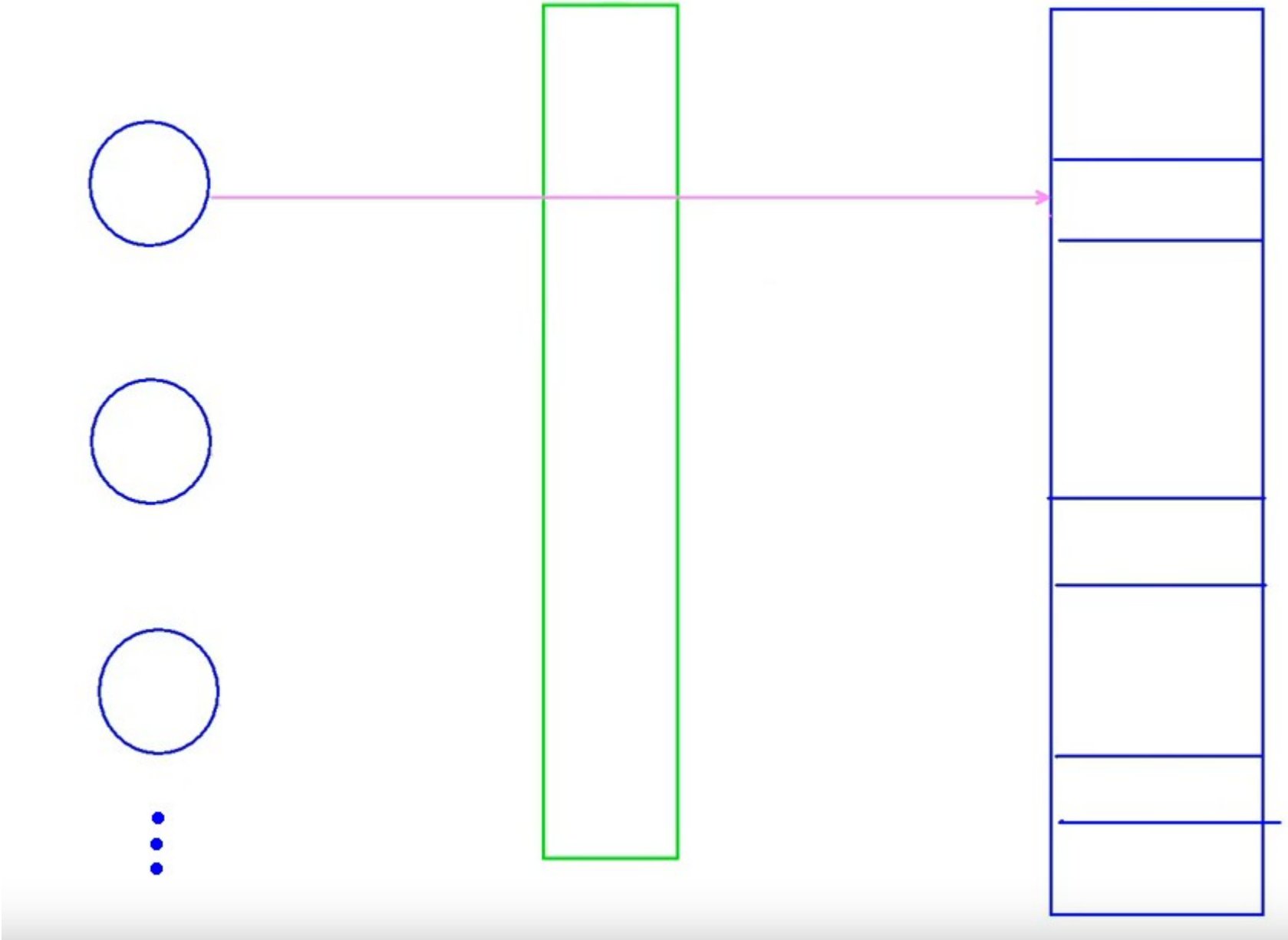
- Procesy ←→ jadrá CPU (`fork`)
- Pamät ←→ fyzická pamäť (`exec`)
- Súbory ←→ diskové bloky (`open`, `read`, …)
- Rúry ←→ zdieľaná fyzická pamäť (`pipe`)
- …

