

OS MMXXI

MIT ;)

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

Zámky

# Prečo nás má táto téma zaujímať?

- Používateľ spúšťa viacero programov naraz
- Jadro by malo vedieť obsluhovať viacero systémových volaní súčasne
- Takže viacero tokov vykonávania súčasne môže pristupovať k rôznym dátovým štruktúram

# Prečo nás má táto téma zaujímať?

- Používateľ spúšťa viacero programov naraz
- Jadro by malo vedieť obsluhovať viacero systémových volaní súčasne
- Takže viacero tokov vykonávania súčasne môže pristupovať k rôznym dátovým štruktúram
- Zámky nám pomáhajú túto situáciu zvládať (spoločne to zvládneme?)
- Zámky znižujú efektivitu paralelného vykonávania (nie vždy to spoločne zvládneme!)

# Prečo nás má táto téma zaujímať?

- Ukážka: zmažme `acquire()`/`release()` v `kfree()`
  - xv6 naštartuje normálne
  - Zbehnú všetky `usertests`, okrem stratenia nejakých pamäťových rámcov... ``usertests reparent2``
- Prečo príde k strate rámcov? Obrázok so súbehom
- Potrebujeme zámkový systém pre zabezpečenie správnosti fungovania, ale zároveň strácame výkon (`kfree()` beží sériovo)



# ADT Lock

- Už sme o ňom hovorili na minulej prednáške
- lock l
- acquire(l)
- $x = x + 1$  // KO (*critical section*)
- release(l)



# ADT Lock

- Ak viacero vlákien (súčasne) vyvolá `acquire( l )`
  - Iba jednému vláknu sa podarí z funkcie vrátiť a pokračovať vo vykonávaní ďalšieho kódu
  - Ostatné musia čakať na uvoľnenie (`release( )`)

# ADT Lock

- Ak viacero vlákien (súčasne) vyvolá `acquire( l )`
  - Iba jednému vláknu sa podarí z funkcie vrátiť a pokračovať vo vykonávaní ďalšieho kódu
  - Ostatné musia čakať na uvoľnenie (`release( )`)
- Zámok nie je automaticky spätý so žiadnou premennou – je na programátorovi, aby samotným kódom určil, na čo bude zámok slúžiť

# Kedy použiť v kóde zámok?

- Ak viac tokov vykonávania môže súčasne pristúpiť k spoločnému pamäťovému miestu a aspoň jeden z týchto tokov vykonáva operáciu zápisu na danom pamäťovom mieste
- Nikdy v programe nepristupuj k zdieľaným údajom bez použitia správneho zámku (každý údaj môže mať vlastný zámok)

# Automatické uzamykanie

- Vyššie programovacie jazyky poskytujú ADT s automatickou podporou uzamykania
- Tento prístup nie je možné použiť na všetky prípady použitia

# Automatické uzamykanie

- Tento prístup nie je možné použiť na všetky prípady použitia
  - Napr. `rename("dir1/file1", "dir2/file2")`
    - `lock(dir1), erase(file1), unlock(dir1)`
    - `lock(dir2), add(file2), unlock(dir2)`
  - Problém: súbor istý časový interval nejestvuje
  - `rename( )` musí byť atomická operácia

# Automatické uzamykanie

- Tento prístup nie je možné použiť na všetky prípady použitia
  - Napr. `rename("dir1/file1", "dir2/file2")`
    - `lock(dir1), erase(file1), unlock(dir1)`
    - `lock(dir2), add(file2), unlock(dir2)`
  - Problém: súbor istý časový interval nejestvuje
  - `rename( )` musí byť atomická operácia:
    - `lock(dir1), lock(dir2)`
    - `erase(file1), add(file2)`
    - `unlock(dir2), unlock(dir1)`
- Programátor musí mať kontrolu nad použitím

# Na čo sú zámký dobré

1. Aby sme sa vyhli stratám údajov pri ich aktualizácii

# Na čo sú zámky dobré

1. Aby sme sa vyhli stratám údajov pri ich aktualizácii
2. Aby sme dokázali z viackrokovej operácie urobiť atomickú operáciu (ukryť interný nekonzistentný medzistav)



