Synchronizace vláken

Další synchronizační problémy, typické chyby, Coffmanovy podmínky

Milan Radojčić

SSPŠaG – Operační systémy

23. 2. 2026

Opakování: Co je to vlákno?

- Vícevláknový program má více míst, odkud spouští instrukce.
- Vlákna mezi sebou **sdílí paměť**.
  - Tedy sdílí adresový prostor.

Adresový prostor s 1 vláknem

Opakování: Semafory

- Semafory *limitují počet vláken, které mohou přistupovat ke sdílenému zdroji*.
- **Inicializace:** `new Semaphore(initial, maximum)`.
  - Čítač uvnitř semaforu je nastaven na `initial`.
- **Do kritické sekce vstoupíme** pomocí `Semaphore.WaitOne()`.
  - Pokud je čítač > 0, tak se sníží a vlákno vstoupí do kritické sekce.
  - Pokud je čítač = 0, tak se vlákno uspí, dokud není čítač inkrementován.
- **Při opuštění kritické sekce** zavoláme `Semaphore.Release()`.
  - Inkrementuje čítač o 1.
- `new Semaphore(1, 1)` se chová jako lock.

Opakování: Použití semaforu na čítač

```
1   using System.Threading;
2   class MyProg {
3       ...
4       static Semaphore sem = new Semaphore(1, 1);
5       static void Main() { ... }
6       static void Increment() {
7           sem.WaitOne();
8           for (int i = 0; i < 10_000_000; i++)
9               counter++;
10          sem.Release();
11      }
12  }
```

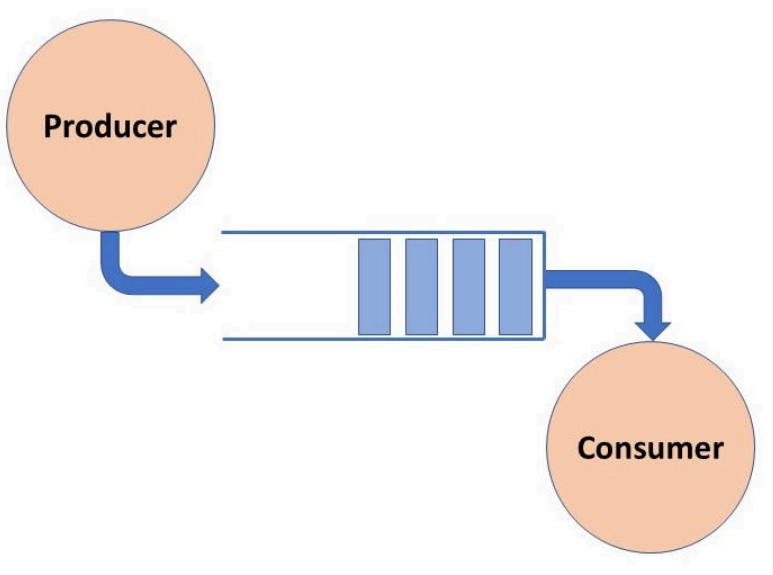
Opakování: Řazení událostí pomocí semaforu

Na kolik nastavit X v inicializaci semaforu?

```
1   class MyProg {
2       static Semaphore sem = new Semaphore(X, 100);
3       static void Main() {
4           Console.WriteLine("Vypise se 1.");
5           var t1 = new Thread(PrintStuff);
6           t1.Start();
7           sem.WaitOne();
8           Console.WriteLine("Vypise se 3.");
9       }
10      static void PrintStuff() {
11          Console.WriteLine("Vypise se 2.");
12          sem.Release();
13      }}
```

Producent/konzument

- Anglicky *producer/consumer* nebo *bounded buffer problem*.
- Máme nějaký počet **producentů** a **konzumentů**.
- Producenti **vkládají data/joby do sdíleného bufferu**.
- Konzumenti **berou data/joby ze sdíleného bufferu** a zpracovávají je.
- Může nastat v mnoha praktických situacích: např. HTTP server.



Producent/konzument

```
1   class MyProg {
2       static Queue queue;
3
4       static void Put(int val) {
5           Debug.Assert(queue.Count ≠ MAX);
6           queue.Enqueue(val);
7       }
8
9       static int Get() {
10          Debug.Assert(queue.Count ≠ 0);
11          return queue.Dequeue();
12      }
13  }
```

Producent/konzument

Proč tohle není dostačující?

```
1   class MyProg {
2       static Mutex mut;
3
4       static void Put2(int val) {
5           mut.WaitOne();
6           Put(val);
7           mut.ReleaseMutex();
8       }
9       ... // stejně Get()
10  }
```

Producent/konzument: řazení

```
1  class MyProg {
2      static Semaphore full = new Semaphore(0, MAX);
3      static Semaphore empty = new Semaphore(MAX, MAX);
4      ...
5  }
```

```
1  static void Put2(int val) {
2      empty.WaitOne(); // sníží empty o 1
3      Put(val);
4      full.Release(); // zvýší full o 1
5  }
```

```
1  static void Get2(int val) {
2      full.WaitOne(); // sníží full o 1
3      Get(val);
4      empty.Release(); // zvýší empty o 1
5  }
```

Na co jsme v tomto řešení zapomněli?