Rozhranie UNIX-like OS

- Procesy ←→ jadrá CPU (`fork`)
  - OS obsadzuje jadrá pre beh procesov (ukladá a obnovuje registre)
  - OS vynucuje výmenu procesov na CPU

- Pamäť ←→ fyzická pamäť (`exec`)
  - Každý proces má svoju „vlastnú” pamäť (obrázok 3)
  - OS rozhoduje, kam sa do ramky umiestni app

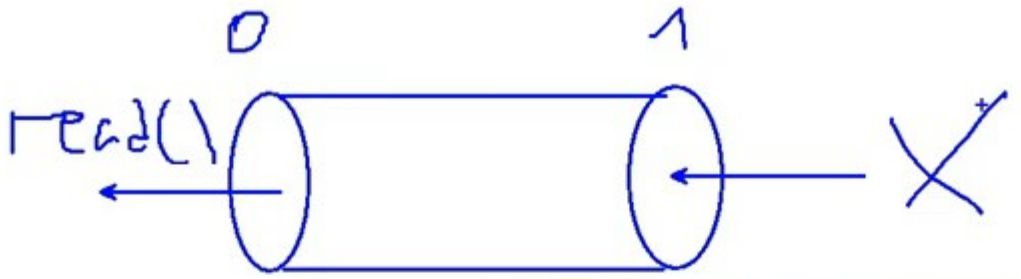
Rozhranie UNIX-like OS

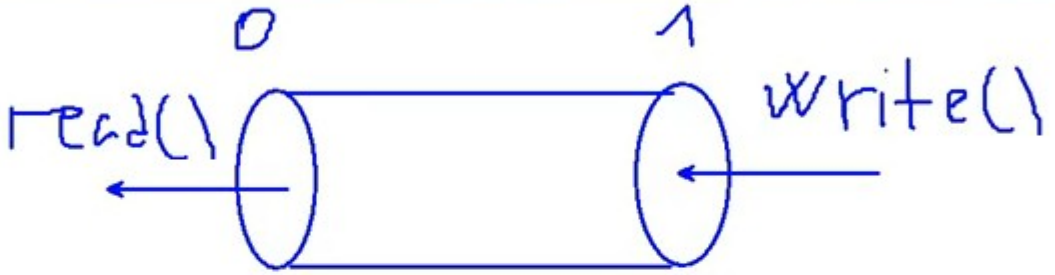


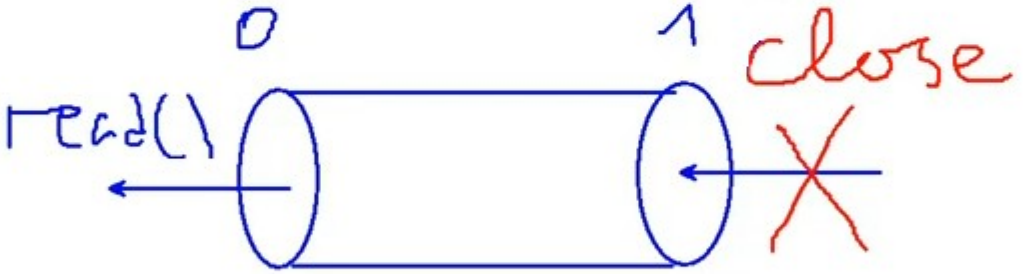
Rozhranie UNIX-like OS

- Súbory ←→ diskové bloky (`open`, `read`, ...)
  - OS poskytuje pohodlné používanie mien súborov a adresárovú štruktúru
  - OS môže umožniť zdieľanie súborov medzi procesmi/používateľmi systému

- Rúry ←→ zdieľaná fyzická pamäť (`pipe`)
  - OS riadi synchronizáciu prenosu (ak je rúra prázdna, čitateľ musí čakať) (obrázok 4)
  - OS môže kedykoľvek ukončiť činnosť čitateľa či zapisovateľa


0
1
read()


0
1
read()
write()


0
1
read()
close


close
0
1
write()

Rola OS

- OS musí byť defenzívny
- Aplikácia by nemala byť schopná spôsobiť pád OS
- Aplikácia by nemala byť schopná preraziť bariéru izolácie a zasahovať priamo do inej aplikácie

Ako?

