

OS MMXXI

MIT ;)

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

a

Peter Tomcsányi: Plánovanie
procesov a vlákien

Niektoré práva vyhradené v zmysle licencie Creative Commons

<http://creativecommons.org/licenses/by-nc-sa/3.0/>

Plánovanie procesov

Téma

Téma

- V súčasnosti sú viacjadrové CPU bežné
- Ale počet jadier je malý
- Používateľ chce spúšťať veľa programov
- Treba vymyslieť, ako malý počet jadier CPU zdieľať medzi veľkým množstvom procesov

Téma

- Zdieľanie by malo byť pre proces transparentné
- Zväčša sa používa abstrakcia – každému procesu sa vytvorí ilúzia procesora, ktorý má iba sám pre seba
- Ide o multiplexing procesov na hw výpočtového systému

Motivácia

- Prečo písať os s podporou behu viacerých úloh súčasne?

Motivácia

- Prečo písať os s podporou behu viacerých úloh súčasne?
- Požiadavka používateľov (hudba, net...)
- Požiadavka algoritmu (sito na prvočísla)
- Požiadavka efektivity (urýchlenie výpočtu)

Vlákno (*Thread*)

- Vlákno je abstrakcia na zjednodušenie programovania
- Vlákno = nezávislé sériové vykonávanie (registre, pc, zásobník)

Vlákno (*Thread*)

- Vlákno je abstrakcia na zjednodušenie programovania
- Vlákno = nezávislé sériové vykonávanie (registre, pc, zásobník)
- Dve hlavné stratégie nasadenia vlákien:
 1. Viac CPU, na každom CPU beží 1 vlákno
 2. Každé CPU “prepína” medzi viacerými vláknami (v 1 čase sa ovšem vykonáva iba 1 vlákno na 1 CPU)

Vlakno (Thread)

- Vlákna môžu, ale nemusia zdieľať pamäť
- Linux: vlákna užívateľského procesu zdieľajú pamäť
- Xv6:
 - Vlákna jadra: **zdieľajú** spoločnú pamäť
 - Vlákna *user* programu:
 - Každý *user* program pozostáva z 1 vlákna
 - Vlákna programov **NEzdieľajú** spoločnú pamäť

Multiplexing

- Multiplexing vytvára ilúziu podobne, ako je to v prípade pamäte pomocou stránkovania
- Ide o prepnutie kódu na procesore
- V xv6 sa prepnutie robí v 2 prípadoch
 - Mechanizmus s `sleep` – `wakeup`
 - Vynútené prepnutie procesu, ak nevyužíva `sleep`

Multiplexing

- Ako urobiť prepnutie jedného procesu na iný?
- Ako to urobiť tak, aby o tom proces “nevedel”? (Xv6 využíva prerušenie časovača)
- Ako urobiť prepnutie bezpečným, keď sa o to pokúšajú súčasne viaceré jadrá naraz?
- Ako správne uvoľniť pamäť ukončeného procesu (nemôže to urobiť on sám, vzhľadom na zásobník jadra...)

Multiplexing

- Ako sa vysporiadať s procesmi, ktoré robia iba výpočty a žiadne systémové volania?
- Čo má robiť procesor, keď plánovač nenájde žiaden vykonateľný proces?
- ...

Výpočtovo orientované procesy

- Každé CPU má časovač, ktorý v pravidelných intervaloch informuje o “tikoch”
- Jadro využíva prerušenie časovača na prerušenie behu výpočtovo orientovaného procesu
- Ide o tzv. “preemptívne” plánovanie (vynútené)
- Jestvuje aj “kooperatívne” plánovanie, pri ktorom sa samotný proces musí dobrovoľne vzdať procesora

Vlákná v xv6

- Brutálny obrázok vlákien xv6 č. 1
- Zásobník pre kód každého CPU
- `entry.S` → `start.c` → `main.c` → `scheduler()`
- Prvý user proces: `userinit()`

