Synchronizace vláken

Milan Radojčić

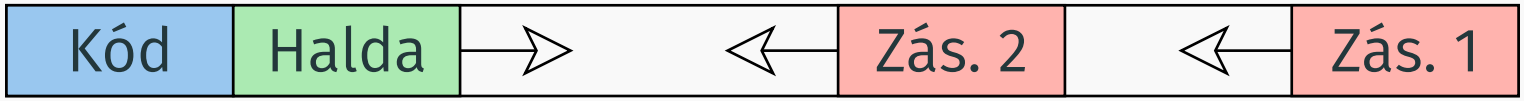
SSPŠaG – Operační systémy

12. 1. 2026

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



Adresový prostor se 2 vlákny

- Každé vlákno má **vlastní zásobník** v adresovém prostoru.

Využití pro vlákna

- Typickým využitím je zrychlování výpočtů, které lze jednoduše paralelizovat.
  - Dekódování videa, násobení matic, atd.
- Dále se vlákna používají pro provádění **pomalého blokujícího IO na pozadí**.[1]

[1] Např. u GUI aplikací nechcete, aby UI přestalo být responzivní když se provádí nějaký síťový požadavek.

Vytváření vláken v C#

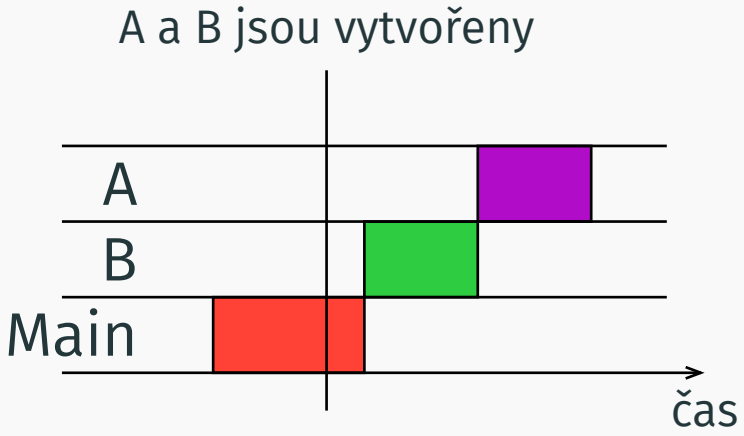
- Pomocí `new Thread(Function)` můžeme vytvořit nové vlákno.
  - Metoda `Thread.Start()` dané vlákno spustí a
  - `Thread.Join()` blokuje aktuální vlákno, než dané vlákno skončí.

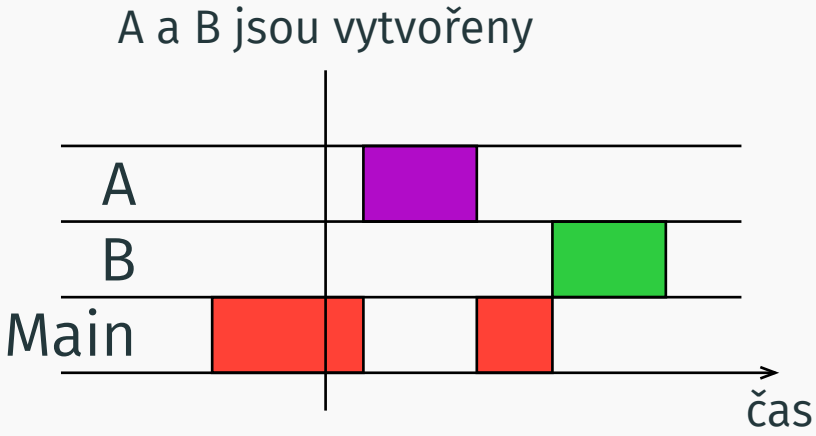
Ukázka: co vypíše program?

```
1   class MyProg {
2       static void Main() {
3           var t1 = new Thread(_ ⇒ PrintName("A"));
4           var t2 = new Thread(_ ⇒ PrintName("B"));
5           t1.Start(); t2.Start();
6           t1.Join(); t2.Join();
7       }
8
9       static void PrintName(string name) {
10          Console.WriteLine("Printing: " + name);
11      }
12  }
```

Ukázka: možné naplánování

My nevíme, jak budou přesně vlákna naplánována.





