# Úlohy z vláknového programování
# PV192

Lukáš Hejtmánek, Petr Holub

## Obsah



Z uvedených příkladů si každý student vybere jeden, který *samostatně* zpracuje. Pokud si dva studenti vyberou stejné zadání, bude v rámci zkoušky posuzována i originalita řešení. Úkoly, které považujeme za obtížnější, jsou označeny hvězdičkou – při diskusi jejich řešení bude brán zřetel na vyšší obtížnost úkolu. V případě, že budete mít vlastní nápad na vhodné zadání, ozvěte se nám – pokud bude rozumné, tak Vám po předchozí dohodě jeho implementaci umožníme.

S případnými dotazy se obracejte na adresy:

- Lukáš Hejtmánek (*xhejtman@ics.muni.cz*) ohledně úkolů v jazyce C/C++,
- Petr Holub (*hopet@ics.muni.cz*) ohledně úkolů v jazycích Java a Ada.

eventuálně na kteroukoli z výše zmíněných adres pokud jde o úkoly nezávislé na jazyku (a také v případě nedostupnosti jednoho z nás).

# 1 Úlohy nezávislé na programovacím jazyce

## Úkol 1: Paralelní floodfill

S pomocí knihovny `libSDL` (případně jiné vhodné knihovny pro práci s grafikou) implementujte algoritmus *flood fill*[1]. Aplikace bude schopna paralelně (parametricky 1 až $n$ vláken) vyplnit obrazec (alespoň pětiúhelník). Každé vlákno bude kreslit svou barvou, již vyplněná plocha se nesmí vyplňovat znova.

## Úkol 2: Přetahovaná ve vláknech

S pomocí knihovny `libSDL` (případně jiné vhodné knihovny pro práci s grafikou) implementuje přetahování o provaz. Vytvořte 4 vlákna, která představují hráče přetahované. Hráči stojí v rozích čtverce, od každého vede lano doprostřed čtverce, kde jsou svázaná dohromady. Vlákna tahají náhodná čísla, která odpovídají přitáhnutí provazu směrem k hráči. Střed lan se bude pohybovat uvnitř čtverce podle výsledků náhodných čísel. Hra končí, jakmile nějaký hráč bude mít střed lan u sebe.

Jako prémiové obtížnější zadání je možno tuto úlohu implementovat pro parametrizovatelný počet $n$ za provaz tahajících vláken.

## Úkol 3: Reflektor UDP paketů

Implementujte paralelní reflektor UDP paketů s následující funkcionalitou:

1. Každý přijatý paket je analyzován na zdrojovou IP adresu a port: pokud tato kombinace ještě není mezi účastníky, je tam automaticky přidána.
2. Každý přijatý paket je rozeslán všem účastníkům.
3. Každý účastník, který déle jak 120 s neposlal na reflektor ani jeden paket je vyloučen ze seznamu účastníků
4. Účastníky je možno přidat na začátku explicitně a tito účastníci nepodléhají 120 s expiraci.

Přijímání, odesílání a housekeeping budou řešeny jako samostatná vlákna. Navrhněte a implementujte vhodné datové struktury, proveďte měření a diskutujte výkonnost implementovaného řešení. Případně můžete srovnat s jednovláknovým řešením.

# 2 C/C++

## Úkol 4: Producent konzument

Naimplementujte 2 varianty klasické úlohy producent/konzument. Producent bude nějakou rychlostí produkovat objekty, konzument je bude nějakou rychlostí spotře-

[1] `http://en.wikipedia.org/wiki/Flood_fill`

bovávat. Maximální množství vyprodukovaných ale nezkonzumovaných objektů bude omezeno – tj. nejen konzument bude čekat na producenta, ale i producent bude čekat na konzumenta, produkuje-li příliš rychle. Jako objekty použijte např. 4 MB kusy paměti vyplněné nějakou hodnotou.

1. Objekty řaďte do fronty, kterou není nutné zamykat. Zkonzumované objekty neuvolňujte a předávejte zpět producentovi k opětovnému použití.

2. Objekty řaďte do obyčejné fronty, kterou při přidání prvku či odebrání prvku zamknete. Objekty alokujte nové a staré uvolňujte.

