

OS MMXXIV

MIT ;)

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

# Prerušená

# Téma

- Stlačenie klávesy
- Pohyb myšou
- Tik časovača
- Údaje pripravené na vstupe sieťovej karty
- ...
- HW si vyžaduje OKAMŽITÚ pozornosť!

# Téma

- CPU musí
  - Odložiť aktuálnu činnosť (uložiť aktuálny stav)
  - Obslúžiť hw (obslúžiť prerušenie)
  - Obnoviť vykonávanie činnosti pred prerušením
- Na spracovanie prerušení na RISC-V sa používa ten istý mechanizmus ako pre
  - Systémové volania (*syscalls*)
  - Výnimky (*exceptions*)

# Komplikácie prerušení

- Prerušená sú **asynchrónne**
- Viac **konkurentne** vykonávaných vecí
- Programovanie **zariadení**

# Komplikácie prerušení

- Prerušená sú **asynchrónne**
  - Kód vykonávaný na CPU pred príchodom prerušená nijako nesúvisí s prerušením! Nie je medzi ním a prerušením žiadna kauzalita
  - Kód obsluhy prerušená nebeží v „kontexte“ procesu (v xv6 nemá zmysel využívať `myproc()`)
- Viac **konkurentne** vykonávaných vecí
  - CPU vykonáva kód, zároveň sa niečo deje na zariadení
- Programovanie **zariadení**
  - Môže byť značne zložitá naprogramovať obsluhu

# Zdroje prerušení



# Zdroje prerušení

- Vodiče zo zariadení napojené na špeciálnu zbernicu (bud' priamo do CPU alebo cez čip, ktorý ďalej spracúva zdroj prerušenía)
- Na SiFive základnej doske (RISC-V CPU) prerušenía zariadení idú cez obvod PLIC

# Zdroje prerušení

- Vodiče zo zariadení napojené na špeciálnu zbernicu (bud' priamo do CPU alebo cez čip, ktorý ďalej spracúva zdroj prerušenia)
- Na SiFive základnej doske (RISC-V CPU) prerušenia zariadení idú cez obvod PLIC
- PLIC ďalej smeruje vzniknuté prerušenie na to jadro CPU, ktoré môže prerušenie obslúžiť
  - CPU môže mať vypnuté spracovanie prerušení
  - Ak nie je žiadne CPU dostupné, PLIC uchováva prerušenie, pokým nejaké CPU nezapne obsluhu

# Mechanismus obsluhy prerušení

# Mechanismus obsluhy prerušení

- Prerušení informuje jádro o tom, že nějaký hw vyžaduje pozornost
- Ovládač (kód v jádře) vie, ako obsluhu zariadenia uskutočniť

# Mechanizmus obsluhy prerušení

- Prerušení informuje jadro o tom, že nejaký hw vyžaduje pozornosť
- Ovládač (kód v jadre) vie, ako obsluhu zariadenia uskutočniť
- Najjednoduchšia obsluha je priame volanie ovládača z obsluhy prerušenía (tak to robí xv6), ale je možná aj sofistikovanejšia schéma – pre obsluhu sa vytvorí a naplánuje vlastné vlákno jadra, prípadne sa obsluha viacerých prerušení spojí do jednej, atď.

# Mechanismus obsluhy prerušení

- Obsluha prerušenía NEBEŽÍ v kontexte procesu
- Čo to znamená pre xv6
  - `myproc()` môže vrátiť 0
  - `copyin()`, `copyout()` sa nedajú použiť
  - Prečo?