Vlákná v xv6

- Brutálny obrázok vlákien xv6 č. 1
- Každý proces má 2 vlákna
 - Užívateľské (kód *user* programu)
 - *Kernel* (systémové volania, obsluha prerušení)
- Všetky vlákna v *kernel* priestore zdieľajú VM (dôsledok: jadro xv6 je multivláknové, konkurentné)

Vlákná v xv6

- V prednáške používame pojem “proces”, “vlákno jadra” a “vlákno” ako synonymá
- Vo všeobecnosti sa pojmy “dávka”, “proces” a “vlákno” rozlišujú
 - Vlákno – najmenšia (nedeliteľná) jednotka toku riadenia (s ktorou môže manipulovať plánovač OS)
 - Proces – zoskupenie vlákien zdieľajúcich spoločný pamäťový priestor
 - Dávka – zoskupenie procesov so spoločnými charakteristikami (napr. terminál, vlastník)

Vlákná v xv6

- Brutální obrázok vláknien xv6 č. 2
- 2x `swtch()`
- Kontext obsluhy prerušení `devintr()`

Vlákná v xv6

- Brutálny obrázok vlákien xv6 č. 2
- TF (trapframe) obsahuje registre užívateľského programu
- CTX (context) obsahuje registre procesora, ktoré sa môžu medzi volaniami funkcií meniť (neobnovujú sa zo zásobníka)
 - ide o tzv. *callee saved* registre v RISC-V architektúre
 - *caller saved* registre ukladá C kód na zásobník

Vlákná v xv6

- Brutálny obrázok vlákien xv6 č. 2
- Prechod z kódu jedného procesu na kód iného procesu je v xv6 nepriamy
 - *user v.* → *kernel v.* (*user* registre sa uložia do TF)
 - *kernel* vlákno → *scheduler* vlákno (*kernel* registre sa uschovajú v CTX)
 - *scheduler* vlákno → *kernel* vlákno (*kernel* registre sa obnovia z CTX)
 - *kernel v.* → *user v.* (*user* registre sa obnovia z TF)

Vlákná v xv6

- Čo je v terminológii xv6 prepnutie kontextu (*context switch*)?
 - Zmena toku riadenia z jedného vlákna jadra na iné
 - Nejedná sa o zmenu *kernel* → *user* alebo *user* → *kernel*
- Zmena *user* vlákna na iné *user* vlákno nie je možná, nakoľko každý *user* program pozostáva z jediného vlákna

Cvičenie

- Prvá úloha cvičenia – rozšírenie *user* programov o viaceré vlákna
- Aktuálne je v xv6 mapovanie vlákien *user:kernel* 1:1
- Cieľom cvičenia je dosiahnuť stav n:1
 - Aké sú výhody viacerých vlákien?
 - Aké sú obmedzenia riešenia n:1?
- Vid' brutálny obrázok č. 3

Cvičenie

- Prvá úloha cvičenia – rozšírenie *user* programov o viaceré vlákna
- Prepínanie vlákien v *user* programoch je **kooperatívne**
- Samotné vlákno musí vyvolať funkciu `thread_yield()`, aby sa vzdalo procesora
 - Jadro nič “nevie” o vláknach implementovaných v *user* priestore

Vlákná plánovača

- Vždy jedno na 1 CPU
 - Zásobník `start.c:stack0`, kontext `proc.h:struct cpu`
- *Kernel* vlákno *user* programu
 - Vždy sa prepne do plánovača toho CPU, na ktorom dané vlákno beží
 - Plánovač hľadá na beh `RUNNABLE` vlákno
 - Nikdy sa nerobí priamy prechod medzi *kernel* vláknami *user* programov!

Vlákna plánovača

- Prečo použiť “medzistupeň” plánovača (keď na cvičení sa prepínanie medzi *user* vláknami robí priamo)?

Vlákná plánovača

- Prečo použiť “medzistupeň” plánovača (keď na cvičení sa prepínanie medzi *user* vláknami robí priamo)?
 - Zjednodušenie návrhu a kódu pri ukončovaní procesu (musí sa uvoľniť zásobník *kernel* vlákna!)
 - Plánovač neustále v cykle hľadá spustiteľné vlákno; ak také nenájde, “zaspí” (doručenie prerušenia ho “zobudí”)
 - Čo ak beží v systéme menej vlákien, ako je CPU? Aký zásobník použije obsluha zvyšných CPU?

