Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Paralelismus, asynchronismus a souběh v moderních programovacích jazycích

Bc. Yaroslav Zenchenko

Diplomová práce

2021

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Yaroslav Zenchenko**
Osobní číslo:	**I19293**
Studijní program:	**N0613A140007 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Paralelismus, asynchronismus a souběh v moderních programovacích jazycích**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je provést rešerši a popis principů asynchronního programování v různých moderních programovacích jazycích. V praktické části budou připraveny ukázky různých technik v jednotlivých jazycích.
Teoretická část se bude zaobírat rešerší pojmů paralelismus, asynchronismus a souběh, vysvětlením rozdílů a implementací ve vybraných programovacích jazycích. Bude popsána základní koncepce přístupu k souběhu, způsob synchronizace a komunikace úloh. Dále budou popsány prostředky vybraných programovacích jazyků a jejich knihoven (zahrnující koncepty async/await, promise/future).
V praktické části budou implementovány vybrané ukázkové úlohy využívající paralelismus/souběh v popsaných jazycích. Hodnocené jazyky (min. 4) budou např. zahrnovat: OOP jazyky (Java, C#), jazyk využívající techniku CSP, funkcionální jazyk, dynamický (skriptovací) jazyk.

Rozsah pracovní zprávy: **50-60 stran**
Rozsah grafických prací: **–**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

*BUTCHER, Paul, 2014. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Pragmatic Bookshelf. ISBN 9781937785659.
*MCCOOL, Michael, James REINDERS a Arch ROBINSON. *Structured Parallel Programming: Patterns for Efficient Computation*. Waltham, USA: Elsevier, 2012. ISBN 9780123914439.

Vedoucí diplomové práce: **Ing. Roman Diviš**
Katedra softwarových technologií

Datum zadání diplomové práce: **6. listopadu 2020**
Termín odevzdání diplomové práce: **15. května 2021**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2020

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne

Bc. Yaroslav Zenchenko

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Romanu Divišovi, Ph.D. za odborné rady a konzultace. Dále chci poděkovat své rodině a přátelům za podporu v celém průběhu studia.

ANOTACE

Diplomná práce se zabývá problematikou paralelismu v moderních programovacích jazycích. První kapitola definuje jednotlivé odborné pojmy. Kapitoly dva až pět zkoumají techniky paralelismu, asynchronismu a souběhu v různých programovacích jazycích. Každý z jazyků uvedených v této práci implementuje jiné programovací paradigma. Patří mezi ně Java, JavaScript, Clojure a Go. Závěr práce provádí srovnávací analýzu uvažovaných technik, jejich vyhody a nevyhody, podobností a rozdílů.

KLÍČOVÁ SLOVA

paralelní programování, asynchronní programování, souběh, vlákna

TITLE

Parallelism, asynchronism and concurrency in modern programming languages

ANNOTATION

The diploma thesis deals with the issue of parallelism in modern programming languages. The first chapter defines the specific terminology. Chapters two through five examine the techniques of parallelism, asynchronism, and concurrency in different programming languages. Each of the languages mentioned in this work implements a different programming paradigm. They are Java, JavaScript, Clojure and Go. The conclusion of the thesis performs a comparative analysis of the considered techniques, their advantages and disadvantages, similarities and differences.

KEYWORDS

parallel programming, asynchronous programming, concurrency, threads

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK

API	Application Programming Iinterface
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DOM	Document Object Model
ES6	ECMAScript 6
HTTP	HyperText Transfer Protocol
IMDB	Internet Movie Database
I/O	Input/Output
IPC	Inter-Process Communication
JIT	Just-in-Time
JVM	Java Virtual Machine
MDN	Mozilla Developer Network
OOP	Objektově Orientované Programování
REPL	Read-Eval-Print Loop
STM	Software Transactional Memory systém

ÚVOD

S rozvojem technologií paralelismu se staly žádanými jazyky, které umožňují tyto technologie využívat naplno. Populární jazyky, které mají k dispozici slabé nebo žádné nástroje paralelismu, je začaly rozvíjet. V této práci budou brány v úvahu přístupy paralelismu, souběžnosti a asynchronismu čtyř jazyků představujících různá programovací paradigmata. To jsou *Java* jako OOP jazyk, *JavaScript* jako skriptovací jazyk, *Clojure* jako funkcionální jazyk a *Go* jako jazyk využívající techniku CSP.

Kapitola *Povaha paralelismu* vysvětluje, co znamenají termíny paralelismus, asynchronismus a souběžnost. Ilustrace v této kapitole demonstrují rozdíl mezi nimi a jejich odlišnosti od sekvenčního zpracování.

Kapitola *Prostředí paralelismu v OOP na příkladu Javy* pojednává o tom, jaké nástroje může moderní Java nabídnout pro řešení paralelismu, souběžnosti a asynchronního volání. Hlavními tématy této kapitoly jsou *Stream API* a *CompletableFuture*. *Stream API* je rozsáhlá knihovna navržená tak, aby usnadňovala práci s kolekcemi. Umožňuje také paralelizovat operace prováděné nad těmito kolekcemi. *CompletableFuture* je třída, která umožňuje zpracovávat asynchronní volání a kombinovat je s užitím paralelismu. Tato kapitola se navíc zabývá tématem *Reaktivního Programování* a metodami, kterými jej Java podporuje.

Kapitola *Prostředí paralelismu ve skriptovacích jazycích na příkladu JavaScriptu* na začátku stručně popisuje principy fungování tohoto jazyka, který se primárně využíval v rámci webových prohlížečů a nabízel programátorovi jenom jediné vlákno. Poté popisuje dlouhou cestu asynchronismu v JavaScriptu od techniky zpětného volání (callback) do *async/await*. Nakonec je popsána technologie *Web Workers*, která poskytuje JavaScriptu přístup k vláknům operačního systému, což mu umožňuje využívat paralelismus.

Kapitola *Prostředí paralelismu ve funkcionálních jazycích na příkladu Clojure* popisuje syntaxi tohoto jazyka a vysvětluje základní pojmy, které jsou tomuto jazyku vlastní. Clojure je jazyk, který byl od základu navržen tak, aby snadno zvládal paralelismus. Podporuje měnná a neměnná data. Pro paralelizaci neměnných dat poskytuje Clojure knihovnu *reducers*. Clojure má několik různých měnných datových typů, to jsou *atomy*, *agenti* a *refy*, každý z nich vhodný pro různé situace.

Kapitola *Prostředí paralelismu v jazycích využívajících techniku CSP na příkladu Go* vysvětluje, jak jazyk Go implementuje techniku CSP (Communicating Sequential Processes). Popisuje základní součásti souběžnosti v Go, hlavními jsou *gorutiny* a *kanály*. Kromě toho je

popsáno běhové prostředí Go, které má svůj vlastní plánovač úloh. Uvažuje se o nebezpečích, která mohou vyplývat z nesprávně naprogramovaného paralelismu, a vzory souběžnosti Go.

1 POVAHA PARALELISMU

Všechny počítače v dnešní době podporují paralelismus. Moderní počítače podporují hardwarový paralelismus prostřednictvím vícejádrových procesorů, paralelních procesorů a j. Vývoj technologie paralelismu vedl k potřebě používat explicitní paralelní programování pro co nejefektivnější využití hardwaru (McCool, 2012, s. 1).

V paralelním programování je jednou ze základních entit úloha. *Úloha (angl. – task)* je blok programového kódu, zodpovědný za zpracování určitých činností. Úlohu lze „zarámovat“ jako samostatný proces nebo vlákno procesu.

Proces je instancí programu za běhu, což je nezávislý objekt, kterému jsou přiděleny systémové prostředky (například paměť a čas CPU). Každý proces běží v samostatném adresním prostoru: jeden proces nemůže přistupovat k proměnným a datovým strukturám jiného procesu bez prostředků meziprocesové komunikace *(angl. – inter-process communication, IPC)*.

Vlákno (angl. – thread) je malá výkonná jednotka v procesu. Obvykle každé vlákno může pracovat (číst a zapisovat) se stejnou oblastí paměti, na rozdíl od procesů, které nemohou přistupovat do paměti jiného procesu. Každé vlákno má své vlastní registry a svůj vlastní zásobník. Každý proces má alespoň jedno vlákno.

Operační systémy spravují úkoly pomocí plánovače. *Plánovač úloh (angl. – scheduler)* je program, který spouští další programy. Nejdůležitějším cílem plánovače je co nejúplnější využití CPU. Na jednom procesoru (jádru) může v jednom časovém okamžiku běžet pouze jedno vlákno. Všechny procesy (vlákna), které nejsou aktuálně spuštěny, jsou zařazeny do prioritní fronty. Plánování spočívá v přiřazení priorit procesům (vláknům) v prioritní frontě. Toto je jen příklad jedné z plánovacích metod, ve skutečnosti jich existuje mnohem více.

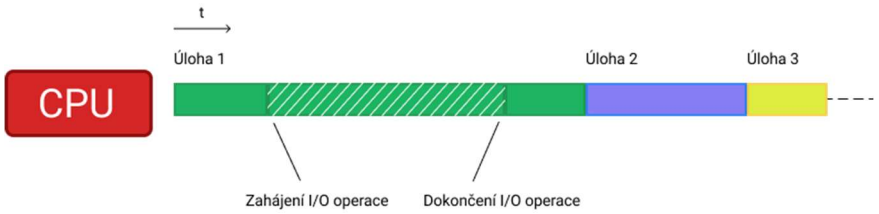
Existují *úlohy vázané na procesor (angl. – CPU bound tasks)* a *úlohy vázané na I/O (angl. – I/O bound tasks)*. Úlohy vázané na procesor jsou úlohy, jejichž rychlost provádění závisí přímo na frekvenci procesoru. Čím vyšší je takt procesoru, tím rychleji je úloha dokončena. Úlohy vázané na I/O jsou úlohy, které závisí na rychlosti operací I/O, jako je čtení ze souboru, komunikace přes počítačovou síť a jiné. Taktovací frekvence procesorů se s vývojem hardwaru zvyšuje rychleji než rychlost procesů souvisejících s I/O. Takové úlohy proto jednoduše nedrží rychlost procesorů a jsou prováděny pomaleji, než by mohly být.

1.1 Sekvenční zpracování

Sekvenční zpracování je, když jsou všechny příkazy provedeny v pořadí, v jakém jsou zapsány v kódu. Dva příkazy nelze provést současně nebo změnit pořadí jejich provedení. Každé spuštění sekvenčního programu vždy povede ke stejnému výsledku. Sekvenční programy jsou deterministické (za předpokladu, že neobsahují například I/O).

Ve skutečnosti je sekvenční provádění v sekvenčním kódu iluzí. Podle McCoolu (2012, s. 2) byl ještě před příchodem vícejádrových procesorů hardware přirozeně paralelní. Kvůli pohodlí programování však byly navrženy s iluzí, že příkazy jsou prováděny postupně, i když ve skutečnosti nejsou. Tento proces se v anglické literatuře nazývá serializace *(angl. – serialization)*. Serializace je vážně vetkána do veškeré programovací praxe, do jazyků, do datových struktur. Ačkoli byla serializace nástrojem pro pohodlnou paralelizaci, není schopna efektivně využívat moderní paralelní hardware sama o sobě.

Sekvenční program běží v jednom vlákně na jednom procesoru. Každá úloha běží od začátku do konce bez přerušení. V případě blokujících systémových volání nebo I/O operací je zablokován celý program. Obrázek 1 schematicky zobrazuje zpracování aplikace s jedním vláknem. Úloha 1 je zde vázaná na I/O, úlohy 2 a 3 jsou vázané na procesor. Čas blokování je označen šrafováním. Při čekání na operaci I/O mohl procesor dokončit celý úkol 2. V případě sekvenčního zpracování však musí počkat na dokončení úlohy 1. Krátká úloha 3 musí zároveň počkat na konec velmi dlouhé úlohy 2. Pokud jsou na sobě nezávislé, lze úkol 3 provést souběžně s úkolem 2 pomocí techniky time-slicing (je popsána v podkapitole 1.3).



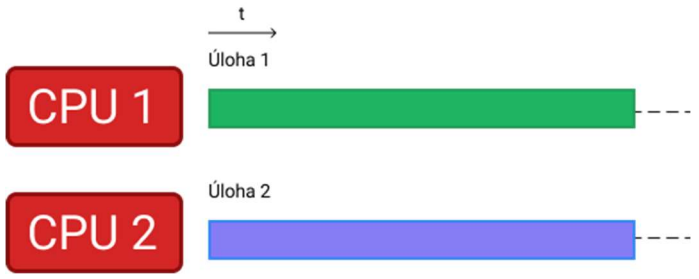
Obrázek 1: Sekvenční zpracování

1.2 Paralelní zpracování

Na rozdíl od sekvenčního zpracování je paralelní zpracování nedeterministické. Několik úkolů je prováděno doslova současně v různých vláknech, která běží na různých jádrech/procesorech. Není možné předpovědět, který z úkolů skončí dříve. Při každém spuštění takového programu bude výsledek jiný.

Paralelismus může urychlit program. Jedním ze způsobů je spouštění nezávislých úkolů na různých jádrech/procesorech. Běžnějším způsobem využití paralelismu je však rozdělení úkolu na více dílčích úkolů a jejich distribuce mezi jádry/procesory. Například distribuce poměrně složitého výpočtu na čtyři jádra by teoreticky mohla úkol zrychlit čtyřikrát.

Obrázek 2 ukazuje, jak běží dvě úlohy na dvou vláknech v systému se dvěma procesory.



Obrázek 2: Paralelní zpracování

Použití paralelismu však není vždy odůvodněné a nemusí vést k významnému zkrácení doby provádění. Faktem je, že komunikace mezi procesory, vytváření vláken a další procesy, které zajišťují paralelismus, také spotřebovávají procesorové a časové prostředky. Čas získaný paralelismem by tedy měl být výrazně kratší než čas strávený udržováním paralelismu. Proto nemá smysl paralelizovat příliš malé objemy úkolů, a naopak u úkolů s velkým množstvím dat paralelismus výrazně zkrátí dobu provádění.

Paralelismus je způsob, jak urychlit úlohy vázané na procesor. Zvýšení počtu procesorů přirozeně zrychlí úkoly, které závisí na frekvenci procesoru (samozřejmě s explicitním paralelním programováním). Jiné způsoby, jak takové úkoly urychlit, jsou například kešování výsledku nákladného úkolu nebo vylepšení algoritmu.

1.3 Souběh neboli souběžnost

„Souběžnost je podobná, ale není totožná s paralelismem“[1] (McCool, 2012, s. 67). Za prvé, na rozdíl od paralelismu může být implementována na jednom procesoru.

Souběh je implementován pomocí techniky zvané *time-slicing* nebo *preemce*. Obrázek 3 jasně ukazuje tuto techniku. Zde jsou v systému s jedním procesorem provedeny dvě úlohy ve

[1] Concurrency is similar but not identical to parallelism.

dvou vláknech. Přestože tyto dva úkoly neběží současně, kvůli častému přepínání mezi nimi to pro uživatele vypadá, že jsou paralelní.

Obrázek 3: Souběh

Plánování vláken se obvykle předává na externí (vzhledem k vláknům) plánovač, jako je plánovač operačního systému nebo vestavěný plánovač jazyka (jako v Go). Protože plánovač může vlákno kdykoli přepnout, je nutné uložit kontext vlákna, tedy všechna data, se kterými vlákno pracovalo. Přepínání kontextu je také časově náročné.

Při souběhu i paralelismu je potřeba chránit data, která jsou k dispozici pro více vláken, aby jedno vlákno nemohlo změnit data, na kterých v té době pracuje jiné vlákno. To lze provést například prováděním atomických operací. Jsou to operace, které nelze přerušit. Dalším způsobem je pracovat s neměnnými *(angl. – immutable)* datovými typy.

Souběh je jedním ze způsobů, jak tyto úlohy vázané na I/O urychlit, pokud existuje více úloh v jednom okamžiku. Badewa (2019) ve svém článku porovnal sekvenční a souběžné provádění mnoha I/O operací. Sekvenční provádění bylo v jeho příkladu desetkrát pomalejší. Většina řešení tohoto problému spočívá v oblasti hardwarové architektury: zmenšení vzdálenosti mezi procesorem a datovými úložišti a jiné. Dalším řešením, které není hardwarové, je použití asynchronismu.

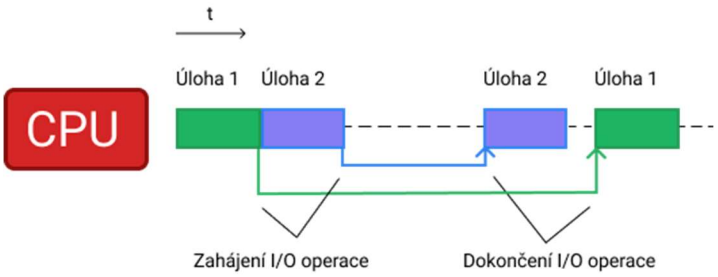
1.4 Synchronismus/asynchronismus

„V modelu synchronního programování se věci dějí jedna po druhé. Když zavoláte funkci, která provádí dlouhotrvající akci, vrátí se pouze po dokončení akce a může vrátit výsledek. Tím se program zastaví na dobu, po kterou akce trvá.

Asynchronní model umožňuje, aby se dělo více věcí najednou. Když spustíte akci, váš program pokračuje v běhu. Po dokončení akce je program informován a získá přístup k výsledku (například k načtení dat z disku).“[2] (Haverbeke, 2018, s. 180)

Rozdíl mezi asynchronismem a souběžností je následující. Když jsou úlohy vázané na I/O prováděny pomocí souběžnosti, část času vláken zabere čekání na dokončení operace. V případě asynchronie vlákna nečekají a vykonávají další užitečnou práci. Jinými slovy, systémová volání a I/O operace neblokují vlákna. Navíc asynchronismus efektivně řeší úlohy vázané na I/O na jednom CPU.

Příklad: je potřeba provést dvě I/O operace a poté spojit jejich výsledky (může to být čtení z disku nebo webový požadavek). Pokud jsou tyto operace prováděny sekvenčně, druhá operace se nespustí, dokud se první operace nedokončí. Pokud je každá operace spuštěna v samostatném vlákně, provedení bude určitě rychlejší. Každé z těchto vláken však bude ztrácet čas CPU čekáním. V asynchronním modelu spustí jedno vlákno první I/O operaci a aniž by čekalo na jeho dokončení, okamžitě spustí druhé. Zatímco operace probíhá, vlákno pokračuje v jiné práci. Po dokončení I/O operací je vlákno upozorněno a může použít výsledek těchto operací.



Obrázek 4: Asynchronismus

Obrázek 4 ukazuje princip asynchronismu. Zde je jedno vlákno běžící na jednom CPU. Úlohy 1 a 2 jsou vázány na I/O. Čáry mimo vlákno představují čas potřebný k dokončení operací I/O. Na obrázku 1, který ukazuje sekvenční zpracování, bylo vlákno nuceno čekat na dokončení I/O. V asynchronním prostředí může vlákno provádět jinou práci, nezávisle na výsledcích I/O operací a bez čekání na konec těchto I/O operací.

2 In a synchronous programming model, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. This stops your program for the time the action takes.
An asynchronous model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

1.5 Souhrn

V praxi se obvykle používá kombinace všeho výše uvedeného. Moderní operační systémy používají souběh, asynchronní volání i paralelismus pokud to hardware umožňuje.

Pro dosažení výše uvedeného na úrovni různých programovacích jazyků existují různé přístupy a abstrakce. Pro asynchronní volání se používají *zpětná volání*, *promises*, *futures*. Pro paralelizace úloh existuje algoritmus *fork/join*. Algoritmus *„Work stealing“* se používá k rovnoměrné distribuci úloh mezi všechny procesory v systému. Tyto a další techniky budou představeny a popsány v následujících kapitolách.