- Keďže aj OS aj aplikácia sú sw, väčšinou sa to robí pomocou hw

1. Hw podpora rôznych úrovní vykonávania inštrukcií na CPU (user / kernel mód)

2. Hw podpora virtuálnej pamäte (vytvorenie zdania „vlastnej” pamäte pre proces)

User / Kernel mód

- Už v minulej prednáške sme spomínali, že kernel mód znamená možnosť vykonávania všetkých inštrukcií, aj tzv. privilegovaných
  - napr. nastavenie módu procesora
  - alebo priamy prístup k hw
- V user móde nie je možné privilegované inštrukcie vykonať
  - Pri pokuse sa vyvolá hw výnimka, ktorú má možnosť obslúžiť programový kód v kernel režime
- OS beží v kernel móde, procesy v user móde

Virtuálna pamäť

- SW používa adresy (ukazatele, angl. *pointer*)
  - Nazývame ich virtuálne
  - Platí to ako pre OS, aj pre aplikácie používateľa
- HW poskytuje tzv. tabuľky stránok, pomocou ktorých sa robí (hardvérovo!) preklad virtuálnej adresy na fyzickú (do ramky)
- OS nastavuje tieto tabuľky pre proces tak, aby nemal prístup k dátam iného procesu
- Tabuľky určujú, do ktorej fyzickej pamäte má tá-ktorá aplikácia prístup

Komunikácia app s OS

- Ak sa využije hw na takto silnú izoláciu, je potrebné vymyslieť mechanizmus, pomocou ktorého môže aplikácia **kontrolovaným spôsobom** pristúpit k hw (alebo k údajom inej aplikácie)
- V podstate sa tento problém redukuje na kontrolovaný prechod z užívateľského režimu procesora do kernel módu procesora
- Na architektúre RISC-V sa to deje pomocou inštrukcie `ecall <n>`

ecall <n>

- Vyvolanie služby OS presne definovaným spôsobom
- Procesy nemajú prístup k funkciám jadra OS priamo!

- Systémové volania nie sú „klasické” funkcie
  - `user/forktest.asm` hľadaj `fork`
  - `user/usys.pl` →`user/usys.S`
  - hw prepnutie do kernel módu; OS určuje, kde sa po prepnutí začne vykonávať kód jadra OS

Štruktúra xv6

- Monolitický kernel
  - Rozhranie user/kernel space: systémove volania
- Zdrojové kódy sú usporiadané modulárne
  - `user/` → aplikácie v user móde
  - `kernel/` → kód v kernel móde
- Samotný kernel pozostáva zo samostatných častí
  - Viď `kernel/defs.h` (proc, fs, ...)
- Kód je vynikajúco dokumentovaný
  - Možnosť pochopiť ho aj bez čítania knihy o xv6

Použitie xv6

- Súbor zostavenia `Makefile` riadi
  - Vytvorenie programu jadra (priečinok `kernel/`)
  - Vytvorenie užívateľských programov (priečinok `user/`)
  - Vytvorenie disku (priečinok `mkfs/`) pre xv6

- Príkaz `` `make qemu` ``
  - Spúšťa xv6 vo virtualizačnom nástroji qemu
  - Qemu emuluje počítač architektúry RISC-V

RISC-V doska

- Ide o reálny hw → možnosť skutočného spustenia xv6!
- Veľmi jednoduchá základná doska (bez grafického výstupu)
  - CPU má 4 jadrá
  - RAM 128MB
  - Podpora prerušení
  - Podpora UART (sériová konzola/klávesnica)
  - Podpora sieťovej karty e1000 (cez zbernicu PCIe)
- Qemu emuluje presne túto konfiguráciu

Prečo qemu a nie hw?

- Museli by ste si ho zakúpiť (a vzhľadom na dodaciu lehotu predmet robiť o rok)

- Pohodlnejšie je využívať služby qemu, nakoľko
  - Qemu emuluje viacero implementácií RISC-V
  - Xv6 využíva počítač “virt” (https://github.com/riscv/riscv-qemu/wiki)
  - Táto implementácia je dosť blízka hw doske SiFive, ale navyše má podporu pre rozhranie VirtIO (https://www.sifive.com/boards)

Čo znamená emulácia?