# Programovanie zariadenia

- Zväčša sa používa mapovanie pamäte

# Programovanie zariadenia

- Zväčša sa používa mapovanie pamäte
- Pomocou virtuálnych adries je možné pristupovať priamo k interným registrom (pamäti) samotného zariadenia
- Priamo sa používajú inštrukcie `load/store`



# Programovanie zariadenia

- Zväčša sa používa mapovanie pamäte
- Pomocou virtuálnych adries je možné pristupovať priamo k interným registrom (pamäti) samotného zariadenia
- Priamo sa používajú inštrukcie `load/store`
- Programovanie UART vid' napr. na: [byterunner.com/16550.html](http://byterunner.com/16550.html)

Prípadová štúdia xv6: \$ Is

# Prípadová štúdia xv6: \$ ls

- Výpis znaku \$ na konzolu
  - Ovládač pošle znak do FIFO **odosielacej** fronty UART zariadenia
  - UART vygeneruje prerušenie, keď sa znak pošle, čím informuje ovládač, že môže poslať ďalší znak

# Prípadová štúdia xv6: \$ ls

- Výpis znaku \$ na konzolu
  - Ovládač pošle znak do FIFO **odosielacej** fronty UART zariadenia
  - UART vygeneruje prerušenie, keď sa znak pošle, čím informuje ovládač, že môže poslať ďalší znak
- Načítanie a výpis 'ls'
  - Používateľ stlačí klávesu, čo spôsobí prerušenie UART
  - Ovládač načíta znak z FIFO **prijímacej** fronty UART

Ako jadro rozozná zariadenia

# Ako jadro rozozná zariadenia

- Každé zariadenie má jedinečné číslo zdroja IRQ (*Interrupt ReQuest*)
- IRQ je definované hw platformou (medzi platformami sa zväčša IRQ čísla líšia)
  - V Qemu má UART0 pridelené IRQ 10 (vid' `kernel/memlayout.h`)
  - Na doske SiFive má UART0 iné IRQ číslo

# Podpora prerušení na RISC-V CPU

# Podpora prerušení na RISC-V CPU

- `sie` (*supervisor interrupt enable register*)
  - bity pre sw prerušenie, externý zdroj a časovač
- `sip` (*supervisor interrupt pending register*)
  - bity pre sw prerušenie, externý zdroj a časovač
- `sstatus` (*supervisor status register*)
  - jeden bit určujúci, či sú prerušenia zapnuté



# Podpora prerušení na RISC-V CPU

- *scause* (*supervisor cause register*)
  - Číslo prerušenia
- *stvec* (*supervisor trap vector register*)
  - Adresa kódu obsluhy prerušení
- *mideleg* (*machine interrupt delegate register*)
  - Všetky výnimky (teda aj prerušenia) sa štandardne obsluhujú obsluhou v **M-móde**
  - Register *mideleg* umožňuje nastaviť automatické smerovanie prerušení do obsluhy v **S-móde**

# Inicializácia prerušení v xv6

kernel/start.c start()

w\_sie(r\_sie() | SIE\_SEIE|SIE\_STIE|SIE\_SSIE)

kernel/main.c main()

consoleinit()

uartinit()

plicinit()/plicinithart()

scheduler()

intr\_on()

w\_sstatus(r\_sstatus() | SSTATUS\_SIE)

# Zobrazenie '\$'

- `user/init.c main()`
  - Init otvára fd 0, 1, 2 pre konzolový vstup/výstup
  - Shell ich pomocou mechanizmu `fork()` zdedí
- Všetky zariadenia sa v systémoch typu UNIX interpretujú ako súbory
  - `printf()` → `putc()` → `write()`

# Zobrazenie '\$'

`sys_write()`

`filewrite()`

`consolewrite()` v `kernel/console.c`

`uartputc()`

- Vloženie znaku do FIFO UART
- Návrat do *userspace*
- Zároveň v tom istom čase UART posiela znak na konzolu

# Zobrazenie '\$'

- Shell v `getcnd()` vyvolal `sys_read()`, čaká na vstup
- UART po dokončení posielania znaku na konzolu vygeneruje prerušenie
- PLIC posunie prerušenie nejakému jadru CPU
- Čo urobí CPU?

Čo robí CPU pri prijatí prerušenia?