2 PROSTŘEDÍ PARALELISMU V OOP NA PŘÍKLADU JAVY

Java je *objektově orientovaný* programovací jazyk, který používá *imperativní styl* programování.

Objektově orientované programování (resp. OOP) je přístup, při kterém je celý program vnímán jako kolekce objektů, které na sebe vzájemně působí.

Imperativní styl programování je paradigma, které se vyznačuje tímto:

- instrukce (příkazy) jsou psány ve zdrojovém kódu programu;
- instrukce musí být provedeny postupně;
- data získaná během provádění předchozích instrukcí lze číst z paměti následnými instrukcemi;
- data získaná provedením instrukce mohou být zapsána do paměti.

Souběžnost a paralelismus v Javě jsou implementovány pomocí *vláken*. Na vícejádrových systémech jsou vlákna schopna vykonávat současně jiný kód. Pokud na stejném jádru běží více vláken, operační systém mezi nimi čas od času přepíná (tomu se říká přepínání kontextu). Vlákna mají sdílenou paměť, proto mají přístup ke stejným objektům a proměnným.

Objekty a proměnné v Javě jsou *mutable*, mohou být změněny. Když několik vláken pracuje se stejným objektem a mění jej, může to vést k neočekávaným výsledkům a chybám ve výpočtech, protože pokud dojde ke změně kontextu, záleží pouze na plánovači operačního systému.

K vyřešení tohoto problému se používají *zámky*. Objekt, se kterým jedno z vláken aktuálně pracuje, je uzamčen. Další vlákno, které chce s tímto objektem pracovat, čeká na uvolnění objektu.

V Javě roli zámku hraje klíčové slovo *synchronized*. Lze jej použít k uzavření objektu i metody.

Od verze Java 5 má jazyk framework Executor. Je určen pro řízení vláken. Uživatel odesílá úkoly Exekutorovi, který je rozděluje mezi vlákna z fondu vláken.

2.1 Moderní nástroje paralelního prostředí v Javě

Od verze Java 1.0 má jazyk třídu *Thread*, která implementuje rozhraní *Runnable*. S její pomocí bylo možné vytvářet vlákna, a tak psát vícevláknové aplikace. Tehdy bylo potřeba

pracovat přímo se vlákny, což je obtížné a náchylné k chybám. Od té doby se toho hodně změnilo.

Java 5 představila rozhraní *Concurrency API* jako balíček java.util.concurrent. Tento balíček poskytoval nová rozhraní, jako je *ExecutorService*, *Callable<T>* a *Future<T>*. *ExecutorService* zavádí koncept fondu vláken. Nyní vývojář nemusí pracovat přímo s vlastními vlákny, protože *ExecutorService* to převezme.

Od verze Java 7 byl do jazyka přidán java.util.concurrent.*RecursiveTask*, který implementoval Fork/Join verze algoritmu „rozděl a panuj“.

Mnoho aktualizací bylo zavedeno v prostředí Java 8. Mezi nimi jsou *Stream API* a třída *CompletableFuture*. *Stream API*, který je nástrojem pro práci s kolekcemi, umožňuje paralelizovat úkoly prováděné na kolekcích. *CompletableFuture* je implementace budoucího rozhraní, které mimo jiné zavádí asynchronismus.

Java 9 představila třídu java.util.concurrent.*Flow*. Tato třída deklaruje vnořená rozhraní určená k implementaci paradigmatu „reaktivního programování“ (Urma, 2019, s. 360–361).

Tato kapitola hovoří o *Stream API* a o tom, jak implementuje paralelizaci. O tom, jak *CompletableFuture* přidává asynchronii k souběžnosti. A o tom, co je reaktivní programování a jak je implementováno v Javě.

2.2 Stream API

Takzvané *Streamy* byly do jazyka Java přidány ve verzi 8. Ačkoli tato práce pojednává o paralelních prostředích, paralelizmus není jedinou funkčností streamů. Balíček java.util.stream je popsán jako: *„Třídy na podporu operací ve funkcionálním stylu u proudů prvků, jako jsou transformace map-reduce u kolekcí.”* [3] (Oracle, ©1993, 2021) Hlavním cílem bylo zjednodušit práci s kolekcemi přidáním do jazyka API, které umožní programovat ve funkcionálním stylu. Výše zmíněné přináší změnu v oblasti manipulace dat, a to možnost manipulovat s daty deklarativním způsobem, kdy již není třeba v každém kroku implementovat, jak se mají data transformovat pro získání požadovaného výsledku.Nyní stačí pouze popsat požadovaný výsledek jako dotaz (Urma, 2019, s. 82).

```
1  List<Item> cheapItems = new ArrayList<>();
2  // filtrování podle ceny (nižší než 1000)
```

[3] Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

```
3  for(Item item : items){
4      if(item.getPrice() < 1000){
5          cheapItems.add(item);
6      }
7  }
8  // třídění
9  Collections.sort(cheapItems, new Comparator<Item>(){
10     @Override
11     public int compare(Item item1, Item item2){
12         return Integer.compare(item1.getPrice(), item2.getPrice());
13     }
14 });
15 // získání původně požadovaného seznamu
16 List<String> cheapItemsNames = new ArrayList<>();
17 for (Item item : cheapItems) {
18     cheapItemsNames.add(item.getName());
19 }
```

Zdrojový kód 1: Příklad imperativního způsobu práce s kolekcí

```
1 import static java.util.Comparator.comparing;
2 import static java.util.stream.Collectors.toList;
3 List<String> cheapItemsNamesFromStream =
4      items.stream()                                  // získání streamu
5           .filter(item -> item.getPrice() < 1000)       // filtrování
6           .sorted(comparing(Item::getPrice))               // třídění
7           .map(Item::getName)         // mapování na požadovaný format
8           .collect(toList()); // získání původně požadovaného seznamu
```

Zdrojový kód 2: Příklad deklarativního způsobu práce s kolekcí

Příkladem může být seznam instancí třídy *Item*. Každá z nich má jméno (*name*) a cenu (*price*). Úkolem je získat seznam názvů objektů, jejichž cena je nižší než 1000 a které jsou seřazené podle ceny. Imperativní řešení tohoto problému by mohlo být provedeno způsobem, jaký je vyobrazen ve zdrojovém kódu 1. Deklarativní způsob manipulace dat, jež je v Javě možný díky rozhraní *Stream API*, však zpracování zmíněného úkolu výrazně usnadňuje (zdrojový kód 2).

Výsledek těchto dvou rozdílně zpracovaných částí kódu, uvedených ve zdrojových kódech 1 a 2, je naprosto stejný. Použití rozhraní *Stream API* tedy umožňuje jak zjednodušení psaní kódu a jeho lepší čitelnost, tak eliminaci potřeby vytváření přechodných proměnných.

Vzhledem k tomu, že práce pojednává především o paralelismu, zmiňuje pouze základy *Stream API* a jeho části související s paralelismem.

2.2.1 Stručně o streamech jako takových

„Streamy jsou klíčovou abstrakcí v Javě 8 pro zpracování kolekcí hodnot a určení toho, co chcete udělat, přičemž plánování operací je ponecháno implementaci.“ [4] (Horstmann, 2014, s. 21)

Stream je rozhraní, které má následující vlastnosti:

- není datovou strukturou (neukládá data);
- „lazy” chování (operace se odkládají na poslední místo);
- může být nekonečný;
- nemutuje zdroj;
- jednorázový;
- uspořádaný/neuspořádaný (ordered/unordered);
- paralelní/sekvenční.

Zdrojový kód 3 vyobrazuje stejný kód uvedený výše ve zdrojovém kódu 2, s rozdílem uvedení očíslovaných řádků pro pohodlnější zkoumání. Níže uvedená proměnná *items* jsou *Collection <Item>*. V jazyce Java verze *8* obsahuje rozhraní Collection metodu *stream()*, která vrací sekvenční stream, jež používá kolekce, která jej volala jako zdroj dat (Oracle, ©1993, 2021).

Instance *Streamu* má i jiné metody, které také vracejí instance *Streamu*. To umožní vytvořit stream zřetězením volání těchto metod, jinými slovy tvorbou pipeline. Metody, které tvoří tento pipeline, s výjimkou úplně první, se nazývají operace. Na ukázce ze zdrojového kódu 3 je na řádku 2 vytvořen stream na základě existující kolekce. Dále je tento stream upravován voláním operací na řádcích 3 až 5. Řádek 3 popisuje operaci filtrování, řádek 4 pak operaci třídění, a nakonec se řádek 5 mapuje na typ dat, jež jsou třeba pro požadovaný výsledek. Argumenty ve všech těchto operacích jsou lambda výrazy. Rámec této diplomové práce však neumožňuje detailněji popsat, jak fungují lambda výrazy. Schematické znázornění pipelinu je zobrazeno na obrázku 5.

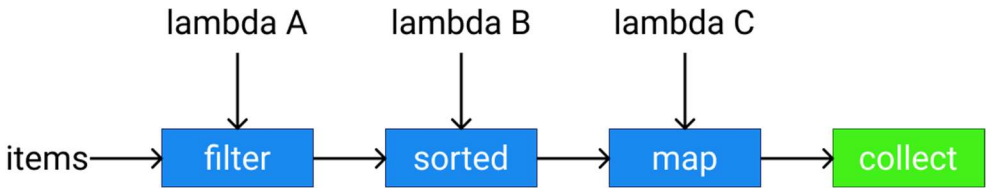
```
1   List<String> cheapItemsNamesFromStream =
2       items.stream()
3            .filter(item -> item.getPrice() < 1000)
4            .sorted(comparing(Item::getPrice))
```

[4] Streams are the key abstraction in Java 8 for processing collections of values and specifying what you want to have done, leaving the scheduling of operations to the implementation.

```
5            .map(Item::getName)
6            .collect(toList());
```

Zdrojový kód 3: Příklad použití *Stream API*

Je však třeba věnovat pozornost skutečnosti, že ve výše zmíněných řádcích 2–5 zatím neprobíhá žádné zpracování dat. Jedná se prozatím o popis toho, jak to stream zvládne. Samotné zpracování bude zahájeno v momentě, kdy dojde k operaci volané na řádku 6. Zde je již známo, jakým způsobem by měla být data zpracována a samotná operace pak určuje, v jakém formátu by měla být uložena (v tomto případě jako *List*). Obrázek 6 graficky popisuje pipeline ze zdrojového kódu 3.



Obrázek 5: Pipeline streamu

Z výše uvedeného vyplývá, že ne všechny operace streamu mají stejnou povahu. Dle jejich chování je lze rozdělit do dvou skupin: neterminální (*angl. – intermediate*) a terminální (*angl. – terminal*) (Urma, 2019, s. 94) Přičemž metody, které vracejí jiný stream jako návratovou hodnotu, se nazývají neterminální. Specifikem neterminálních metod je, že při volání nejsou prováděny, jelikož se vyznačují odloženým vyhodnocováním (jsou tzv. „lazy“). To znamená, že generují nebo zpracovávají prvky pouze v případě potřeby. Tyto „lazy“ metody začnou být aktivní až v momentě, kdy řetězec volání dosáhne poslední terminální metody a jenom v případě potřeby. Tak knihovna *Streams* provede optimalizace, díky nimž budou operace prováděny co nejefektivněji. Terminální operace vytvářejí výsledek z pipelinu streamu. Výsledkem je jakákoli hodnota, která není streamem, například *List*, *Integer* nebo dokonce *void*.

Tato práce neposkytne podrobný popis všech metod, pouze stručný popis základních operací v tabulkách 1 a 2.

Tabulka 1: Neterminální operace *Stream API*

Signatura metody	**Popis**
Stream<T> filter(Predicate *predicate*)	Vrací stream skládající se z prvků tohoto streamu, které odpovídají danému *predikátu*.
Stream<T> map(Function *mapper*)	Vrací stream skládající se z výsledků použití *mapperu* na prvky tohoto streamu.
Stream<T> flatMap(Function *mapper*)	Podobné map(), funguje však nejlépe pro seznam kolekcí.
Stream<T> distinct()	Vrací stream skládající se z unikátních prvků.
Stream<T> sorted(); Stream<T> sorted(Comparator *comparator*)	Vrací stream skládající se z prvků tohoto streamu, seřazený podle přirozeného pořadí nebo podle zadaného *komparátoru*.
Stream<T> peek(Consumer *action*)	Vrací stream skládající se z prvků tohoto datového proudu, navíc provede poskytovanou *akci* na každém prvku.
Stream<T> limit(long *maxSize*)	Vrací stream s danou omezenou *velikostí*.
Stream<T> skip(long n)	Tato metoda přeskočí dané *n* prvků a vrátí stream.

Tabulka 2: Terminální operace *Stream API*

Signatura metody	**Popis**
void forEach(Consumer *action*)	Provede *akci* pro každý prvek tohoto streamu.
T reduce(T *identity*, BinaryOperator<T> *accumulator*)	Provede redukci prvků tohoto streamu pomocí poskytnuté hodnoty *identity* a asociativní *akumulační* funkce a vrátí redukovanou hodnotu.
T collect(Collector *collector*)	Metoda collect() akumuluje prvky ve streamu do kontejneru, jako je kolekce.
Optional<T> min(Comparator *comparator*)	Vrací minimální prvek tohoto streamu podle poskytnutého *komparátoru*.
Optional<T> max(Comparator *comparator*)	Vrací maximální prvek tohoto streamu podle poskytnutého *komparátoru*.
long count()	Vrací počet prvků v tomto streamu.
boolean anyMatch(Predicate *predicate*)	Vrátí true či false, dle toho, zda se některé prvky tohoto streamu shodují s poskytnutým *predikátem*.
boolean allMatch(Predicate *predicate*)	Vrátí true či false, dle toho, zda se všechny prvky tohoto streamu shodují s poskytnutým *predikátem*.
boolean noneMatch(Predicate *predicate*)	Vrátí true či false, dle toho, zda se žádné prvky tohoto streamu neshodují s poskytnutým *predikátem*.

2.2.2 Paralelismus v Stream API

Streamy se dělí na sekvenční a paralelní, kdy standardně metoda *stream()* vytvoří sekvenční stream. Převod streamu z postupného na paralelní lze provést dvojím způsobem. Lze buď vytvořit stream voláním metody *parallelStream()* namísto *stream()*, nebo přidat volání metody *parallel()* do pipelinu. Druhý zmíněný způsob je proveditelný díky „lazy“ povaze neterminálních operací. Metodu *parallel()* lze vložit kamkoli do pipelinu a celý stream se bude chovat jako paralelní. Ve skutečnosti tato metoda samotný stream nijak neovlivňuje, pouze nastavuje interní příznak, který signalizuje, že všechny operace by měly být spuštěny paralelně. Dokonce je možné do pipelinu vložit zároveň jak metodu *parallel()*, tak metodu *sequential()*. V tomto případě je vítěznou metodou ta, která je umístěna později (Urma, 2019, s. 175).

Urma (2019, s. 174) poskytl názorný příklad, jak paralelní stream funguje. Tento kód je znázorněn ve zdrojovém kódu 4. Metoda *iterate()* v řádku 2 vrací stream hodnot typu Long, které se donekonečna zvyšují (což je opět možné díky „lazy“ chování). Operace *limit()* na řádku 3 potom omezuje daný stream na n prvků. V řádku 4 je stream označen jako paralelní. Poslední operací je terminální operace *reduce()* volaná v řádku 5, která vrací sumu prvků tohoto streamu jako typ *Long*.

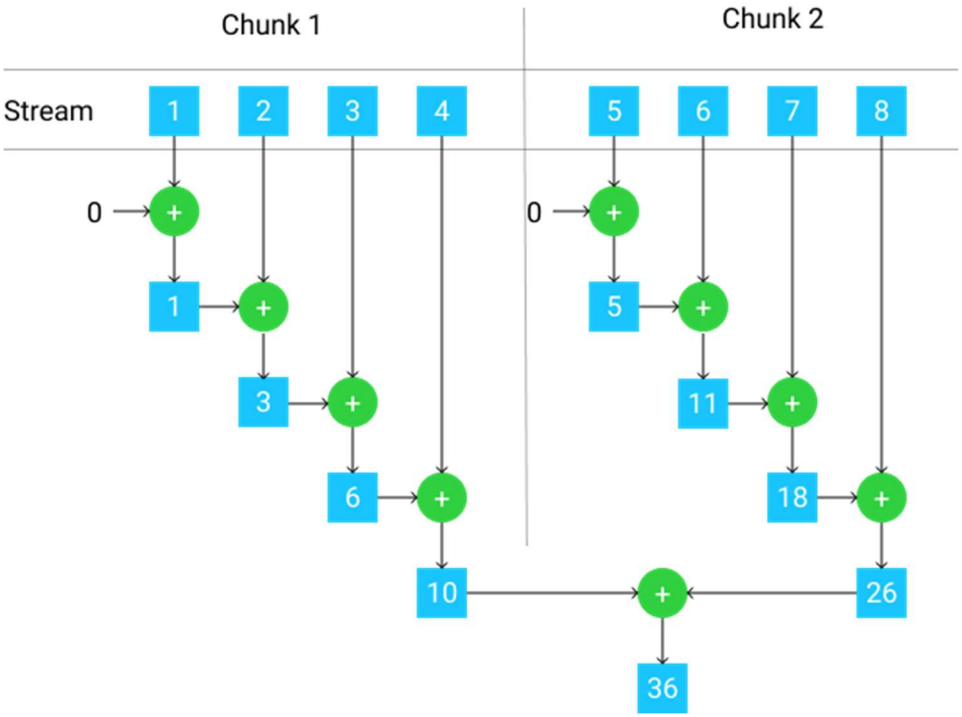
```
1    public long parallelSum(long n) {
2        return Stream.iterate(1L, i -> i + 1)
3                     .limit(n)
4                     .parallel()
5                     .reduce(0L, Long::sum);
6    }
```

Zdrojový kód 4: Paralelní stream

Pro paralelní zpracování dat je stream rozdělen do několika kusů *(angl. – chunks)*. Každý z nich běží na samostatném vlákně a na data v různých vláknech aplikuje operaci *reduce()* samostatně. Poté se operace *reduce()* použije na výsledky všech těchto operací *reduce()* a vrátí konečný výsledek (obrázek 7).

Různé kusy výpočtu se tedy provádějí paralelně v různých vláknech, což by mohlo naznačovat, že je tento paralelní kód rychlejší než analogický sekvenční kód. Kupodivu však není. Urma (2019, s. 178) porovnal čas provedení tohoto kódu a jeho sekvenčního analogu na čtyřjádrovém počítači. Ukázalo se, že paralelní verze kódu byla asi pětkrát pomalejší. Vše spočívá v metodě *iterate()*, která generuje nikoli čísla, ale zabalené objekty, které je třeba před

přidáním rozbalit. Kromě toho je stream, vytvořený metodou *iterate()*, obtížné rozdělit na kusy, které by mohly běžet paralelně.



Obrázek 6: Schéma pracování paralelního streamu

Programátor pracující s paralelními streamy by proto měl mít na paměti, že některé streamy a jejich operace jsou pro paralelizaci méně vhodné než jiné. *„Zejména je obtížné operaci iterate() rozdělit na kusy, které lze provádět nezávisle na sobě, protože vstup jedné iterace vždy závisí na výsledku předchozí iterace.“*[5] (Urma, 2019, s. 178) To samozřejmě neznamená, že je skutečně nemožné provést tuto úlohu v paralelitě pomocí streamů. Je nutné pouze použít jiné metody, které *Stream API* nabízí.

Kromě samotného rozhraní *Stream*poskytuje *Stream API* jeho primitivní specializace, tedy rozhraní *DoubleStream, IntStream* a *LongStream*. Zmíněný problém lze vyřešit rozhraním LongStream, jež obsahuje metodu *rangeClosed()*. Na rozdíl od *iterate()* pracuje přímo s primitivy typu long, které není třeba rozbalovat; mimo to také generuje rozsahy čísel, které lze snadno rozdělit na kusy. Přepsaná verze kódu (zdrojový kód 5) bude fungovat v paralelitě efektivně.

