

OS MMXXI

MIT ;)

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

# Štruktúra prednášky

Úvod do OS

Štart prvého procesu xv6

Izolácia procesov: systémové volania

# Opakovanie

- Načo slúži OS?

# Opakovanie



...

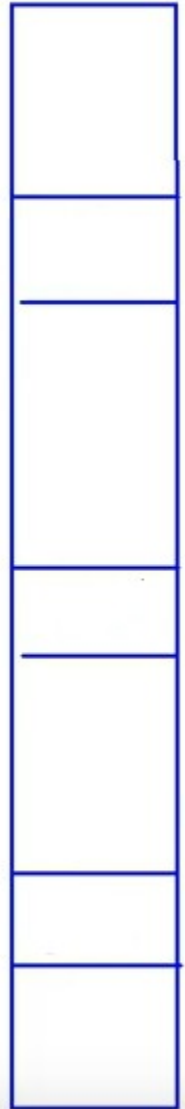


...

# 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

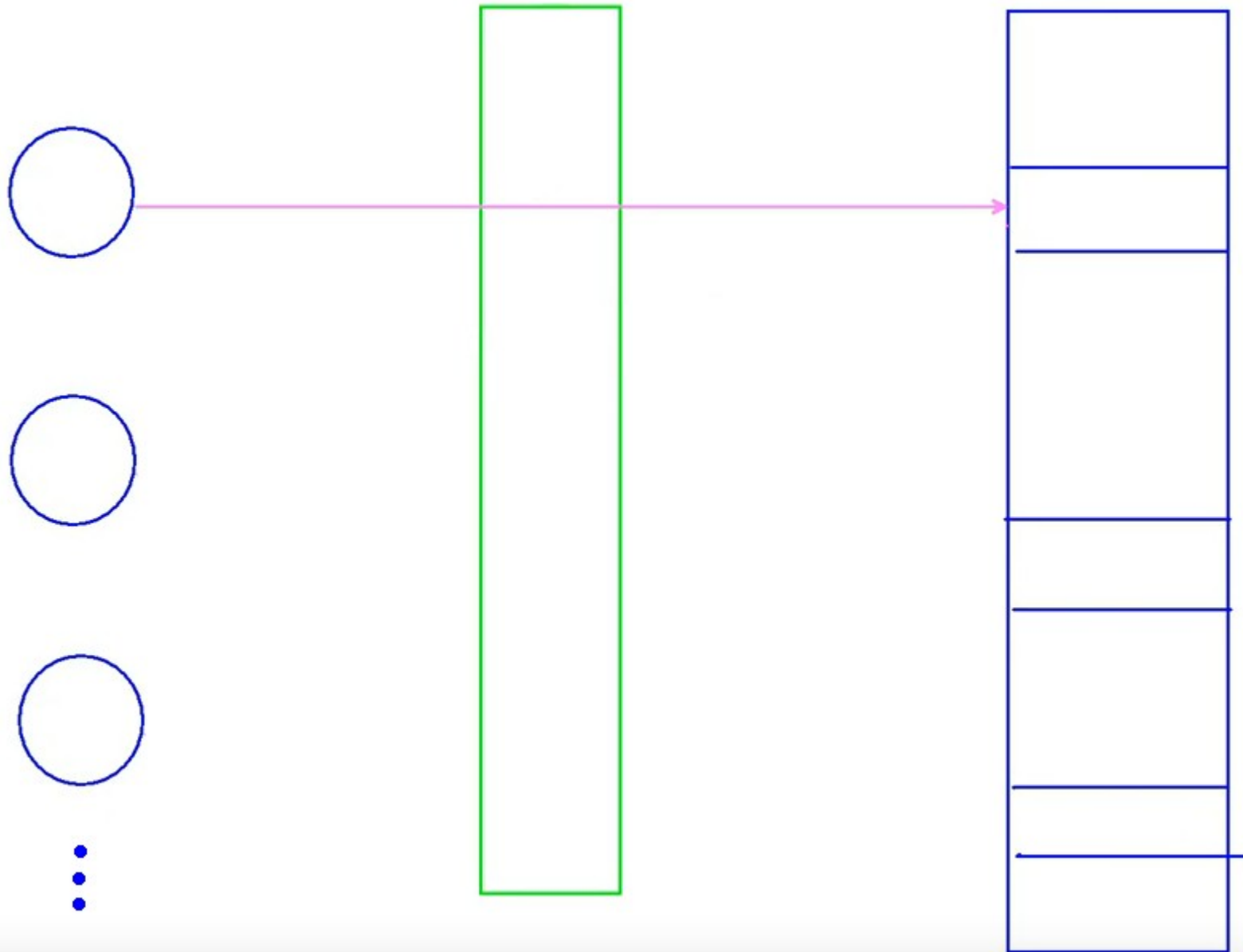
# Rozhranie UNIX-like OS

- OS vytvára abstrakciu hardvéru
- Procesy  $\leftrightarrow$  jadrá CPU (fork)
- Pamät  $\leftrightarrow$  fyzická pamät' (exec)
- Súbory  $\leftrightarrow$  diskové bloky (open, read, ...)
- Rúry  $\leftrightarrow$  zdieľaná fyzická pamät' (pipe)
- ...

# Rozhranie UNIX-like OS

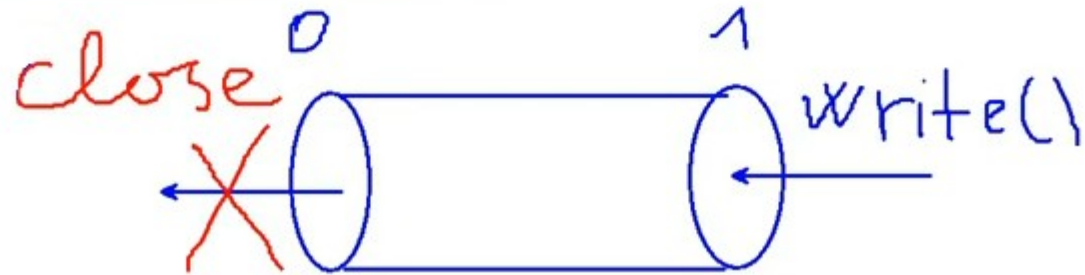
- Procesy  $\leftrightarrow$  jadrá CPU (fork)
  - OS obsadzuje jadrá pre beh procesov (ukladá a obnovuje registre)
  - OS vynucuje výmenu procesov na CPU
- Pamäť  $\leftrightarrow$  fyzická pamäť (exec)
  - Každý proces má svoju “vlastnú” pamäť (obrázok 3)
  - OS rozhoduje, kam sa do ramky umiestni app

# Rozhranie UNIX-like OS



# Rozhranie UNIX-like OS

- Súbory  $\leftrightarrow$  diskové bloky (open, read, ...)
  - OS poskytuje pohodlné používanie mien súborov a adresárovú štruktúru
  - OS môže umožniť zdieľanie súborov medzi procesmi/používateľmi systému
- Rúry  $\leftrightarrow$  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



# 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** pristúpiť 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: systémové volania
- Zdrojové kódy sú usporiadané modulárne
  - `user/` → aplikácie v user móde
  - `kernel/` → kód v kernel móde
- Samotný kernel pozostáva zo samostatných častí
  - Vid' `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 (priečinkov kernel/)
  - Vytvorenie užívateľských programov (priečinkov user/)
  - Vytvorenie disku (priečinkov 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?