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE



# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v \$sstatus

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v \$sstatus
5. Nastav \$scause podľa zdroja prerušenia

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v \$sstatus
5. Nastav \$scause podľa zdroja prerušenia
6. Nastav mód CPU na *supervisor*

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v \$sstatus
5. Nastav \$scause podľa zdroja prerušenia
6. Nastav mód CPU na *supervisor*
7. Skopíruj \$stvec do \$pc

# Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj \$pc do \$sepc
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v \$sstatus
5. Nastav \$scause podľa zdroja prerušenia
6. Nastav mód CPU na *supervisor*
7. Skopíruj \$stvec do \$pc
8. Pokračuj vykonávaním inštrukcie podľa \$pc

# Čo robí CPU pri prijatí prerušenia?

- `$stvec` obsahuje buď adresu `kernelvec()` alebo `uservec()` (podľa toho, či je prerušenie vyvolané z *user* alebo *kernel* priestoru)
- Rovnaký mechanizmus sa používa aj pre výnimky a inštrukciu `ecall` (systémové volanie)

# Čo robí CPU pri prijatí prerušenia?

kerneltrap()/usertrap() volajú devintr()

ak ide o externé prerušenie

plic\_claim() zistí, o ktoré zariadenie ide

ak UART, uartintr()

pokým je znak na vstupe, vypíš ho

pošli na výstup aj znaky z vyrovn. pamäte

plic\_complete()

return z kernelvec()/uservec() obnoví prerušené  
vykonávanie kódu



# Viaceré prerušenia súčasne

- Čo keď sa v jednom čase vyskytne viacero prerušení?

# Viaceré prerušenia súčasne

- Čo keď sa v jednom čase vyskytne viacero prerušení?
- PLIC zabezpečuje, že každé zariadenie môže vygenerovať iba 1 prerušenie, pokiaľ nie je obsluha dokončená
- To znamená, že súčasne sa môžu vyskytnúť prerušenia od rôznych zariadení

# Viaceré prerušenia súčasne

- PLIC dokáže pridelit' (alebo skôr CPU si dokáže prevziať od PLIC vo funkcii `plic_claim()`) spracovanie prerušenia rôznym jadrám CPU

# Viaceré prerušenia súčasne

- PLIC dokáže pridelit' (alebo skôr CPU si dokáže prevziať od PLIC vo funkcii `plic_claim()`) spracovanie prerušenia rôznym jadrám CPU
- Takže spracovanie viacerých prerušení MÔŽE prebiehať súčasne! (t. j. **paralelne**)

# Viaceré prerušenia súčasne

- PLIC dokáže pridelit' (alebo skôr CPU si dokáže prevziať od PLIC vo funkcii `plic_claim()`) spracovanie prerušenia rôznym jadrám CPU
- Takže spracovanie viacerých prerušení MÔŽE prebiehať súčasne! (t. j. **paralelne**)
- Ak žiadne jadro CPU neprevezme prerušenie na spracovanie, prerušenie ostáva nespracované (angl. *pending*), pokým ho niektoré jadro nespracuje

# Prerušená a konkurentnosť

- Prerušená vnášajú problematiku viacerých typov konkurencie (angl. *concurrency*) vykonávania činnosti

# Prerušená a konkurentnosť

- Prerušená vnášajú problematiku viacerých typov konkurencie (angl. *concurrency*) vykonávania činnosti
1. Medzi zariadením a CPU (problém producent/konzument)

# Prerušená a konkurentnosť

- Prerušená vnášajú problematiku viacerých typov konkurencie (angl. *concurrency*) vykonávania činnosti
  1. Medzi zariadením a CPU (problém producent/konzument)
  2. Prerušenie používateľského programu OK, ale čo prerušenie kódu jadra? Napr. `userret()` (riešenie – vypínanie prerušení, aby sme dosiahli atomicitu operácie)