[5] Specifically, the iterate operation is hard to split into chunks that can be executed independently, because the input of one function application always depends on the result of the previous application.

```
1   public static long parallelSum(long n) {
2        return LongStream.rangeClosed(1, n)
3                         .parallel()
4                         .reduce(0L, Long::sum);
5    }
```

Zdrojový kód 5: Opravený paralelní stream

Z výše uvedeného příkladu je zřetelné, že při práci s paralelismem ve streamech můžete program místo zrychlení zpomalit. A nejen to. Například, Dokumentace Oracle (©1993, 2021) zmiňuje o operaci *forEach()* následující: *„Chování této operace je výslovně nedeterministické. U pipelinů paralelních streamů tato operace nezaručuje respektování pořadí prvků streamu, protože by to obětovalo výhodu paralelismu. U libovolného daného prvku lze akci provést kdykoli a v jakémkoli vlákně, které si knihovna vybere. Pokud akce přistupuje ke sdílenému stavu, je odpovědná za zajištění požadované synchronizace.“*[6] Programátor, který chce pracovat s paralelními streamy (ve skutečnosti s jakýmikoli streamy), by měl vždy nahlédnout do dokumentace, aby se ujistil, že zvolená operace je vhodná pro paralelizaci.

Kromě výše uvedeného je při paralelizaci úkolu třeba vzít v úvahu mnoho dalších úskalí. Příkladem mohou být operace jako *findFirst()*, které fungují lépe sekvenčně, protože závisí na pořadí dat v streamu. Zároveň operace jako *findAny()* pracuje v paralelním streamu pohodlněji. Mimo to, některé kolekce se paralelizují lépe než jiné. Dále je možné zmínit, že náklady na paralelizaci by měly být nižší než výhody.

2.2.3 Jak Stream API pracuje s vlákny. Framework fork/join

Až dosud nebylo zmíněno, jak *Stream API* implementuje paralelismus a jak paralelizuje úkoly mezi vlákny. K paralelnímu provádění operací používají paralelní streamy framework *fork/join*.

„Framework fork/join byl navržen tak, aby rekurzivně rozdělil paralelizovatelný úkol na dílčí úkoly a poté spojil výsledky každého dílčího úkolu a vytvořil celkový výsledek. Jedná se

[6] The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

o implementaci rozhraní ExecutorService, které distribuuje tyto dílčí úkoly do pracovních vláken ve fondu vláken, který se nazývá ForkJoinPool.“[7] (Urma, 2019, s. 184)

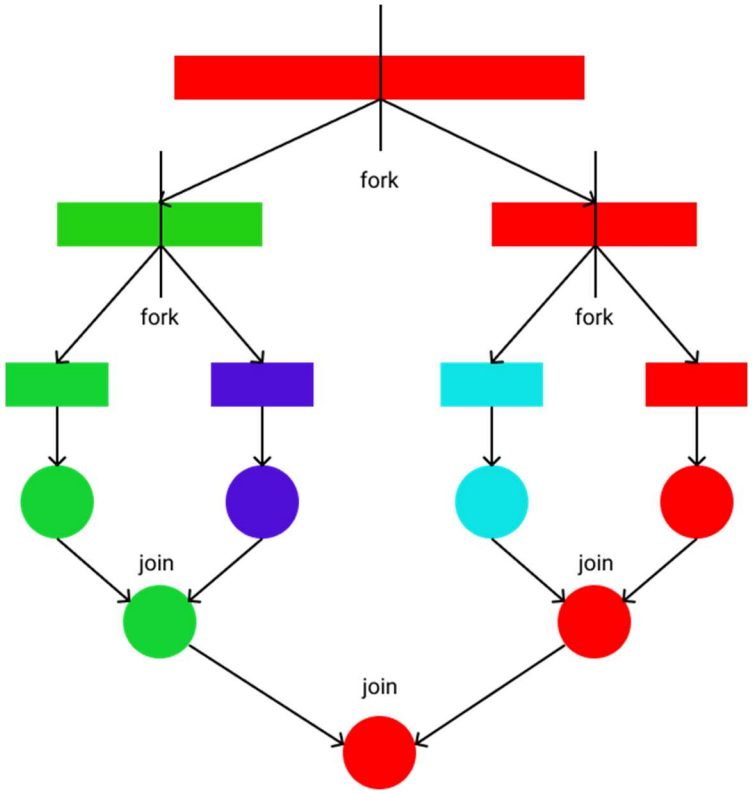
Třída *ForkJoinPool* má k dispozici metodu *invoke()*, která přijímá jako argument paralelizovatelný úkol. Pro vytvoření paralelizovatelného úkolu je třeba vytvořit podtřídu třídy *RecursiveTask <R>*. Tato podtřída potřebuje implementovat abstraktní metodu *compute()*. Tato metoda testuje, jak malý úkol je. Pokud je jeho velikost menší než zadaná prahová hodnota, úkol se provede sekvenčně. V druhém případě je úkol rozdělen na dvě části. Pro jednu z částí se metoda *compute()* volá rekurzivně ve stejném vlákně, pro druhou v jiném. Je však možné použít stejné vlákno ještě jednou, což snižuje náklady na vytvoření vláken. Poté metoda čeká na výsledek z obou částí a kombinuje jejich výsledky.

Princip činnosti je stejný jako algoritmy metody rozděl a panuj. Ve skutečnosti se jedná o stejnou metodu, pouze ve světě paralelismu. Obrázek 8 obsahuje schematické znázornění fungování *RecursiveTask*. Poté je volána metoda *invoke()* z *ForkJoinPool*, která jako argument přebírá instance nově vytvořené podtřídy *RecursiveTask*, čímž spustí paralelizovanou úlohu.

Jeden úkol provedený například na čtyřjádrovém procesoru lze snadno rozdělit na 1000 dílčích úkolů. V tomto případě framework fork/join implementuje tzv. algoritmus „Work stealing“. Každý pracovník má frontu dílčích úkolů. Pokaždé, když je úkol jednoho pracovníka rozdělen dvěma, volný pracovník si „ukradne“ jeden z těchto dílčích úkolů. Pokud během provozu jeden z pracovníků dokončí celou frontu dílčích úkolů, neukončí svoji činnost, ale ukradne úkol z konce fronty jiného pracovníka stejným způsobem. Tento proces se opakuje, dokud nebudou zpracovány všechny dílčí úkoly.

Při implementaci úkolu v rámci fork/join je třeba explicitně určit, na jakou velikost má být úkol rozdělen. V rozhraní *Stream API* však není třeba nic podobného specifikovat. Pro tyto účely používá *Stream API* tzv. *Spliterator* (Urma, 2019, s. 190).

[7] The fork/join framework was designed to recursively split a parallelizable task into smaller tasks and then combine the results of each subtask to produce the overal result. It’s an implementation of the ExecutorService interface, which distributes those subtasks to worker threads in a thread pool, called ForkJoinPool.



Obrázek 7: Princip činnosti *RecursiveTask*

2.2.4 Spliterator

„Stejně jako Iteratory se i Spliteratory používají k procházení prvků zdroje, jsou ale také navrženy k provádění této činnosti paralelně.“[8] (Urma, 2019, s. 190)

Pro individuální nebo sekvenční procházení prvků obsahuje *Spliterator* metody *tryAdvance()* a *forEachRemaining()* (Oracle, ©1993, 2021).

Z hlediska paralelismu je zajímavá metoda *trySplit()*, jež rozdělí *Spliterator* na dva a vrací jeden z nich. Odděluje některé prvky *Spliteratoru* do jiného *Spliteratoru*, který se vrací jako návratová hodnota metody, a to jim umožňuje paralelní běh. Poté oba tyto Spliteratory znovu zavolají metodu *trySplit()*, dokud všechna volání této metody nevrátí hodnotu null, což znamená, že další dělení již není možné.

Spliterator používá metodu *estimateSize()* k rozhodnutí, zda data dále rozdělit či spustit sekvenční řešení. V případě, že je původně známá velikost kolekce, která volala Spliterator, metoda jednoduše vrátí velikost kolekce. V opačném případě metoda vrátí přibližný počet

[8] Like Iterators, Spliterators are used to traverse the elements of a source, but they’re also designed to do this in parallel.

prvků. I přibližná hodnota umožňuje rozhodnout, jak efektivněji rozdělit kolekci a zda je to nutné.

Kromě toho má *Spliterator* metodu *characteristics()*, jež vrátí sadu charakteristik popisujících strukturu Spliteratoru jako int. Spliterator má následující vlastnosti: ORDERED, DISTINCT, SORTED, SIZED, NONNULL, IMMUTABLE, CONCURRENT a SUBSIZED. Například rozdělovač na základě Set má charakteristiku DISTINCT, rozdělovač na bázi *List* má charakteristiku SIZED a tak podobně. Znalosti těchto vlastností datové struktury umožňují optimalizovat práci *Spliteratoru*.

2.2.5 Výhody a nevýhody používání Stream API

Stream API, které přišlo s prostředím Java 8, výrazně usnadnilo práci programátorům pracujícím s paralelními výpočty. Navzdory zjevným výhodám skrývá *Stream API* i řadu nástrah.

Výhody používání *Stream API*:

1. méně kódu, lepší čitelnost;
2. zjednodušuje práci s kolekcí;
3. umožňuje psát kód ve funkcionálním stylu;
4. není nutné pracovat s vlákny přímo, Stream API se o to postará;

Nevýhody používání *Stream API*:

1. stále je vhodné rozumět tomu, jak vlákna fungují;
2. existují metody a zdroje, které nejsou vhodné pro paralelizaci;
3. nesprávné použití může snížit výkon;
4. nesprávné použití může ztížit ladění.

2.3 CompletableFuture

Třída CompletableFuture, zavedená v prostředí Java 8, implementuje rozhraní *Future*, které existuje již od zavedení prostředí Java 5. Hlavní myšlenkou je, že metoda, která vyžaduje dlouhý úkol, jej odešle do exekutoru, a v tuto chvíli okamžitě vrátí objekt typu Future, aniž by čekala na výsledek tohoto dlouhého úkolu. Poté je pomocí metody *get()* možné získat řešení z dané instance Future, jakmile je připraveno (zdrojový kód 5). Tato technologie může být užitečná například v situaci, kdy je potřeba souběžně přijímat data z různých zdrojů, ať už jde o webovou službu, databázi nebo výsledek dlouhé matematické operace, a kombinovat je (zdrojový kód 6).

Příklad složitější posloupnosti volání, které lze zrychlit souběžným spuštěním, může být následující: výsledek metody *a()* je vyžadován metodami *b()* a *c()*. Metoda *d()* pak pracuje s výsledky provádění metod *b()* a *c()*. Obrázek 9 je grafické znázornění podobné sekvence (v anglické literatuře se takové diagramy nazývají „diagramy typu box-and-channel“). Z ukázky je zřetelné, že metody *b()* a *c()* mohou být spuštěny souběžně, před nimi musí být provedena metoda *a()*, a v poslední řadě je provedena metoda *d()*. S využitím Future lze kód napsat tak, jak je uvedeno ve zdrojovém kódu 8.

```
1 ExecutorService executorService = Executors.newFixedThreadPool(1);
2 Future<Integer> x = executorService.submit(() -> someLongLogic());
3 // Some logic dosn't need x
4 System.out.println(x.get());
5 executorService.shutdown();
```

Zdrojový kód 6: Získání hodnoty z *Future*

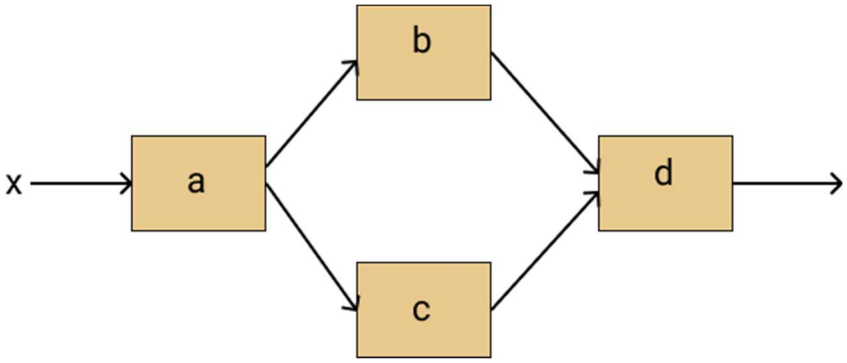
```
1 ExecutorService executorService = Executors.newFixedThreadPool(2);
2 Future<Integer> x = executorService.submit(() -> someLongLogic());
3 Future<Integer> y = executorService.submit(() -> someOtherLongLogic());
4 // Some logic dosn't need x and y
5 System.out.println(x.get() + y.get());
6 executorService.shutdown();
```

Zdrojový kód 7: Kombinování dvou hodnot z *Future*

Problém však nastává v situaci, kdy hodnota ještě nebyla přijata, metoda *get()* na ni čeká a tím blokuje vlákno. Pokud se v systému nachází pouze menší množství souběžnosti, nejedná se o tolik zásadní překážku. Pokud je ale systém plný diagramů typu box-and-channel, o problém už se jedná. *„V této situaci může mnoho úkolů čekat (s voláním get()) na dokončení Future, a výsledkem může být nedostatečné využívání hardwarového paralelismu nebo dokonce deadlock.“*[9] (Urma, 2019, s. 374)

Implementace CompletableFuture tyto problémy řeší. Poskytuje asynchronní API pro vyhýbání se blokování, kombinátory pro kombinování složitých řetězců souběžných úkolů, a navíc také nástroje pro zpracování chyb.

[9] In this situation, many tasks might be waiting (with a call to get()) for a Future to complete, and as discussed in section 15.1.2, the result may be underexploitation of hardware parallelism or even deadlock.



Obrázek 8: Diagram typu box-and-channel

```
1 int a1 = a(x);
2 Future<Integer> b1 = executorService.submit(() -> b(t));
3 Future<Integer> c1 = executorService.submit(() -> c(t));
4 int d1 = d(b1.get(), c1.get());
```

Zdrojový kód 8: Kombinace sekvenčních a souběžných volání s *Future*

2.3.1 Přechod z Future na CompletableFuture.

CompletableFuture získal své pojmenování díky metodě *complete()*. *„Obyčejná Future se obvykle vytváří pomocí Callable, která se spouští, a výsledek se získá pomocí metody get(). Ale CompletableFuture vám umožňuje vytvořit Future, aniž byste jí dali spustit jakýkoli kód, a metoda complete () umožňuje, aby ji nějaké jiné vlákno později doplnilo hodnotou (proto dané pojmenování), aby get() měl přístup k této hodnotě.“*[10] (Urma, 2019, s. 375) Zdrojový kód 9 je variace zdrojového kódu 7, s rozdílem použití CompletableFuture a metody *complete()*. Tato verze však stále používá metodu *get()*, která může blokovat vlákno, pokud výpočet *x* trvá déle než *y*. Řešením je použití kombinujících metod. V tomto případě je vhodná metoda *thenCombine()* (zdrojový kód 10[11]). Tato metoda vezme výsledky dvou *CompletableFuture* a použije na ně funkci *fn*, když jsou výsledky již k dispozici, ale bez blokování vlákna (tato ukázka je vyobrazena ve zdrojovém kódu 11). *„Metoda thenCombine vytváří výpočet, jehož spuštění ve fondu vláken je naplánováno, až když budou dokončeny oba dva první výpočty.“*[12]

[10] An ordinary Future is typically created with a Callable, which is run, and the result is obtained with a get(). But a CompletableFuture allows you to create a Future without giving it any code to run, and a complete() method allows some other thread to complete it later with a value (hence the name) so that get() can access that value.
[11] CompletionStage je rozhraní, které CompletableFuture implementuje.
[12] The thenCombine method creates a computation that's scheduled to run in the thread pool only when both of the first two computations have completed.

```
1 ExecutorService executorService = Executors.newFixedThreadPool(10);
2 CompletableFuture<Integer> x = new CompletableFuture<>();
3 executorService.submit(() -> x.complete(someLongLogic()));
4 int y = someOtherLongLogic();
5 System.out.println(x.get() + y);
6 executorService.shutdown();
```

Zdrojový kód 9: Kombinace sekvenčních a souběžných volání s *Future*

```
1 CompletableFuture<V> thenCombine(CompletionStage<U> other,
2                                   BiFunction<T, U, V> fn)
```

Zdrojový kód 10: Metoda *thenCombine()*

```
1 ExecutorService executorService = Executors.newFixedThreadPool(10);
2 CompletableFuture<Integer> x = new CompletableFuture<>();
3 CompletableFuture<Integer> y = new CompletableFuture<>();
4 CompletableFuture<Integer> z = x.thenCombine(y, (a, b) -> a + b);
5 executorService.submit(() -> x.complete(someLongLogic()));
6 executorService.submit(() -> y.complete(someOtherLongLogic()));
7 System.out.println(z.get());
8 executorService.shutdown();
```

Zdrojový kód 11: Používání metody *thenCombine()*

2.3.2 Nastavení fondu vláken

Jednou z nejjednodušších metod je statická metoda supplyAsync() (zdrojový kód 12). „Vrátí nový CompletableFuture, který je asynchronně dokončen úlohou spuštěnou ve ForkJoinPool.commonPool() s hodnotou získanou voláním daného Suppliera.“[13] (Oracle, ©1993, 2021) Metoda pouze odešle úkol přijatý jako Supplier do fondu vláken k provedení. Mnohem zajímavější je přetížená verze této metody, která umožní spustit úlohu s vlastním exekutorem, což znamená, že umožňuje uživateli přizpůsobit fond vláken pro své účely.

Při použití *Stream API* neexistuje způsob, jak konfigurovat fond vláken. Fond je jednoduše vybaven tolika vlákny, kolik je v systému jader. To je odůvodněno skutečností, že v této situaci je pro rychlejší provedení úkolu nutná skutečná simultánnost provádění, tj. paralelismus. Pokud existuje více vláken než jader, vlákna mezi sebou budou bojovat za čas

[13] Returns a new CompletableFuture that is asynchronously completed by a task running in the ForkJoinPool.commonPool() with the value obtained by calling the given Supplier.

použití a čas se bude plýtvat přepínáním kontextu. Jinými slovy, vlákna se budou navzájem rušit.

```
1 CompletableFuture<U> supplyAsync(Supplier<U> supplier)
2 CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
```

Zdrojový kód 12: Metody *supplyAsync()*

Větší počet vláken může být užitečný, pokud je například potřeba získat informace o velkém počtu objektů z prostředku třetí strany. Například metoda *supplyAsync()* spustí úlohu, která načte hodnocení konkrétního filmu z webu IMDB. Systém potřebuje hodnocení stovek filmů. Je zřejmé, že sekvenční řešení bude několikrát pomalejší než souběžné. Aby však bylo souběžné řešení nejefektivnější, je nutné vybrat vhodný počet vláken.

„Ideální velikost fondu vláken závisí na typech úkolů, které budou odeslány, a vlastnostech systému nasazení. Velikost fondu vláken by měla být zřídka pevně zakódována; místo toho by měly být velikosti fondu poskytovány konfiguračním mechanismem nebo vypočítávány dynamicky konzultováním Runtime.availableProcessors.“[14] (Goetz, 2006, s. 170) Aby zjistil velikost fondu vláken, Goetz zadá následující vzorec:

$$N_{threads} = N_{CPU} * U_{CPU} * \left(1 + \frac{W}{C}\right), \quad (1)$$

kde N_{CPU} je počet procesorů nebo jader; U_{cpu} je cílové využití CPU, $0 \leq U_{CPU} \leq 1$; $\frac{W}{C}$ je poměr doby čekání ku času výpočtu.

