Operační systémy

Správa procesoru

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

Procesy (1/2)

- neformálně: proces = běžící program (vykonává činnost)
- proces:
  - paměťový prostor
  - kód programu
  - data – statická a dynamická (halda)
  - zásobník
  - registry

Obecný životní cyklus procesu

- nový (new) – proces byl vytvořen
- připravený (ready) – proces čeká, až mu bude přidělen CPU
- běžící (running) – CPU byl přidělen procesor a právě provádí činnost
- čekající (waiting) – proces čeká na nějakou událost (např. na vyřízení I/O požadavku)
- ukončený (terminated) – proces skončil svou činnost

Procesy (2/2)

- multi-tasking: (zdánlivě) souběžný běh více procesů
- informace o procesu má OS uloženy v **process control block** (PCB)
- obsahuje informace:
  - identifikace procesu
  - informace o stavu
  - adresu instrukce, kterou bude program pokračovat
  - stav registrů
  - informace k plánování procesů
  - informace o přidělené paměti
  - informace o používaných I/O zařízeních, otevřených souborech, atd.
- zásadní role při předávání procesoru mezi procesy
- uložení stavu CPU (context switch)
- $\Longrightarrow$ jde řešit softwarově nebo s podporou HW
- kooperativní vs. preemptivní multitasking
- preemptivní multitasking $\Longrightarrow$ velikost časového kvanta
- potřeba plánování

Plánování procesů (1/2)

Požadavky

- férovost – každému procesu by v rozumné době měl být přidělen CPU
- efektivita – maximální využití CPU
- maximalizace odvedené práce (throughput)
- minimalizace doby odezvy
- minimalizace doby průchodu systémem (turnaround)

Round robin

- každý proces má pevně stanovené kvantum
- připravené procesy jsou řazeny ve frontě a postupně dostavají CPU
- spravedlivý

Priority

- připravené procesy jsou řazeny podle priorit
- riziko vyhladovění

- reálně se používají heuristické algoritmy sdílející obě vlastnosti

Plánování procesů (2/2)

Různé typy úloh

- interaktivní
- dávkové zpracování
- pracující v realném čase
- intenzivní na CPU vs. I/O

Symmetric Multi-processing

- přibývá problém, jak vybrat procesor
- $\Longrightarrow$ přesun procesu $\Longrightarrow$ vyprázdnění cache
- řešení: oddělené plánování pro každý procesor
- nebo maska affinity (definuje procesor/jádro, na kterém může proces běžet)

Vlákna (Threads)

- OS umožňují rozdělení procesu na víc vláken
- základní entitou vykonávající program není proces, ale vlákno
- proces se skládá z jednoho a více vláken
- každé vlákno má svůj zásobník + registry $\Longrightarrow$ přepnutí vláken v rámci procesu rychlejší
- vlákna v rámci procesu sdílí paměťový prostor, data, prostředky $\Longrightarrow$ jednoduché sdílení prostředků $\Longrightarrow$ nové problémy se synchronizací

Realizace

- vztah kernel thread - user thread
- kernel thread – vlákno, jak jej vídí OS
- user thread – vlákno z pohledu uživatelského prostoru
  - 1:N – vlákna jsou realizovaná v uživatelském prostoru (problém s plánováním)
  - 1:1 – k jednomu vlánku v uživatelském prostoru existuje jedno v jaderném prostoru (nejčastější použití)
  - M:N – vlákna v uživatelském prostředí jsou mapována na menší počet jaderných vláken

Procesy a vlákna v Unixech

- původně základní entitou byl proces
- procesy tvoří hierarchii rodič-potomek (navrcholu je proces init)
- vytvoření procesu pomocí volání `fork()`, které vytvoří klon procesu
- přepsání procesu pomocí `exec`
- komunikace mezi procesy: zasílání signálů, roury, ...
- možnost nastavit *niceness* procesu (prioritu od -20 do 19)
- vlákna do Unixu přidána až později (implementace se v rámci různých OS liší)
- v Linuxu (pthreads) vnitřně implementovány stejně jako procesy (task), ale sdílí paměťový prostor

Procesy a vlákna ve Windows

- Windows NT navrženy s vlákny jako základní entitou pro běh aplikace
- proces sdružuje vlákna, plánování se účastní vlákno
- sofistikovaný systém priorit při plánování vláken
- priorita vlákna je odvozená od třídy priority procesu
- proměnlivá velikost kvant
- priority boost – dochází k dočasnému zvýšení priority
  - po dokončení I/O operace
  - po dokončení čekání na semafor
  - probuzení vlákna, jako reakce na UI
  - vlákno již dlouhodobu něběželo