# Na čo sú zámky dobré

1. Aby sme sa vyhli stratám údajov pri ich aktualizácii
2. Aby sme dokázali z viackrokovej operácie urobiť atomickú operáciu (ukryť interný nekonzistentný medzistav)
3. Vo všeobecnosti na zachovanie invariantov pri rôznych ADT
  - Invariant je platný pred začatím operácie
  - Zámky “ukryjú” dočasné porušenie invariantu
  - Na konci operácie pred uvoľnením zámku sa platnosť invariantu obnoví

# Uviaznutie

- Majme verziu `rename ( )` s 2 zámkami

vlákno A:

```
rename("a/1","b/2")
```

```
lock(a)
```

```
lock(b)
```

...

vlákno B:

```
rename("b/3","a/4")
```

```
lock(b)
```

```
lock(a)
```

...

- Aké môže byť riešenie?

# Uviaznutie

- Uzamykanie v tom istom poradí
  - Musíme vedieť, o ktoré zámky sa jedná
  - Zoradiť ich, a až potom volať `acquire()`
  - Príliš komplexné riešenie
- Programátor2 musí poznať kód, význam jednotlivých zámkov a miesta ich použitia

# Zámky versus paralelné vykonávanie

- Zámky ZNEMOŽŇUJÚ paralelné vykonávanie

# Zámky versus paralelné vykonávanie

- Zámky ZNEMOŽŇUJÚ paralelné vykonávanie
- Na umožnenie (čiastočného) paralelizmu je často nutné rozdeliť dátovú množinu (ktorú zámok chráni) na menšie časti
- Každá menšia časť bude mať vlastný zámok
- Vid' úlohy cvičenia
- V zmysle delenia dátovej množiny hovoríme o granularite (hrubá versus jemná)

# Zámky versus paralelné vykonávanie

- Nájdenie optimálnej granularity je netriviálna úloha

# Zámky versus paralelné vykonávanie

- Nájdenie optimálnej granularita je netriviálna úloha
  - Celý FS / per adresár a súbor / per diskový blok
  - Celé jadro systému / každý subsystém / každý objekt

# Zámky versus paralelné vykonávanie

- Nájdenie optimálnej granularity je netriviálna úloha
  - Celý FS / per adresár a súbor / per diskový blok
  - Celé jadro systému / každý subsystém / každý objekt
- Úloha 1 z cvičenia
  - 1 zoznam voľných rámcov RAM rozdeliť na toľko zoznamov, koľko je CPU
  - Každé CPU bude využívať svoj zoznam
  - Cieľ: môžu pristupovať k zoznamom súčasne



# Granularita zámkov

- Ako nájsť správnu granularitu?

# Granularita zámkov

- Ako nájsť správnu granularitu?
- Návrh začneme jedným veľkým zámkom (tzv. *big lock*)
  - Menšia šanca uviaznutia
  - Menej dokazovania správnosti invariantov

# Granularita zámkov

- Ako nájsť správnu granularitu?
- Návrh začneme jedným veľkým zámkom (tzv. *big lock*)
  - Menšia šanca uviaznutia
  - Menej dokazovania správnosti invariantov
- Zmeriame efektivitu riešenia
  - Často riešenie s veľkými zámkami postačuje

# Granularita zámkov

- Ako nájsť správnu granularitu?
- Návrh začneme jedným veľkým zámkom (tzv. *big lock*)
  - Menšia šanca uviaznutia
  - Menej dokazovania správnosti invariantov
- Zmeriame efektivitu riešenia
  - Často riešenie s veľkými zámkami postačuje
- Jemnejšiu granularitu riešime, iba ak je to nutné (s ohľadom na zmeranú efektivitu riešenia)

# Implementácia zámkov

```
1 struct lock { int locked; }
2 acquire(l) {
3     while(1) {
4         if (l->locked == 0) {
5             l->locked = 1;
6             return;
7         }
8     }
9 }
```

# Implementácia zámkov

- Ako vyriešiť súbeh?
- Potrebujeme atomickú operáciu

# Implementácia zámkov

- Ako vyriešiť súbeh?
- Potrebujeme atomickú operáciu
  - Výmeny obsahu pamäťového miesta
  - Môže byť aj zložitejšia: CAS (*compare-and-swap*)