U výše uvedeného příkladu s filmy je poměr W/C přibližně 99 % z důvodu, že aplikace většinou čeká na odpověď z webu. Požadované využití CPU je 100 %. V systému se 4 jádry udává tento vzorec přibližně 400 vláken. Urma (2019, s. 401) však navrhuje zvolit počet vláken jiným způsobem. Koneckonců, nemá smysl mít více vláken, než je počet souběžných požadavků. Navíc příliš mnoho požadavků může server zhroutit. Z tohoto důvodu navrhuje Urma přijmout tolik vláken, kolik má systém požadavků ve stejný čas, ne však více než 100.

Pro příklad s filmy lze nakonfigurovat exekutora jako ve zdrojovém kódu 13. Při konfiguraci jsou vlákna označena jako démoni (*angl.* – *daemon*). Výše uvedené je vyžadováno, protože vlákna-démoni jsou vždy zničena po ukončení hlavního programu.

[14] The ideal size for a thread pool depends on the types of tasks that will be submitted and the characteristics of the deployment system. Thread pool sizes should rarely be hard-coded; instead pool sizes should be provided by a configuration mechanism or computed dynamically by consulting Runtime.availableProcessors.

```
1 private final Executor executor =
2             Executors.newFixedThreadPool(
3                  Math.min(films.size(), 100),
4                  (Runnable r) -> {
5                      Thread t = new Thread(r);
6                      t.setDaemon(true);
7                      return t;
8                  });
```

Zdrojový kód 13: Nastavení vlastního fondu vláken

Přístup, podle kterého je nalezena vhodná velikost fondu vláken, samozřejmě závisí na stanovených cílech. Nejvhodnější variantou je, když je možné spustit aplikaci několikrát s různým nastavením fondu a poté porovnat výkon.

Volání *supplyAsync()* k načtení hodnocení *filmu* může vypadat podobně jako ve zdrojovém kódu 14. Metoda *getRating()* instance *film* odešle požadavek na server. Metoda *supplyAsync()* asynchronně spustí provádění metody *getRating()* na dříve vytvořeném fondu vláken.

```
1 CompletableFuture.supplyAsync(() -> film.getRating(), executor);
```

Zdrojový kód 14: Použití metody *supplyAsync()*

```
1 List<Integer> ratings =
2                films.stream()
3                     .map(film -> CompletableFuture.supplyAsync(
4                            () -> film.getRating(), executor))
5                     .collect(Collectors.toList());
```

Zdrojový kód 15: Používání *CompletableFuture* se *Stream API*

Pro spuštění *getRating()* na všech požadovaných filmech je možné použít *Stream API*, které bylo popsáno v podkapitole 2.2. Ukázka je vyobrazena ve zdrojovém kódu 15, kde *films* je kolekce potřebných filmů. Jedná se o stejné použití *supplyAsync()* jako ve zdrojovém kódu 14, ale pro všechny filmy v kolekci. U každého filmu je metoda *getRating()* odeslána do fondu vláken. Pokud je v kolekci méně než sto filmů, pak každé vlákno provede jeden požadavek, v opačném případě jsou požadavky rozděleny mezi sto vláken.

2.3.3 Zřetězení a kombinování CompletableFuture

Metoda *thenApply()*, znázorněna ve zdrojovém kódu 16, přebírá instanci *Function* a aplikuje ji na hodnotu volajícího *Future*. Metoda vrátí *CompletableFuture*, který obsahuje výsledek

provádění *fn*. Ve zdrojovém kódu 17 přijímá metoda *thenApply()* výsledek metody *someComputation()* a posílá jej do metody *someAnotherComputation()*.

```
1 CompletableFuture<U> thenApply(Function<T, U> fn)
```

Zdrojový kód 16: Metoda *thenApply()*

```
1 CompletableFuture<Integer> result =
2       CompletableFuture.supplyAsync(() -> someComputation())
3                        .thenApply(s -> someAnotherComputation(s));
```

Zdrojový kód 17: Používání metody *thenApply()*

Podobnou metodou je *thenAccept()*, která je uvedena ve zdrojovém kódu 18. Rozdíl je v tom, že přijímá nikoli *Function*, ale *Consumera*, který vrací void. To může být užitečné například v situaci, když není potřeba vrátit hodnotu. Ve zdrojovém kódu 19 vytiskne *thenAccept()* výsledek metody *someAnotherComputation()* do konzole a vrátí *CompletableFuture<Void>*.

```
1 CompletableFuture<Void> thenAccept(Consumer<T> action)
```

Zdrojový kód 18: Metoda *thenAccept()*

```
1 CompletableFuture<Void> result =
2         CompletableFuture.supplyAsync(() -> someComputation())
3                          .thenAccept(
4                            s -> System.out.println(someAnotherComputation(s)
5                          ));
```

Zdrojový kód 19: Používání metody *thenAccept()*

Metoda *thenCombine()*, znázorněná ve zdrojovém kódu 20, kombinuje výsledky operací prováděných dvěma nezávislými *CompletableFuture*. Při použití *thenCombine()* jsou dvě *CompletableFutures* prováděny paralelně a poté je na ně použita funkce *fn* (zdrojové kódy 20 a 21).

```
1 CompletableFuture<V> thenCombine(CompletionStage<U> other,
2                                                 BiFunction<T, U, V> fn)
```

Zdrojový kód 20: Metoda *thenCombine()*

```
CompletableFuture<Integer> result =
1 CompletableFuture.supplyAsync(() -> someComputation())
2                  .thenCombine(
```

```
3                   CompletableFuture.supplyAsync(
4                       () -> someAnotherComputation()),
5                       (a, b) -> a + b
6                   );
```

Zdrojový kód 21: Používání metody *thenCombine()*

V této podkapitole již byla zmíněna schopnost paralelně spouštět *CompletableFuture* pomocí *Stream API*. Další možností je použít metodu *allOf()*, kterou lze nalézt ve zdrojovém kódu 22. Tato metoda vrátí nový *CompletableFuture*, který se dokončí v momentě, až budou dokončeny všechny předávané *CompletableFuture*. Analogická metoda *anyOf()* (zdrojový kód 23) vrací *CompletableFuture*, který se dokončí, když se dokončí některý z předaných *CompletableFuture*, a to se stejným výsledkem.

```
1 CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
```

Zdrojový kód 22: Metoda *allOf()*

```
1 CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

Zdrojový kód 23: Metoda *anyOf()*

Většina metod má dvě další verze s postfixem *Async* v názvu. Metody bez tohoto postfixu běží na volajícím vlákně. Metody s touto příponou fungují stejným způsobem jako *supplyAsync()*, což znamená, že posílají úkol do fondu vláken. Toto téma je detailněji popsáno v podkapitole 2.3.2.

2.3.4 Výhody a nevýhody používání CompletableFuture

Tato podkapitola samozřejmě nepokrývá všechna možná použití *CompletableFuture*, ale dotýká se pouze hlavních bodů. Na tomto základě je však již možné učinit závěr o výhodách a nevýhodách této technologie.

Výhody použití *CompletableFuture*:

1. poskytuje asynchronní API neblokující vlákno;
2. umožňuje použít vlastní fond vláken;
3. poskytuje mnoho metod pro řetězení a kombinování paralelních úkolů;
4. dává možnost odeslat úkol do fondu vláken nebo jej spustit na volajícím vlákně.

Nevýhody použití *CompletableFuture*:

1. přes své plusy funguje *CompletableFuture* stále s multithreadingem a zneužití může vést k neočekávanému chování.

2.4 Reaktivní programování v Java

Reaktivní programování je programovací paradigma, které umožňuje zpracovávat a kombinovat proudy datové položky z různých systémů a zdrojů asynchronním způsobem (Urma, 2019, s. 417). Zdůvodnění, proč je toto paradigma zapotřebí, popisuje Reaktivní Manifest následovně: *„Dnešní aplikace běží na všem od mobilních telefonů po cloudové clustery s tisíci multi-core procesory. Uživatelé očekávají odezvu v rámci milisekund a 100% dostupnost; data měříme v petabytech. Dnešní požadavky nelze zkrátka uspokojit včerejšími strukturami.“* (Bonér, © 2014)

Reaktivní Manifest byl vytvořen k popisu principů, které musí všechny reaktivní systémy dodržovat. Podle manifestu by měl být reaktivní systém *responzivní*, *odolný*, *pružný* a *zprávami-řízený*. To znamená, že by takový systém měl rychle reagovat na uživatele, zůstávat responzivní i v případě výpadku či při proměnlivé zátěži, a komponenty systému musí komunikovat pomocí asynchronního zasílání zpráv (Bonér, © 2014).

Reaktivní programování je programování, které využívá reaktivní proudy. Reaktivní streamování je technika založená na návrhovém vzoru publikování – odběr *(angl. – Publish-Subscribe)*. To znamená, že některé programové komponenty se mohou přihlásit k odběru aktualizací dalších programových komponent a včas na ně reagovat. Každý odběratel se může přihlásit k odběru libovolného počtu vydavatelů a každý vydavatel může mít libovolný počet odběratelů.

Jakmile vydavatel aktualizuje stav, okamžitě informuje všechny své odběratele formou zaslání zpráv. Zde se zavádí koncept zpětného tlaku (nebo protitlaku, *angl. – backpressure*). Může se stát, že vydavatel aktualizuje svůj stav příliš rychle, a ne všichni předplatitelé mají čas na zpracování každé aktualizace. Protitlakový mechanismus může být implementován různými způsoby. Jednou z variant je, že vydavatel odesílá data pouze předplatitelům, kteří jsou právě teď ochotni je přijmout. Nebo je pošle každému jednotlivému předplatiteli v okamžiku, kdy požaduje data.

2.4.1 Flow API

Java 9 zavádí novou třídu java.util.concurrent.*Flow*, která poskytuje rozhraní pro implementaci reaktivního programování. Obsahuje pouze statické komponenty a nelze ji vytvořit instancí. Třída *Flow* obsahuje čtyři vnořená rozhraní:

- *Publisher*;
- *Subscriber*;
- *Subscription*;
- *Processor*.

```
1 @FunctionalInterface
2 public interface Publisher<T> {
3     void subscribe(Subscriber<? super T> s);
4 }
```

Zdrojový kód 24: Rozhraní *Publisher*

V dokumentaci Oracle (©1993, 2021) je třída Flow popsána následovně: „Vzájemně propojená rozhraní a statické metody pro vytváření komponent řízených tokem, ve kterých Publishery produkují položky spotřebované jedním nebo více Subscribery, každý spravovaný Subscriptionem.“[15]

Publisher je funkční rozhraní, tj. rozhraní, které obsahuje pouze jednu abstraktní metodu. Jedná se o metodu *subscribe()*, kterou se *Publisher* přihlásí k odběru *Subscribera* (zdrojový kód 24).

Rozhraní *Subscriber* má čtyři metody (zdrojový kód 25). Tyto metody musí být volány striktně podle protokolu popsaného následujícím regulárním výrazem:

```
onSubscribe onNext* (onError | onComplete)?
```

Metoda *onSubscribe()* je vždy volána jako první. Volá se *Publisherem*, jakmile se *Subscriber* přihlásí k odběru a prostřednictvím ní odešle instanci Subscription. Pak se metoda *onNext()* volá tolikrát, kolikrát to bude potřeba, potenciálně donekonečna. Odběr je poté dokončen voláním jedné z metod *onComplete()* nebo *onError()*.

[15] Interrelated interfaces and static methods for establishing flow-controlled components in which Publishers produce items consumed by one or more Subscribers, each managed by a Subscription.

```
1 public interface Subscriber<T> {
2     void onSubscribe(Subscription s);
3     void onNext(T t);
4     void onError(Throwable t);
5     void onComplete();
6 }
```

Zdrojový kód 25: Rozhraní *Subscriber*

Rozhraní *Subscription* deklaruje dvě metody: *request()* a *cancel()* (zdrojový kód 26). S první metodou *Subscriber* upozorní *Publishera*, že je připraven přijmout data (realizace protitlaku), a druhou pak informuje, že se chce odhlásit.

```
1 public interface Subscription {
2     void request(long n);
3     void cancel();
4 }
```

Zdrojový kód 26: Rozhraní *Subscription*

Rozhraní *Processor* rozšiřuje rozhraní *Publisher* i *Subscriber*, ale nedeklaruje žádné doplňující metody (zdrojový kód 27). Je zodpovědný za přeměnu reaktivního proudu a stojí mezi *Publisherem* a *Subscriberem*. Může například formátovat data pocházející od *Publishera* nebo nějak reagovat na chybu.

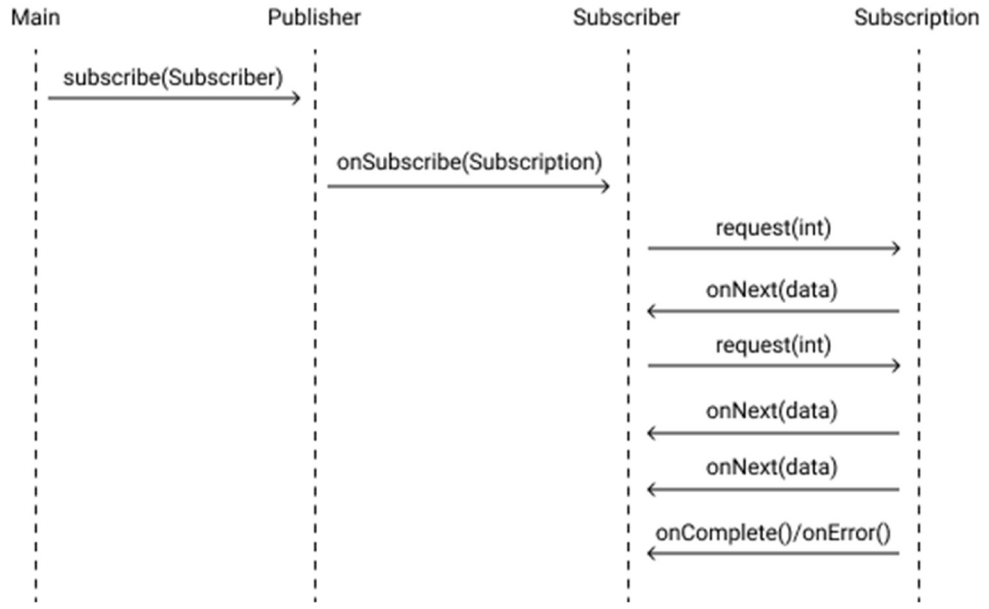
Standardní cyklus reaktivní aplikace využívající *Flow API* bude tedy vypadat podobně jako na obrázku 10.

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

Zdrojový kód 27: Rozhraní *Processor*

Java obvykle vždy poskytuje hotové implementace svých rozhraní. Rozhraní ze třídy Flow jsou však výjimkou. Důvodem je, že již existovaly knihovny reaktivních vláken třetích stran pro Javu. Flow API je primárně určeno ke standardizaci konceptu reaktivního programování pro Javu. Tento standard umožňuje větší spolupráci mezi různými knihovnami (Urma, 2019, s. 431).

Je tedy obtížné si představit, že by jakýkoli vývojář psal vlastní implementaci rozhraní z *Flow API*. Pravděpodobnější variantou je zvolení jedné z knihoven třetích stran, jichž dnes existuje velký výběr, jako například *RxJava*, *Spring WebFlux*, *Akka*, *Ratpack* a další. Všechny výrazně rozšiřují a doplňují možnosti rozhraní *Flow API*.



Obrázek 9: Cyklus reaktivní aplikace s *Flow API*

2.4.2 Výhody a nevýhody využití reaktivního programování v Javě

Reaktivní programování je tématem tak širokým, že by vydalo na samostatnou knihu. Tato práce však poskytuje pouze stručný přehled nativně poskytovaných nástrojů Java pro práci s reaktivními proudy. Je čas zvážit výhody a nevýhody této technologie v kontextu Javy.

Výhody využití reaktivního programování v Javě:

1. je představeno rozhraní standardizující paradigma;
2. existuje velký výběr knihoven třetích stran.

Nevýhody využití reaktivního programování v Javě:

1. žádná vlastní nativní implementace.

2.5 Souhrn

Zatímco v prostředí Java 1.0 museli vývojáři pracovat přímo s vlákny, od zavedení prostředí Java 8 je přímá interakce s nimi minimalizována. Přes svůj věk (první verze Javy byla vydána v roce 1996) splňuje moderní požadavky a poskytuje rozhraní pro implementaci moderních architektur, jako jsou reaktivní systémy.

Stream API je vhodným řešením pro paralelizaci úkolů. Hladce rozdělí úkoly mezi hardwarová vlákna tak, že se uživatel nemusí starat o vlákna. Při práci s paralelními streamy by však uživatel stále měl mít na mysli, že ne všechny zdroje dat a ne všechny metody *Stream API* jsou vhodné pro paralelizaci.

Třída *CompletableFuture<T>* umožňuje synchronizovat souběžné úkoly a řetězit je. A zároveň, pokud je aplikace naprogramována správně, vlákna nebudou blokována, což znamená, že aplikace běží asynchronně.

Další metodou programování asynchronních aplikací je využití paradigmatu reaktivního programování. Java pro něj poskytuje *Flow API*, které implementuje mnoho knihoven třetích stran, jako jsou *RxJava*, *Akka* a tak dále.

3 PROSTŘEDÍ PARALELISMU VE SKRIPTOVACÍCH JAZYCÍCH NA PŘÍKLADU JAVASCRIPTU

JavaScript je široce používán pro skriptování webových stránek. Zvláštností jazyka je single-threading, přesto jazyk podporuje souběžnost a asynchronii i paralelismus.

Na začátku kapitoly je popsáno prostředí, ve kterém je spuštěn JavaScript. Pak tato kapitola popisuje technologie, které implementují souběžnost a asynchronii, od zpětných volání po *async/await*. Na konci kapitoly je popsáno, jak je možné implementovat paralelismus v JavaScriptu pomocí *Web Workers*.

3.1 Běhové prostředí JavaScriptu

Před zvážením souběžnosti v JavaScriptu je třeba pochopit, jak tento jazyk funguje a jak funguje prostředí, ve kterém běží jeho kód. Faktem je, že samotný JavaScript se velmi liší od jazyků, jako je Java. Má jenom jedno vlákno a tato skutečnost sama o sobě by měla naznačovat, že nemá smysl uvažovat o paralelismu. Běhové prostředí JavaScriptu však umožňuje souběžnost a asynchronii.

V současné době jsou nejpopulárnější běhová prostředí JavaScriptu prohlížeč a Node.js. Node.js přináší do jazyka API pro interakci s operačním systémem. V rámci této práce byl jako popsané prostředí JavaScriptu zvolen prohlížeč.

Boduch (2015, s. 16–21) zdůrazňuje následující komponenty: samotné běhové prostředí, v němž běží interpret JavaScriptu, fronty úkolů (*angl. – task queues*) a smyčku událostí (*angl. – event loop*). Navíc k výše zmíněným má interpret k dispozici fronty úloh (*angl. – job queues*) a zásobník volání (*angl. – call stack*), nazývaný také kontextový zásobník (*angl. – context stack*).

Běhové prostředí je kontejner, který obsahuje vše, s čím bude kód JavaScript interagovat. Vytváří se při otevření webové stránky. Toto je takzvaný sandbox, ze kterého se JavaScript nemůže dostat ven.

Běhové prostředí interně obsahuje jednu nebo více front úkolů. Fronty se liší podle priority a typu úkolů. Například fronta s nejvyšší prioritou obsahuje úkoly pro interakci s uživatelem, další fronta obsahuje úkoly pro vykreslení a tak dále.

Tyto fronty jsou obsluhovány smyčkou událostí. Protože zde existuje pouze jedno vlákno, běhové prostředí má pouze jednu smyčku událostí. Smyčka událostí čeká na nové úkoly a vždy obsluhuje úkoly ve frontě s nejvyšší prioritou jako první.

Události, jako jsou kliknutí na prvek nebo načtení skriptu, vytvoří úkol, který vyvolá interpret jazyka JavaScript. Ten spustí určitou funkci, která bude pracovat, dokud se nedokončí. Z toho důvodu se v anglické literatuře JavaScript nazývá „run-to-completion language“. Když interpret spustí funkci, tlačí ji na vrchol zásobníku volání. To je aktuální kontext spuštění. Až funkce dokončí své provádění, interpret ji odebere z vrcholu zásobníku volání. Pokud je další funkce volána z aktuální funkce, pak je tato nová funkce také přesunuta na vrchol zásobníku a po dokončení odstraněna.

Fronta úloh je analogická s frontou úkolů, ale nachází se uvnitř interpretu a prohlížeč s nimi nijak neinteraguje. Existují pouze dvě základní fronty úloh: pro vytváření nových zásobníků volání a pro řešení funkcí zpětného volání (*angl. – callback function*) spuštěných promisy (popsáno detailněji v podkapitole 3.3.1).

Nyní, když již existuje představa o tom, jak funguje mechanismus běhového prostředí JavaScriptu, je možné přejít k tomu, jak jsou v tomto jazyce implementovány paralelismus, souběžnost a asynchronismus.

3.2 Callback funkce

„Jedním z přístupů k asynchronnímu programování je umožnit funkcím, které provádějí pomalou akci, další argument, funkci zpětného volání. Akce je spuštěna a po dokončení je volána funkce zpětného volání s výsledkem.“[16] (Haverbeke, 2018, s. 183)

```
1 var data = getSomeData("http://resourse.name");
2 console.log(data);
```

Zdrojový kód 28: Nefunkční příklad práce s asynchronní funkcí

Jako příklad lze uvést metodu *getSomeData()*, která vytvoří asynchronní požadavek na síť a poté se přijatá data vytisknou do konzole, jak je znázorněno ve zdrojovém kódu 28. V době, kdy se volá metoda *console.log()*, však metoda *getSomeData()* ještě nevrátila data. To je místo, kde je vhodné využít funkci zpětného volání (tzv. callback funkce), která se předává asynchronní metodě jako druhý argument (zdrojový kód 29). Tato callback funkce bude volána pouze po dokončení spuštění *getSomeData()* (Simpson, 2015).

[16] One approach to asynchronous programming is to make functions that perform a slow action take an extra argument, a callback function. The action is started, and when it finishes, the callback function is called with the result.

