Prepínanie procesov a vlákien

Prečo je potrebná podpora viacerých úloh vykonávaných konkurentne?

- Viacero používateľov systému súčasne,
- viacero programov jedného používateľa spustených súčasne,
- viacero vlákien bežiacich na jednom procesore (na jadrách) súčasne.

Zmysel multi-taskingu (podpory viacerých konkurentne bežiacich tokov riadenia):

- Izolácia (kvôli úmyselným či neúmyselným chybám aplikácií),
- zdieľanie zdrojov výpočtového systému (CPU, pamäť, disk, sieť, ...),
- efektivita (keď CPU čaká na dokončenie I/O, môže bežať iný proces).

Dva základné prístupy k multi-taskingu:

1. Vlákna (procesy): vykonávanie programu ide sekvenčne (riadok po riadku).
2. Udalosti: tok programu je riadený prichádzajúcimi udalosťami (viď model prerušení).

Unix-like systémy a xv6 používajú prvý model (vlákna v procesoch).
Príklad sekvenčného kódu dvoch procesov vhodného na tento model:

```
P1:                                 P2:
while(1){                           while(1){
        x = wait_for_input();               compute();
        x = x+1;                    }
        printf(x);
}
```

Problémy pri navrhovaní modelu vlákien:

- Ako prepínať vlákna pri malom počte CPU? Ako spraviť toto prepínanie transparentne (aby mal programátor aplikácií čo najmenej starostí)?
- Čo má procesor robiť pri procese P1, pokým sa čaká na vstup?
- Ako predísť tomu, aby si proces P2 neprivlastnil výpočtový čas procesora na úkor iných procesov?

Najprv sa zamyslime nad prípadom procesu P2. Takéto procesy označujeme *compute-bound* (výpočtovo orientované).

- Jadro OS nebeží stále: ako teda môže prinútiť výmenu vlákien na CPU?
- Každé CPU má hw časovač, ktorý je možné použiť na pravidelné vyvolanie prerušenia!
- Túto možnosť využíva jadro, aby sa vynútil prechod do jadra OS aj pri procesoch, ktoré na svoj beh nepotrebujú žiadnu službu OS.
- Takejto technike, kde sa vynucuje prerušenie behu procesov v používateľskom režime, sa hovorí *preemptívne* plánovanie procesov.

Teraz sa vráťme k prípadu procesu P1, ktorý nie je schopný vykonávať programový kód (slangovo hovoríme "bežať"), pokým nebude mať k dispozícii nejaký zdroj:

- Potrebujeme niekam uložiť jeho stav: registre CPU, zásobník, pamäť, otvorené súbory, ...
- Pamäť nikam nepôjde, to je jasné (treba však prepnúť tabuľku stránok!).
- Registre CPU treba uložiť, pretože druhé vlákno zmení ich hodnoty. Tak aby sme vedeli obnoviť stav procesora v čase, keď odkladané vlákno bude znovu plánované na beh.

- Zásobník musí mať každý tok riadenia úplne samostatne, pretože na zásobníku je uložený sled volaní (návratové adresy funkcií) spolu s lokálnymi premennými funkcií.
- Ostatné informácie o procese (otvorené súbory...) sa uchovávajú v štruktúrach jadra a/alebo vo virtuálnom priestore procesu, ktorý sa prepína pomocou stránkovania.

Xv6 poskytuje vlákna pre jadro OS aj používateľa, a to nasledovným spôsobom:

- Jedno používateľské vlákno pre proces, určené na beh kódu užívateľskej aplikácie;
- jedno vlákno jadra pre proces, určené na vykonávanie systémových volaní užívateľskej aplikácia;
- jedno vlákno plánovača pre každý procesor zvlášť.

Čo je teda proces?

- Používateľský proces pozostáva z dvoch vlákien (jedno v jadre, jedno v užívateľskom priestore); ide o vykonateľný programový kód načítaný z disku do pamäte.
- Okrem toho má pridelené ďalšie zdroje výpočtového systému (pamäť, otvorené súbory, sieťové pripojenia, zámky, semafory atď).
- **Rozdiel medzi programom a procesom**: program je kód spolu s údajmi uložený napr. na disku (ide o pasívnu entitu). Proces je aktívna entita využívajúca zdroje výpočtového systému pomocou OS (vykonávajúca programový kód zavedený do pamäte; pristupujúca k súborom, monitoru, sieti, ...).