Vypíše se **B, A**.

Vypíše se **A, B**.

Ukázka: co vypíše program?

```
1   class MyProg {
2       static int counter = 0;
3       static void Main() {
4           var t1 = new Thread(Increment);
5           var t2 = new Thread(Increment);
6           t1.Start(); t2.Start();
7           t1.Join(); t2.Join();
8           Console.WriteLine("Counter: " + counter);
9       }
10
11      static void Increment() {
12          for (int i = 0; i < 10_000_000; i++)
13              counter++;
14      }
15  }
```

Ukázka: co se děje

- Abychom pochopili předchozí ukázku, musíme si ukázat jak funguje zvyšování čítače na úrovni instrukcí.
  1. Nejprve se **načte hodnota čítače z paměti do nějakého registru**.
  2. Poté se **hodnota v registru inkrementuje**.
  3. Nová hodnota v registru se až poté **zapisuje zpět do paměti**.
- Co když se stane, že se vlákna přeplánují po načtení z paměti, ale **před zápisem zpět**?

Ukázka: co se děje

- Řekněme, že hodnota v čítači je 1000, pak může nastat následující situace.
  1. Vlákno A načte z paměti hodnotu 1000 do registru.
  2. A inkrementuje hodnotu v registru.
  3. **Přeplánuje se na vlákno B.**
  4. B načte z paměti **také 1000**.
  5. B inkrementuje hodnotu v registru.
  6. B zapíše do paměti hodnotu 1001.
  7. **Přeplánuje se zpět na vlákno A.**
  8. **A taky zapíše do paměti hodnotu 1001.**

Proč v operačních systémech?

- Abychom psali korektní vícevláknové programy, budeme **potřebovat podporu od OS**.
- **Samotný OS je vícevláknový program** — musí být správně synchronizován.

Synchronizační nástroje: hardware

- Abychom mohli vytvářet vyšší synchronizační nástroje[2] potřebujeme nějakou podporu od hardwaru.
- Potřebujeme, aby procesor podporoval instrukci, která **atomicky vymění hodnotu v registru s pamětí**.[3]
- Obecně se této instrukci říká **test-and-set**, **compare-and-swap** nebo **atomic exchange**.
- V každé architektuře se jmenuje trochu jinak:
  - SPARC: `ldstub` – load and store unsigned byte
  - x86: `xchg` – exchange
  - RISC-V: `amocas` – atomic compare and swap

[2] Semafory, mutexy, bariéry, atd.

[3] Atomicky znamená „v jednom kroku.“

Jednoduchý spinlock

```
1   class SpinLock {
2     int flag;
3
4     void init() {
5       flag = 0;
6     }
7
8     void lock() {
9       while (TestAndSet(ref flag, 1) == 1) {}
10    }
11
12    void unlock() {
13      flag = 0;
14    }
15  }
```

Jednoduchý spinlock

- Předchozí je spíš takový pseudokód, ktarý ilustruje jak fungují spinlocky.
  - Funkce `TestAndSet` v C# neexistuje — je náhrada za naší test-and-set instrukci.
- Když chce vlákno získat zámek, tak **aktivně testuje hodnotu pořád dokola**.
  - Může být žádoucí, když čekáme, že se zámek brzo uvolní.
  - Nemusí být žádoucí, pokud tohle budeme dělat dlouho a budeme tím brát čas na CPU.
- Později si ukážeme jiný způsob, jak postavit zámek na blokování.

Pojmy spojené se synchonizací 1

- **Kritická sekce** je místo, kde vlákna přistupují ke *sdíleným zdrojům*.
  - V našem příkladu s čítačem je to funkce `Increment`:

```
1  static void Increment() {
2      for (int i = 0; i < 10_000_000; i++)
3          counter++; }
```

- **Race condition** (nebo *data race*) nastává, když se dvě vlákny najednou pokusí upravit sdílené data.
- Vlákna by měly využívat nějaký typ **vzájemného vyloučení** (angl. *mutual exclusion*), aby jen jedno z nich mohlo být v kritické sekci.