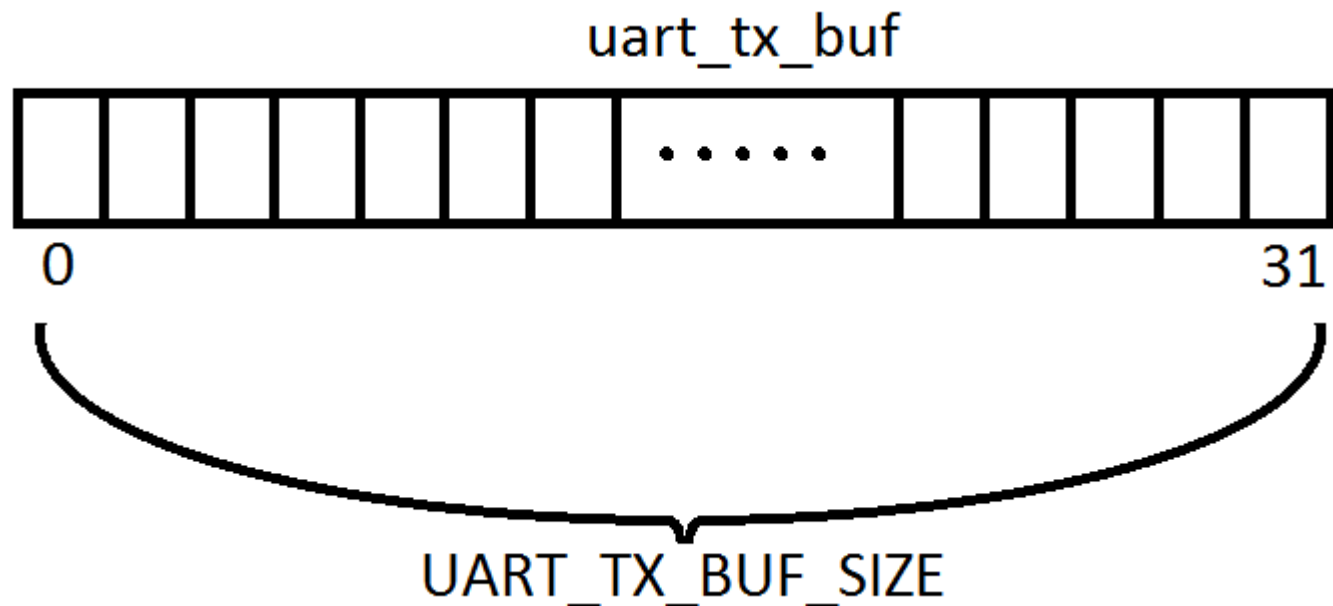


# Prerušená a konkurentnosť

- Prerušená vnášajú problematiku viacerých typov konkurencie (angl. *concurrency*) vykonávania činnosti
  1. Medzi zariadením a CPU (problém producent/konzument)
  2. Prerušenie používateľského programu OK, ale čo prerušenie kódu jadra? Napr. `userret()` (riešenie – vypínanie prerušení, aby sme dosiahli atomicitu operácie)
  3. Paralelné vykonávanie rôznych častí kódu využívajúce tú istú pamäť (riešenie – zámky)

# 1. Producent/konzument

- Napríklad vypisovanie na monitor
  - Shell je producent
  - Zariadenie UART je konzument



# 1. Producent/konzument

- `sys_write()` → ... → `uartputc()` → `uartstart()`
  - Vkladá do `uart_tx_buf`
  - Ak je *buf* plný, čaká (stav procesu *SLEEPING*)
  - **Beží v kontexte procesu!!!** (systémové volanie)
- `devintr()` → ... → `uartintr()` → `uartstart()`
  - Vyberá z `uart_tx_buf` a posiela na zariadenie
  - Ak je *buf* prázdny, nič neurobí
  - Budí z čakania producentov!
  - Vyvolané z `devintr()`, **nebeží v kontexte procesu!!!**

# 1. Producent/konzument

- Už vieme, ako sa vypíše prompt '\$ '
- Teraz sa pozrieme na načítavanie znakov (napr. 'ls')

# 1. Producent/konzument

- Už vieme, ako sa vypíše prompt '\$ '
- Teraz sa pozrieme na načítavanie znakov (napr. 'ls')
- Načítanie znakov z klávesnice
  - *shell* je konzument (`sys_read()`)
  - klávesnica je producent (pri stlačení sa generuje hw prerušenie)
  - Príslušný kód xv6 v `kernel/console.c`

# 1. Producent/konzument

- `sys_read()` → `fileread()` → `consoleread()`
  - Využíva `cons.buf`
  - Ak je *buffer* prázdny, proces sa uspí
  - **Volá sa v kontexte procesu!!!**

# 1. Producent/konzument

- `sys_read()` → `fileread()` → `consoleread()`
  - Využíva `cons.buf`
  - Ak je *buffer* prázdny, proces sa uspí
  - **Volá sa v kontexte procesu!!!**
- `devintr()` → ... → `uartintr()` → `consoleintr()`
  - Vždy **mimo kontext procesu!!!**
  - Vkladá do `cons.buf` (čo ak je *buf* plný?)
  - Budí konzumentov čakajúcich na vstup z klávesnice
    - Za akej podmienky nastane prebudenie?