```
1 getSomeData("http://resourse.name", (data) => console.log(data));
```

Zdrojový kód 29: Příklad práce s asynchronní funkcí pomocí zpětného volání

I když se to na první pohled může zdát jako dobré, ne-li elegantní řešení, ve skutečnosti může ve velkých systémech zpětné volání vést ke značným problémům.

3.2.1 „Callback hell“

Zdrojový kód 30 ukazuje typický příklad vnořených zpětných volání. K tomu dochází, když chce vývojář nastavit synchronizaci pomocí zpětných volání. Takový kód se mezi programátory často nazývá „callback hell“, neboli „peklo zpětného volání“ (Simpson, 2015; Boduch, 2015, s. 31).

```
1 doFirst( () => {
2     doSecond();
3     doThird( () => {
4         doFourth();
5     })
6     doFifth();
7 });
8
9 doSixth();
```

Zdrojový kód 30: „Callback hell“

Problém pekla zpětného volání není pouze o hnízdění, to je spíše vedlejší účinek. Pořadí volání ve zdrojovém kódu 30 bude následující: *doFirst() => doSixth() => doSecond() => doThird() => doFifth() => doForth()*. Ale to pouze v případě, že *doFirst()* a *doThird()* jsou skutečně asynchronní. Navíc pořadí volání je zde pevně zakódováno, což povede k problémům, pokud nějaká metoda selže. Tento styl programování to samozřejmě může zajistit, ale tento kód bude nečitelný, neškálovatelný a také nebude znovupoužitelný (Simpson, 2015).

Další problém se zpětnými voláními se nazývá „Problémy důvěryhodnosti“ (*angl. – Trust issues*). Jde o to, že asynchronní funkce, ke kterým odesíláte zpětná volání, jsou často dodávány třetími stranami. Není tedy možné přesně vědět, jak funguje nástroj poskytovaný třetí stranou (Simpson, 2015).

3.2.2 Výhody a nevýhody zpětného volání

Se všemi výše uvedenými skutečnostmi je docela obtížné najít nevýhody používání zpětných volání. Asynchronní kód využívající pouze zpětná volání je obtížné škálovat a číst. Vede k mnoha problémům, jako jsou „Callback hell“ a „Problémy důvěryhodnosti“.

Bez ohledu na to je užitečné pochopit, jak zpětná volání fungují. Nejen proto, že se jedná o základ asynchronismu JavaScriptu, ale také proto, že další pohodlnější a pokročilejší techniky jsou založeny na použití callbacků.

3.3 Objekty Promise

Promise je taková asynchronní akce, která slibuje, že se někdy v budoucnosti dokončí a vrátí hodnotu. Je možné si představit kontejner pro hodnotu, která zatím neexistuje, ale jednou bude. Objekty *Promise* umožňují psát lepší souběžný kód bez velkého množství boilerplate kódů pro kontrolu stavů (Boduch, 2015, s. 33; Haverbeke, 2018, s. 185). V podstatě *Promise* ze JavaScriptu ztělesňuje stejnou myšlenku jako *CompletableFuture* z Javy, který je popsán v podkapitole 2.3, ale jiným způsobem.

3.3.1 Struktura Promise

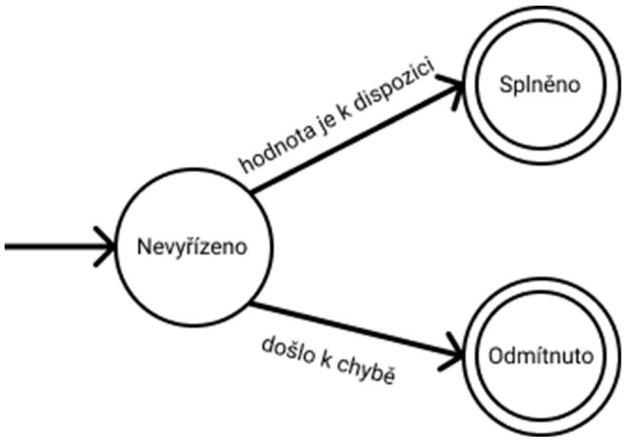
Promise je objekt, který momentálně neobsahuje hodnotu. Po nějaké době v budoucnosti se tato hodnota objeví a bude k dispozici pro callback funkce *then()*. Nebo z nějakého důvodu bude promise odmítnut, což povede k chybě. Promise tedy může být v jednom ze tří stavů: Nevyřízeno (*angl. – Pending*), Splněno (*angl. – Fulfilled*) a Odmítnuto (*angl. – Rejected*).

Promise je ve stavu Nevyřízeno od okamžiku, kdy byl vytvořen, dokud není vyřešen nebo odmítnut. Z tohoto stavu pak může přejít na Splněno nebo Odmítnuto. Pokud je všechno v pořádku a data jsou přijata, přejde do stavu Splněno, pokud dojde k neočekávané chybě, pak do stavu Odmítnuto. Od tohoto okamžiku zůstane navždy v jednom z těchto dvou stavů. To může být reprezentováno jednoduchým konečným automatem, jako je vyobrazeno na obrázku 11.

Konstruktor *Promise* přebírá jeden argument, čímž je funkce exekutora, která je pak odpovědná za to, jak je hodnota přijímána. Exekutor zase vezme dva argumenty, kterými jsou funkce resolveru a funkce rejektoru. Po předání resolveru exekutorovi jej lze volat kdekoli v kódu. Jakmile k tomu dojde, *Promise* přejde do stavu Splněno a poté se provedou callbacky, které byly předány *then()*. Rejektor funguje stejně jako resolver, s tím rozdílem, že pracuje

s chybou, ne s hodnotou. Když je volán, Promise přejde do stavu Odmítnuto a pak bude proveden callback zpracování chyby předaný *then()* nebo *catch()*, pokud existuje. Obecný případ vytvoření *Promise* může být podobný jak je uvedeno ve zdrojovém kódu 31.

V kontextu *Promise* je také využíván termín thenable, který označuje objekt, který má metodu *then()*. (Boduch, 2015, s. 33–34).



Obrázek 10: Stavy objektu *Promise*

```
1 new Promise((resolve, reject) => {...});
```

Zdrojový kód 31: Obecné vytvoření objektu *Promise*

Zdrojový kód 32 ukazuje příklad objektu *Promise*, který se vždy splní. V tomto kódu exekutora vezme pouze první parametr (resolver) a předá jej jako funkci zpětného volání *setTimeout()*, který jej za sekundu spustí. Hodnotou tohoto *Promise* bude řetězec „fulfilled“ předaný *resolve()*.

Jakmile k tomu dojde, spustí se metoda *then()*. Nepustí však v tento moment svou funkci zpětného volání. Jak je již popsáno na začátku podkapitoly 3.1, interpret jazyka JavaScript má k dispozici dvě fronty úloh. První z nich, hlavní fronta, se používá k inicializaci nových kontextových zásobníků. Do druhé fronty metoda *then()* přidá svá zpětná volání, když je splněn *Promise*. Zvláštností této druhé fronty je, že interpret z ní začne odesílat úkoly k provedení, až když bude hlavní fronta prázdná. Metoda *then()* je podrobněji popsána v podkapitole 3.3.2.

Výsledkem zdrojového kódu 32 bude řádek *„fulfilled“* vytištěný v konzoli jednu sekundu po spuštění.

Stejná sémantika funguje pro callbacky pro zachycení chyb, jako je na znázorněno ve zdrojovém kódu 33. *Promise* v této ukázce funguje stejně jako zdrojový kód 32, pouze je

vždy odmítnut. Rozdíl je v tom, že odmítnuté sliby mají jako hodnotu chybu. Tato chyba je zachycena buď zpětným voláním metody *catch()*, jako v kódu, nebo zpětným voláním, které je předáno metodě *then()* jako druhý argument.

```
new Promise((resolve) => {
    setTimeout(() => {
        resolve('fulfilled');
    }, 1000);
}).then((value) => console.log(value));
```

Zdrojový kód 32: *Promise*, který se vždy splní

```
new Promise((resolve, reject) => {
    setTimeout(() => {
        reject('rejected');
    }, 1000);
}).catch((e) => console.error(e));
```

Zdrojový kód 33: *Promise*, který je vždy odmítnut

Při implementaci požadavku HTTP na *Promises* může exekutor svázat posluchače událostí s požadavkem, takže v případě, kdy je spuštěna událost „load“, volá se *resolve()*; v případě, kdy je spuštěna událost „error“, volá se *reject()* a nakonec, v případě, kdy je spuštěna událost „abort“, pak se volá *resolve()* bez předání hodnoty. Příklad implementace tohoto přístupu je uveden ve zdrojovém kódu 34. *„Namísto toho, abychom vždy museli vytvářet obslužné rutiny pro události „load“, „error“ a „abort“, musíme si dělat starosti pouze s jedním rozhraním – promise.“*[17] (Boduch, 2015, s. 43)

```
function getData(address){
    return new Promise( (resolve, reject) => {
        var request = new XMLHttpRequest();
        request.open('GET', address, true);

        request.addEventListener('load', (data) => {
            resolve(JSON.parse(data.target.responseText));
        });

        request.addEventListener('error', (error) => {
            reject(error.target.statusText || 'unexpected error');
```

[17] Instead of always having to create handlers for the load, error, and abort events, we only have one interface to worry about–the promise.

```
12          });
13
14          request.addEventListener('abort', resolve);
15
16          request.open();
17          request.send();
18
19      });
20  }
```

Zdrojový kód 34: Implementace požadavku na server pomocí promise

3.3.2 Zřetězení objektů Promise

Metoda then vrací instanci Promisu. To umožní vytvářet libovolně dlouhé řetězy *then().then().then()* atd. Každé takové volání *then()* nezmění původní promise, ale vrátí nový. Objekty Promis vytvořené pomocí *then()* jsou však vázány na původní promise. Z toho vyplývá, že když je vyřešen první promise, je okamžitě vyřešen promise spojený s ním a tak dále podél řetězu. (Boduch, 2015, s. 48, 51).

Metoda *then()* přijímá jeden nebo dva parametry. První je funkce zpětného volání splnění (volá se, pokud promise přijde do stavu Splněno), druhé je funkce zpětného volání odmítnutí (volá se, pokud promise přijde do stavu Odmítnuto). Pokud je některý z těchto parametrů vynechán, použije se výchozí callback. Výchozí callback splnění jednoduše předá zprávu, zatímco výchozí callback odmítnutí jednoduše znovu vyvolá důvod chyby, který obdrží (Simpson, 2015).

```
1 new Promise((resolve) => {
2     resolve('fulfilled');
3 }).then((value) => {
4     console.log('value 1', value);
5 }).then((value) => {
6     console.log('value 2', value);
7 });
```

Zdrojový kód 35: Zřetězení objektů *Promise*

Ve zdrojovém kódu 35 je uveden příklad zřetězení promise pomocí metody *then()*. Zde je hodnota prvního promise předána první funkci zpětného volání *then()*. Není však dostatečně zřejmé, že dále tato hodnota není předána poslednímu *then()*. Faktem je, že zatímco první promise implicitně předává hodnotu přijatou resolverem, callback funkce *then()* musí

explicitně vrátit hodnotu, aby mohl jít dále v řetězci. Aby tato hodnota byla předána dál, měl by kód vypadat jako ve zdrojovém kódu 36.

```
1 new Promise((resolve) => {
2     resolve('fulfilled');
3 }).then((value) => {
4     console.log('value 1', value);
5     return value;
6 }).then((value) => {
7     console.log('value 2', value);
8     return value;
9 });
```

Zdrojový kód 36: Zřetězení objektů *Promise* s návratovou hodnotou

```
1  new Promise((resolve, reject) => {
2      reject('rejected');
3  }).then((value) => {
4      console.log(value);
5      return value;
6  }, e => {
7      console.error(e);
8  }).catch( e => {
9      console.error(e);
10 });
```

Zdrojový kód 37: Chytání chyb pomocí metod *then()* a *catch()*

Zpracování chyb lze provést jak metodou *then()*, tak metodou *catch()*, jak je znázorněno ve zdrojovém kódu 37. Jak je uvedeno výše, druhým parametrem metody *then()* je zpětné volání odmítnutí. Metoda *catch()* zase přijímá pouze jeden parametr a toto je také zpětné volání odmítnutí. Na základě všeho, co je známo o metodách *then()* a *catch()*, lze volání metody *catch(rejected)* přirovnat k volání metody *then(null, rejected)* (Simpson, 2015).

3.3.3 Užitečné statické metody třídy Promise

Kromě metod *then()* a *catch()* poskytuje třída Promise řadu statických metod. Mezi nimi jsou *Promise.resolve()* a *Promise.reject()*. Berou nějakou hodnotu jako argument a s touto hodnotou vrátí splněný (respektive odmítnutý) promise.

To může být užitečné v případech, kdy je pohodlnější pracovat s objekty *Promise*, a ne se samotnými hodnotami. Například stejná funkce může být asynchronní nebo synchronní, v závislosti na situaci. Použití *Promise.resolve()* a *Promise.reject()* umožní vrátit instanci

Promise, i když je funkce prováděna synchronně. Tímto způsobem je možné standardizovat návratovou hodnotu, která bude vždy typu *Promise* (Boduch, 2015, s. 59–62).

Další dvojice statických pomocných metod je *Promise.all()* a *Promise.race()*. Jako argument se berou pole objektů *Promise*.

Promise.all() vrací promise, který bude splněn, jakmile budou splněny všechny přijaté objekty *Promise*, nebo bude odmítnut, jakmile bude odmítnut alespoň jeden z přijatých objektů. V prvním případě hodnotou tohoto nového promise bude pole splněných hodnot přijatých promise, v druhém případě to bude první hodnota důvodu odmítnutí. Pokud *Promise.all()* přijme prázdné pole, vrácený promise bude splněn okamžitě. Tato metoda je užitečná, když je potřeba nějakým způsobem kombinovat výsledky více asynchronních funkcí.

Řešením *Promise.race()* bude první dokončený promise z pole, ať už byl promise splněn nebo zamítnut. Pokud je přijaté pole prázdné, vrácený promise nebude nikdy vyřešen. Tato metoda je užitečná v případech, kdy při splnění jednoho z objektů *Promise* jsou ostatní objekty *Promise* již irelevantní. (Simpson, 2015).

3.3.4 Výhody a nevýhody Promise

Třída *Promise* nabízí mnoho výhod oproti běžným zpětným voláním při práci s asynchronními funkcemi. Umožňují zacházet s asynchronními hodnotami jako s normálními primitivními typy. Poskytuje kompaktní a uživatelsky přívětivé rozhraní.

Přestože *Promise* používají zpětná volání, neposílají je k provedení do nástrojů třetích stran. Za spouštění zpětných volání je zodpovědná metoda *then()*, která je odesílá do speciální fronty úloh. Tím se vyřeší problém důvěryhodnosti popsaný v podkapitole 3.2.

Přes všechny tyto výhody není *Promise* dokonalým řešením. Lepší řešení je popsáno v následující části.

3.4 Generátory

Podkapitola 3.1 vysvětluje, že smyčka událostí vykonává funkci od začátku do konce. Nic ji nemůže přerušit, až do konce práce zabírá vlákno. Z tohoto důvodu se JavaScript nazývá „run-to-complete language“. S generátory získá JavaScript novou funkcionalitu. *Generátory* jsou funkce, které lze přerušit uprostřed provádění. Pouze rozhodnutí o přerušení není učiněno zvenčí, ale je provedeno samotnou funkcí generátoru. Jedná se o kooperativní formu

souběžnosti, ve které vlákno, které drží zámek, rozhoduje, v jakém okamžiku přenese zámek na vlákno čekající. Generátory nejsou „run-to-complete“.

3.4.1 Iterátory

Iterátor je objekt, který přijímá data od producenta dat pomocí metody *next()*. Při každém volání metoda *next()* vrací nový datový prvek poskytnutý producentem. Nejjednodušším příkladem takového producenta je pole.

Iterátor nelze použít na každý objekt, ale pouze na objekt, který je „iterovatelný“ (*angl. – iterable*). Iterovatelný objekt musí mít ve své struktuře funkci nazvanou Symbol.iterator. Jedná se o speciální hodnotu poskytovanou ES6[18]. Tato funkce by měla vrátit instanci iterátoru. Všechny integrované datové struktury JavaScriptu jsou iterovatelné.

Iterátory lze implicitně použít v jazyce. Smyčka *for..of* používá iterátor poskytovaný polem k iteraci nad prvky pole. Tudíž lze prvky pole procházet jak metodou zobrazenou ve zdrojovém kódu 38, tak metodou zobrazenou ve zdrojovém kódu 39.

```
1 let arr = [1, 2, 3, 4];
2
3 for(let value of arr){
4     console.log(value);
5 }
```

Zdrojový kód 38: Implicitní iterace prvků pole

```
1 let arr = [1, 2, 3, 4];
2 let it = arr[Symbol.iterator]();
3 console.log(it.next().value);
4 console.log(it.next().value);
5 console.log(it.next().value);
6 console.log(it.next().value);
```

Zdrojový kód 39: Explicitní iterace prvků pole

Je důležité si uvědomit, že metoda *next()* nevrací samotný prvek pole, ale vrací objekt s dvěma vlastnostmi. První je *value*, obsahuje prvek pole. Druhá je vlastnost typu boolean *done*, obsahuje true v případě, že byly předány všechny prvky pole.

[18] ES6 nebo ECMAScript 6 je šestá verze jazyka ECMAScript, který je základem JavaScriptu.

3.4.2 Generátory, iterátory a jejich komunikace

K deklaraci generátoru se používá následující syntaxe: *function* generator()*. V těle takové funkce je povoleno klíčové slovo *yield*. Jakmile interpret dosáhne slovo *yield*, funkce je pozastavena (zdrojový kód 40).