Vlákná plánovača - invarianty

1. Jedno jadro CPU v jednom čase vykonáva iba jedno vlákno (buď *scheduler* alebo *kernel* vlákno *user* programu)
2. Buď vlákno beží na práve jednom jadre CPU, alebo sú jeho registre uchované v kontexte
3. Ak vlákno jadra nie je práve vykonávané, jeho kontext uchováva stav z volania `switch()`

Štruktúra proc v xv6

- $p \rightarrow \text{trapframe}$
- $p \rightarrow \text{context}$
- $p \rightarrow \text{kstack}$
- $p \rightarrow \text{state}$
- $p \rightarrow \text{lock}$

Štruktúra proc v xv6

- $p \rightarrow \text{trapframe}$: uchováva registre *user* vlákna
- $p \rightarrow \text{context}$: uchováva registre *kernel* vlákna
- $p \rightarrow \text{kstack}$: ukazuje na *kernel* zásobník vlákna
- $p \rightarrow \text{state}$: *running, runnable, sleeping, ...*
- $p \rightarrow \text{lock}$: chráni integritu $p \rightarrow \text{state}$ a iných

Exkurz *lock*

- Čo je zámok (*lock*)? Synchronizačný mechanizmus umožňujúci implementovať serializáciu (radenie za sebou)

Exkurz *lock*

- Čo je zámok (*lock*)? Synchronizačný mechanizmus umožňujúci implementovať serializáciu (radenie za sebou)
- Ako?
 - Nadobúda 2 stavy: odomknutý, zamknutý
 - Manipulácia so stavmi pomocou metód: `acquire()` a `release()`

Exkurz *lock*

- Metóda `acquire()`
 - Ak je zámok voľný/odomyknutý, volajúci môže pokračovať ďalej vo vykonávaní kódu a zároveň sa stav zámku zmení na “zamknutý”
 - Ak je zámok obsadený/zamknutý, volajúci nepokračuje vo vykonávaní kódu ďalej, ale čaká na uvoľnenie/odomyknutie zámku

Exkurz *lock*

- Metóda `acquire()`
 - Ak je zámok voľný/odomyknutý, volajúci môže pokračovať ďalej vo vykonávaní kódu a zároveň sa stav zámku zmení na “zamknutý”
 - Ak je zámok obsadený/zamknutý, volajúci nepokračuje vo vykonávaní kódu ďalej, ale čaká na uvoľnenie/odomyknutie zámku
- Čakanie je pre volajúceho transparentné (nevie o ňom, nijako sa o to nestará)
- Podľa typu čakania rozoznávame ADT *Spinlock* a *Sleeplock*

Exkurz *lock*

- Rozdiel medzi *Sleeplock* a *Spinlock*
 - *Sleeplock* pri čakaní na uvoľnenie zámku nevytáča CPU; stav procesu sa v xv6 označí ako SLEEPING, takže ho plánovač nebude plánovať
 - *Spinlock* pri čakaní využíva cyklus (v cykle neustále testuje dostupnosť zámku)

Exkurz *lock*

- Rozdiel medzi *Sleeplock* a *Spinlock*
 - *Sleeplock* pri čakaní na uvoľnenie zámku nevyťažuje CPU; stav procesu sa v xv6 označí ako SLEEPING, takže ho plánovač nebude plánovať
 - *Spinlock* pri čakaní využíva cyklus (v cykle neustále testuje dostupnosť zámku)
- Metóda `release()`
 - Mení stav zámku na voľný/odomyknutý
 - V prípade ADT *Sleeplock* zobudí procesy čakajúce (ak nejaké vôbec sú) v metóde `acquire()` na získanie zámku