- Qemu je sw (program), ktorý implementuje RISC-V procesor

```
for (;;) {
  1) read next instruction
  2) decode instruction
  3) execute instruction (updating
  processor state)
}
```

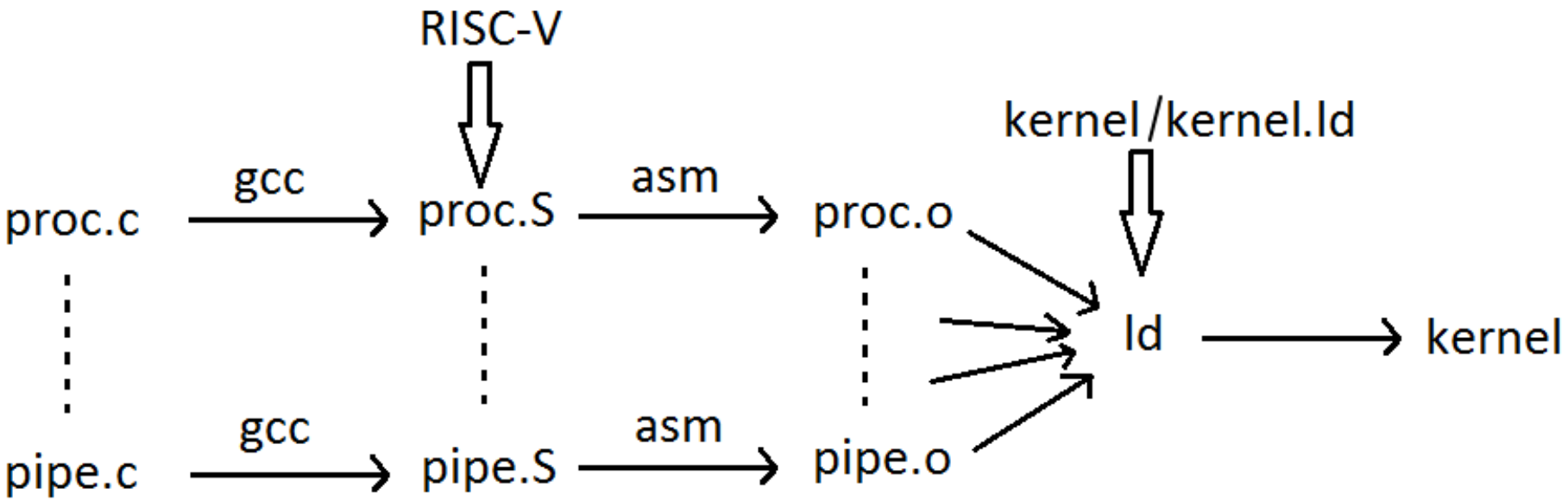
Štart systému xv6

- Využijeme gdb
- Spustíme xv6 s jedným jadrom (nie ako je prednastavená hodnota 4), aby sme mohli sledovať jedno vlákno v gdb

```
$ make CPUS=1 qemu-gdb
```

- Qemu začne vykonávať kód xv6, ktorý sa nachádza v `kernel/entry.S`
  - Viď `kernel/kernel.ld` symbol `_entry` (riadok 2)
  - Čo je `kernel.ld`?

Zostavenie xv6



prevzaté a upravené z https://pdos.csail.mit.edu/6.828/2020/lec/l-os-boards.pdf [21. september 2020]

`Makefile` je nastavený tak, aby vytvoril `asm` súbory (viď napríklad `kernel.asm`)

Štart systému xv6

- `b _entry`
  - Porovnajme inštrukcie so súborom `kernel.asm`
  - `info reg, x/10i _entry`
- `b main`
  - si, continue, next next next next…, tui enable
- step do funkcie `userinit()`
  - Next cez funkciu
  - Viď `proc.h`
  - Step do `allocproc()`
  - Viď `initcode.S/initcode.asm` a `od -t xC initcode`

Štart systému xv6

- `b forkret`
  - Next po `usertrapret()`
- `b syscall`
  - `print num`
  - Step do `syscalls[num]()`
  - Nachádzame sa v `` `exec "/init"` ``
- `symbol-file user/init.o`
  - `b main`
  - `continue`

Nejaké otázky?

Domáca úloha

- Prečítať kapitolu 2
- *Operating system organization*

- https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

MIT ;)