Srovnejte rychlost obou řešení.

### Úkol 5: Kancelační body

Implementujte pomocí vláken vyhledávání v textu. Vygenerujte si nějaký řádově stomegabytový soubor s textem a pomocí několika vláken v něm vyhledávejte nějaký vzor. Jakmile některé vlákno vzor najde, pošle ostatním *cancel* a oznámí úspěch (každé ukončené vlákno po sobě musí „uklidit“, tj. zavřít soubor a podobně. Vlákna musí prohledávat vždy jinou část souboru. V další části programu vytvořte nová vlákna a spočítejte četnost nalezeného vzoru. Výsledek opět oznamte na terminál.

### Úkol 6: Procházení seznamu

Implementujte následující aplikaci. Master vlákno prochází seznam objektů. Každý objekt má tři atributy: `int x`, `int y`, `char data[9000]`. Master vlákno odebere ze seznamu objekt a dá ho na zpracování workerům. Workeři sdílí jeden velký buffer velikosti `int w` * `int h`. Worker převezme objekt, podívá se, jestli `x * y` není již za hranicí bufferu ($x * y + 9000 < w * h$), pokud není, zavolá `memcpy(buffer + x*y, data, 9000)`. Pokud je, uvolní buffer a alokuje nový tak, aby se do něj objekt vešel a následně do něj data nakopíruje.

Pozor na to, že nelze uvolnit buffer, do kterého jiný worker právě kopíruje data. Aplikace by měla škálovat na více procesorů.

### Úkol 7: Paralelní quicksort

Implementujte paralelně algoritmus třízení quicksort[2]. Porovnejte rychlost paralelní implementace se sekvenční verzí. Paralelní verze by měla škálovat na více procesorech. Třiďte objekty typu double.

### Úkol 8: Lock-free datové struktury

Implementujte korektně lock-free RCU strukturu. Tedy strukturu, která je převážně čtena a občas aktualizována. Aktualizace se provádí pomocí kopie prvku nebo celé

[2] `http://cs.wikipedia.org/wiki/Quicksort`

struktury a aktualizace je provedena v nové kopii. Původní prvek nebo strukturu je potřeba uvolnit až ve chvíli, kdy se nepoužívá. Navrhněte vhodný způsob dodatečného uvolňování těchto objektů či struktur.

Pro test použijte mapu nebo seznam o řádově miliónech položkách, po cca 1000 přečtených položkách jednu změnte. Porovnejte se sekvenčním řešením a ukažte, jak aplikace škáluje pro 1-4 CPU jádra.

# 3 Java

### Úkol 9: Měření výkonu zámků

Navrhněte (najděte) a implementujte vhodný algoritmus pro porovnání výkonu

- `synchronized`
- `ReentrantLock`
- `ReadWriteLock`

Proveďte a zpracujte měření v rozsahu 1 až 32 vláken, a to v režimech nízkého až středního versus vysokého soutěžení o zámek.

### Úkol 10: Měření výkonu sychronizace pomocí atomických proměnných

Navrhněte (najděte) a implementujte vhodný algoritmus pro porovnání výkonu

- `synchronized/ReentrantLock`
- `Atomic`*`XXX`*
- `ThreadLocal`

Proveďte a zpracujte měření v rozsahu 1 až 32 vláken, a to jak v režimu nízkého a vysokého „soutěžení o zámek“ (i tam, kde se ve skutečnosti nezamyká ;-) ).

### Úkol 11: Měření výkonu kolekcí

Zpracujte studii porovnávající výkon synchronizovaných a paralelních kolekcí, zejména

- `CopyOnWrite`*`XXX`* v porovnání se synchronizovanými seznamy a množinami,
- `ConcurrentHashMap` v porovnání se `synchronizedMap` a `Hashtable`,
- srovnání s `WaitFreeReadQueue` a `WaitFreeWriteQueue`, podaří-li se získat přistup k funkční implementaci.

V rámci studie je třeba vybrat a implementovat několik vhodných algoritmů, na kterých bude srovnání provedeno.