# Implementácia zámkov

- Ako vyriešiť súbeh?
- Potrebujeme atomickú operáciu
  - Výmeny obsahu pamäťového miesta
  - Môže byť aj zložitejšia: CAS (*compare-and-swap*)
- Atomická inštrukcia `swap(addr, reg)`
  1. lock addr (globálne zamkne adresu (zbernicu))
  2. `temp := *addr`
  3. `*addr := reg`
  4. `reg := temp`
  5. unlock addr



# Implementácia zámkov

- Implementácia ADT Spinlock pomocou CAS

# Implementácia zámkov

- Implementácia ADT Spinlock pomocou CAS

```
acquire(l) {  
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)  
        ;  
}
```

# Implementácia zámkov

- Implementácia ADT Spinlock pomocou CAS

```
acquire(l) {  
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)  
        ;  
}
```

- Ak `lk->locked` bolo rovné 1, inštrukcia znovu nastaví hodnotu 1; zároveň vráti hodnotu 1 (pôvodnú hodnotu pred zmenou)
- Ak `lk->locked` bolo rovné 0, inštrukcia nastaví hodnotu 1 a vráti hodnotu 0 (pôvodnú hodnotu pred zmenou)

# Poradie inštrukcií

- Doteraz sme predpokladali, že program sa vykonáva sekvenčne
  - T. j. majme sekvenciu ASM inštrukcií
  - Inštrukcie sa vykonávajú za sebou

# Poradie inštrukcií

- Doteraz sme predpokladali, že program sa vykonáva sekvenčne
  - T. j. máme sekvenciu ASM inštrukcií
  - Inštrukcie sa vykonávajú za sebou
- Toto nie je pravda
  - GCC môže optimalizáciou poprehadzovať inštr.
  - CPU môže pri vykonávaní poprehadzovať inštr.

# Poradie inštrukcií

Program 1:

locked = 1

x = x + 1

locked = 0

Program 2:

while(locked == 1)

....

locked = 1

x = x + 1

locked = 0

=====

locked = 1

locked = 0

x = x + 1

# Poradie inštrukcií

- Aby sme zabránili poprehadzovaniu inštrukcií narábajúcich s pamäťou (ukazateľmi), využívame ďalšiu gcc *intrinsic* funkciu `__sync_synchronize()`

# Poradie inštrukcií

- Aby sme zabránili poprehadzovaniu inštrukcií narábajúcich s pamäťou (ukazateľmi), využívame ďalšiu gcc *intrinsic* funkciu `__sync_synchronize()`
- Kompilátor nemôže presunúť inštrukcie pracujúce s pamäťou za volanie tejto “funkcie”
- Kompilátor by mal vygenerovať inštrukciu “*memory barrier*” pre cieľové CPU, aby ani CPU neprehodilo žiadnu pamäťovú inštrukciu pred bariérou za ňu



# Načo je dobrý ADT Spinlock

- Neustále vyťažuje na 100% CPU; je potrebný?
- Využíva sa pri čakaní na “krátky” čas
  - Krátky čas znamená kratšie, ako je režia nutná k preplánovaniu vlákna pri ADT Sleeplock
- Zároveň pri ADT Spinlock NESMIE prísť ku preplánovaniu
  - Pri možnosti preplánovania by mohlo nastať uviaznutie

# Uviaznutie pri držaní ADT Spinlock

- Počas `sched()` P1 drží *spinlock* L1
- Obnoví sa beh procesu P2, a ten sa pokúsi vykonať `acquire(L1)`
- Keďže `acquire()` beží s vypnutými prerušeniami
  - tak časovač nemôže doručiť prerušenie,
  - preto sa P2 nemôže vzdať CPU,
  - takže P1 nemôže byť naplánovaný na beh,
  - a teda nemôže byť uvoľnený zámok L1.
- Nastáva uviaznutie (*deadlock*)

# Načo je dobrý ADT Sleeplock

- Ak je čakanie dlhšie než réžia nutná k použitiu ADT Sleeplock
- Ak je nutné preplánovanie
- Čakajúce vlákno uvoľní CPU
- Zväčša sa čaká na externé udalosti (údaje zo zariadení)
- Klávesnica, disk, sieťová karta...

# Domáce čítanie

## Chapter 6

### Locking

xv6: a simple, Unix-like teaching operating system

Implementácia ADT Sleeplock samoštúdium