Schéma prepínania procesov v xv6:

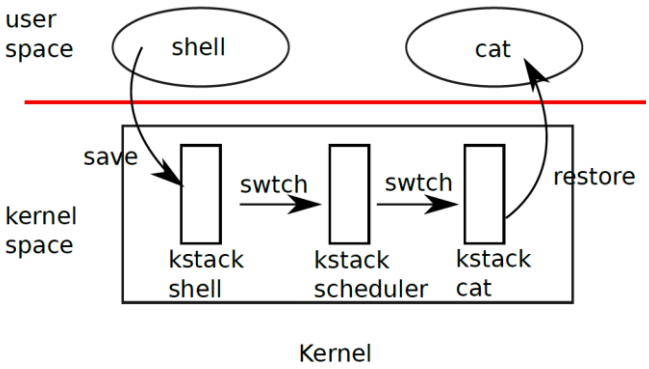
1. user thread -> kernel thread (pomocou systémového volania alebo prerušenia od časovača); vlákno jadra sa vzdá procesora v dôsledku preempcie alebo čakania na dokončenie I/O;
2. kernel thread -> scheduler thread; vlákno plánovača hľadá nejaké vlákno jadra, ktoré je v stave RUNNABLE;
3. scheduler thread -> kernel thread
4. kernel thread -> user thread

Štruktúra proc v proc.h:

- trapframe uchováva odložené registre užívateľského vlákna
- context uchováva odložené registre vlákna jadra
- kstack uchováva registre vlákna plánovača

Procesy xv6 môžu nadobúdať nasledovné stavy (viď položka proc->state):

1. RUNNING
2. RUNNABLE
3. SLEEPING
4. ZOMBIE
5. UNUSED



Obr. 1: Prepnutie jedného vlákna používateľa na druhé. Jedno CPU. Zdroj: book-riscv-rev0.pdf, str. 62.

Vlákna jadra xv6 zdieľajú spoločný virtuálny priestor. Užívateľské vlákna majú virtuálne adresné priestory odlišné. Xv6 implementuje iba jedno užívateľské vlákno per proces. Kto chce, môže rozšíriť podporu xv6 o viac užívateľských vlákien per proces.

Urobiť správne prepínanie kontextu v xv6 bolo jednou z najťažších vecí; prepínanie zahŕňa viac CPU, uzamykanie, prerušenia, špeciálna stavy procesu (prvý proces, ukončenie procesu)...

Ukážka spin.c

1. CPUS=1 make qemu-gdb; gdb; c; spin; ctrl+c... kde sme?
2. CPUS=1 make qemu-gdb; gdb; b trap.c:81; c; spin... kde sme?
    - (gdb) p p->name
    - (gdb) p p->pid
    - (gdb) p/x *(p->tf)
    - (gdb) p/x p->tf->epc; viď epc v user/spin.asm
    - (gdb) c
    - (gdb) p/x p->tf->epc; viď epc v user/spin.asm
    - (gdb) c
    - (gdb) p/x p->tf->epc
    - (gdb) c
    - ...
3. CPUS=1 make qemu-gdb; gdb; b trap.c:81; c; spin
    - (gdb) step
    - (gdb) next
    - (gdb) print p->state (RUNNING)
    - (gdb) next --- yield() uzamyká zámok, pretože modifikuje stav procesu a zároveň chráni pred spustením tohto vlákna na inom procesore
    - (gdb) next
    - (gdb) print p->state (RUNNABLE) --- vzdanie sa CPU, ale označenie procesu, že chce znovu získať prístup ku CPU a bežať ďalej; keby nebol zamknutý zámok, na inom procesore by bolo od tohto okamihu možné naplánovať na beh toto vlákno, a to by bol problém, pretože toto vlákno pokračuje ešte vo vykonávaní kódu ďalej na tomto procesore, a zároveň by sa spustilo jeho vykonávanie na inom procesore... a to je véééééľmi zléééé (veeeerY baaaad)!
    - step --- ideme do sched(), pozrime kód; sched() po základných kontrolách integrity spúšťa funkciu swtch(); chceme vyvolať výmenu kontextu (obsah registrov CPU) jedného vlákna jadra za iné (za jadro plánovača daného CPU). Výmena kontextu znamená uloženie obsahu CPU a nahratie nového obsahu registrov CPU z iného miesta, kde boli predtým uložené. Funkcia swtch() ukladá obsah 32 registrov CPU na miesto dané prvým argumentom (p->context) a obnovuje obsah registrov CPU z pamäťového miesta, ktoré je druhým argumentom funkcie (mycpu()->scheduler). Vzhľadom na cvičenie je dobré si všimnúť, ktoré registre sa ukladajú/obnovujú.
    - pred spustením swtch() si ešte všimnime obsah štruktúr p->context a mycpu()->scheduler; mycpu() nevieme použiť, preto skúsime p/x cpus[0]->scheduler a všimneme si hodnoty registrov. Aká je hodnota registra ra? Kde to je? Viď kernel/kernel.asm... Pokračujeme funkciou swtch().
    - Funkciu swtch() krokujeme pomocou si (step instruction)!
    - Pred inštrukciou ret zastaňme a urobme nasledovné kroky v gdb:
        - (gdb) p/x $pc
        - (gdb) p/x $ra
        - (gdb) p/x $sp
        - (gdb) stepi --- mala by sa vykonať inštrukcia ret
        - (gdb) p/x $pc
        - (gdb) p/x $ra
        - (gdb) p/x $sp
        - (gdb) where
        - (gdb) stepi
    - Vynoríme sa v kernel/proc.c:468. Sme vo funkcii scheduler(), vo vlákne plánovača, na jeho zásobníku. Ako sme sa tu ocitli? Plánovač musel vyvolať funkciu swtch() na riadku 464, čím spustil vykonávanie

