

Synchronizační primitiva

Paralelní a distribuované systémy, přednáška 3

Tomáš Urbanec

Katedra informatiky PŘF UPOL

7.10.2025

Složené atomické akce

Složené atomické akce

- Doted' čtení a zápis proměnné jako dvě akce.
- Lze provádět obojí najednou (atomicky).
- Více možností:
 1. test-and-set,
 2. exchange (swap),
 3. fetch-and-add,
 4. compare-and-swap,
 5. load-link/store-conditional (pár operací),
 6. ...
- Jsou různě silné (4 a 5 silnější).
- Vedou k zajímavým úvahám (např. ABA problém).

Složené atomické akce a KS

Tabule - řešení kritické sekce pomocí:

1. test-and-set,
2. exchange,
3. fetch-and-add,
4. compare-and-swap.

Složené atomické akce
ooo

Zámek
●ooooooooooooooooo

Semafor
ooo

Monitor
ooooooooo

Další nástroje a vzory
ooooo

Zámek

Zámek (Mutex)

- Nejjednodušší synchronizační primitivum.
- Dva stavy: zamknuto nebo odemknuto.
- Jednoduché použití, ale pozor na přehnanou synchronizaci.

KS zámek

Sdílená paměť

lock ← unlocked

Proces i

opakuj:

- 1: Nekritická sekce
 - 2: lock.Lock()
 - 3: Kritická sekce
 - 4: lock.Unlock()
-

Příklad

Zámek

Pokus

- n lidí hází kostkou.
- Počítají kolikrát (všem dohromady) padla trojka.
- Komu padne šestka, ten končí.

Základní kód

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

Proces *i*

```
1: repeat
2:     vysledek  $\leftarrow$  HodKostkou()
3:     if vysledek = 3
4:         pocet  $\leftarrow$  pocet + 1
5:     if vysledek = 6
6:         break
```

→ sdílený čítač nějakého jevu (padla trojka)

Pokus 1

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces *i*

1: lock.Lock()

2: repeat

3: vysledek $\leftarrow$ HodKostkou()

4: if vysledek = 3

5: pocet $\leftarrow$ pocet + 1

6: if vysledek = 6

7: break

8: lock.Unlock()

Analýza pokusu 1

Zámek

- Funguje.
- Umožňuje jen serializovaný běh procesů (jeden po druhém).

Analýza pokusu 1

Zámek

- Funguje.
 - Umožňuje jen serializovaný běh procesů (jeden po druhém).
- Zamyká celý pokus.
- Přehnaná synchronizace zrušila veškeré efekty paralelizace.

Pokus 2

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces i

1: repeat

2: lock.Lock()

3: vysledek $\leftarrow$ HodKostkou()

4: if vysledek = 3

5: pocet $\leftarrow$ pocet + 1

6: if vysledek = 6

7: lock.Unlock()

8: break

9: lock.Unlock()

Analýza pokusu 2

Zámek

- Funguje.
- Umožňuje předávat kostku mezi hody.

Analýza pokusu 2

Zámek

- Funguje.
 - Umožňuje předávat kostku mezi hody.
- Zamyká práci s kostkou.
- Lepší, ale jde to i lépe.

Pokus 3

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces i

1: repeat

2: lock.Lock()

3: vysledek $\leftarrow$ HodKostkou()

4: if vysledek = 3

5: pocet $\leftarrow$ pocet + 1

8: lock.Unlock()

6: if vysledek = 6

7: break

Analýza pokusu 3

Zámek

- Nefunguje. Deadlock. Kde?

Analýza pokusu 3

Zámek

- Nefunguje. Deadlock. Kde?
- Musíte si být jistí, které scénáře synchronizací připouštíte.

Analýza pokusu 3

Zámek

- Nefunguje. Deadlock. Kde?
- Musíte si být jistí, které scénáře synchronizací připouštíte.
- Pro reálné příklady intuitivně těžké.
- Formální logika.

Pokus 4

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces i

1: repeat

2: vysledek $\leftarrow$ HodKostkou()

3: lock.Lock()

4: if vysledek = 3

5: pocet $\leftarrow$ pocet + 1

6: lock.Unlock()

7: if vysledek = 6

8: lock.Unlock()

9: break

Analýza pokusu 4

Zámek

- Nefunguje.
- Deadlock. Kde?

Analýza pokusu 4

Zámek

- Nefunguje.
 - Deadlock. Kde?
- Co když padne 4?

