Procesy a vlákna POSIX

Z FITwiki

Obsah

Procesy

Program
: je statický kód s počátečními daty uložený v souboru.

Proces
: je instance programu v paměti (vlastny adresovy prostor), která se vykonává. Má jednoznačnou identifikaci (PID). Může být více procesů pro jeden program.

Stav procesu
: registry, paměť, zásobník a systémové prostředky (deskriptory apod.).
je definován stavem proměnných a pozicí v programu.

Hierarchie procesů

- nové procesy vznikají duplikací běžícího rodičovského procesu (`fork()`)
- existuje vztah otec-syn
- Nejvyšším prarodičem je proces `init`
- Při ukončení otce se synové přesouvají k `init`
- Při ukončení syna si otec vybere stav. Pokud otec na stav nečeká, stav visí v paměti a ze syna se stává zombie.

Paralelní provádění
: některé části procesu mohou být prováděny souběžně. Podstatný je stav na začátku a na konci procesu.

Determinismus
: pokud je výsledek paralelního provádění vždy stejný bez ohledu na pořadí provádění (časová nezávislost)

Nedeterminismus
: pokud výsledek závisí na pořadí provádění částí

Maximální paralelismus
: pokud už nelze více paralelizovat bez porušení determinismu

Vlákna

Vlákno

- samostatně prováděná část programu v rámci jednoho procesu
- Takto může jeden proces běžet na více procesorech paralelně.
- Vlákna jednoho procesu sdílí logický adresový prostor a systémové prostředky.
- Registry, zásobník a stav provádění programu se uchovává pro každé vlákno samostatně.

Typické použití
: I/O vlákno vedle výpočtů, GUI, u více procesorů nutnost pro výkon, ...

Proces s vlákny

- je pak jen obalová jednotka pro vlákna
- vlákna která jsou "procesy" z pohledu procesoru
- proces tak pouze uchovává vnější stav (PID, deskriptory apod.) společný všem vláknům
- V UNIXu je tedy proces jednotkou přidělování prostředků a vlákno jednotkou přidělování procesoru.

Výhody

- Rychlost:
  - spuštění vlákna je 10-100x rychlejší než `fork()` (jinak by neměla vlákna moc smysl).
- Prostředky:
  - vlákna sdílejí celý adresový prostor (procesy jen kód popř sdílenou paměť) (není zde ochrana!),
  - Zásobník je jediné místo, kde vlákno nemusí řešit vícenásobný přístup.

HyperThreading
: přináší více kontextů na jednom procesoru, ale běh je stále bez paralelizace, procesor se pak tváří jako 2 CPU.

Přidělování procesoru vláknům

- **globální** (každé vlákno je nezávislé)
- **lokální** (přidělován procesu, proces rozhoduje o přidělování vláknům)

Implementace vláken

N:N (1:1)

Všechna vlákna jsou na úrovni OS: OS/2, WinNT, LinuxThreads, NPTL

- [+] volání jádra je přímé a rychlé
- [+] plné využití multiprocessingu
- [-] jádro musí všechna vlákna evidovat, zabírají jadernou paměť
- [-] při přepnutí vlákna větší režie přepnutí (jde přes jádro)

N:1

OS vidí jen procesy, vlákna jsou v userspace pomocí knihoven: FreeBSD < 5.x

- [+] nízká režie vlákna (paměťová i časová)
- [+] plná kontrola plánování (nezasahuje OS)
- [-] blokující volání jádra se zapouzdřují
- [-] nejde použít pro více procesorů (CPU dostane celý proces)

N:M

kombinovaný přístup, OS vidí M vláken z N: Solaris, AIX, Irix, FreeBSD

- [+] OS vidí jen potřebná vlákna (většinou podle počtu CPU)

- [+] čekající vlákna už OS nevidí, bez režie
- [+] zároveň lze přepínat kontext i v userspace
- [+] lze simulovat N:N i N:1
- [-] velmi problematická implementace

LWP (Lightweight Process)

je implementace M:N (Solaris).

- Jedno vlákno na úrovni procesoru obsahuje staticky několik uživatelských
- Plánování je problém, userspace neví moc o tom, co se děje v jádře s LWP.
- LWP jsou pro jádro normálními procesy (chová se k nim jako ke starým procesům).
- proces běží jako jednu LWP vlákno

KSE (Kernel Scheduler Entities)

je nová implementace pomocí konceptu **Scheduler Activations** (FreeBSD > 5.x).

- Procesor se přiděluje skupinám KSE.
- Skupina obsahuje 1 až (počet CPU) aktivních KSE.
- Vlákna, která je potřeba provádět, dostanou přiděleno jedno KSE.
- Pokud je vlákno pozastaveno, probudí se jiné a je to ohlášeno userspace.

Pthreads

je implementace vláken podle POSIX.