Exkurz *lock*

- Využitie zámkov
 1. Ochrana integrity dát
 2. Výlučný prístup ku zdroju

Exkurz *lock*

- Využitie zámkov
 1. Ochrana integrity dát
 2. Výlučný prístup ku zdroju
- Kód medzi metódami `acquire()` a `release()` označujeme ako KO (kritická oblasť)
- Vykonávať ho môže vždy iba JEDEN tok riadenia (viď prechod cez turniket – vždy iba jeden)

Exkurz *lock*

- Vzhľadom na získanie a uvoľnenie zámku rozlišujeme 2 prístupy
 1. Vlákno, ktoré získalo zámok, zámok uvoľňuje
 2. Jedno vlákno získa zámok, iné vlákno uvoľňuje

Exkurz *lock*

- Vzhľadom na získanie a uvoľnenie zámku rozlišujeme 2 prístupy
 1. Vlákno, ktoré získalo zámok, zámok uvoľňuje
 - Najčastejší prípad využitia zámkov, vid' napríklad druhá a tretia úloha cvičenia
 2. Jedno vlákno získa zámok, iné vlákno uvoľňuje
 - Veľmi účinný mechanizmus na odovzdávanie “poverenia” (tokenu) – vid' napr. Petriho siete
 - Napríklad prepínanie vlákien v jadre xv6

Ukážka prepnutia v xv6

- user/spin.c
- Dva procesy, ktore vyt'azujú procesor (tzv. *CPU-bound* procesy)
- Spustíme qemu s iba 1 CPU
- Budeme pozorovať, ako xv6 urobí prepnutie medzi nimi

Ukážka prepnutia v xv6

- `make CPUS=1 qemu-gdb`
- V druhom okne spustíme gdb a zadáme ``continue``
- V okne qemu spustíme program `spin`
- Na výstupe vidíme, ako sa striedajú napriek tomu, že v kóde je nekonečný cyklus
- Xv6 vynucuje ich striedanie na 1 CPU, ktoré je k dispozícii

Ukážka prepnutia v xv6

- Dáme *breakpoint* do prerušenia časovača
- (gdb) Ctrl+c
- (gdb) b trap.c:207
- (gdb) c
- (gdb) finish
- (gdb) where

Ukážka prepnutia v xv6

- Kde sme? V `usertrap()`, po spracovaní prerušenia z časovača
- Aký užívateľský program bežal v čase vyvolania prerušenia?
 - `(gdb) print p->name`
 - `(gdb) print p->pid`
 - `(gdb) print/x *(p->trapframe)`
 - `(gdb) print/x p->trapframe->epc`

Ukážka prepnutia v xv6

- Kde sme? V `user trap()`, po spracovaní prerušenia z časovača
- Aký užívateľský program bežal v čase vyvolania prerušenia?
 - `(gdb) print p->name`
 - `(gdb) print p->pid`
 - `(gdb) print/x *(p->trapframe)`
 - `(gdb) print/x p->trapframe->epc`
 - Pozrime do `user/spin.asm`, kde prišlo k prerušeniu užívateľského kódu prerušením časovača...

Ukážka prepnutia v xv6

- Pomocou `step` sa presuňme v gdb do funkcie yield()
- (gdb) next
- (gdb) print p->state

- O 2 riadky nižšie sa mení stav procesu z RUNNING na RUNNABLE
- Aby neprišlo k zlej situácii (akej?), je potrebné použiť zámok

Ukážka prepnutia v xv6

- (gdb) next 2
- (gdb) step // vojdeme do funkcie sched()
 - Najprv sú kontroly, tie preskočíme, aby sme sa dostali pred vykonanie swtch()
- (gdb) next 7 // pripadne ešte 1-2x `step`
- V tomto volaní swtch() sa prepína kontext medzi *kernel* vláknom a vláknom plánovača

Ukážka prepnutia v xv6

- `swtch()`
 - Uloží aktuálne hodnoty registrov do prvého argumentu (`p->context`)
 - Obnoví hodnoty registrov uložené na adrese druhého argumentu (`c->context`)
 - Pozri `cpus[0].context.ra` – obsahuje adresu, kam sa po vykonaní `swtch()` odovzdá riadenie
 - Pozri `cpus[0].context.sp` – obsahuje adresu vrcholu zásobníka, ktorý sa použije
 - Ide o funkciu v ASM, aby nemusela používať zásobník