Pokus 4'

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces i

1: repeat

2: vysledek $\leftarrow$ HodKostkou()

3: lock.Lock()

4: if vysledek = 3

5: pocet $\leftarrow$ pocet + 1

6: if vysledek = 6

7: lock.Unlock()

8: break

9: lock.Unlock()

Analýza pokusu 4'

Zámek

- Funguje.
- Umožňuje házet všem naráz.
- Každý musí mít svou kostku.

Analýza pokusu 4'

Zámek

- Funguje.
- Umožňuje házet všem naráz.
- Každý musí mít svou kostku.
- Zamyká zpracování výsledku hodu.
- HodKostkou() nesmí spoléhat na vzájemné vyloučení.
- Lepší, ale jde to i lépe.

Pokus 5

Zámek

Počet trojek

Sdílená paměť

pocet $\leftarrow$ 0

lock $\leftarrow$ unlocked

Proces *i*

1: repeat

2: vysledek $\leftarrow$ HodKostkou()

3: if vysledek = 3

4: lock.Lock()

5: pocet $\leftarrow$ pocet + 1

8: lock.Unlock()

6: if vysledek = 6

7: break

Analýza pokusu 5

Zámek

- Funguje.
- HodKostkou() nesmí očekávat vzájemné vyloučení.
- Zamyká jen společný čítač (prací s papírem s aktuálním výsledkem pokusu).
- Lépe už to nepůjde.

Složené atomické akce
ooo

Zámek
oooooooooooooooo

Semafor
●oo

Monitor
oooooooo

Další nástroje a vzory
ooooo

Semafor

Semafor

≈ chráněný čítač.

- Zámek je binární semafor.
- Stav 0 až n .
- Operace:
 - Čekání (též dekrementace, P)
 - Signalizace (též inkrementace, V)
- Čekání - proces přijde k semaforu a:
 - buď vidí stav 0 a čeká až stav bude > 0 ,
 - nebo dekrementuje stav o 1 a pokračuje dál.
- Signalizace - proces inkrementuje stav semaforu o 1.
- Lze implementovat jako (zapouzdrěný) čítač chráněný zámkem (zkuste si).
- Implementace nevyžaduje aktivní čekání (podpora OS).
- Pořád ale relativně nízkoúrovňové (nestrukturovaný stav).

Kritická sekce a semafor

KS semaforem

Sdílená paměť

semaphore $\leftarrow$ 1

Proces i

opakuj:

- 1: Nekritická sekce
 - 2: semaphore.Wait()
 - 3: Kritická sekce
 - 4: semaphore.Signal()
-

- Nic nebrání vyhladovění.
- potřebujeme silnou férovost plánovače/OS
nebo férovou implementaci (fronta čekajících)

Složené atomické akce
ooo

Zámek
oooooooooooooooo

Semafor
ooo

Monitor
●oooooooo

Další nástroje a vzory
ooooo

Monitor

Monitor

≈ strukturovaný synchronizační nástroj

- Jeden pohled:
 - Zobecnění jádra OS ve smyslu, že se KS řeší v privilegovaném režimu (tj. v monitoru).
 - Místo jednoho „jádra“ má každý objekt/modul/... svůj monitor.
 - V jednom okamžiku s monitorem pracuje (používá jeho operace) nejvýše jeden proces.
- Druhý pohled:
 - Zobecnění objektu z OOP, ve smyslu, že každý monitor je instance nějaké třídy (má vnitřní data, má operace).
 - Navíc ale s každou instancí může v jednom okamžiku pracovat nejvýše jeden proces.
 - Všechna data monitoru musí být zapouzdřená - stav nelze měnit jinak než definovanými operacemi.

Monitor

- Dost jazyků monitory neposkytuje.
 - Mají je např:
 - Java a další JVM jazyky*,
 - C# a další .NET jazyky*,
 - Ada, Ruby, ...
 - Ostatní obvykle nabízí zámky a podmíněné proměnné nebo jiné nástroje, kterými lze monitor modelovat.
- * není to tak jednoduché.

Kritická sekce a monitor

KS monitorem

Sdílená paměť

monitor M:

data monitoru (čísla, flagy, ...)

operation criticalOperation():

Kritická sekce

Proces *i*

opakuj:

1: Nekritická sekce

2: M.criticalOperation()

- Kód kritické sekce je schovaný v monitoru (tj. ne v každém procesu).
- Procesy nemusí řešit explicitní zámky atp.
- Ani zde nic nebrání vyhladovění.