Synchronizace vláken a procesů

- příklad: přičtení jedničky dvěma procesy
  1. proměnná X = 1
  2. A: načte hodnotu do registru
  3. A: zvýší hodnotu v registru o jedna

     přepnutí procesu

  4. B: načte hodnotu do registru
  5. B: zvýší hodnotu v registry o jedna
  6. B: uloží hodnotu do paměti (X = 2)

     přepnutí procesu

  7. A: uloží hodnotu do paměti (X = 2)

- řešení – atomické operace nebo synchronizace
- je potřeba zajistit, aby v průběhu manipulace s určitými zdroji s nimi nemohl manipulovat někdo jiný (vzájemné vyloučení)

Atomický přístup do paměti

- vícevláknové aplikace/víceprocesorové počítače (+ out-of-order provádění operací)
- obecně přístupy do paměti nemusí být atomické (záležitost CPU)
- klíčové slovo `volatile` – často záleží na překladači
- *memory barriers* umožňují vynutit si synchronizaci (záležitost CPU)

Atomické operace

- Compare-and-Swap (CAS): ověří, jestli se daná hodnota rovná požadované, a pokud ano, přiřadí ji novou (CMPXCHG)
- Fetch-and-Add: vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna (XADD)
- Load-link/Store-Conditional (LL/CS): načte hodnotu, a pokud během čtení nebyla změněna, uloží do ní novou hodnotu
- umožňují implementovat synchronizační primitiva na vyšší úrovni

Synchronizační primitiva

- slouží k řízení přístupu ke sdíleným prostředkům
- Petersonův algoritmus (umožňuje implementovat vzájemné vyloučení dvou procesů, pomocí běžných operací)

Mutex & Kritická sekce

- potřebujeme zajistit, aby kód pracující se sdílenými zdroji mohlo použít jen jedno vlákno/proces $\Longrightarrow$ kritická sekce
- realizace pomocí mutexu
- dvě operace:
- vstoupit do kritické sekce (pokud je v kritické sekci jiný proces/vlákno, proces čeká)
- opustit kritickou sekci (pokud na vstup do kritické sekce čeká jiné vlákno, je vpuštěno)

Kritická sekce pomocí SWAP

- uvažujme následující **atomickou** operaci, která prohodí dvě hodnoty

```
void swap(bool *a, bool *b) {
   bool tmp = *a;
   *a = *b;
   *b = tmp;
}
```

- a kód

```
key = true;
while (key == true)
   swap(&lock, &key);
// kritická sekce
lock = false;
```

Semafor (1/2)

- chráněná proměnná obsahující počítadlo s nezápornými čísly
- operace P: pokud je hodnota čísla nenulová, sníží hodnotu o jedna, jinak čeká, až bude hodnota zvýšena
- operace V: zvýší hodnotu o jedna
- operace P a V se provádí atomicky

```
void P(Semaphore s) {
  while (s <= 0) { }
  s--;
}

void V(Semaphore s) {
  s++;
}
```

Semafor (2/2)

```
struct sem {
   int value;
   struct process * list;
};
void P(struct sem * s) {
   s->value--;
   if (s->value < 0) {
      // přidej proces do s->list;
      block(); // uspi aktuální proces
   }
}
void V(struct sem * s) {
   s->value++;
   if (s->value <= 0) {
      // odeber process P z s->list
      wakeup(P);
   }
}
```

Monitor

- modul nebo objekt
- v jeden okamžik může kteroukoliv metodu používat pouze jeden proces/vlákno
- nutná podpora prog. jazyka
- Java (`synchronized`), .NET (`lock`)
- rozšíření o podporu čekání (`Wait`, `Pulse`, `PulseAll`) $\Longrightarrow$ možnost odemčít zámek společně s čekáním
- monitor v C#

```
lock (obj) {
   // kriticka sekce
}
```

Deadlock

- uváznutí – systém se dostal do stavu, kdy nemůže dál pokračovat

Užívání prostředků

- request – požadavek na prostředek, není-li k dispozici, proces čeká
- use – proces s prostředkem pracuje
- release – uvolnění prostředku pro další použití

Podmínky vzniku

- Mutual exclusion – alespoň jeden prostředek je výlučně užíván jedním procesem
- Hold & wait – proces vlastní alespoň jeden prostředek a čeká na další
- No preemption – prostředek nelze násilně odebrat
- Circular wait – cyklické čekání (proces A vlastní prostředek 1, chce prostředek 2, který drží B a současně žádá o 1)

Řešení deadlocku (1/4)

Ignorace

- ,,neřešení", v praxi často používané

Detekce (detection & recovery)

- pokud vznikne deadlock, je detekován a některý proces odstraněn
- k detekci se používá *alokační graf prostředků* a *graf čekání*
- alokační graf:
  - orientovaný graf
  - dva typy uzlů – prostředek, proces
  - hrana proces-prostředek – proces čeká na prostředek
  - hrana prostředek-proces – prostředek je vlastněn procesem
- graf čekání vznikne vynecháním uzlů prostředků a přidáním hran $P_n \to P_m$ pokud existovaly hrany $P_n \to R$ a $R \to P_m$, kde P_n a P_m jsou procesy a R je prostředek
- deadlock vznikne, pokud je v grafu čekání cyklus
- Kdy má smysl provádět detekci?
- Jak vybrat oběť?