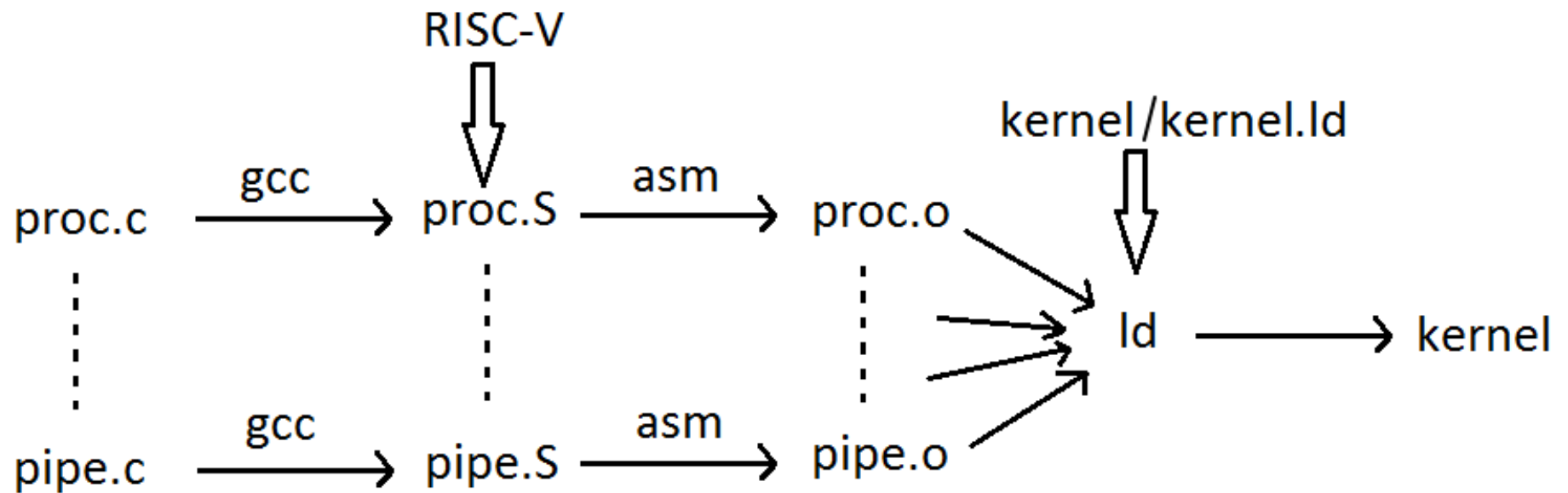
- Qemu je sw (program), ktorý implementuje RISC-V procesor

```
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`
  - Vid' `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 (vid' napríklad `kernel.asm`)

# Štart systému xv6

- `b _entry`
  - Porovnajme inštrukcie so súborom `kernel.asm`
  - `info reg`
- `b main`
  - Next next next next ...
- step do funkcie `userinit()`
  - Next cez funkciu
  - Vid' `proc.h`
  - Step do `allocproc()`
  - Vid' `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?

# Von Neumann architektúra

- CPU

- Memory



- I/O

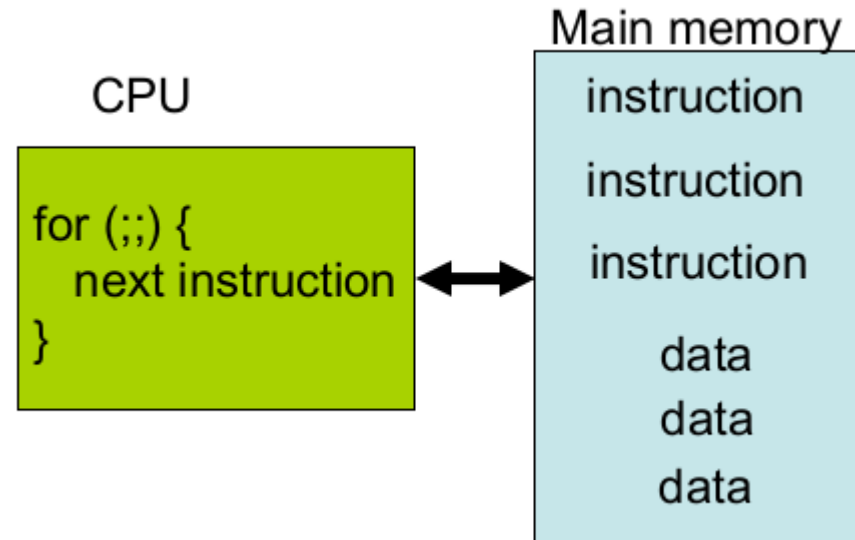