Ukážka prepnutia v xv6

- Podme do funkcie swtch
- (gdb) tbreak swtch
- (gdb) c
 - Sme vo funkcii kernel/swtch.S
 - a0 obsahuje prvý argument (p->context)
 - a1 obsahuje druhý argument (cpus[0].context)
 - Funkcia ukladá hodnoty registrov CPU do 1. arg
 - Načítava hodnoty registrov CPU z 2. arg
 - Potom vyvolá návrat pomocou `ret`

Ukážka prepnutia v xv6

- Otázka 1
- `swtch()` neukladá ani neobnovuje `$pc` (*program counter*); ako potom “vie”, kde má pokračovať vykonávanie po zmene kontextu?
- Otázka 2
- Prečo `swtch()` ukladá iba 14 registrov (`$ra`, `$sp`, `$s0` až `$s11`) a ostatné nie?
 - Registre *user* vlákna sú VŠETKY odložené v TF; tu hovoríme o registroch *kernel* vlákna

Ukážka prepnutia v xv6

- Zobrazme si registre na začiatku funkcie
 - (gdb) p/x \$pc // swtch
 - (gdb) p/x \$ra // sched
 - (gdb) p/x \$sp // proc[pid-1].kstack+???
- Podme na koniec funkcie a zobrazme si ich znovu
 - (gdb) stepi 28 // mali by sme byť pred `ret`
 - (gdb) p/x \$pc // swtch
 - (gdb) p/x \$ra // scheduler !!!
 - (gdb) p/x \$sp // stack0+???

Ukážka prepnutia v xv6

- (gdb) where
- (gdb) stepi
- Vykonávanie je v scheduler (), vo vlákne plánovača jadra 0, kód beží na zásobníku tohto vlákna!

Ukážka prepnutia v xv6

- Pre vlákno plánovača sa “javí” beh kódu ako obyčajný návrat z funkcie `swtch()`
 - Túto fnc musel plánovač vyvolať niekedy v minulosti, čím spôsobil “prepnutie” na kód vlákna jadra procesu
 - Toto predošlé zavolanie `swtch()` uložilo kontext vlákna plánovača (všimni si argumenty funkcie – tu je to opačne ako v `sched()`)
 - Premenná `p` ukazuje na proces, ktorého beh bol prerušený

Ukážka prepnutia v xv6

- (gdb) print p->name
- (gdb) print p->pid
- (gdb) print p->state

Ukážka prepnutia v xv6

- Funkcia `yield()` získala zámok (*lock*), plánovač ho teraz uvoľní
- Z pohľadu kódu sa zdá, že `scheduler()` zamkne aj odomkne hneď po sebe
- Ale v skutočnosti
 - `scheduler()` zamkne, `yield()` odomkne
 - `yield()` zamkne, `scheduler()` odomkne
- Netypické využitie zámku – **posunutie tokenu** niekomu inému (“prihraj loptu”)

Ukážka prepnutia v xv6

- Je možné uvoľniť zámok $p \rightarrow \text{lock}$ tesne pred zavolaním `switch()`? (či už vo funkcii `scheduler()` alebo `yield()`)

Ukážka prepnutia v xv6

- Je možné uvoľniť zámok $p \rightarrow \text{lock}$ tesne pred zavolaním `swtch()`? (či už vo funkcii `scheduler()` alebo `yield()`)
 - Vyvolajme funkciu `swtch()` a CPU1 stihne uložiť iba pár registrov (alebo aj žiaden), ale nie všetky
 - Keďže $p \rightarrow \text{status}$ je `RUNNABLE`, tak CPU2 naplánuje tento proces na beh (v `scheduler()` sa vyvola `swtch()`)
 - Vlákno jadra na CPU2 sa obnoví s poškodeným kontextom, čo môže viesť ku chybe (ktorá sa veeeeeeelmi ťažko hľadá)