### Úkol 12: Implementace webového crawleru

Pomocí vzoru *kradení práce s vlastní frontou úloh pro každé vlákno* implementujte webový crawler, který bude procházet web počínaje zadanou stránkou do parametricky zadané hloubky a bude počítat statistiku výskytu HTML elementů na stránkách. Pro počítání statistik zvolte vhodnou strukturu (tak, abyste byli své rozhodnutí schopni zdůvodnit!). Crawler si bude vnitřně měřit statistiky, jak často se použil

prostý model práce producent-konzument a jak často bylo využito mechanismu kradení práce.

### Úkol 13: Signalizace

Pomocí signalizačních mechanismů (`wait()`, `notify()`, `notifyAll()`) implementujte ekvivalenty následujících synchronizačních mechanismů:

- BlockingQueue
- BlockingDequeue
- semafor
- CountDownLatch
- bariéru
- Exchanger

### Úkol 14: Paralelizace rozvrhování

Úkolem studenta je zparalelizovat sekvenční optimalizační cyklus. Tento cyklus se používá na optimalizaci rozvrhu či plánu – například školního rozvrhu nebo plánu pořadí výpočtu úloh na (super)počítačích.

Jeden cyklus se skládá že tři kroku:

1. výběr úlohy(předmětů) z rozvrhu
2. přesunutí úlohy na jiné místo v rozvrhu
3. vyhodnocení této změny a její přijetí/zamítnutí v závislosti na zlepšení/zhoršení rozvrhu

Takovýto algoritmus lze velice dobře paralelizovat. Nejdůležitějším úkolem studenta tak bude efektivně vyřešit problém reprezentace rozvrhu, který musí pro jednotlivá vlákna zůstat konzistentní, tj. nesmí být ovlivněn kroky ostatních vláken. Triviální řešení představuje duplikace této datové struktury pro každé nové vlákno, což ale nemusí být paměťové efektivní. Dalším zajímavým problémem je otázka duplikovaných kroku, kdy bez vzájemné komunikace vláken hrozí, že jednotlivá vlákna budou duplikovat již jednou provedené optimalizační kroky jiných vláken. Ve finále je potom potřeba provést sadu experimentálních vyhodnocení, kdy student změří výslednou kvalitu řešení a časově nároky paralelní i původní sekvenční verze programu. Dále pak podle potřeby vyhodnotí jednotlivé možnosti paralelizace a na základě experimentálních výsledků (např. časová vs. paměťová náročnost) zvolí nejvhodnější variantu řešení.

Sekvenční algoritmus pro zájemce poskytne dr. Klusáček.

# 4 Ada

### Úkol 15: Implementace semaforu, závlačky a bariéry v Adě

Implementujte semafor, závlačku (ekvivalent `CountDownLatch` z Javy) a bariéru dvěma způsoby:
(1) pomocí tasků,
(2) pomocí protected typů.
Pro každý z uvedených typů implementujte jednoduchou aplikaci, která jej bude používat.

### Úkol 16: Neblokující zásobník a fronta

Implementujte v Adě následující dvě struktury:

- neblokující zásobník Treiberovým algoritmem[3] – implementace pro Javu je dostupná na adrese `http://www.javaconcurrencyinpractice.com/listings/ConcurrentStack.java`,
- neblokující frontu algoritmem Michael-Scott[4] – probráno na přednášce.

Pro implementaci využijte generika. Vytvořte jednoduchý příklad, který struktury bude využívat.

### *Úkol 17: Implementace thread pools v Adě

Navrhněte a implementujte mechanismus ekvivalentní thread pools v jazyce Ada. Využijte předávání práce pomocí access typů na procedury. Thread pools budou mít jako parametry minimální počet vláken, maximální počet vláken a hodnotu keep-alive, po jejímž uplynutí bez vykonání práce budou vlákna ukončena.

[3] R. K. Treiber. "Systems Programming: Coping with Parallelism.", *Technical Report RJ 5118*, IBM Almaden Research Center, April 1986.

[4] M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking ad Blocking Concurrent Queue Algorithms." In *Symposium on Principles of Distributed Computing*, pages 267-275. 1996. `http://citeseer.ist.psu.edu/michael96simple.html`