## 2. Prerušenie preruší bežiaci kód

- Čo v prípade, keď nejaké prerušenie preruší vykonávaný kód?



## 2. Prerušenie preruší bežiaci kód

- Čo v prípade, keď nejaké prerušenie preruší vykonávaný kód?
- Majme napríklad kód, ktorý alokuje na zásobníku miesto a uloží tam návratovú adresu:
  1. `addi sp, sp, -48`
  2. `sd ra, 40(sp)`
- Môže sa vykonať nejaký kód MEDZI riadkami 1 a 2?

## 2. Prerušenie preruší bežiaci kód

- Čo v prípade, keď nejaké prerušenie preruší vykonávaný kód?
- Majme napríklad kód, ktorý alokuje na zásobníku miesto a uloží tam návratovú adresu:
  1. `addi sp, sp, -48`
  2. `sd ra, 40(sp)`
- Môže sa vykonať nejaký kód MEDZI riadkami 1 a 2?
  - Áno, obsluha prerušení! Napríklad časovač, `uart...`

## 2. Prerušenie preruší bežiaci kód

- Čo „hrozí“ v takom prípade používateľskému procesu?

## 2. Prerušenie preruší bežiaci kód

- Čo „hrozí“ v takom prípade používateľskému procesu?
  - Vykonávanie obsluhy pobeží v priestore jadra
  - Stav používateľského programu bude obnovený v tej podobe, v akej bol pri vyvolaní výnimočného stavu spôsobeného prerušením

## 2. Prerušenie preruší bežiaci kód

- Čo „hrozí“ v takom prípade používateľskému procesu?
  - Vykonávanie obsluhy pobeží v priestore jadra
  - Stav používateľského programu bude obnovený v tej podobe, v akej bol pri vyvolaní výnimočného stavu spôsobeného prerušením
- Čo „hrozí“ kódu jadra? Je to podobne jednoduché ako v prípade používateľského kódu?

## 2. Prerušenie preruší bežiaci kód

- Majme nasledovné kódy jadra: bežiaci kód jadra a kód vykonávaný obsluhou prerušenia

bežiaci\_kód:

```
x = 0
```

```
if (x == 0)
```

```
    f()
```

obsluha\_prerušenia:

```
x = 1
```

- Ako je to s volaním funkcie **f()**?

## 2. Prerušenie preruší bežiaci kód

- Aby sa funkcia  $f()$  vždy zaručene vykonala, nesmie nastať prerušenie!
- Prečo?

## 2. Prerušenie preruší bežiaci kód

- Aby sa funkcia  $f()$  vždy zaručene vykonala, nesmie nastať prerušenie!
- Prečo?
  - Obsluha prerušenia môže byť vyvolaná iba na hranici inštrukcií, nie uprostred vykonávania nejakej inštr.
  - Predpokladáme, že nastavenie premennej ( $x = 0$ ) a testovanie premennej ( $\text{if } x == 0$ ) sú minimálne 2 inštrukcie
  - Ak je na CPU povolená obsluha prerušení, medzi týmito dvoma riadkami kódu sa môže spustiť obsluha prerušenia, ktorá zmení hodnotu premennej  $x$ !



## 2. Prerušenie preruší bežiaci kód

- Aby sa funkcia  $f()$  vždy zaručene vykonala, nesmie nastať prerušenie!
- Ako zabezpečiť „atomické“ vykonanie nejakého bloku inštrukcií (riadkov kódu)?