```
1 function* generator(){
2     console.log('start');
3     yield;      // pause
4     console.log('end');
5 }
```

Zdrojový kód 40: Generátor a klíčové slovo *yield*

```
1 let it = generator();
2 it.next();      // start
3 it.next();      // end
```

Zdrojový kód 41: Generátor je spuštěn iterátorem

Jak je to vidět ve zdrojovém kódu 41, generátor nelze spustit jen jako normální funkce. První řádek ve zdrojovém kódu 41 nespustí funkci *generator()*. Tím se vytvoří instance generátoru, která vrací instanci iterátoru, která je uložena v proměnné *it*. Ke spuštění kódu zapsaného v těle funkce generátoru se používá metoda *next()* iterátoru. Po prvním volání *next()* funkce generator() vytiskne řetězec "start" a pozastaví se kvůli slovu *yield*. Po druhém volání *next()* bude funkce generator() pokračovat tam, kde přestala, vytiskne řetězec "end" a ukončí se.

Podkapitola 3.4.1 vysvětluje, že metoda iterátoru *next()* vrací objekt, obsahující běžný prvek, který byl poskytnut producentem dat, a informace o tom, zda existuje další prvek. Iterátor generátoru se v tomto ohledu neliší od všech ostatních iterátorů. Tato podkapitola používá vzorový kód zobrazený ve zdrojovém kódu 42 k vysvětlení, jak tento mechanismus funguje v kontextu generátorů.

```
1  function* generator(a){
2      let b = a * (yield "It's first yield");
3      let c = (yield "It's second yield");
4      return b * c;
5  }
6
7  let it = generator(5);
8
9  console.log(it.next().value);   // It's first yield
10 console.log(it.next(5).value);  // It's second yield
```

```
11 console.log(it.next(5).value);  // 125
```

Zdrojový kód 42: Komunikace mezi generátorem a iterátorem

Ve zdrojovém kódu 42 je nyní nějaký řádek po slovu *yield*. Tento řetězec bude hodnotou vlastnosti value objektu, který vrátí metoda *next()*. Slovo *yield* tedy předává data iterátoru. Lze to považovat za přechodný *return*.

Počínaje druhým voláním má metoda *next()* nějakou hodnotu jako parametr. Tato hodnota nahradí slovo *yield* tam, kde generátor naposledy přestal. To lze přirovnat k obousměrnému kanálu pro zasílání zpráv. Generátor odesílá zprávy do iterátoru pomocí slova *yield* a iterátor odesílá zprávy pomocí metody *next()*.

Z výše uvedeného vyplývá, že ve zdrojovém kódu 42 nastává následující:

1. Řádek `let it = generator(5);` vytvoří instanci *generator*, jíž je předána hodnota 5 jako argument, a iterátor tohoto generátoru je uložen v proměnné *it*;
2. Je volána metoda `it.next()`, která poprvé spustí funkci `generator(5)`;
3. Metoda provede `let b = a *` a zastaví se uprostřed výrazu. V současné době není známo, co by mělo být na místě *yield*, a metoda bude pokračovat v provádění, jakmile bude známo;
4. `(yield "It's first yield")` vrací objekt s řetězcem "It's first yield" do iterátoru;
5. Konzola vytiskne hodnotu *value* získanou iterátorem objektu (a to je "It's first yield");
6. Je volána metoda `it.next(5)`. Číslo 5, předané jako argument, nahradí naposledy použitý *yield* v kódu generátoru;
7. Generátor se znovu zastaví při příštím *yield*, který volající metodě vrátí „Je to druhý yield“;
8. Třetí volání `it.next(5)` stejným způsobem pošle číslo 5 do generátoru, ale ve vlastnosti *value* získá to, co vrátí klíčové slovo *return*.

3.4.3 Souběžnost s generátory

Podkapitola 3.2 ve zdrojovém kódu 28 ukazuje nefunkční kód. Díky generátorům je však možné dosáhnout toho, že podobný kód fungoval.

Ve zdrojovém kódu 43 *getDataAsync()* je asynchronní funkce třetí strany, která načítá data z adresy URI. Přijímá URI a funkce zpětného volání jako argumenty. Řádky 17-18 jsou ve skutečnosti stejné jako ve zdrojovém kódu 28, ale v tomto případě fungují podle očekávání.

Ve zdrojovém kódu 43 dochází k následujícímu:

1. Na řádku 26 iterátor *it* spustí funkci *generator()*;
2. Až generátor dosáhne řídku 17, zastaví se. V tomto případě slovo výtěžek nevrátí nic iterátoru, ale spustí funkci *getData()*;
3. Funkce *getData()* volá asynchronní funkci třetí strany *getDataAsync()*. Až *getDataAsync()* dokončí svůj běh, zavolá svou funkci zpětného volání;
4. Pokud *getDataAsync()* selže, funkce zpětného volání zavolá metodu iterátoru *throw()*, která vyvolá chybu (řádek 6). Tato chyba je zachycena v bloku *catch* generátoru (řádek 20);
5. V opačném případě funkce zpětného volání zavolá metodu iterátoru *next()* a odešle přijatá data do generátoru (řádek 9);
6. Řízení se vrací do generátoru a pokračuje v provádění tam, kde skončilo (řádek 17). Uloží přijatá data do proměnné *data* a okamžitě je vytiskne do konzoly.

```
1  function getData(address){
2      getDataAsync(
3          address,
4          function(err, data){
5              if(err){
6                  it.throw(err);
7              }
8              else{
9                  it.next(data);
10             }
11         }
12     );
13 }
14
15 function* generator(){
16     try {
17         var data = yield getData('http://resourse.name');
18         console.log(data);
19     } catch (e) {
20         console.error(e);
21     }
22 }
23
24 var it = generator();
25
26 it.next();
```

Zdrojový kód 43: Souběžnost s generátorem

Řádky 17–18 zdrojového kódu 43 jsou tedy asynchronní kód napsaný synchronním stylem.

Tento způsob použití zpětných volání bohužel vede k problémům popsaným v podkapitole 3.2. To lze napravit kombinací generátorů a objektů *Promise*, jak je znázorněno ve zdrojovém kódu 44.

K vyžádání serveru se použije zdrojový kód 34.

Navzdory zcela odlišnému způsobu přijímání dat ze serveru se kód generátoru vůbec nezměnil. Stále jen volá funkci a čeká, až vrátí odpověď. Tentokrát však yield vrátí hodnotu iterátoru a toto je instance *Promisu*.

Díky tomu iterátor může uložit objekt promisu do proměnné, což umožňuje později použít metodu *then()* tohoto promisu. Metoda *then()* spustí výraz `it.next(data)`, pokud je promise splněn, a v opačném případě pak `it.throw(err)`. Rozdíl s předchozím příkladem spočívá v tom, že tam byly tyto výrazy spuštěny jako funkce zpětného volání pro funkci jiného výrobce a nyní jako funkce zpětného volání pro metodu *then()*.

```
1  function getData(address){
2      return new Promise( (resolve, reject) => {
3          var request = new XMLHttpRequest();
4          request.open('GET', address, true);
5
6          request.addEventListener('load', (data) => {
7              resolve(JSON.parse(data.target.responseText));
8          });
9
10         request.addEventListener('error', (error) => {
11             reject(error.target.statusText || 'unexpected error');
12         });
13
14         request.addEventListener('abort', resolve);
15
16         request.open();
17         request.send();
18     });
19 }
20
21 function* generator(){
22     try {
23         var data = yield getData('http://resourse.name');
24         console.log(data);
25     } catch (e) {
26         console.error(e);
27     }
28 }
29
30 var it = generator();
```

```
31
32 var promise = it.next().value;
33
34 promise.then(
35     (data) => {
36         it.next(data);
37     },
38     (err) => {
39         it.throw(err);
40     }
41 );
```

Zdrojový kód 44: Spolupráce generátoru a promise

3.4.4 Výhody a nevýhody generátorů

Generátory vám umožňují provádět věci, které dříve nebyly v JavaScriptu možné. Jedná se o přerušení funkce uprostřed provádění.

Výše uvedená inovace otevírá velmi důležitou příležitost: psát asynchronní souběžný kód, jako by byl synchronní postupný.

Před napsáním sekvenčního kódu je bohužel potřeba napsat velký doplněk v podobě implementace Promisu. *Async* funkce, které jsou popsány v následující podkapitole, tento problém řeší.

3.5 async/await

ES7 přinesl do jazyka dvě nová klíčová slova: *async* a *await*. Slovo *async* označuje funkci, ze které budou asynchronní funkce volány. Slovo *await* je povoleno pouze v „*async* funkcích“. Je umístěno před voláním asynchronní funkce a signalizuje, že je nutné počkat na výsledek. *„Async funkce je speciální typ generátoru. Při volání produkuje promise, který je splněn, když se vrátí (dokončí), a odmítnut, když vyvolá výjimku. Klíčové slovo await vrátí promise, a výsledek tohoto promisu (hodnota nebo vyvolaná výjimka) je výsledkem výrazu čekání.“*[19] (Haverbeke, 2018, s. 197)

Ve skutečnosti se jedná o generátor, ve kterém byla hvězdička nahrazena klíčovým slovem *async*, klíčové slovo *yield* bylo nahrazeno slovem *await*. Navíc vývojář nemusí

[19] An async function is a special type of generator. It produces a promise when called, which is resolved when it returns (finishes) and rejected when it throws an exception. Whenever it yields (awaits) a promise, the result of that promise (value or thrown exception) is the result of the await expression.

implementovat promise a iterátor, generátor to činí sám. Již použitý příklad ze zdrojového kódu 28, při používání konstrukce *async*/*await*, by mohl vypadat jako ve zdrojovém kódu 45. Zde slovo *await* zastaví funkci *getData()*, aby počkala na ukončení *getDataAsync()* a na vrácení hodnoty. Když *getDataAsync()* něco vrátí, *getData()* bude pokračovat.

```
1 async function getData(address){
2     var data = await getDataAsync(address);
3     return data;
4 }
```

Zdrojový kód 45: Používání konstrukci *async*/*await*

```
1 getData("http://resourse.name")
2     .then(data => anotherAsyncOperation(data))
3     .then(data => console.log(data))
4     .catch(e => console.error(e));
```

Zdrojový kód 46: *async* funkce vratí instance promise

Návratová hodnota async funkce je promise, což znamená, že všechno, co platí pro promise, platí i pro takové funkce. Například při volání *getData()* lze použít metody *then()* a *catch()*, jak je znázorněno ve zdrojovém kódu 46.

3.5.1 Asynchronní kód v synchronním stylu

Konstrukce *async/await* umožňuje psát asynchronní kód jako by byl synchronní. To zvyšuje čitelnost a srozumitelnost kódu. Zdrojový kód 46 ukazuje příklad, kde lze výsledek *async* funkce prodloužit řetězením metod *then()* a *catch()*. I když je to možné, tento styl není nutný. Posloupnost několika příkazů *await* v rámci *async* funkce lze srovnat s řetězcem volání *then()*. Zdrojový kód 47 funguje stejně jako zdrojový kód 46. I když se jedná o asynchronní kód, posloupnost volání a chycení chyb vypadá stejně jako při psaní synchronního kódu.

Příklad uvedený na obrázku 47 řeší situaci, kdy každé následující asynchronní volání používá výsledek předchozího. Často však existují situace, kdy je třeba kombinovat data z různých zdrojů, a je výhodné mít tyto asynchronní funkce spuštěné souběžně.

```
1 async function getData(address){
2     try {
3         var data = await getDataAsync(address);
4         data = await anotherAsyncOperation(data);
5         console.log(data);
```

```
6     } catch (e) {
7         console.error(e);
8     }
9 }
```

Zdrojový kód 47: Kód *async* funkce vypadá jako synchronní

```
1 async function concurrentAsync(){
2     const promise1 = await getDataAsync(address1);
3     const promise2 = await getDataAsync(address2);
4     const promise3 = await getDataAsync(address3);
5
6     const result = combineData(promise1, promise2, promise3);
7     return result;
8 }
```

Zdrojový kód 48: Neefektivní příklad souběhu s *async/await*

Toho samozřejmě nelze dosáhnout pomocí zdrojového kódu 48. Tady každé následné volání asynchronní funkce čeká na dokončení předchozího, což je zbytečná ztráta času.

Skutečnost, že *await* výrazy také vrací promisy, pomáhá vyřešit tento problém. Ve zdrojovém kódu 49 jsou asynchronní volání spuštěna bez použití klíčového slova *await*. Proto každá následující asynchronní funkce běží bez čekání na konec předchozí, což dává souběh. Funkce *concurrentAsync()* se zastaví pouze na řádku s *Promise.all()*, což znamená, že čeká na dokončení všech asynchronních volání současně.

```
1 async function concurrentAsync(){
2     const promise1 = getDataAsync(address1);
3     const promise2 = getDataAsync(address2);
4     const promise3 = getDataAsync(address3);
5
6     const result = await Promise.all([promise1, promise2, promise3]);
7     return result;
8 }
```

Zdrojový kód 49: Souběžnost v *async* funkci pomocí *Promise.all()*

3.5.2 Výhody a nevýhody používání async/await

Konstrukce *async/await* je v podstatě založena na všech asynchronních technikách popsaných dříve v této kapitole. Každá nová technologie, která se objevila v JavaScriptu, byla navržena tak, aby usnadňovala práci s asynchronním kódem. Hlavním cílem bylo usnadnit psaní a čtení

kódu. Asynchronní kód, podle definice, může být nejen obtížně pochopitelný, ale může také fungovat různě podle toho, jaký vývojář ho napsal. Díky *async/await* již tento problém není.

S *async/await* je možné nyní psát asynchronní kód synchronním stylem. A to bez programování promisů, generátorů a iterátorů. Proto se vývojář může soustředit na základní problém, aniž by byl rozptylován implementací asynchronie.

Kromě toho je učení *async/await* mnohem jednodušší než například funkce zpětného volání. Aby však bylo možné použít *async/await* efektivně, měl by si vývojář také být vědom promisů.

3.6 Web Workery

V této kapitole je opakovaně řečeno, že JavaScript je jazyk s jedním vláknem. Navzdory tomu JavaScript efektivně implementuje souběžnost a asynchronii. S jediným vláknem však není možné efektivně používat vícejádrové procesory ani implementovat skutečný paralelismus. Web Workery poskytují východisko z této situace.

Web Worker je kód, který běží na vlákně na úrovni operačního systému. Běh Web Workerů je tedy řízen operačním systémem a jeho plánovačem, což vede k nejúčinnějšímu způsobu využití procesoru (Boduch, 2015, s. 94).

Data mezi pracovníkem a hlavním vláknem programu jsou přenášena pomocí zpráv. Tyto zprávy jsou před odesláním serializovány a po doručení deserializovány. Přijímač tedy neobdrží samotný objekt, ale jeho kopii, proto hlavní vlákno a workery nikdy nepracují přímo se stejným objektem. Kromě toho to omezuje práci workerů: je nemožné předat funkci ve zprávě, nemohou spolu komunikovat – to je možné pouze s hlavním vláknem (Boduch, 2015, s. 97, 102).

Kód workera běží na jiném vlákně a jeho objekty jsou vytvářeny v jiném globálním kontextu. Worker proto nemůže použít proměnnou *window* k získání aktuálního globálního kontextu, musí k tomu použít proměnnou *self*. Worker také nemůže s DOM nijak interagovat (MDN Web Docs, © 2005–2021; Boduch, 2015, s. 99).

3.6.1 Typy Web Workerů

Podle MDN Web Docs (© 2005–2021) existují následující typy web workerů:

- Dedikované (*angl. – dedicated*) Workery jsou workery, které jsou k dispozici pouze skriptu, který je volal;

- Sdílené (*angl. – shared*) Workery jsou workery dostupné pro různé skripty, které mohou být umístěny v různých oknech a rámcích, nebo pro stejný skript, který je několikrát otevřen na různých kartách (MDN Web Docs, © 2005–2021; Boduch, 2015);
- Služební (*angl. – service*) Workery v podstatě fungují jako proxy, které jsou umístěny mezi webovými aplikacemi, prohlížečem a sítí (pokud existují);
- Chrome Workery slouží k vývoji doplňků a rozšíření;
- Audio Workery se používají pro přímé zpracování zvuku v kontextu web workera.

Boduch (2015, s. 97) také rozlišuje sub-workera jako samostatný typ, i když ve skutečnosti se jedná o worker vytvořený jiným workerem.

Jak je vidět z výše uvedeného krátkého popisu, každý typ workeru řeší různé problémy. V této práci jsou uvažovány dedikované workery, protože jejich funkčnost umožňuje řešení paralelních úkolů.

3.6.2 Paralelismus s dedikovanými workery

„Dedikované workery jsou pravděpodobně nejběžnějším typem workerů. Jsou považovány za výchozí typ web workera. Když naše stránka vytvoří nového workera, je věnován kontextu provádění stránky a ničemu jinému. Když naše stránka zmizí, udělají to i všechny dedikované workery vytvořené touto stránkou.“[20] (Boduch, 2015, s. 96)

Z hlediska jazyka je worker objekt a je vytvořen pomocí konstruktoru. Zpráva je workeru odeslána pomocí metody *postMessage()*. Odpověď workera je zachycena pomocí obslužné rutiny události „onmessage“. Kód pro vytvoření pracovníka, odeslání zprávy a přijetí odpovědi je uveden ve zdrojovém kódu 50. Takový kód by měl být zabalen podmíněným blokem `if (window.Worker)`, aby bylo zajištěno, že prohlížeč podporuje workery.

Kód workera je podobný kódu hlavního programu, jak ukazuje zdrojový kód 51. Zde worker používá obslužnou rutinu události k přijetí zprávy z hlavního programu a okamžitě odešle odpověď.

```
1 var worker = new Worker('worker.js');
2 worker.postMessage(`It's message from main thread`);
```

[20] Dedicated workers are probably the most common worker type. They're considered the default type of web worker. When our page creates a new worker, it's dedicated to the page's execution context and nothing else. When our page goes away, so do all the dedicated workers created by the page.