Ukážka prepnutia v xv6

- Úloha zámku $p \rightarrow \text{lock}$

Ukážka prepnutia v xv6

- Úloha zámku $p \rightarrow \text{lock}$
- Atomicita nasledovných operácií
 - $p \rightarrow \text{state} = \text{RUNNABLE}$
 - Uloženie registrov do $p \rightarrow \text{context}$
 - Ukončenie používania zásobníka jadra $p \rightarrow \text{kstack}$

Ukážka prepnutia v xv6

- Úloha zámku $p \rightarrow \text{lock}$
- Atomicita nasledovných operácií
 - $p \rightarrow \text{state} = \text{RUNNABLE}$
 - Uloženie registrov do $p \rightarrow \text{context}$
 - Ukončenie používania zásobníka jadra $p \rightarrow \text{kstack}$
- Atomicita a neprerušiteľnosť operácií
 - $p \rightarrow \text{state} = \text{RUNNING}$
 - Presun hodnôt registrov z $p \rightarrow \text{context}$ do CPU; nesmie nastať intr, lebo by sa kontext prepísal ešte neinicializovanými hodnotami registrov CPU

Ukážka prepnutia v xv6

- Prejdime na miesto, kde scheduler () nájde proces, ktorý je RUNNABLE
 - (gdb) tbreak proc.c:454
 - (gdb) c
 - (gdb) print p->name // meno je rovnaké... fork ()
 - (gdb) print p->pid // ide o iný proces!!!

Ukážka prepnutia v xv6

- Prejdime na miesto, kde scheduler () nájde proces, ktorý je RUNNABLE
 - (gdb) tbreak proc.c:454
 - (gdb) c
 - (gdb) print p->name // meno je rovnaké... fork ()
 - (gdb) print p->pid // ide o iný proces!!!
- Pozrime, kde bude pokračovať vykonávanie vlákna tohto procesu v jadre po switch ()
 - (gdb) print/x p->context
 - (gdb) x/4i p->context->ra // funkcia sched ()

Ukážka prepnutia v xv6

- Podme znovu do funkcie `sched()`
 - `(gdb) tbreak swtch`
 - `(gdb) c`
 - `(gdb) stepi 28 // sme pred vykonaním inštrukcie ret`
 - `(gdb) print/x $ra`
 - `(gdb) where`
 - Sme po obsluhu prerušenia časovača v kontexte iného procesu ako sme začali!!!
 - Proces bol prerušený, zavolať `yield()`, `sched()`, `swtch()`
 - Teraz sa jeho beh obnovil a vráti sa do *user* priestoru

Ukážka prepnutia v xv6

- Poznámka1
 - Iba `switch()` prepisuje kontexty (okrem inicializácie)
 - Iba `sched()` a `scheduler()` volajú `switch()`
 - Preto platí, že `context.ra`
 - vlákna jadra procesu vždy ukazuje do `sched()`
 - vlákna plánovača vždy ukazuje do `scheduler()`

Ukážka prepnutia v xv6

- Poznámka1
 - Iba `swtch()` prepisuje kontexty (okrem inicializácie)
 - Iba `sched()` a `scheduler()` volajú `swtch()`
 - Preto platí, že `context.ra`
 - vlákna jadra procesu vždy ukazuje do `sched()`
 - vlákna plánovača vždy ukazuje do `scheduler()`
- Poznámka 2
 - `sched()` → `swtch()` → `scheduler()` → `swtch()` → `sched()`
 - Vo všeobecnosti sa nejedná o návrat do funkcie `sched()` rovnakého vlákna jadra

Exkurz koprogram/korutina

- Korutina (koprogram, *co-routine*) vs. podprogram

Exkurz koprogram/korutina

- Korutina (koprogram, *co-routine*) vs. podprogram
 - Podprogram po svojom zavolaní začne na začiatku svojej definície a 1x skončí; nedrží stav premenných medzi jednotlivými vyvolaniami

Exkurz koprogram/korutina

- Korutina (koprogram, *co-routine*) vs. podprogram
 - Podprogram po svojom zavolaní začne na začiatku svojej definície a 1x skončí; nedrží stav premenných medzi jednotlivými vyvolaniami
 - Koprogram pri svojom prvom spustení začne od začiatku svojej definície a potom “zdanlivo” skončí volaním iného koprogramu; iný koprogram môže “odovzdať riadenie” do tohto bodu “skončenia” prvého koprogramu. Medzi jednotlivými obnoveniami behu udržiava stav premenných.