## 2. Prerušenie preruší bežiaci kód

- Aby sa funkcia  $f()$  vždy zaručene vykonala, nesmie nastať prerušenie!
- Ako zabezpečiť „atomické“ vykonanie nejakého bloku inštrukcií (riadkov kódu)?
- Vypnutím spracovania inštrukcií
  - Vid' funkcia `kernel/riscv.h: intr_off()`
  - `w_sstatus(r_sstatus() & ~SSTATUS_SIE);`

## 2. Prerušenie preruší bežiaci kód

- Aby sa funkcia  $f()$  vždy zaručene vykonala, nesmie nastať prerušenie!
- Ako zabezpečiť „atomické“ vykonanie nejakého bloku inštrukcií (riadkov kódu)?
- Vypnutím spracovania inštrukcií
  - Vid' funkcia `kernel/riscv.h: intr_off()`
  - `w_sstatus(r_sstatus() & ~SSTATUS_SIE);`
- Kde v kóde sa táto funkcia využíva? Môže jadro obsluhovať prerušenie v kóde trampolíny?

## 2. Prerušenie preruší bežiaci kód

- Vráťme sa k príkladu načítania vstupu z klávesnice – v akej funkcii sa nachádza kód jadra?

## 2. Prerušenie preruší bežiaci kód

- Vráťme sa k príkladu načítania vstupu z klávesnice. V akej funkcii sa nachádza kód jadra?  
\$ (shell je v `sys_read()`, aby získal vstup z kláv.)

## 2. Prerušenie preruší bežiaci kód

- Vráťme sa k príkladu načítania vstupu z klávesnice. V akej funkcii sa nachádza kód jadra?

\$ (shell je v `sys_read()`, aby získal vstup z kláv.)

`usertrap()` – vyvolané systémovým volaním (`ecall`)

`w_stvec((uint64)kernelvec) !!!!!!!`

...

`consoleread()`

`sleep()`

`scheduler()`

`intr_on()` // kde sa volá?

## 2. Prerušenie preruší bežiaci kód

\$ I (používateľ stlačil klávesu 'I', UART prerušenie)

## 2. Prerušenie preruší bežiaci kód

\$ I (používateľ stlačil klávesu 'I', UART prerušenie)

kernelvec() – pretože \$stvec obsahuje túto adresu!!!

– na aký zásobník sa uložia registre CPU?



## 2. Prerušenie preruší bežiaci kód

kernelvec()

kerneltrap()

devintr()

uartintr()

c = uartgetc()

consoleintr(c)

obsluha špeciálnych sekvencií (ctrl)

poslanie znaku 'l' na výstup (uartputc\_sync() v consputc())

vloženie c do cons.buf

zobudenie konzumentov čakajúcich v consoleread()

návrat z consoleintr()

návrat z uartintr()

návrat z devintr()

návrat z kerneltrap()

obnovenie stavu registrov CPU

sret

## 2. Prerušenie preruší bežiaci kód

kernelvec()

kerneltrap()

devintr()

uartintr()

c = uartgetc()

consoleintr(c)

obsluha špeciálnych sekvencií (ctrl)

poslanie znaku 'l' na výstup (uartputc\_sync() v consputc())

vloženie c do cons.buf

zobudenie konzumentov čakajúcich v consoleread()

návrat z consoleintr()

návrat z uartintr()

návrat z devintr()

návrat z kerneltrap()

obnovenie stavu registrov CPU

sret

**KAM sa vráti tok vykonávania inštrukciou sret?**

## 2. Prerušenie preruší bežiaci kód

- Kam sa vráti tok riadenia inštrukciou `sret`?
- Tam, kde prišlo k prerušeniu bežiaceho kódu pri príchode prerušenia
- V našom prípade to je cyklus vo funkcii `scheduler()`
- Vid' pomocou ``make CPUS=1 qemu-gdb``