```
3 worker.addEventListener('message', (e) => {
4     console.log(e.data);
5 })
```

Zdrojový kód 50: Komunikace s workerem: kod hlavního programu

```
1 addEventListener('message', (e) => {
2     console.log(e.data);
3     postMessage(`It's message from worker`);
4 })
```

Zdrojový kód 51: Komunikace s workerem: kod workera

Hlavní vlákno může vytvořit libovolný počet těchto dedikovaných workerů, stejně jako dedikovaný worker může vytvořit libovolný počet sub-workerů. Zvláštností sub-workera je, že nemůže komunikovat přímo s hlavním vláknem, ale pouze s workerem, který jej vytvořil. Tím se otevírá možnost vytvoření více workerů nebo sub-workerů, které budou provádět dílčí úkoly paralelně v různých vláknech.

Pro paralelizaci úkolu bude muset vývojář implementovat princip podobný principu popsanému v podkapitole 2.2.3. Tady je programátor zodpovědný za všechno: od rozhodování, kolik workerů by mělo být vytvořeno, až po algoritmus pro rozdělení úkolu na dílčí úkoly.

Pro nejefektivnější paralelizaci stojí za to vytvořit tolik workerů, kolik má počítač jader. Většina prohlížečů poskytuje proměnnou *navigator.hardwareConcurrency* k určení počtu jader v systému (MDN Web Docs, © 2005-2021). Pokud prohlížeč tuto proměnnou nepodporuje, programátor bude muset uhodnout optimální počet workerů. Boduch (2015, s. 128) v tomto případě doporučuje použít číslo čtyři, protože se jedná o nejběžnější počet jader v moderních procesorech.

Vytváření pracovníků, stejně jako vytváření vláken, je náročné na zdroje. Proto by si měl vývojář pro každý úkol rozmyslet, zda má smysl paralelizovat nebo ne. Při rozhodování musí programátor vzít v úvahu dva faktory: velikost datového pole a časovou složitost operace pro jeden prvek (Boduch, 2015, s. 126).

Vývojář může rozhodnout, zda budou dílčí úkoly řešeny workery nebo sub-workery. V prvním případě se logika, která rozhoduje o tom, zda má být úkol paralelní, rozdělí úkol na dílčí úkoly a odešle je workerům, nachází v hlavním kódu programu. V druhém případě hlavní program odešle celý úkol jednomu workeru. Výše popsaná logika je umístěna uvnitř workera a sub-workery řeší dílčí úkoly.

3.6.3 Výhody a nevýhody využití workerů pro paralelismus

Před příchodem workerů v JavaScriptu nebylo možné spustit kód ve více vláknech, pokud jde o prostředí prohlížeče. Pracovníci odpovídají vláknům na úrovni operačního systému, což umožňuje efektivní využití procesoru uživatele aplikace.

Bohužel se může ukázat, že prohlížeč uživatele nepodporuje pracovníky. Další nevýhodou pracovníků je, že programátor musí implementovat paralelizační algoritmy sám, což odvádí pozornost od řešení okamžitých problémů.

3.7 Souhrn

Navzdory skutečnosti, že JavaScript je jazyk s jedním vláknem, tak asynchronie se souběžností i paralelismus jsou v něm plně realizovatelné. Během existence jazyka se princip psaní asynchronního kódu vyvinul z extrémně nepohodlných funkcí zpětného volání na *async/await*, což umožňuje psát asynchronní kód stejným způsobem jako synchronní.

V současné době lze dosáhnout paralelismu pomocí Web Workerů. Umožňují používat vlákna operačního systému, což zase umožňuje využívat plný výkon moderních vícejádrových procesorů. Weboví pracovníci spolu nekomunikují a pracují s kopiemi původních objektů, což eliminuje situace, kdy několik vláken soutěží o právo používat stejný objekt.

JavaScript je moderní jazyk, který plně implementuje možnosti asynchronismu, paralelismu a souběžnosti.

4 PROSTŘEDÍ PARALELISMU VE FUNKCIONÁLNÍCH JAZYCÍCH NA PŘÍKLADU CLOJURE

Clojure je funkcionální programovácí jazyk založený na Lispu, který běží na Java Virtual Machine. Clojure vylepšuje Lispovu syntaxi při zachování plné síly Lispu. Běh na JVM umožňuje použití všech tříd a metod Javy (Miller, 2018).

Standardně jsou všechny datové struktury v Clojure *immutable* (*doslova* z *angl. – neměnné nebo nemutující*). „*Podpora funkcionálního programování od Clojure usnadňuje psaní kódu bezpečného pro vlákna. Protože neměnné datové struktury se nikdy nemohou změnit, nehrozí nebezpečí poškození dat na základě aktivity jiného vlákna.“*[21] (Miller, 2018)

Přes zaměření na nemutující data podporuje také mutující data. To jsou refy, agenti a atomy. Programátor se tedy může rozhodnout použít mutující data, pokud si myslí, že je to v této situaci oprávněné. Výhodou nemutujících dat v Clojure je, že jsou orientována na souběžnost. „*Pokud potřebujete odkazy na proměnlivá data, Clojure je chrání prostřednictvím softwarové transakční paměti (STM)“*[22] (Miller, 2018). Na rozdíl od mnoha jiných funkcionálních jazyků má Clojure dynamickou typovou kontrolu. Volání Java není v zásadě funkcionální.

STM je mechanismus pro ochranu mutable dat pomocí transakce, nikoli pomocí zámků. To umožňuje bezpečné použití souběžnosti, a to i při mutaci dat.

Zdrojový kód Clojure muže být spuštěn v prostředí *„read-eval-print loop“* (REPL). Vypadá jako terminál, kde lze zadávat příkazy Clojure a okamžitě získat návratovou hodnotu.

Jazyk Clojure umožňuje psát jednoduchý paralelní kód bezpečný pro použití ve více vláknovém režimu (*angl. – thread-save code*).

4.1 Základy jazyka: syntaxe, kolekce, sekvence

„Clojure je Lisp. Clojure zaujímá nový přístup k Lispu tím, že zachovává základní myšlenky a zároveň zahrnuje sadu vylepšení syntaxe, díky nimž je Clojure přátelštější k programátorům, kteří nepoužívají Lisp.“[23] (Miller, 2018)

[21] Clojure’s support for functional programming makes it easy to write thread-safe code. Since immutable data structures cannot ever change, there’s no danger of data corruption based on another thread’s activity.

[22] When you need references to mutable data, Clojure protects them via software transactional memory (STM).

[23] Clojure is a Lisp. Clojure takes a new approach to Lisp by keeping the essential ideas while embracing a set of syntax enhancements that make Clojure friendlier to non-Lisp programmers.

4.1.1 Základy syntaxe

Clojure je funkcionální jazyk, proto jsou funkce hlavním prvkem jazyka. V anglické literatuře se píše, že funkce jsou *„first-class objects“*. *„Funkce mohou být vytvořeny za běhu, předány, vráceny a obecně použity jako jakýkoli jiný datový typ.“*[24] (Miller, 2018)

Jednou z datových struktur Clojure je *list*. Seznam je uzavřen v kulatých závorkách, jak je uvedeno ve zdrojovém kódu 52. Za normálních okolností však Clojure zachází s takovým listem jako s funkcí. První prvek seznamu je považován za název funkce, zbytek pak za argumenty funkce. Ve zdrojovém kódu 53 je funkce + s argumenty *1* a *2*. Symbol => označuje návratovou hodnotu. V souladu s tím bude *list* ve zdrojovém kódu 52 také považován za funkci *1,* s argumenty *2*, *3* a *4*. Listy jsou podrobněji popsány v podkapitole 4.1.2.

```
1 (1 2 3 4)
```

Zdrojový kód 52: *list* interpretovaný jako funkce *1*

```
1 (+ 1 2)
2 => 3
```

Zdrojový kód 53: Funkce +

```
1 (+ 1 2 4 5 10)
2 => 22
```

Zdrojový kód 54: Funkce + s pěti argumenty

```
1 (+)
2 => 0
```

Zdrojový kód 55: Funkce + bez argumentů

Mohlo by se zdát, že funkce + je podobná operátoru + z jazyků, jako je Java. Operátor + však může přijmout pouze dva operandy, například *2 + 3*. Tento styl umístění operátoru (mezi operandy) se nazývá *infixová notace*. Styl používaný v Clojure je *prefixová notace*, která umožňuje použít funkci + na neurčitý počet argumentů, dokonce bez argumentů, jak je uvedeno ve zdrojových kódech 54 a 55.

[24] Functions can be created at runtime, passed around, returned, and in general, used like any other datatype.

Klíčová slova *def* a *defn* se používají k definování konstant a funkcí (zdrojové kódy 56 a 57). Tato operace vrátí název namespace, ve kterém byl nový prvek vytvořen, a název samotného prvku (*user* je výchozí namespace).

Název *immutable-var* je nyní vázán na hodnotu *21*. Pro získání této hodnoty je stačí zadat do REPL *immutable-var*. Zadáním *(immutable-var)* se však vrátí chyba, protože hodnota v závorkách je považována za funkci.

```
1 (def immutable-var 21)
2 => #'user/immutable-var
```

Zdrojový kód 56: Definice konstanty

Název *simple-multiplication* je nyní vázán s funkcí, která vynásobí dvě čísla. Zde je jednoduchá syntaxe: za názvem funkce jsou její argumenty uvedeny v hranatých závorkách, potom je tělo funkce v kulatých závorkách. Volání této funkce je znázorněno ve zdrojovém kódu 58.

```
1 (defn simple-multiplication [a b]
2   (* a b))
3 => #'user/simple-multiplication
```

Zdrojový kód 57: Definice funkce

```
1 (simple-multiplication 2 3)
2 => 6
```

Zdrojový kód 58: Volání funkce

Na konec seznamu argumentů lze napsat symbol *&* a název dalšího argumentu. Tento název je vázán na neomezený počet argumentů. Tímto způsobem je možné při volání funkce tento argument úplně vynechat nebo lze předat neomezený počet argumentů. Příklad definování a volání funkce s takovým argumentem je uveden ve zdrojovém kódu 59.

Tělo funkce *multiple-multiplication* (zdrojový kód 59) používá funkci *reduce*. *Reduce* přijímá funkci, počáteční hodnotu (volitelně) a kolekci. Potom kumulativně použije funkci na všechny prvky v kolekci. Ve výsledku vrátí jeden objekt.

```
1 (defn multiple-multiplication [& args]
2   (reduce * args))
3 (multiple-multiplication)
4 => 1
5 (multiple-multiplication 1 2)
```

```
6 => 2
7 (multiple-multiplication 1 2 3 4 5 6 7)
8 => 5040
```

Zdrojový kód 59: Funkce s neurčitým počtem argumentů a *reduce*

Existují funkce s více aritami. Tyto funkce mají více seznamů argumentů a těl. Příklad definování takové funkce je uveden ve zdrojovém kódu 60. Toto je funkce hledání plochy čtyřúhelníku. Pokud přijímá pouze jeden argument, pak se plocha vypočítá pomocí vzorce pro plochu čtverce, pokud dva, pak pomocí vzorce pro plochu obdélníku.

```
1 (defn area
2   ([a] (* a a))
3   ([a b] (* a b)))
```

Zdrojový kód 60: Funkce s dvěma těly

Dalším typem funkce je anonymní funkce. Používá se pro stejné účely jako anonymní funkce v Javě. Definuje se klíčovým slovem *fn*. Příklad použití anonymní funkce je ve zdrojovém kódu 61.

Stejný zdrojový kód používá funkci *map*. Přijímá funkci a kolekci. Aplikuje funkci na každý prvek v kolekci a vrátí *liné sekvence* (líné sekvence jsou popsány v podkapitole 4.1.4).

Funkce přijímá argument typu kolekce. Ve zdrojovém kódu 61 je *list* předán jako argument.

```
1 (defn multiplication-by-two [numbers]
2   (map (fn [a] (* a 2)) numbers))
3
4 (multiplication-by-two '(1 2 3 4))
5 => (2 4 6 8)
```

Zdrojový kód 61: Anonymní funkce a *map*

4.1.2 Kolekce

Clojure podporuje čtyři typy kolekcí: *listy*, *vektory*, *sety* a *mapy*.

Listy jsou sekvenční kolekce uložené jako lineární spojový seznam. Jak je uvedeno výše, *list* je uzavřen v kulatých závorkách. Aby jazyk interpretoval *list* jako datovou strukturu, a ne jako funkci, musí být použita funkce *quote*. Funkce *quote* má tvar čtecího makra (*angl.* –

reader macro form), kterým je apostrof. Vytvoření *listu* oběma způsoby je ukázáno ve zdrojovém kódu 62. Zdrojový kód 61 také používal tvar čtecího makra.

```
1 (quote (1 2 3))
2 => (1 2 3)
3 '(1 2 3)
4 => (1 2 3)
```

Zdrojový kód 62: *List*

Vektory jsou sekvenční indexované kolekce. *Vektor* je uzavřen v hranatých závorkách. Vytvoření *vektoru* je ukázáno ve zdrojovém kódu 63.

```
1 [1 2 3]
2 => [1 2 3]
```

Zdrojový kód 63: *Vektor*

Sety neboli *množiny* jsou neuspořádané kolekce, které neobsahují duplicitní prvky. Uzavřeny jsou ve složených závorkách, před kterými je mřížka typu *#{}*. Vytvoření *setu* je ukázáno ve zdrojovém kódu 64. Vzhledem k tomu, že *set* je neuspořádaná kolekce, mohou být položky v návratové hodnotě v libovolném pořadí.

```
1 #{1 2 3}
2 => #{1 3 2}
```

Zdrojový kód 64: *Set*

```
1 {:one 1 :two 2 :three 3 :four 4}
2 => {:one 1, :two 2, :three 3, :four 4}
```

Zdrojový kód 65: *Mapa*

Mapy jsou kolekce párů klíč/hodnota,uzavřené ve složených závorkách. Syntaxe pro vytvoření mapy je následující: *{klíč1 hodnota1 klíč2 hodnota2 klíč3 hodnota3}*. Clojure zachází s čárkami jako s mezerami, takže je možné pro usnadnění použít čárku jako oddělovač: *{klíč1 hodnota1, klíč2 hodnota2, klíč3 hodnota3}*. Vytvoření *setu* je ukázáno ve zdrojovém kódu 65. Jako klíč v mapách se může používat libovolný typ dat, ale často se používají *klíčová slova*. *Klíčová slova* začínají dvojtečkou, jak je vidět ve zdrojovém kódu 65.

4.1.3 Sekvence

Všechny datové struktury v Clojure jsou *sekvence*. *Sekvence* je jednoduchá abstrakce, pomocí které lze dostat přístup ke všem výše uvedeným datovým strukturám. Neboli každou implementaci *sekvence*, jako je *list* nebo *vektor*, lze považovat za *sekvenci*.

Existují tři základní funkce, které lze použít na sekvence: *first*, *rest* a *cons*. Funkce *first* umožňuje získat první prvek sekvence. Funkce *rest* vrací sekvenci bez prvního prvku. Funkce *cons* vloží nový prvek na začátek sekvence. Použití těchto funkcí je uvedeno ve zdrojovém kódu 66. Tyto funkce samozřejmě nemění sekvence, ale vracejí nové sekvence založené na původních. Sekvence v Clojure jsou immutable.

```
1 (first '(1 2 3))
2 => 1
3 (rest '(1 2 3))
4 => (2 3)
5 (cons 1 '(2 3))
6 => (1 2 3)
```

Zdrojový kód 66: Funkce *first, rest* a *cons*

Výše uvedené funkce lze aplikovat také na mapy a sety. *„Mapy a sety mají stabilní pořadí procházení, ale toto pořadí závisí na podrobnostech implementace a neměli byste se na něj spoléhat. Prvky množiny se nemusí nutně vrátit v pořadí, ve kterém jste je vložili.“*[25] (Miller, 2018)

4.1.4 Líné (lazy) sekvence

„Většina Clojure sekvencí je líná (angl. – lazy); jinými slovy, prvky se nepočítají, dokud nejsou potřeba.“[26] (Miller, 2018) Líné sekvence umožňují vytvářet doslova nekonečné datové sady, aniž by zabíraly místo v paměti.

V Clojure existuje mnoho funkcí pro generování sekvencí. Většina z nich vrací líné sekvence. Mezi nimi jsou *range, repeat, iterate, take* a další.

Funkce *range* má následující verze: *(range)*, *(range end)*, *(range start end)* a *(range start end step)*. Jak název napovídá, *range* vrací sekvence čísel od *start* do *end*, s krokem *step*.

[25] Maps and sets have a stable traversal order, but that order depends on implementation details, and you shouldn't rely on it. Elements of a set will not necessarily come back in the order that you put them in.
[26] Most Clojure sequences are lazy; in other words, elements are not calculated until they're needed.

Pokud je pouze argument *start*, sekvence se začíná od *0* s krokem *1*. Zdrojový kód 67 obsahuje příklady použití této funkce. Funkce range bez argumentů vytvoří posloupnost čísel od 0 do nekonečna. Z tohoto důvodu ji nelze takto zadávat do REPL, protože to bude tisknout, dokud nedojde paměť.

```
1 (range 10)
2 => (0 1 2 3 4 5 6 7 8 9)
3 (range 10 20)
4 => (10 11 12 13 14 15 16 17 18 19)
5 (range 10 20 2)
6 => (10 12 14 16 18)
```

Zdrojový kód 67: Funkce *range*

Funkce *repeat* má dvě varianty: *(repeat x)* a *(repeat n x)*. Vrací *x* *n*krát (zdrojový kód 68). Pokud *n* chybí, vrátí nekonečnou sekvenci.

```
1 (repeat 3 1)
2 => (1 1 1)
```

Zdrojový kód 68: Funkce *repeat*

Funkce *iterate* má pouze jednu možnost: *(iterate f x)*. Vrátí nekonečnou línou sekvenci *(x, (f x), (f (f x)), ...)*. Pomocí této funkce lze vytvořit sekvenci popsanou funkcí. Protože je sekvence nekonečná, má smysl ji uložit pro pozdější použití.

Zdrojový kód 69 představuje funkci, která vrací sekvenci dvojic Fibonacciho čísel. Anonymní funkce přijímá vektor *[a b]* a vrátí vektor *[b (+ a b)]*. Funkce *iterate* odešle počáteční hodnotu *[0N 1N]* anonymní funkci, poté udělá totéž pro návratovou hodnotu a opakuje to donekonečna, přesněji, pokud to je potřeba. *N* je suffix označující, že číslo je typu *BigInt*.

```
1 (defn fibo-pairs []
2   (iterate (fn [[a b]] [b (+ a b)]) [0N 1N]))
```

Zdrojový kód 69: Dvojice Fibonacciho čísel pomocí *iterate*

```
1 (take 5 (fibo-pairs))
2 => ([0N 1N] [1N 1N] [1N 2N] [2N 3N] [3N 5N])
```

Zdrojový kód 70: Funkce *také*

Funkce *(take n coll)* vrací prvních *n* prvků kolekce *coll*. Ve zdrojovém kódu 70 funkce *take* vrací prvních 5 prvků sekvence *fibo-pairs*.

```
1 (defn fibo []
2   (map first (fibo-pairs)))
```

Zdrojový kód 71: Sekvence Fibonacciho čísel

```
1 (take 10 (fibo))
2 => (0N 1N 1N 2N 3N 5N 8N 13N 21N 34N)
```

Zdrojový kód 72: Prvních deset Fibonacciho čísel

```
1 (nth (fibo) 50)
2 => 12586269025N
```

Zdrojový kód 73: Padesáté Fibonacciho číslo

Existují také funkce, které transformují sekvence. Patří mezi ně *map* a *reduce,* již popsané v této kapitole. Funkce *map* umožňuje změnit sekvence použitím funkce na každý prvek. Zdrojový kód 71 zachovává novou sekvenci, která je vytvořena z fibo-pairs aplikací funkce *first* na každý z jejích prvků. Tato nová sekvence je sekvencí Fibonacciho čísel. Nyní lze tuto sekvenci snadno použít k získání prvních *n* Fibonacciho čísel pomocí funkce *take* (zdrojový kód 72) nebo *n*-té Fibonacciho číslo pomocí funkce *nth* (zdrojový kód 73).

Ve výše uvedených zdrojových kódech jsou nekonečné sekvence uloženy pomocí *defn*. Nezabírají paměť a každá hodnota se vypočítá, pouze když je to potřeba. Clojureova deklarativní syntaxe umožňuje implementovat do několika řádků funkce, které by v imperativním jazyce vyžadovaly více než 10 řádků (Miller, 2018).

Tato podkapitola popisuje jen několik funkcí sekvencí, jádro Clojure jich poskytuje mnohem více.

4.2 Paralelismus a souběh s immutable daty

4.2.1 Paralelní zpracování pomocí funkce pmap a knihovny reducers

Předchozí podkapitola popisuje funkci *map*, která vytváří sekvenci aplikací funkce na každý prvek vstupní sekvence. Funkce *pmap* funguje stejným způsobem, ale paralelně. Přístup používaný touto funkcí se nazývá *„semi-líný“* (*angl. – semi-lazy*), protože paralelní výpočet

předčí spotřebu, ale neimplementuje celý výsledek, pokud není vyžadován (Butcher, 2014, s. 61).