niektorého vlákna jadra. Spustením swtch() sa uložil stav vlákna plánovača, a obnovil sa obsah registrov CPU podľa vlákna jadra, ktoré plánovač vybral na beh na procesore.

- Premenná p stále ukazuje na prerušený proces:
  - (gdb) print p->name
  - (gdb) print p->pid
  - (gdb) print p->state
- Na riadku 472 vlákno plánovača uvoľňuje zámok procesu; ten bol uzamknutý prerušeným vláknom vo funkcii yield(). Ak by sme sledovali iba kód, zdalo by sa, že zámok uzamklo vlákno plánovača na riadku 457! To však nie je pravda!!! Keď plánovač chce skontrolovať stav procesu, aby zistil, či je niektoré vlákno spustiteľné (na riadku 456 prechádza poľom všetkých štruktúr proc), tak pred examinovaním stavu uzamkne zámok príslušného vlákna. Ak je stav procesu RUNNABLE (riadok 458), tak vlákno plánovača zmení stav kandidáta na beh na CPU na RUNNING (riadok 462) a spustí výmenu kontextu swtch() (riadok 464). Tým sa vymení obsah registrov CPU, vymení sa zásobník, príde k prepnutiu kontextu na vlákno jadra, ktoré bude pokračovať... poďme sa pozrieť ako.
- (gdb) tbreak swtch --- nastavíme najbližší bod prerušenia gdb do funkcie swtch()
- (gdb) c
- (gdb) where --- sme vo funkcii swtch() vyvolanej z vlákna plánovača; nemáme dostupnú premennú p; preto sa "presunieme" v rámci sledu volaných funkcií o úroveň vyššie
- (gdb) up
- (gdb) print p->name
- (gdb) print p->pid
- (gdb) print p->state
- **Kde bude pokračovať vykonávanie prerušeného vlákna jadra po ukončení funkcie swtch()? Kde bude poračovať vykonávanie prerušeného vlákna užívateľského priestoru?** Pozor na rozdiel štruktúr context a tf!!!
- (gdb) print p->context --- všimnime si register ra a zásobník sp; konzultujme súbor kernel/kernel.asm
- posuňme sa na koniec swtch(), na inštrukciu ret
- (gdb) stepi 28 --- 28x sa zopakuje stepi; ak sme na inštrukcii ret, ešte raz stepi
- (gdb) where --- mali by sme byť v kernel/riscv.h:287
- (gdb) up 3 --- presunieme sa do rámca funkcie sched, aby sme mali k dispozícii premennú p
- (gdb) print p->pid
- (gdb) print p->name
- (gdb) print/x p->tf->epc --- takže sa skutočne nachádzame v tom druhom procese forku, ktorý bol prerušený časovačom, vyvolal yield(), ten vyvolal zmenu kontextu vlákien jadra, spustil sa plánovač, vybral iné vlákno na beh, plánovač spustil swtch() na zmenu kontextu vlákien jadra na iný proces, ten sa vrátil do užívateľského priestoru, tam sa znovu vyvolalo prerušenie časovača... atď atď atď

Aký vzor vidíme pri striedaní užívateľských procesov?