- Vlákno se spouští voláním funkce, která obsahuje kód vlákna (volající pak pokračuje dál jako původní vlákno)
- Funkce vlákna má signaturu `void *vlakno(void *arg)`
- Volá se pomocí, je možné nastavit atributy vlákna.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, /* atributy */, void *
(*func)(void *) /* vlákno */, void *arg)/* parametr */
```

- Čekání na ukončení a převzetí stavu se `pthread_join()`
- Každé vlákno je identifikováno strukturou `pthread_t`.

Atributy vláken

- lokální/globální plánování
- plánovací algoritmus
- možnost ukončení bez čekání na stav
- velikost zásobníku
- apod.

Problémy implementace
- vlákna sdílejí paměť, starší konstrukce pak kolidují (errno, localtime, strtok), nutnost definovat `_REENTRANT`.

Reentrantnost
Schopnost alokovat dynamická data na zásobníku. Díky tomu může být funkce volána vícekrát (rekurze, sdílení kódu více procesy nebo vlákny) a jednotlivé instance si tak nepřepisují svá data.

Synchronizace, vzájemné vyloučení

Tohle teoreticky do okruhu nepatří (loni to byl samostatný okruh), ale pro jistotu to sem dám. není to kompletní a není to revidovaný

Souběžnost procesů (vláken)

- **Jednoprocesorový systém** - pseudosouběžnost
  - **Preemptivní** - procesor může být procesu odebrán kdykoliv
  - **Nepreemptivní** - proces se sám musí vzdát procesoru
- **Víceprocesorový systém** - reálná souběžnost

Nedeterminismus
: se projevuje v paralelně běžících procesech (a vláknech), při provádění není jasná posloupnost akcí.

Časově závislé chyby (Race condition)
: chyba vznikající kvůli různé rychlosti paralelních procesů, je potřeba zavést synchronizaci.

Synchronizace
: je zajištění kooperace mezi procesy pomocí synchronizačních okamžiků (procesy čekají na ostatní).

Atomická operace
: je taková, která se provede celá najednou nebo vůbec. Zajišťuje konzistenci dat.

- **Čtení nebo zápis z/do paměti**
- **Speciální instrukce** (Test&Set)

: pozn.: při práci s pamětí je nutné si dát pozor na zarovnání slov

Vzájemné vyloučení (mutual exclusion)
: užívá synchronizaci k tomu, aby danou operaci mohl provádět pouze jediný proces.

Kritická sekce
: je úsek programu, jehož provádění je vzájemně vyloučené. Je důležité aby se do ní nedostal dva procesy naráz.

- KS musí být provedena v konečném čase (ošetření ukončení procesu).
- Proces může do KS vstoupit nekonečně krát.
- Pokud v KS nikdo není, musí do ní jít vstoupit okamžitě.
- Přidělování procesoru je spravedlivé.

Spravedlnost (fairness) přidělení CPU

- **Unconditional** - každý aktivní nepodmíněný atomický příkaz bude někdy proveden
- **Weak** - každý podmíněný atomický příkaz bude proveden, pokud nabude hodnoty TRUE a nebude se měnit (pokud se mění nemusí být proveden nikdy)
- **Strong** - provedou se i příkazy, jejichž hodnoty podmínek se libovolně krát změní(prakticky neproveditelný)

Bezpečný (safe) algoritmus
: zaručuje vzájemné vyloučení.

Živý (live) algoritmus
: je bezpečný, bez uváznutí, blokování a stárnutí

Uváznutí (deadlock)
: procesy čekají v synchronizaci na stav, který by mohl nastat, kdyby jeden z nich mohl pokračovat. Obtížná detekce, nenastává deterministicky.

Blokování (blocking)

postup procesu je blokován jiným procesem i když to není nutné. Nebezpečné pokud např. pokud druhý proces skončí, může trvale blokovat proces první

Stárnutí/vyhladovění (starving)
: je dlouhé čekání na splnění podmínky vstupu do KS, která nenastává v okamžiku testování (např. předbíhání). Může být tolerováno, pokud je vyloučeno praktickým nasazením.

Synchronizační nástroje

- zámky
- semafory
- monitory
- zprávy

Synchronizační techniky
: čtení/zápis, zákaz přerušení, instrukce (implementují nástroje)

- **Aktivní čekání** - neustálé testování nějaké podmínky, stále zaměstnává procesor, není vhodné pro dlouhodobé čekání
- **Pasivní čekání** - uspání procesu procesu a provádění procesu jiného, režie spojená s pozastavením procesu je mnohem vyšší u krátkodobého čekání než při použití aktivního čekání, pasivní čekání se tak používá především u dlouhodobého čekání

Přidělování procesoru
: znamená přepnutí kontextu mezi dvěma procesy, vykonává dispatcher.

- Všechny čekající procesy se obvykle ukládají do hashovací tabulky, kde je klíčem adresa objektu, na který se čeká.

Plánování
: (scheduling) je volba strategie a řazení procesů do fronty.

- Minimalizuje odezvu programů (u desktopových systémů menší časová kvanta).
- Efektivní využití prostředků (u serverů delší kvanta a priority).
- Spravedlivé přidělování.
- Zvýšení průchodnosti (transakcí v čase).

Implementace vzájemného vyloučení

Vzájemné vyloučení pro 1 CPU

Atomický kód
: je úsek kódu, při kterém nemůže dojít k přerušení nebo přepnutí kontextu.

Zakázání přerušení

: zaručuje, že obsluha přerušení nenaruší KS

- Problém je, že komunikace s řadičem trvá moc dlouho, takže vypnutí a zapnutí má vysokou režii. Lze obejít tak, že se zaznamená blokování a pokud IRQ přijde, tak se neobslouží, ale uloží k obsloužení později.

Zakázání přepnutí kontextu v jádře

- a) **pomocí zákazu přerušení** - ale tak se zakáže i I/O

- b) **zakázat preemptivní přepínání kontextu v jádře** (Unix)
  - Kontext se může přepnout: synchronně při zahájení čekání, při ukončení služby jádra, při návratu z přerušení
  - Nevýhodou je delší reakce na události na úrovni prostorů
- c) **Semafory a povolení preemptivního přepínání v jádře** (dlouhodobé vyloučení v klasickém UNIXu)

Uživatelské procesy

- a) **služby jádra** - používaly nástroje v jádře (semafory apod.), dříve používáno, ale volání jádra má příliš velkou režii
- b) **futex** (Fast Userspace Mutex), který jádro volá pouze, pokud je již zámek uzamčen, jinak se pouze atomicky testuje hodnota.

Vzájemné vyloučení pro více CPU

Víceprocesorové prostředí přináší další problémy. Vypnutí přerušení nezabrání běhu procesů na ostatních procesorech.

Krátkodobé vyloučení

- Pro kritické sekce, které jsou **krátké, neblokující a bez preempce** (nesmí se přerušit její běh)
- Je možné i **aktivní čekání**, protože přepnutí by mělo vysokou režii
- Je potřeba pro implementaci dalších nástrojů.

- a) **Implementace čtením a zápisem** - pro více procesorů moc složité
- b) **Speciální atomické instrukce** - instrukce nedělitelného současného čtení a zápisu (RMW)
  - **Test &Set** (XCHG, Compare&Swap) - bezpečné, bez uváznutí a blokování, hrozí stárnutí
  - **Load Linked a Store Conditional** - (RISC), Load Linked čte obsah paměti, Store Conditional ukládá do paměti pouze pokud do paměti nikdo nezapsal od poslední Load Linked
  - **Fetch & Add** - FA(var, incr) = <t = var; var = var + incr; return t>

Problémy:

- instrukce zatěžují paměťovou sběrnici (stále něco testují)
- u HT se blokuje i druhý procesor aktivním čekáním
- možnost stárnutí

Dlouhodobé vyloučení

využívá **binární zámek** (první sekce zamkne a odemkne při odchodu).

Komplikace - Přístup do paměti

- **Silná konzistence** - Přístupy v pořadí, v jakém procesory žádaly (neefektivní)
- **TSO** (Total Store Ordering) - Pořadí zachováno pro jednotlivé procesory.
- **PSO** (Partial Store Ordering) - Pořadí zápisů proházeno i na jednom procesoru.

Instrukce pro vzájemné vyloučení

- Oboje implementované v procesorech 80x86 jako atomická instrukce

- Test-and-set