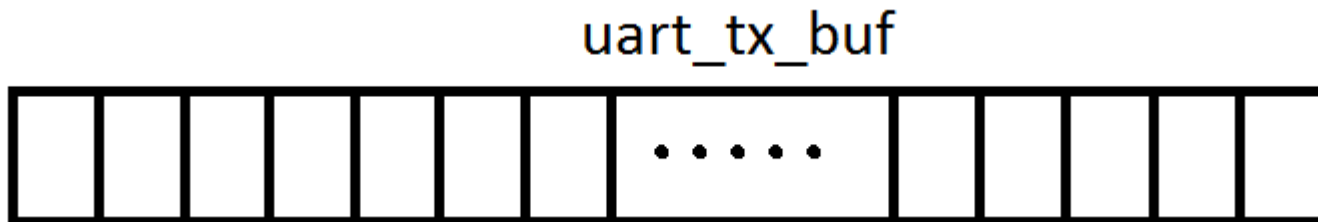
# 3. Konkurentný prístup k údajom

- Poslednou úrovňou konkurencie je prístup k tým istým pamäťovým oblastiam konkurentne/paralelne z rôznych tokov vykonávania kódu

# 3. Konkurentný prístup k údajom

- Príklad

- majme dva používateľské programy, každý sa vykonáva na samostatnom jadre CPU
- nech sa oba programy v tom istom časovom okamihu pokúsia vykonať `printf("ahoj %d\n", pid)`
- kernel/uartc.c: `uartputc()`



# 3. Konkurentný prístup k údajom

- Riešenie – použitie zámkov (angl. *lock*)
- Vid' funkcie `acquire()` a `release()`  
v `kernel/uart.c: uartputc()`

# 3. Konkurentný prístup k údajom

- Riešenie – použitie zámkov (angl. *lock*)
- Vid' funkcie `acquire()` a `release()`  
v `kernel/uart.c: uartputc()`
- Opakovanie: zámkový systém sa využíva na vynútenie vykonávania istej časti kódu iba jediným tokom riadenia

# Vývoj prerušení

- Kedysi bol tento prístup navrhnutý a vyvinutý, aby urýchlil činnosť CPU
- V súčasnosti sú prerušenia príliš pomalé pre niektoré zariadenia
  - Napr. gigabit ethernet dokáže preniesť 1.5 milióna paketov za sekundu
  - To je viac než 1 za mikrosekundu
  - Spracovanie prerušenia trvá rádovo v mikrosekundách
  - Ako potom takéto zariadenia obsluhovať?



# Vývoj prerušení

- Ak je obsluha prerušenía príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa“, tzv. *polling*

# Vývoj prerušení

- Ak je obsluha prerušenía príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa“, tzv. *polling*
- CPU neustále v cykle kontroluje, či niektoré zariadenie nevyžaduje pozornosť
  - Toto čakanie v cykle je neefektívne (nevyužije sa CPU naplno), ak je zariadenie pomalé
  - Jeden príklad v xv6: `uartputc_sync()`
  - Ale ak je zariadenie mega super rýchle, šetrí sa čas CPU (žiadna zmena kontextu atď.)

# Vývoj prerušení

- Ak je obsluha prerušenía príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa“, tzv. *polling*
- Prečo alebo kedy používať túto techniku?

# Vývoj prerušení

- Ak je obsluha prerušenía príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa“, tzv. *polling*
- Prečo alebo kedy používať túto techniku?
- Ak je generovanie udalostí tak rýchle, že musia neustále čakať na spracovanie – vtedy nie je nutné o vygenerovaní udalosti informovať, pretože vieme, že vždy je k dispozícii nejaká udalosť čakajúca na spracovanie

# Prerušená vs *polling*

# Prerušená vs *polling*

- Pre zariadenia, ktoré chríia udalosti – *polling*
- Pre pomalé zariadenia (typu klávesnica) – *irq*

# Prerušená vs *polling*

- Pre zariadenia, ktoré chrliá udalosti – *polling*
- Pre pomalé zariadenia (typu klávesnica) – *irq*
- Automatické prepínanie medzi oboma módmí činností
- Presmerovanie spracovania prerušení do používateľského priestoru
  - Výpadky stránok
  - Obsluha nejakých zariadení (napr. disk, sieť)

# Domáce čítanie a pozeranie

## Chapter 5

### Interrupts and device drivers

xv6: a simple, Unix-like teaching operating system

<https://www.youtube.com/watch?v=Fcjychg4Tvk>