- beh užívateľského vlákna v jadre
- prechod do jadra (vďaka prerušeniu časovača alebo žiadosťou systémového volania)
- vyvolanie yield(); uzamknutie zámku procesu
- prechod do plánovača; odomknutie zámku procesu
- výber iného procesu na beh; uzamknutie zámku procesu
- prechod do vlákna procesu; odomknutie zámku procesu
- ukončenie funkcie yield()
- prechod do užívateľského priestoru; obnovenie behu užívateľského vlákna

Aký vzor vidíme v uzamykaní zámku?

- scheduler() zamkne; yield() odomkne
- yield() zamkne; scheduler() odomkne

- Zámok sa najbežnejšie používa tak, že ten, kto ho zamkne (t.j. vlákno X), ho aj odomyká. V tomto synchronizačnom vzore vidíme, že vlákno X zamyká, ale vlákno Y odomyká! Takémuto vzoru sa hovorí "podaj ďalej" :)
- Ide o tzv. odovzdávanie "tokenu" medzi vláknami. Viac info na predmete PPaDS :D

yield() a scheduler() sú tzv. koprogramy (*co-routines*). Koprogram je taký tok riadenia, ktorý nemá len jeden vstupný a jeden výstupný bod. Volajúci tok "vie", kam sa riadenie odovzdáva, volaný tok "vie", odkiaľ riadenie prišlo. Tento prístup je odlišný od klasického prepínania kontextu, kde žiadna zo zúčastnených strán výmeny nevie, ktoré vlákno bude nasledovať, alebo ktoré bolo predtým.

Otázky ohľadom algoritmu výberu plánovača:

- Akú politiku plánovania plánovač xv6 implementuje? Aký algoritmus?
- Môže byť znovu hneď spustené vlákno, ktoré práve vyvolalo yield()?
- Je dobré, ak to tak je?

Otázka ohľadom prerušení: Prečo plánovač zapína prerušenia funkciou intr_on()?

- Čo ak všetky vlákna čakajú na dokončenie I/O, a teda žiadne vlákno nie je v stave RUNNABLE?
- Zapnutie prerušení umožní zariadeniam signalizovať pripravenie údajov pre nejaké vlákno, takže sa bude môcť naplánovať na beh na CPU. Inak by systém zamrzol...

Otázka na potrebu špeciálneho vlákna plánovača: Je možné navrhnúť jadro tak, aby neprichádzalo ku prepnutiu kontextu medzi vláknom jadra a plánovačom, ale v kontexte vlákna jadra by sa priamo urobilo prepnutie na iné vlákno? Tak, že by sme zrušili vlákno plánovača, a výber vlákna, ktoré bude bežať, by sa robil napr. vo funkcii sched()?

- Áno, dá sa to, a bolo by to istotne rýchlejšie riešenie (bez prepínania kontextu vlákien jadra).
- Ale... prinieslo by to veľa starostí ohľadom správnosti riešenia; bolo by treba premyslieť a prepracovať systém uzamykania a prepínania zásobníkov.
- Kto chce, môže sa o túto zmenu pokúsiť.

Zatiaľ sme v prednáške spomenuli preemptívne plánovanie pre užívateľské vlákna (t.j. hardvérovým prerušením časovača sa vynúti prechod do jadra, a v rámci obsluhy tohto prerušenia sa vyvolá výmena prerušeného užívateľského procesu za iný). Je však takáto preempcia implementovaná aj pre vlákna jadra?

- Áno, xv6 implementuje preempciu aj pre vlákna jadra, a to rovnakým spôsobom ako pre vlákna v užívateľskom priestore - využíva sa prerušenie časovača a vyvolanie yield() v rámci jeho obsluhy.
- Viď funkciu kerneltrap() v kernel/trap.c
- Kam sa však ukladajú registre prerušeného kódu vlákna jadra? Nemôžu sa uložiť do p->tf, pretože by sa stratila informácia o uloženom stave užívateľského vlákna... Takže sa ukladajú na zásobník aktuálne spusteného vlákna jadra.

Je preempcia v jadre užitočná?

- V xv6 určite nie je nutná.
- Určite je užitočná v prípade dlhšieho spracovania nejakého systémového volania.
- Podobne je užitočná, ak máme plánovač, ktorý zohľadňuje priority procesov, a potrebuje nejaký mechanizmus, pomocou ktorého by ich vynucoval.