Producent/konzument: vzájemné vyloučení

```
1  class MyProg {
2      static Semaphore full = new Semaphore(0, MAX);
3      static Semaphore empty = new Semaphore(MAX, MAX);
4      static Mutex mut = new Mutex();
5      ... }
```

```
1  static void Put2(int val) {
2      mut.WaitOne();
3      empty.WaitOne();
4      Put(val);
5      full.Release();
6      mut.Release(); }
```

```
1  static void Get2(int val) {
2      mut.WaitOne();
3      full.WaitOne();
4      Get(val);
5      empty.Release();
6      mut.Release(); }
```

Je nějaký problém s tímto přístupem?

Producent/konzument: správné vzájemné vyloučení

```
1  class MyProg {
2      static Semaphore full = new Semaphore(0, MAX);
3      static Semaphore empty = new Semaphore(MAX, MAX);
4      static Mutex mut = new Mutex();
5       ... }
```

```
1  static void Put2(int val) {
2      empty.WaitOne();
3      mut.WaitOne();
4      Put(val);
5      mut.Release();
6      full.Release(); }
```

```
1  static void Get2(int val) {
2      full.WaitOne();
3      mut.WaitOne();
4      Get(val);
5      mut.Release();
6      empty.Release(); }
```

Čtenáři/písaři

- Chceme synchronizovat přístup k nějaké datové struktuře, databázi, atd.
- **Několik vláken může současně číst.**
- Ale jen **jedno vlákno může v jednu chvíli zapisovat!**
  - A zároveň v tu chvíli nemůže žádné vlákno číst.

Čtenáři/písaři: řešení

```
1  class MyProg {
2    static Mutex mainlock, writelock;
3    int readers;
4       ... }
```

```
1  void AcquireReadLock() {
2    mainlock.WaitOne();
3    readers++;
4    if (readers == 1)
5      writelock.WaitOne();
6    mainLock.Release();
7  }
```

```
1  void ReleaseReadLock() {
2    mainlock.WaitOne();
3    readers--;
4    if (readers == 0)
5      writelock.Release();
6    mainLock.Release();
7  }
```

Jak bude vypadat kód pro získání zámku pro zapisování?

Čtenáři/písaři: dokončení

```
1  void AcquireWriteLock() {
2    writelock.WaitOne();
3  }
```

```
1  void ReleaseWriteLock() {
2    writelock.Release();
3  }
```

Toto řešení funguje, má ale jiný problém. Jaký?

Čtenáři/písaři: hladovění

Čtenář 0	Čtenář 1	Písař 0	Čtenář 2
chce číst			
získá zámek			
získá writelock			
		chce zapisovat	
	chce číst		
	získá zámek		
vzdá se zámku			
			chce číst
			získá zámek
			vzdá se zámku
chce číst			
získá zámek			
vzdá se zámku			
	vzdá se zámku		
	uvolní writelock		
		získá writelock	