```
int testAndSet (target)
int *target;
{ int value = *target;
    *target = 1;
```

```
    return (value);
}
```

- Swap

```
void Swap(a, b)
int *a;
int *b;
{ int temp = *a;
    *a = *b;
    *b = temp;
}
```

Použití instrukcí pro vyloučení

- Test-and-set solution

```
shared var lock = 0;
repeat
    while testAndSet(&lock) do skip;
    critical section
    lock = 0;
    remainder section
until false;
```

- Atomic Swap Solution

```
shared var lock = 0;
repeat
    key := 1;
    repeat
        swap (&lock, &key);
    until key=0;
    critical section
    lock := 0;
    remainder section
until false;
```

Algoritmus Bounded Wait (Test & Set)

- pro libovolný počet procesů

```
shared var
waiting: array [0..n-1] of boolean; // Want to enter
lock: boolean
var j: 0..n-1;
key: boolean;

repeat
  waiting[i] := true;                 // Wants to enter
  key := true;
  while waiting[i] and key do
    key := testAndSet (&lock);                  i
  waiting[i] := false;
  critical section
  j := i+1 mod n;
  while (j<>i) and (not waiting[j])
    do j := j+1 mod n;
  if j=i then lock := 0
    else waiting[j] := false;
  remainder section
until false;
```

Ticket algoritmus (serializer)

- Simple solution for pro n processes

```
shared var number = 1, next = 1, turn: array [1..n];
repeat
       <turn[i] = number; number = number + 1>
       <await turn[i] = next>
       critical section
       <next = next + 1>
until false;
```

Implementation using fetch_and_add

```
FA(var, incr) = <t = var; var = var + incr; return t>

shared var ...
repeat
       turn[i] = FA(number, 1);
       while turn[i] <> next do skip;
       critical section
       next = next + 1; {need not be atomic}
until false;
```

Citováno z „http://wiki.fituska.eu/index.php?title=Procesy_a_vl%C3%A1kna_POSIX&oldid=12347“
Kategorie: Státnice 2011 | Pokročilé operační systémy

- Stránka byla naposledy editována 16. 6. 2014 v 19:21.