Vráťme sa ešte k zámku p->lock. Jeho úlohou je ochrániť nasledovné invarianty (invariant je pravdivý, keď zámok NIE JE uzamknutý; invariant môže nadobúdať logickú hodnotu false v čase, keď zámok JE uzamknutý; význam slova invariant: niečo, čo je stále, čo sa nemení; v informatike ide o podmienku v algoritme, ktorá musí byť splnená počas celej doby behu algoritmu):

1. Ak je stav procesu RUNNING, registre CPU obsahujú platné hodnoty pre proces (nejde o p->context).
2. Ak je stav procesu RUNNABLE, p->context uchováva odložené platné hodnoty procesu.
3. Ak je stav procesu RUNNABLE, žiadne CPU nepoužíva zásobník procesu.

Uzamknutie zámku v yield() a scheduler() má za následok, že:

1. Žiadne CPU nemôže vykonávať proces p, pokým aktuálne CPU neprestane manipulovať so zásobníkom procesu p.
2. Keďže sú vypnuté prerušenia, nemôže prísť ku spracovaniu prerušenia (napr. časovača), a tým ku vnorenému vyvolaniu yield() počas vykonávania funkcie swtch(), ktorá manipuluje so zásobníkom vlákna jadra.

Ako vieme, že sú vypnuté prerušenia vo funkcii swtch()? Pretože xv6 vypína prerušenia na aktuálnom CPU vždy, keď vlákno získa zámok (iné CPU stále môžu nejaké prerušenia spracovať, pokiaľ ich tiež nemajú vypnuté). Funkcia acquire() vypína prerušenia, funkcia release() obnovuje stav prerušení (ak boli predtým zapnuté, zapne, inak ostanú vypnuté). Prečo je to tak? Nielen aby neprišlo ku nekonzistencii údajov vlákna (napr. zásobník vo funkcii swtch()), ale aby ani neprišlo ku uviaznutiu! Príklad:

- consoleread() --- čaká na vstup z klávesnice
    - acquire(cons.lock), kuk do cons.buf, release(cons.lock)
- Čo keď príde k prerušeniu od klávesnice práve v čase, keď drží zámok (robí "kuk" do cons.buf)?
- Funkcia consoleintr() sa pokúša získať zámok, aby mohla zapísať znak do cons.buf. Avšak obsluha prerušenia od klávesnice beží na tom istom zásobníku ako prerušený kód jadra (consoleread). Consoleread() nemôže skončiť skôr, než skončí consoleintr(). Ale consoleintr() nemôže pokračovať, lebo funkcia consoleread() uzamkla zámok! Takže uviaznutie (*deadlock*).
- Ak by napríklad jadro xv6 implementovalo obsluhu prerušení zvlášť v samostatných vláknach jadra (podobne, ako implementuje obsluhu plánovača pre každé CPU zvlášť v samostatnom vlákne), k takejto situácii by nemohlo prísť. Skúste prepracovať xv6, aby to tak bolo.

Synchronizácia je veľmi ťažká vec... Skúsme sa na záver zamyslieť ešte nad jednou "maličkosťou": Prečo sa vo funkcii sched() vyžaduje, aby proces nedržal okrem p->lock žiaden zámok (viď kernel/proc.c:496)? Pozrime sa na modelovú situáciu:

- Proces P1 drží L1, uvoľňuje CPU pre proces (vyvolá yield()) P2.
- Proces P2 sa pokúsi získať zámok L1:
    1. Ako už vieme, prerušenia sú vo funkcii acquire() vypnuté;
    2. podobne ako už z predošlej prednášky vieme, funkcia acquire() beží v cykle (ide o spinlock), vyťažuje CPU neustálym testom hodnoty zámku, či sa konečne náhodou neuvoľnil.
- Takže proces P2 v neustálom cykle beží na CPU s vypnutými prerušeniami:
    1. Prerušenie od časovača na toto CPU nepríde,
    2. kód acquire() je tak napísaný, že P2 sa procesora nevzdá volaním yield(),
    3. P1 nemôže získať CPU,
    4. P1 nemôže uvoľniť L1,
    5. uviaznutie.
- Ovšem, niekto by namietal, že tvrdenie "P1 nemôže získať CPU" je zavádzajúce, pretože to bude platiť iba v prípade spustenia xv6 na jednom CPU. To je síce pravda, ale xv6 nemá zavedené obmedzenie, že nesmie byť spúšťaný iba na jednom CPU. A keby to aj tak bolo, pre N CPU vieme nájsť podobný scenár s N+1 procesmi tak, aby k uviaznutiu prišlo. Z tohto dôvodu sa pri výmene kontextu vlákien jadra vyžaduje, aby vlákno okrem presne kontrolovaného zámku p->lock nedržalo žiadny iný zámok.