Použití spinlocku na čítač

```
1   using System.Threading;
2   class MyProg {
3       ...
4       static SpinLock s1 = new SpinLock();
5       static void Main() { ... }
6       static void Increment() {
7           for (int i = 0; i < 10_000_000; i++) {
8               bool lockTaken = false;
9               do {
10                  s1.Enter(ref lockTaken);
11              } while (!lockTaken);
12              counter++;
13              s1.Exit(); }
14      }
15  }
```

Použití spinlocku na čítač

- Předchozí přístup **nemusí být optimální** — spousta architektur podporuje atomickou inkrement instrukci.

Mutex

- **Mutex** je zkratka pro mutual exclusion lock.
- Mutexy většinou **blokují vlákna, které chtějí získat zámek**.
- Pokud nějaké vlákno A chce získat zámek, ale už ho vlastní vlákno B, tak je A uspáno dokud B zámek neuvolní.
- Efektivnější než spinlock, když víme, že budeme dlouho čekat.
  - Na druhou stranu, pokud budeme čekat jen chvíli, může být spinlock lepší.[4]

[4] Uspání vlákna totiž vyžaduje nějakou režii od OS.

Použití mutexu na čítač

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

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.

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  }
```

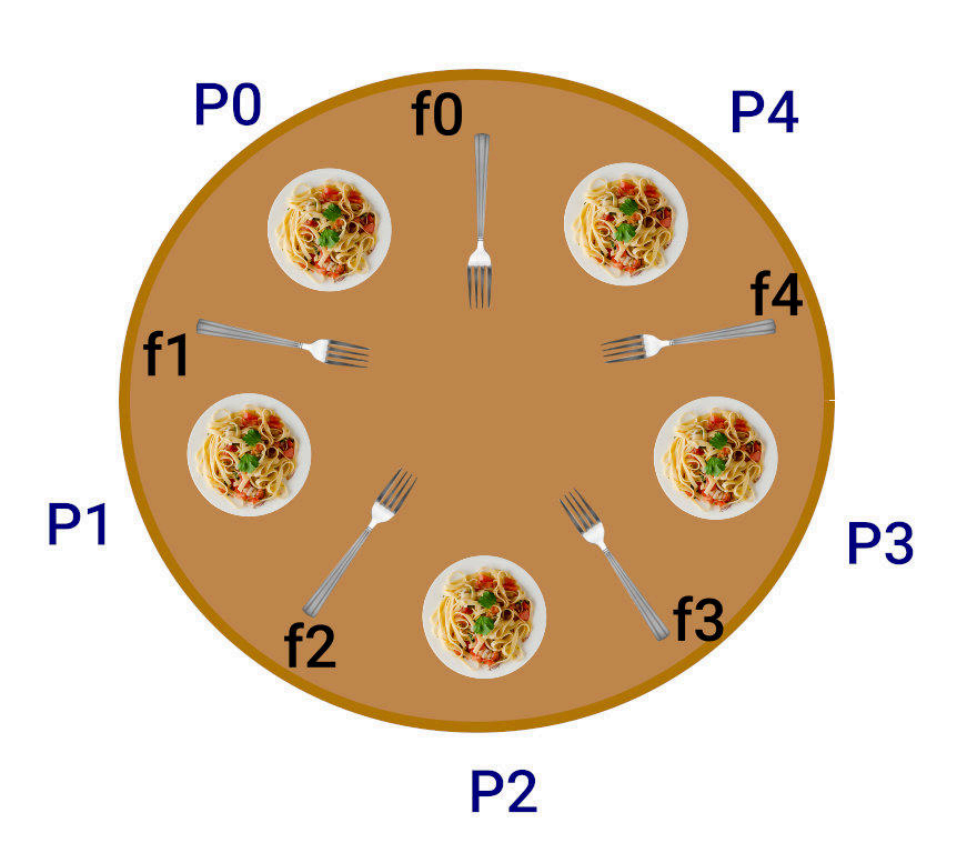
Řazení událostí pomocí semaforu

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      }}
```

Večeřící filosofové

- Ukážeme si jeden klasický[5] synchronizační problém.