- Bus



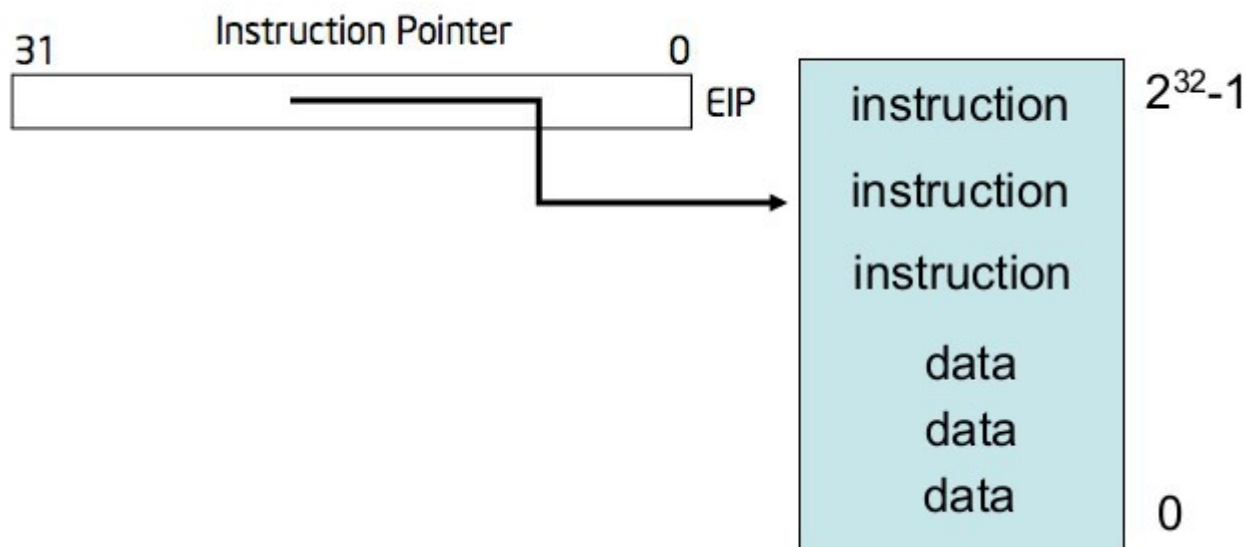
# Vykonávanie programu

- Pamäť
  - Inštrukcie
  - Údaje
- CPU
  - Interpretácia
  - Manipulácia

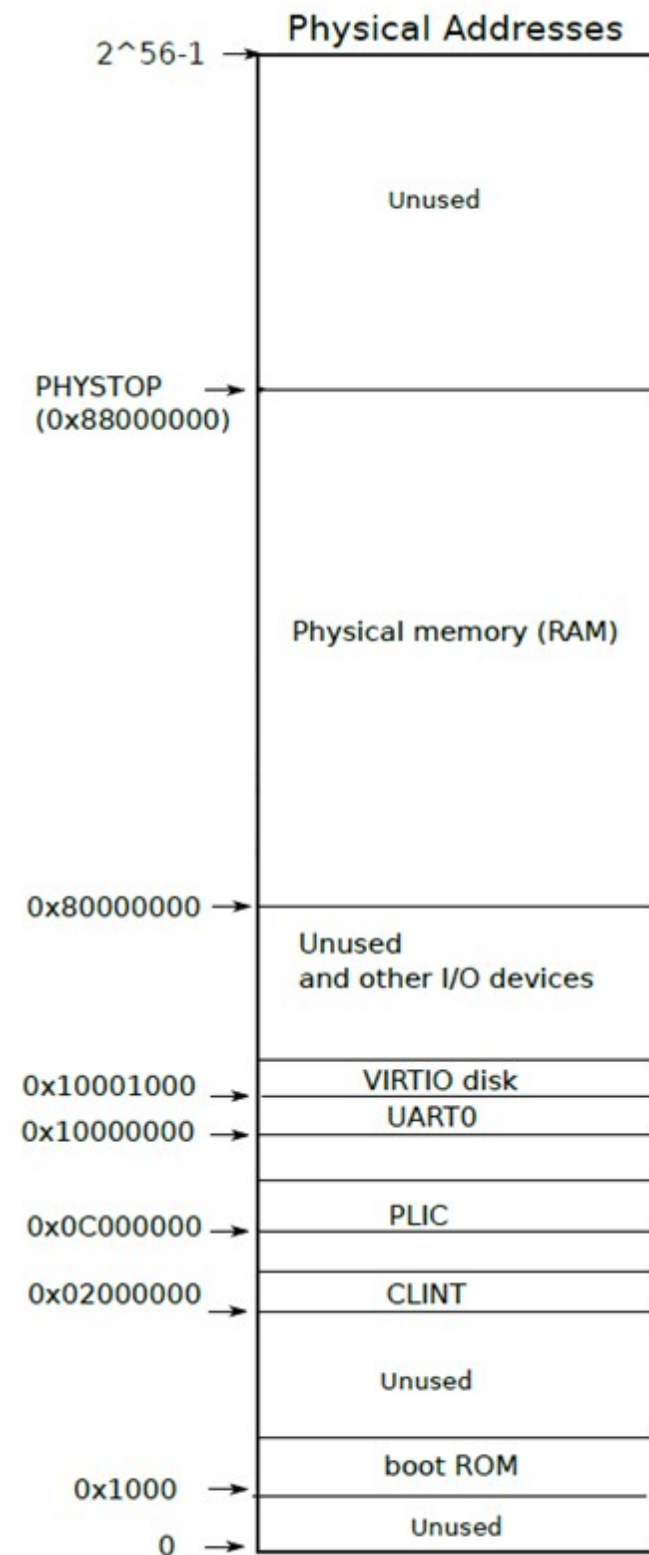


# Vykonávanie programu

- Ako sa CPU dostane k ďalšej inštrukcii?
  - Špeciálny register CPU, ktorý sa zvyšuje po každej inštrukcii
  - Automaticky modifikovaný niektorými inštrukciami (volanie procedúry, návrat z procedúry, skok)



# Fyzické adresy: organizácia RISC-V



prevzaté a upravené z

<https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf>

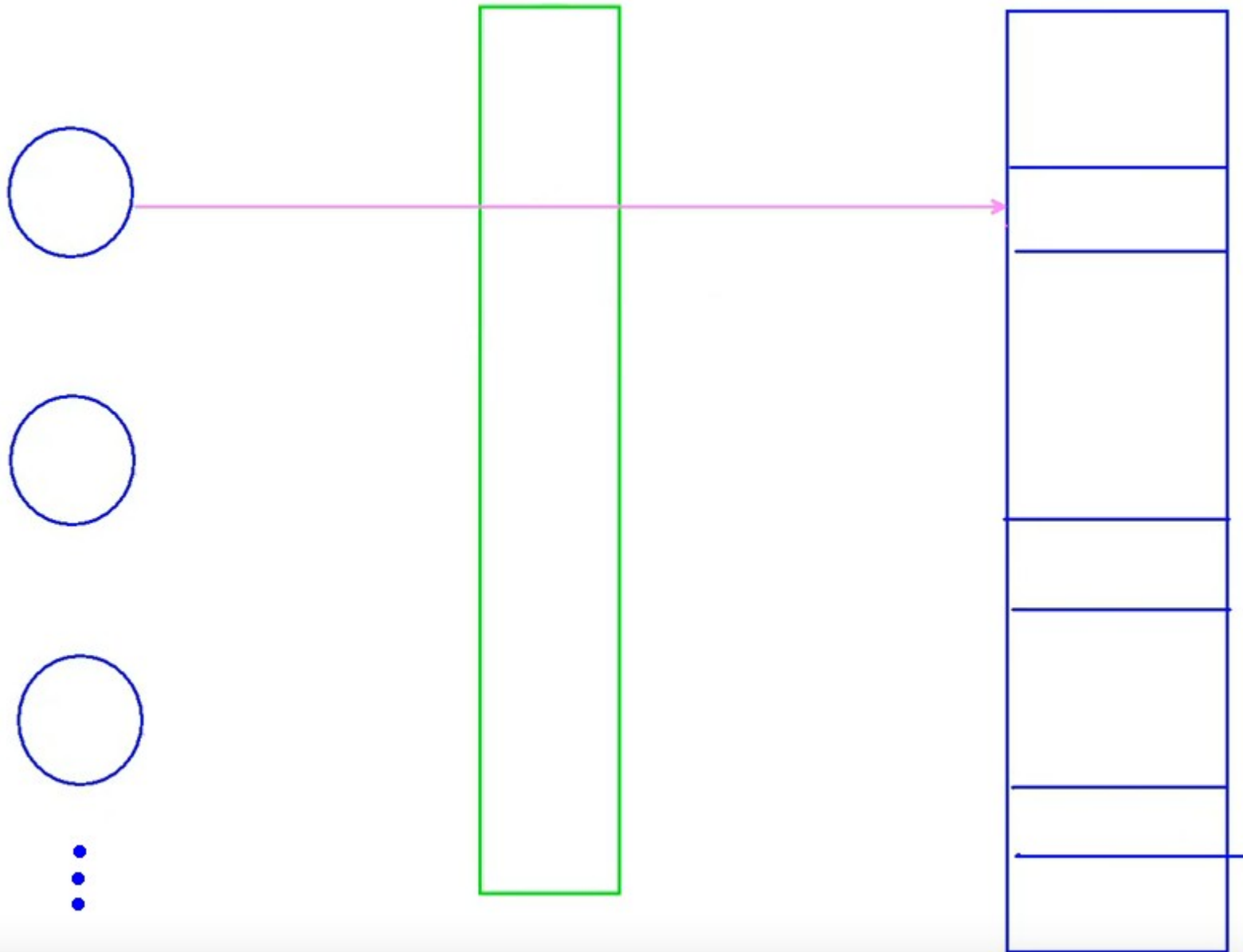