Exkurz koprogram/korutina

- Vzájomné vyvolanie korutín je v inom vzťahu ako volanie obyčajného podprogramu
 - Pri obyčajnom podprograme je tento volaný, vzťah je asymetrický (volajúci – volaný)
 - Pri dvoch korutinách sa tieto navzájom volajú, vzťah je symetrický (volajúci – volajúci alebo volaný – volaný, je to jedno)

Exkurz koprogram/korutina

- Vzájomné vyvolanie korutín je v inom vzťahu ako volanie obyčajného podprogramu
 - Pri obyčajnom podprograme je tento volaný, vzťah je asymetrický (volajúci – volaný)
 - Pri dvoch korutinách sa tieto navzájom volajú, vzťah je symetrický (volajúci – volajúci alebo volaný – volaný, je to jedno)
- Viac o synchronizačných mechanizmoch, koprogramoch a iných veciach na voliteľnom predmete PPaDS inžinierskeho štúdia

Korutiny v xv6

- `sched()` a `scheduler()` sú navzájom korutiny
- Navzájom sa “poznajú” – “vedia”, ktorá kam odovzdáva riadenie a skadiaľ riadenie príde
- Navzájom kooperujú v rámci zdieľania stavu ($p \rightarrow \text{lock}$ a $p \rightarrow \text{state}$)

Ako je to v jadre xv6?

- Je preemptívne plánovanie platné aj pre beh vlákién jadra (nielen užívateľských procesov)?

Ako je to v jadre xv6?

- Je preemptívne plánovanie platné aj pre beh vlákien jadra (nielen užívateľských procesov)?
- Áno, pri prerušení časovača (vid' `kerneltrap()`)
- Kam sa ukladajú registre v tomto prípade?
 - Do `p->trapframe` sa nemôžu (užívateľský kód)
 - Do `p->context` sa nemôžu (používajú ho iba korutiny `sched()` a `scheduler()` pri volaní `switch()`)

Ako je to v jadre xv6?

- Je preemptívne plánovanie platné aj pre beh vlákien jadra (nielen užívateľských procesov)?
- Áno, pri prerušení časovača (vid' `kerneltrap()`)
- Kam sa ukladajú registre v tomto prípade?
 - Do `p->trapframe` sa nemôžu (užívateľský kód)
 - Do `p->context` sa nemôžu (používajú ho iba korutiny `sched()` a `scheduler()` pri volaní `switch()`)
 - Odpoveď je v súbore `kernelvec.S` – na aktuálny zásobník jadra

Ako je to v jadre xv6?

- Prečo plánovač čím skôr zapína prerušenia pomocou `intr_on()`?

Ako je to v jadre xv6?

- Prečo plánovač čím skôr zapína prerušenia pomocou `intr_on()`?
 - Čo keď sú všetky procesy čakajúce (napr. na disk alebo konzolu)?
 - Zapnutie prerušení dáva možnosť zariadeniam signalizovať pripravenosť dát, takže sa stav čakajúcich vlákien môže zmeniť na `RUNNABLE`
 - V opačnom prípade hrozí uviaznutie systému

Ako je to v jadre xv6?

- Prečo je tak prísna kontrola ohľadom držania zámkov v `sched()`? Konkrétne, iba jeden jediný môže byť držaný, a to `p→lock`?

Ako je to v jadre xv6?

- Prečo je tak prísna kontrola ohľadom držania zámkov v `sched()`? Konkrétne, iba jeden jediný môže byť držaný, a to `p→lock`?
 - Súvisí to s princípom, ako funguje *spinlock*; o tom ešte budeme hovoriť na inej prednáške
 - V krátkosti prijmime fakt, že funkcia `acquire()` musí bežať s vypnutými prerušeniami
 - Majme nasledovnú situáciu
 - Ukážka sa týka behu jadra s 1 CPU, ale platí všeobecne

Ako je to v jadre xv6?