- **Filozofové** (P0, P1, ...) sedí v kruhu a mají mezi sebou **vidličky** (f0, f1, ...).
- Filosof může být v jednom ze **2 stavů**:
  - Buď *přemýšlí a nic nedělá*.
  - Nebo může dostat hlad, v tom případě musí **chytnout obě vidličky**, aby mohl jíst.

[5] Poprvé publikován a vyřešen Edsgerem Dijkstrou v roce 1971.

Večeřící filosofové: naivní řešení

```
1   void Philosopher(int i) {
2       while (true) {
3           Think();
4           TakeFork(Left(i));
5           TakeFork(Right(i));
6           Eat();
7           PutFork(Left(i));
8           PutFork(Right(i));
9       }
10  }
```

- `TakeFork(i)` se pokusí získat `i`-tou vidličku.
  - Pokud je zabraná tak filosof čeká.
- `PutFork(i)` uvolní `i`-tou vidličku.

Vidíte nějaký problém s tímto řešením?

Večeřící filosofové: funkční řešení

```
1   Mutex mut;
2   void Philosopher(int i) {
3       while (true) {
4           Think();
5           mut.WaitOne();
6           TakeFork(Left(i));
7           TakeFork(Right(i));
8           Eat();
9           PutFork(Left(i));
10          PutFork(Right(i));
11          mut.ReleaseMutex();
12      }
13  }
```

Vidíte nějaký problém s tímto řešením?

Večeřící filosofové: řešení pomocí semaforů

```
1  Semaphore[] sems;
2  void Philosopher(int i) {
3      while (true) {
4          Think();
5          TakeForks(i);
6          Eat();
7          PutForks(i);
8      };
9  }
```

```
10 void TakeForks(int i) {
11     sems[L(i)].WaitOne();
12     sems[R(i)].WaitOne();
13 }
14
15 void PutForks(int i) {
16     sems[L(i)].Release();
17     sems[R(i)].Release();
18 }
```

- Semafory jsou inicializovány na 1.
- Vidíte nějaký problém s tímto řešením?

Večeřící filosofové: řešení pomocí semaforů

- Řekněme, že je 5 filosofů, v předchozím řešení může nastat deadlock následovně:

1. Filosof 0 dostane hlad, vezme vidličku 0.
2. Filosof 1 dostane hlad, vezme vidličku 1.
3. ...
4. Filosof 4 dostane hlad, vezme vidličku 4.

- Dostali jsme se do situace, kdy každý filosof drží levou vidličku.
- Každý z nich ale chce získat pravou vidličku, ale nikdo ji neuvolní — **deadlock**.

Máte nápad jak předejít deadlocku?

Večeřící filosofové: finální řešení

- Abychom zamezili kruhovému čekání, můžeme změnit způsob jakým jeden filosof bere vidličky.
- Např. poslední filosof vezme nejprve pravou a pak levou vidličku.

```
1 void TakeForks(int i) {
2     if (i == 4) {
3         sems[R(i)].WaitOne();
4         sems[L(i)].WaitOne();
5     } else {
6         sems[L(i)].WaitOne();
7         sems[R(i)].WaitOne();
8     }
9 }
```

Večeřící filosofové: finální řešení

- Ukažme si, jak řeší poslední úprava problém kruhového čekání.

1. Filosof 0 dostane hlad, vezme vidličku 0.
2. Filosof 1 dostane hlad, vezme vidličku 1.
3. ...
4. Filosof 4 dostane hlad, **zkusí vzít vidličku 0**.
   - Tu má filosof 0, takže čeká na uvolnění.

- V této chvíli nenastane deadlock.
  - Filosof 3 může totiž vzít vidličku 4 — tím získá obě vidličky a může se najíst.
  - Pak může získat obě vidličky filosof 2 atd.

Shrnutí

- Co jsou to vlákna.
- Odkud se berou synchronizační bugy.
  - Co se děje na úrovni HW.
- Synchronizační primitiva, podpora od HW.
- Jednoduché synchronizační příklady, problém večeřících filosofů.