# Opakovanie

- Dve hw technológie umožňujúce izoláciu procesov od jadra a vzájomne medzi sebou
1. Systémové volania (umožňujú procesom presne definovaným spôsobom vykonať obsluhu služby v jadre OS)
  2. Virtuálna pamäť (vytvára zdanie vlastnej pamäte pre proces; znemožňuje zasahovať do pamäte iného procesu alebo jadra)

# Pamät'

- Majme v pamäti umiestnené procesy (a kód OS)
- Nech sa v nejakej aplikácii nachádza chyba, ktorá spôsobuje občasné zapísanie na náhodnu adresu v RAM-ke
- Ako zabezpečíme, aby sa neprepísali údaje/kód nejakej aplikácie alebo jadra OS?

# Pamät'



# Adresný priestor

- Izoláciou prístupu procesu k pamäťovému priestoru
- Každý proces bude mať “vlastnú” pamäť
- Čo to znamená?
  - Môže do nej zapisovať a čítať z nej
  - Nemôže zapisovať a čítať z pamäte iného procesu alebo jadra OS
- Otázka: ako vtesnať (multiplexovať) viacero takýchto samostatných pamäťových boxov do jednej RAM-ky a zachovať izoláciu medzi nimi?

# Adresný priestor

4 GB

4 GB

4 GB

□ 128 MB  
RAM  
RISC-V



# Adresný priestor

- Jestvujú viaceré možnosti, najrozšírenejšie sú

1. Segmentácia

2. Stránkovanie

3. Kombinácia seg+str alebo str+seg

# Domáca úloha

- Prečítať kapitolu 2
- *Operating system organization*
- <https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf>

MIT ;)