Řešení deadlocku (2/4)

Zamezení vzniku (prevention)

- snažíme se zajistit, že některá z podmínek není splněna
- zamezení výlučnému vlastnění prostředků (často nelze z povahy prostředku)
- zamezení držení a čekání
  - proces zažádá o všechny prostředky hned na začatku
  - problém s odhadem
  - plítvání a hladovění
  - množství prostředků nemusí být známé předem
  - jde použít i v průběhu procesu (ale proces se musí vzdát všech prostředků)
- zavedení možnosti odejmout prostředek – vhodné tam, kde lze odejmout prostředky tak, aby nešlo poznat, že byly odebrány
- zamezení cyklickému čekání – zavedení globálního číslování prostředků a možnost žádat prostředky jen v daném pořadí

Řešení deadlocku (3/4)

Vyhýbání se uváznutí (avoidance)

- procesy žádají prostředky libovolně
- systém se snaží vyhovět těm požadavkům, které nemohou vést k uváznutí
- **bezpečný stav** – existuje pořadí procesů, ve kterém jejich požadavky budou vyřízeny bez vzniku deadlocku
- tomu je přizpůsobeno plánovaní procesů
- je potřeba znát předem, kolik prostředků bude vyžádáno
- systém, který není v bezpečném stavu, nemusí být v deadlocku
- systém odmítne přidělení prostředků, pokud by to znamenalo přechod do nebezpečného stavu (proces musí čekat)

Řešení deadlocku (4/4)

Algoritmus na bázi alokačního grafu

- vhodný pokud existuje jen jedna instance každého prostředku
- do alokačního grafu přidáme hrany (proces-prostředek) označující potenciální žádosti procesu a prostředky
- žádosti a prostředek se vyhoví pouze tehdy, pokud konverze hrany na hranu typu (prostředek-je vlastněn-procesem) nepovede ke vzniku cyklu

Bankéřův algoritmus

- vhodný tam, kde je větší počet prostředků daného typu
- na začátku každý proces oznamí, kolik prostředků jakého typu bude maximálně potřebovat
- při žádosti o prostředky systém ověří, jestli se nedostane do nebezpečného stavu
- pokud nelze vyhovět, je proces pozdržen
- porovnávají se volné prostředky, s aktualně přidělenými a maximálními

Bankéřův algoritmus (1/2)

- uvažujme m prostředků a n procesů
- matice $n \times m$
  - max – počet prostředků, které bude každý proces žádat
  - $assigned$ – počet přiřazených prostředků jednotlivým procesům
  - $needed$ – počet prostředků, které bude každý proces ještě potřebovat (evidentně $needed = max - assigned$)
- vektory velikosti m
  - E – počet existujících prostředků
  - P – počet aktuálně držených prostředků
  - A – počet dostupných zdrojů (evidentně $A = E - P$)

Algoritmus

1. najdi řádek i v $needed$ takový, že $needed[i] \leq A$, pokud takový není, systém není v bezpečném stavu
2. předpokládej, že proces skončil a uvolnil své zdroje, i.e., $A \leftarrow A + assigned[i]$
3. opakuj body 1 a 2, dokud nejsou odstraněny všechny procesy nebo není jasné, že systém není v bezpečném stavu

Bankéřův algoritmus (2/2)

	K	L	M	N
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

assigned

	K	L	M	N
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

needed

$E = \langle 6, 3, 4, 2 \rangle$
$P = \langle 5, 3, 2, 2 \rangle$
$A = \langle 1, 0, 2, 0 \rangle$

- podmínku splňuje proces $D \Longrightarrow$ odebrán a $A \leftarrow \langle 2, 1, 2, 1 \rangle$
- podmínku splňuje proces $A \Longrightarrow$ odebrán a $A \leftarrow \langle 5, 1, 3, 2 \rangle$
- podmínku splňuje proces $B \Longrightarrow$ odebrán a $A \leftarrow \langle 5, 2, 3, 2 \rangle$
- podmínku splňuje proces $C \Longrightarrow$ odebrán a $A \leftarrow \langle 6, 3, 4, 2 \rangle$
- podmínku splňuje proces $E \Longrightarrow$ odebrán a $A \leftarrow \langle 6, 3, 4, 2 \rangle$

Samostudium

- Keprt A. Operační systémy.
- Kapitoly 9–11, tj. strany 84–128.