Podmíněná proměnná

- Monitor $\approx$ implicitní vzájemné vyloučení (volání operací, přístup k datům).
- Občas je potřeba explicitní synchronizaci (např. producent/konzument a plný/prázdný buffer).
- Potřeba čekání se dá vyjádřit podmínkou (tj. čekám na její splnění).
- Pak můžeme:
 - explicitně čekat, až podmínka bude splněna (operace *waitC*),
 - explicitně signalizovat, že podmínka je splněna (operace *signalC*).
- Podmínka má frontu procesů, které na ni čekají.
- Nepotřebuje aktivní čekání.
- Nižší úroveň než monitor.
- Často dostupná práv v montioru.
- Monitor lze implementovat pomocí podmíněné proměnné a zámku.

Operace

Podmíněná proměnná

Podmíněná proměnná

Sdílená paměť

monitor M:

cond $\leftarrow$ Queue()

operation waitC(cond):

cond.Enqueue(process)

process.state $\leftarrow$ blocked

M.lock.unlock()

operation signalC(cond):

if not(cond.isEmpty()):

process $\leftarrow$ cond.Dequeue()

process.state $\leftarrow$ ready

-
- Pokud proces začne čekat, uvolní monitor. Proč?

Příklad

Semafor pomocí monitoru

Sdílená paměť

monitor Sem:

$s \leftarrow n$

$\text{notZero} \leftarrow \text{Queue}()$

operation Wait():

if $s = 0$:

$\text{waitC}(\text{notZero})$

$s \leftarrow s - 1$

operation Signal():

$s \leftarrow s + 1$

$\text{signalC}(\text{notZero})$

Proces i

opakuj:

1: Nekritická sekce

2: Sem.Wait()

3: Kritická sekce

4: Sem.Signal()

Monitor

Kdo pokračuje při signalC?

- V příkladu ok, ale obecně i za signalC může být další příkaz.
- Netriviální otázka.
- Proces S vyvolá signalC, čímž uvolní na waitC čekající proces W. Kdo pokračuje?
 - S? A co s W?
 - W? A co s S?
 - A co ostatní čekající (E)?
 - Jeden musí počkat - přiřadíme jim prioritu.
 - klasicky: $W > S > E$ (Signal and urgent wait, IRR)
 - možné i $S > W = E$ (uvolněný musí znovu ověřit)

Čekání na Monitor vs. Semafor

Semafor	Monitor
<code>wait</code> nemusí blokovat	<code>waitC</code> vždy blokuje
<code>signal</code> vždy něco udělá	<code>signalC</code> nemusí udělat nic
<code>signal</code> uvolní nějaký proces	<code>signalC</code> uvolní první ve frontě
uvolněný proces okamžitě pracuje	$W > S$ nebo $S > W$?

Složené atomické akce
ooo

Zámek
oooooooooooooooo

Semafor
ooo

Monitor
oooooooo

Další nástroje a vzory
●oooo

Další nástroje a vzory

Bariéra

≈ místo v programu, kde se musí sejít více procesů.

- Tj. čekáme na ostatní a až pak pokračujeme.
- Dvě fáze
 - procesy přicházejí (a čekají),
 - propuštění procesů.
- Existuje více možností, jak to provést:
 - Lineární, stromová, motýlová.
 - Různé způsoby propouštění (jde o rychlost).

Jendorázová lineární bariéra

Sdílená paměť

semaphore barrier $\leftarrow$ 0, counterMutex $\leftarrow$ 1

count $\leftarrow$ 0

threadCount $\leftarrow$ N

Proces i

1: counterMutex.wait()

2: count $\leftarrow$ count + 1

3: if count = threadCount

4: barrier.signal()

5: counterMutex.signal()

6: barrier.wait()

7: barrier.signal()

ThreadPool

- Návrhový vzor i běžný nástroj
- „Pool“ vláken.
- Vlákna přiřazována úkolům.
- Nové úkoly přidávány do fronty.
- Když je ve frontě úkol a uvolní se vlákno, tak se úkol předá vláknu.
- Nemusíme znovu vytvářet vlákna (efektivita).
- Běžně dostupné v jazycích.
- Často používané.



M. Ben-Ari. 2006. *Principles of concurrent and distributed programming*. Druhé vydání. Prentice-Hall, USA.



Per Brinch Hansen. 1972. *Structured multiprogramming*. Commun. ACM 15, 7, 574–578. Dostupné z: <https://doi.org/10.1145/361454.361473>



Per Brinch Hansen. 1972. *A comparison of two synchronizing concepts*. Acta Inf. 1, 3, 190–199. Dostupné z: <https://doi.org/10.1007/BF00288684>



C. A. R. Hoare. 1974. *Monitors: an operating system structuring concept*. Commun. ACM 17, 10, 549–557. Dostupné z: <https://doi.org/10.1145/355620.361161>

Changelog

- 9.10.2025
 - Pokus 4 a 4'.