- 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 zamok L1.
- Nastáva uviaznutie (*deadlock*)

Plánovacia politika

- Akú stratégiu pri plánovaní implementuje xv6?

Plánovacia politika

- Akú stratégiu pri plánovaní implementuje xv6?
- “Ide pieseň dokola” (*Round Robin*)

Plánovacia politika

- Akú stratégiu pri plánovaní implementuje xv6?
- “Ide pieseň dokola” (*Round Robin*)
- Aké ďalšie stratégie poznáme?
 - FCFS (“kto prv príde, ten prv melie”)
 - SJF (“najkratší najskôr”)
 - Prioritné plánovanie
 - ...

Plánovacia politika

Peter Tomcsányi: <http://edu.fmph.uniba.sk/~tomcsanyi/06a.pdf>

- FCFS (*First-Come First-Served*)
 - Do fronty sa procesy radia podľa času príchodu
 - Každý proces beží, pokým neskončí, alebo pokým neodíde do stavu čakania (na niečo)
 - Znevýhodňuje vstupno/výstupné procesy
- SJF (*Shortest Job First*)
 - Do fronty sa procesy radia podľa dĺžky svojho behu
 - V praxi nerealizovateľné bez nejakej modifikácie, nakoľko dopredu nevieme určiť čas behu procesu

Plánovacia politika

Peter Tomcsányi: <http://edu.fmph.uniba.sk/~tomcsanyi/06a.pdf>

- SRTN (*Shortest Remaining Time Next*)
 - Preemptívna verzia algoritmu SJF
 - Čas do skončenia sa prepočítava (napr. pri príchode ďalšej úlohy do fronty, pri odložení na čakanie atď.)
 - Vyberá sa ten proces, ktorému ostáva najmenej času do skončenia
- RR (*Round Robin*)
 - Preemptívny; na základe časového kvanta
 - Pri veľkom kvante rastie čas odozvy, pri malom klesá efektivita (veľa réžie na zmenu kontextu)

Plánovacia politika

Peter Tomcsányi: <http://edu.fmph.uniba.sk/~tomcsanyi/06a.pdf>

- Prioritné plánovanie
 - Zvýhodňuje procesy na základe priority
 - Znižuje férovosť
 - Dynamická priorita upravuje férovosť (napr. prepočítanie priority podľa času strávenom na CPU)
 - V praxi často verzia, kde je viacero prioritných front

Plánovacia politika

Peter Tomcsányi: <http://edu.fmph.uniba.sk/~tomcsanyi/06a.pdf>

- SPN (*Shortest Process Next*)
 - Alternatíva základného algoritmu SJF
 - Za najkratší sa považuje ten, ktorý najmenej čakal v dobe medzi dvoma čakaniami na beh
- Algoritmus Lotérie
 - Každý proces má “los”
 - Niektoré môžu mať viac “losov”
 - Spolupracujúce procesy si môžu “losy” odovzdávať
 - Ide o alternatívu prioritného plánovania – má predvídateľnejšie správanie

Plánovacia politika

Peter Tomcsányi: <http://edu.fmph.uniba.sk/~tomcsanyi/06a.pdf>

- Algoritmus férového podielu
 - Procesy sa plánujú na beh tak, aby každý používateľ systému “minul” rovnaký podiel času na procesore
 - Ak má užívateľ¹ 4 procesy a užívateľ² 6 procesov, tak podľa tohto algoritmu $\frac{2}{5}$ času na CPU strávia procesy užívateľa 1 a $\frac{3}{5}$ času procesy užívateľa 2
- Garantované plánovanie
 - Funguje na základe dosahovania podmienky
 - Napríklad pri N procesoch každý má mať $\frac{1}{N}$ času CPU

Domáce čítanie

Chapter 7

Scheduling

xv6: a simple, Unix-like teaching operating system

Niečo aj pre náročných

<https://graphitemaster.github.io/fibers/>