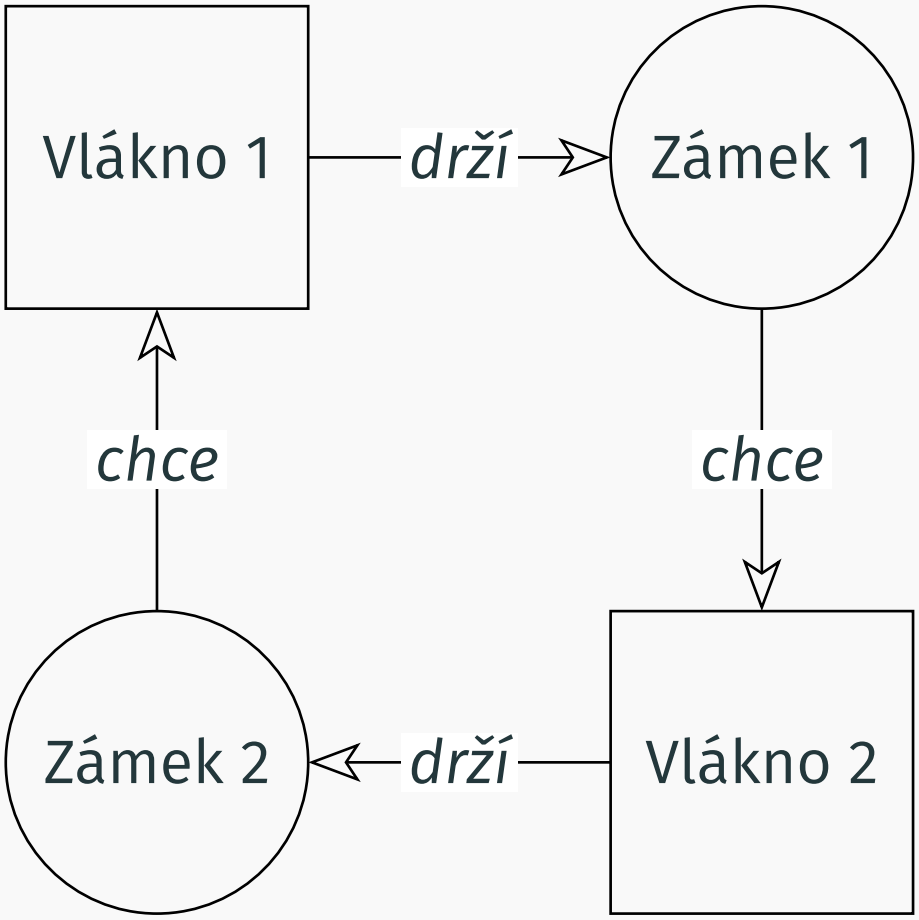
Typy chyb

Na nějaké chyby spojené se synchronizací jsme už narazili, jaké to jsou?

- **Chybí synchronizace přístupu** – např. náš čítač na začátku
- **Chybějící synchronizace pořadí** – např. jeden thread používá proměnnou, dřív něž ji druhý inicializuje
- **Uváznutí** (*deadlock*)

Deadlock

- Využití zámků vlákny se dá vizualizovat pomocí **alokačního grafu**.
- Pokud **vlákno drží zámek** tak jde šipka směrem **vlákno – zámek**.
- Pokud **vlákno chce získat zámek** tak jde šipka směrem **zámek – vlákno**.
- Deadlock odpovídá **cyklu** v tomto grafu.



Deadlock 2 vláken

Alokační graf pro večeřící filosofy

- Vzpomeňte si na naše řešení problému večeřících filosofů s deadlockem.
- Jaké zdroje v něm máme?
- Jak vypadá alokační graf pro případ deadlocku?

Coffmanovy podmínky

- Jsou nutnou a postačující podmínkou pro deadlock.
- Prakticky jsou užitečné v tom, že stačí jednu z nich porušit aby nemohl deadlock nastat.

Coffmanovy podmínky

1. **Vzájemné vyloučení**: Vlákna používají nějaký typ vzájemného vyloučení např. zámky.
2. **Podmínka drž a čekej**: Vlákna drží zdroje (zámky atd.) a mezitím čekat na další.
3. **Podmínka neodnímatelnosti**: Vláknům se nedají zdroje odebrat silou.
4. **Kruhové čekání**: Existuje smyčka vláken, kde každé vlákno čeká na zdroje, které má další vlákno ve smyčce.

Prevence kruhového čekání

- Pokud nějaká funkce potřebuje víc zámků, tak pevně definujeme v jakém pořadí se budou zámky získávat.
  - To jsme udělali u večeřících filosofů.
- Problém může být jak seřadit zámky v komplikovanějších případech.

Porušení podmínky drž a čekej

- Deadlock může nastat pouze tehdy, když vlákno může žádat o další prostředky když už drží zámek.
- Pro prevenci můžeme přinutit vlákna, aby **všechny zámky získávaly najednou**.

```
1  Mutex[] locks;
2  Mutex prevention;
3  void TakeForks(int i) {
4      prevention.WaitOne();
5      locks[L(i)].WaitOne();
6      locks[R(i)].WaitOne();
7      prevention.ReleaseMutex();
8  }
9  // zbytek stejně
```

Prevence uváznutí

- Můžeme přidat **timeout k pokusu o získání zámku**. (V C# je na to nepovinný parametr.)
  - Pokud zámek do té doby nedostaneme, tak vše resetujeme a pokusíme se získat všechny zámky zase.

```
1 void TakeForks(int i) {
2 top:
3     locks[L(i)].WaitOne();
4     if (!locks[R(i)].WaitOne(1000)) {
5       locks[L(i)].Release();
6       goto top;
7     }
8 }
```

- Novým problémem je tady tzv. **livelock** – vlákna se můžou stále dokola dostávat do deadlocku a pak resetovat.

Prevence uváznutí

- Někdy se můžeme obejít bez mutexů.
  - Např. v našem příkladu s čítačem: existuje atomická inkrement instrukce, kterou bychom mohli využít.
- Existuje např. i **Petersonův algoritmus**, který poskytuje vzájemné vyloučení bez zámků.
  - Nefunguje však na moderních superskalárních procesorech.
- Můžeme také **plánovat chytřeji** – např. nenaplánujeme zároveň dva filosofy, co sdílí vidličku.
  - Viz Dijkstrův *bankéřův algoritmus*.

Tyhle přístupy nemusí být vždycky uplatnitelné. S posledními dvěmi se můžete spíše setkat třeba u vestavěných (embedded) systémů.

2. cvičení: Autobusová zastávka

- Pasažéři (thready) můžou přicházet na zastávku, tam čekají než přijede autobus.
- Jakmile přijede autobus (další vlákno) tak nastoupí max 10 threadů.
  - Pokud je na zastávce míň threadů (klidně 0) tak autobus odjede s tímto počtem.
- Za každého nastupujícího pasažéra musíte zavolat `PassengerBoard()`.
  - Když bus odjíždí tak musíte zavolat `BusDepart()`.