```
1 (defn long-area-of-square [a]
2   (Thread/sleep 1000)
3   (* a a))
```

Zdrojový kód 74: Funkce *long-area-of-square*

```
1 (time (doall (map long-area-of-square [5 10 7 25 45 65 12 11])))
2 => "Elapsed time: 8079.412 msecs"
3 => (25 100 49 625 2025 4225 144 121)
4 (time (doall (pmap long-area-of-square [5 10 7 25 45 65 12 11])))
5 => "Elapsed time: 1012.2091 msecs"
6 => (25 100 49 625 2025 4225 144 121)
```

Zdrojový kód 75: Porovnání výkonu *map* a *pmap*

Zdrojový kód 74 definuje funkci *long-area-of-square*, která emuluje dlouhý výpočet pomocí metody *Thread.sleep()* z Javy. Zdrojový kód 75 měří čas běhu *map* a *pmap* pro tuto funkci pomocí funkce *time*. Funkce *doall* je nutná pro provedení experimentu, protože eliminuje lazy chování. Zdrojový kód 75 běží na čtyřjádrovém procesoru.

Knihovna *reducers* poskytuje dostatek příležitostí pro paralelní výpočty. Poskytuje alternativní funkce sekvencí pro paralelní provoz. Zdrojový kód 76 importuje knihovnu *reducers* do prostředí jako *r*.

```
1 (require '[clojure.core.reducers :as r])
```

Zdrojový kód 76: Importování knihovny *reducers*

```
1 (map (fn [a] (* a a)) [5 10 7 25 45 65 12 11])
2 => (25 100 49 625 2025 4225 144 121)
```

Zdrojový kód 77: Funkce *map* vrací sekvenci

Funkce *map* z jádra Clojure vrací sekvenci, jak je vidět ve zdrojovém kódu 77. Analogická funkce *r/map* z knihovny *reducer* vrací objekt *reducible*, jak je znázorněno ve zdrojovém kódu 78. „*Objekt reducible není přímo použitelná hodnota; je to jen něco, co lze následně*

předat funkci reduce.“[27] (Butcher, 2014, s. 64) Obvykle to vypadá tak, jak je zobrazeno ve zdrojových kódech 79 nebo 80, které jsou navzájem rovnocenné.

```
1 (r/map (fn [a] (* a a)) [5 10 7 25 45 65 12 11])
2 => #object[clojure.core.reducers$folder$reify__214 0x255e5e2e
3                          "clojure.core.reducers$folder$reify__214@255e5e2e"]
```

Zdrojový kód 78: Funkce *r/map* vrací objekt *reducible*

```
1 (reduce conj [] (r/map (fn [a] (* a a)) [5 10 7 25 45 65 12 11]))
2 => [25 100 49 625 2025 4225 144 121]
```

Zdrojový kód 79: Funkce *r/map* a *reduce*

```
1 (into [] (r/map (fn [a] (* a a)) [5 10 7 25 45 65 12 11]))
2 => [25 100 49 625 2025 4225 144 121]
```

Zdrojový kód 80: Funkce *r/map* a *into*

Funkce *conj* ze zdrojového kódu 79 přijímá kolekci a alespoň jednu položku, kterou má přidat do této kolekce. Funkce *into* ze zdrojového kódu 80 funguje přesně stejným způsobem, ale interně používá *reduce*. Díky tomu jsou výrazy *reduce conj []* a *into []* zaměnitelné.

Knihovna reducers poskytuje vlastní verzi funkce r/reduce. Vývojáři jazyka však místo toho doporučují použít funkci r/fold. „Obecně většina uživatelů nebude volat r/reduce přímo a místo toho by měla preferovat r/fold, která implementuje paralelní redukci a kombinování.“[28] (Clojure, ©2008-2021)

Pro srovnání výkonu *reduce*, *r/fold* a *r/reduce* jsou do zdrojového kódu 81 zapsány tři jednoduché funkce sčítání. Zdrojový kód 82 vytváří sekvence čísel od nuly do deseti milionů s názvem *numbers*. Ve zdrojovém kódu 83 se funkce *sum*, *fold-sum* a *reduce-sum* provádějí dvakrát, aby kompilátor JIT provedl své optimalizace. V důsledku provedení je jasně vidět, že verze používající *r/fold* je téměř dvakrát rychlejší než verze s *r/reduce*. Zároveň je *r/reduce* pomalejší než *reduce*.

Pro paralelizaci úkolů *r/fold* používá framework *fork/join* z Java 7, který je popsán v podkapitole 2.3.3 (Butcher, 2014, s. 67).

[27] A reducible isn’t a directly usable value — it’s just something that can subsequently be passed to reduce.
[28] In general most users will not call r/reduce directly and instead should prefer r/fold, which implements parallel reduce and combine.

```
(defn sum [numbers]
  (reduce + numbers))
(defn fold-sum [numbers]
  (r/fold + numbers))
(defn reduce-sum [numbers]
  (r/reduce + numbers))
```

Zdrojový kód 81: Funkce *fold-sum* a *reduce-sum*

```
(def numbers (into [] (range 0 10000000)))
```

Zdrojový kód 82: Sekvence čísel od nuly do deseti milionů.

```
 (time (sum numbers))
 => "Elapsed time: 131.4211 msecs"
 => 49999995000000
 (time (fold-sum numbers))
 => "Elapsed time: 139.7427 msecs"
 => 49999995000000
 (time (reduce-sum numbers))
 => "Elapsed time: 123.2661 msecs"
 => 49999995000000
(time (sum numbers))
=> "Elapsed time: 86.9194 msecs"
=> 49999995000000
(time (fold-sum numbers))
=> "Elapsed time: 56.4056 msecs"
=> 49999995000000
(time (reduce-sum numbers))
=> "Elapsed time: 104.4756 msecs"
=> 49999995000000
```

Zdrojový kód 83: Porovnání funkcí *fold-sum* a *reduce-sum*

Funkce z knihovny *reducers* nevracejí výsledek, ale recept na výsledek. Tento recept musí být předán buď *reduce* nebo *fold*. Butcher (2014, s. 65) zdůrazňuje dvě výhody tohoto přístupu: je efektivnější než řetězení funkcí, které vracejí líné sekvence, protože není nutné vytvářet přechodné sekvence; to umožňuje funkci *fold* paralelizovat celý řetězec operací na výchozí kolekci.

Knihovna *reducers* poskytuje následující funkce: *r/map*, *r/mapcat*, *r/filter*, *r/remove*, *r/flatten*, *r/take-while*, *r/take* a *r/drop* (Clojure, ©2008-2021).

4.2.2 Souběh s future a promise

Java používá k implementaci souběžnosti rozhraní *Future* a JavaScript používá objekty *Promise*. Ve skutečnosti jsou *Future* z Javy a *Promise* z JavaScriptu dvě implementace stejné myšlenky. V Clojure je souběžnost rozdělena mezi objekty *future* a *promise*. Fungují trochu odlišně a navzájem se doplňují.

```
1 (defn future-simple (future (Thread/sleep 10000) (+ 1 2 3 4 5)))
2 (time @future-simple)
3 => "Elapsed time: 2894.6924 msecs"
4 => 15
```

Zdrojový kód 84: Fungování *future*

Funkce *future* přijímá tělo kódu, spustí jej asynchronně v jiném vlákně a vrátí objekt *future*. Pro získání hodnoty z *future* je třeba ji dereferovat pomocí *deref* nebo @. Při pokusu o dereference bude vlákno blokováno, dokud nebude k dispozici hodnota. Zdrojový kód 84 ukazuje nejjednodušší příklad použití *future*.

Tato syntaxe usnadňuje implementaci problému popsaného v podkapitole 2.3 a ilustrovaného na obrázku 9. Zdrojový kód 85 deklaruje funkce *a*, *b*, *c* a *d*. Funkce *sleep* a *println* se používají jenom pro demonstrace.

```
1 (defn a [x y]
2   (Thread/sleep 1000)(println "a function done")(+ x y))
3 (defn b [x]
4   (Thread/sleep 1000)(println "b function done")(* x x))
5 (defn c [x]
6   (Thread/sleep 1000)(println "c function done")(* x 2))
7 (defn d [x y]
8   (Thread/sleep 1000)(println "d function done")(* x y))
```

Zdrojový kód 85: Funkce *a, b, c* a *d*

Zdrojový kód 86 spouští deklarované funkce v pořadí znázorněném na obrázku 9; *let* implementuje vazbu. V tomto případě je `a-result` vázán na `(time (a 5 10))`, `b-result` vázán na `(future (time (b a-result)))` a dál. Pak *let* provede výraz. Pro přehlednost jsou funkce *a*, *b*, *c* a *d* volány z funkce *time*.

```
1  (let [a-result (time (a 5 10))
2        b-result (future (time (b a-result)))
3        c-result (future (time (c a-result)))]
```

```
4    (time (d @b-result @c-result)))
5  => a function done
6  => "Elapsed time: 1004.6414 msecs"
7  => c function doneb function done
8  =>
9  => ""EEllaappsseedd  ttiimmee::  11000055..35204155  mmsseeccs"
10 => s"
11 => d function done
12 => "Elapsed time: 2021.9261 msecs"
13 => 6750
```

Zdrojový kód 86: Souběžnost pomocí *future*

Výstup jasně ukazuje, že funkce *a* fungovala jako první. Poté funkce *b* a *c* dokončily svoji práci téměř současně, výstupy funkcí *time* se navzájem překrývaly. Funkce *d* běžela dvě sekundy, protože nejprve čekala na dokončení funkce *a* (jednu sekundu), poté funkcí *b* a *c* (také jednu sekundu).

Na rozdíl od objektu *future* objekt *promise* nespustí žádný kód. Místo toho získává svou hodnotu pomocí *deliver*. Ve zdrojovém kódu 87 je vytvořen objekt promise, poté je pomocí *future* vytvořeno nové vlákno. Toto nové vlákno vytiskne hodnotu *promise-objekt*, až bude k dispozici. Ve zdrojovém kódu 88 získá *promise-objekt* svou hodnotu a okamžitě jí vytiskne druhé vlákno.

```
1 (def promise-objekt (promise))
2 (future (println "The value of promise-objekt is:" @promise-objekt))
3 => #object[clojure.core$future_call$reify__8477 0x2f04105
4                                          {:status :pending, :val nil}]
```

Zdrojový kód 87: Vytvoření objektu *promise*

```
1 (deliver promise-objekt 20)
2 => The value of promise-objekt is: 20
3 => #object[clojure.core$promise$reify__8524 0x1ffcf674
4                                            {:status :ready, :val 20}]
```

Zdrojový kód 88: Objekt *promose* získá svou hodnotu

Nástroje uvedené v této podkapitole umožňují psát souběžné a paralelní aplikace s immutable daty. Clojure však není čistě funkcionální jazyk a podporuje zohledňující souběžnost mutable data, o nichž pojednává další podkapitola.

4.3 Mutable data zohledňující souběžnost

Clojure podporuje následující referenční datové typy, které jsou mutable:

- *refy* řídí koordinační, synchronní změny sdíleného stavu;
- *atomy* řídí nekoordinační, synchronní změny sdíleného stavu;
- *agenti* řídí asynchronní změny sdíleného stavu;

Každý z těchto typů funguje mírně odlišně a používá se v různých situacích.

4.3.1 Refy

Vytvoření *refu* ukazuje zdrojový kód 89. Hodnotu *refu* lze získat pomocí *@/deref*; *ref-set* se používá k nastavení nové hodnoty *refu*. Samotný *ref-set* však nelze použít. V takovém případě bude vyvolána výjimka *IllegalStateException*. Protože *ref* je mutable, musí být změny chráněny. Clojure to činí prostřednictvím transakcí. Transakce se volá uvnitř *dosync* (zdrojový kód 90).

```
1 (def mutable-ref (ref 21))
```

Zdrojový kód 89: Vytvoření *refu*

```
2 (dosync (ref-set mutable-ref 60))
3 => 60
```

Zdrojový kód 90: *dosync*

```
1 (def ref1 (ref 20))
2 (def ref2 (ref 21))
3 (dosync
4  (ref-set ref1 40)
5  (ref-set ref2 41))
6 => 41
```

Zdrojový kód 91: Transakce

Za transakce v Clojure odpovídá softwarová transakční paměť (STM). Transakce zajišťují atomicitu, konzistenci a izolovanost změn. Databáze obvykle navíc zaručují trvalost, ale STM ne. *„Pokud v jedné transakci změníte více než jeden ref, jsou všechny změny koordinovány*

tak, aby „se staly současně“ z pohledu jakéhokoli kódu mimo transakci“[29] (Miller, 2018). Zdrojový kód 91 ukazuje, jak změny nastanou přes dva *refy* v rámci jedné transakce.

Transakce nefunguje s *refy* přímo. Místo toho bere číslo, které STM představuje jako časové razítko, a pracuje s kopiemi *refů* připojených k tomuto časovému razítku. Pokud během operace STM zjistí, že *ref* byl změněn jinou transakcí, transakce se pokusí znovu vykonat jednotlivé operace. Pokud blok *dosync* vyvolá výjimku, transakce se nezopakuje (Miller, 2018).

Ref lze aktualizovat nejen pomocí *set-ref*, ale také pomocí *alter* a *commute*. Oba přijímají *ref*, funkci aktualizace a její argumenty. Zdrojový kód 91 demonstruje jejich použití. Poznámka: funkce *conj* se používá ke kombinaci kolekce a prvku, nikoli *cons*. Faktem je, že funkce *conj* přijímá sekvenci jako první argument a *cons* jako druhý. To je důležité: *alter* a *commute* používají *ref* jako první argument funkce aktualizace.

```
1 (def ref-list (ref ()))
2 (dosync (alter ref-list conj :alter-item))
3 => (:alter-item)
4 (dosync (commute ref-list conj :commute-item))
5 => (:commute-item :alter-item)
```

Zdrojový kód 92: *alter a commute*

Commute poskytuje lepší výkon než *alter*, protože si nedělá starosti s možnou další transakcí, která změní *ref*. Používá se v případech, kdy nezáleží na tom, v jakém pořadí jsou akce prováděny na *ref*. I přes výkon by měl být preferován *alter*. „*Pokud použijete alter, když by commute stačil, nejhorší věcí, která se může stát, je snížení výkonu. Pokud ale použijete commute, když je potřeba alter, zavedete jemnou chybu, kterou je obtížné odhalit pomocí automatických testů.“*[30] (Miller, 2018)

4.3.2 Atomy

Atom je entita lehčí než *ref*. Rozdíl je v tom, že pomocí transakce lze změnit několik *refů* jakoby současně, zatímco *atom* lze změnit pouze jeden po druhém (Miller, 2018). „*Atomy Clojure jsou postaveny na java.util.concurrent.atomic.“* (Butcher, 2014, s. 86)

---

[29] If you change more than one ref in a single transaction, the changes are all coordinated to “happen at the same time” from the perspective of any code outside the transaction.
[30] If you use alter when commute would suffice, the worst thing that might happen is performance degradation. But if you use commute when alter is required, you’ll introduce a subtle bug that’s difficult to detect with automated tests.

K vytvoření *atomu* se používá syntaxe podobná syntaxi použité k vytvoření *refu* (zdrojový kód 93). Hodnotu *atomu* lze získat pomocí *@/deref*. Hodnotu *atomu* lze aktualizovat pomocí *reset!* a *swap!*. *Reset!* nastaví novou hodnotu *atomu*, *swap!* použije funkci na hodnotu *atomu* (zdrojový kód 94). *Atomy* se neúčastní transakcí, a proto nevyžadují *dosync*.

```
1 (def mutable-atom (atom 24))
```

Zdrojový kód 93: Vytvoření *atomu*

```
1 (reset! mutable-atom 32)
2 => 32
3 (swap! mutable-atom inc)
4 => 33
5 (swap! mutable-atom * 2)
6 => 66
```

Zdrojový kód 94: *reset!* a *swap!*

Atomy interně používají metodu *compareAndSet()* z *java.util.concurrent.AtomicReference*, což znamená, že jsou velmi rychlé a neblokují vlákno. Tudíž nemůže dojít k deadlocku. Z tohoto důvodu musí *swap!* zkontrolovat, zda byl *atom* změněn jiným vláknem mezi voláním funkce aktualizace a pokusem o aktualizaci hodnoty. Pokud ano, *swap!* to zkusí znovu pro novou hodnotu atomu. Proto funkce, které *swap!* provádí, musí být bez vedlejších účinků (Butcher, 2014, s. 92).

Hodnotu atomu lze omezit pomocí validátoru. Zdrojový kód 95 vytvoří atom, jehož hodnota může být pouze sudé číslo. *#(...)* je zkratka pro *fn*, *%* označuje jediný argument anonymní funkce. Validátor je proveden před změnou hodnoty, takže pokud je *swap!* proveden vícekrát, i validátor je volán vícekrát. Proto by ani validátor neměl mít vedlejší účinky (Butcher, 2014, s. 92).

```
1 (def even (atom 0 :validator #(= 0 (mod % 2))))
2 (reset! even 5)
3 => Execution error (IllegalStateException) at user/eval154 (REPL:1).
4 => Invalid reference state
```

Zdrojový kód 95: Validátor

```
1 (def watcher-atom (atom 25))
2 (add-watch watcher-atom :print-update
3     #(println "Old value: " %3 ", new value: " %4))
4 => #object[clojure.lang.Atom 0x35636217 {:status :ready, :val 25}]
```

```
5 (swap! watcher-atom inc)
6 => Old value:  25 , new value:  26
7 => 26
```

Zdrojový kód 96: Watcher

Další možností práce s atomy je, že k nim můžete připojit pozorovatele (*angl. – watcher*). Tento pozorovatel je spuštěn po provedení každé změny, a pouze jednou za změnu, tudíž může mít vedlejší účinky. Pozorovatel je vázán na *atom* pomocí *add-watch* a má čtyři argumenty: klíč, který lze poté použít k odebrání pozorovatele, odkaz na atom, předchozí hodnotu a novou hodnotu. Příklad pozorovatele je uveden ve zdrojovém kódu 96. Zde *%3* a *%4* představují třetí a čtvrtý argument anonymní funkce, které odpovídají předchozí a nové hodnotě, jak je uvedeno výše (Butcher, 2014, s. 92–93).

4.3.3 Agenti

Agent je vytvořen stejným způsobem jako *ref* a *atom* a jeho hodnotu lze získat stejným způsobem pomocí *@/deref*. Hodnotu agenta lze aktualizovat pomocí *send*, který funguje jako *swap!*. Rozdíl je v tom, že send provádí funkci aktualizace asynchronně – odešle ji do fondu vláken. Jak je tedy vidět ve zdrojovém kódu 97, *send* nevrací novou hodnotu *agenta*, ale samotný objekt *agenta*. „*Pokud více vláken volají send souběžně, provedení funkcí předaných send je serializováno: provede se pouze jednou. To znamená, že nebudou znovu zkoušeny, a proto mohou obsahovat vedlejší účinky.*“[31] (Butcher, 2014, s. 98)

```
1 (def mutable-agent (agent 30))
2 (send mutable-agent * 2)
3 => #object[clojure.lang.Agent 0x2b56f5f8 {:status :ready, :val 60}]
4 @mutable-agent
5 60
```

Zdrojový kód 97: *Agent* a *send*

Aby bylo zajištěno, že aktualizace *agenta* je kompletní, lze použít *await*, který blokuje vlákno, dokud *agent* neobdrží novou hodnotu. *Await-for* blokuje vlákno s nastaveným časovým limitem.

[31] If multiple threads call send concurrently, execution of the functions passed to send is serialized: only one will execute at a time. This means that they will not be retried and can therefore contain side effects.

Stejně jako atom lze i agenta nakonfigurovat pomocí validátoru. Pokud výsledek funkce aktualizace neodpovídá funkci validátoru, bude vyvolána výjimka. Agent má nástroje pro zpracování chyb.

Agent má dva režimy reakce na chybu: :fail a :continue. Pokud není nakonfigurováno zpracování chyb, agent je v režimu :fail. Když je agent v tomto režimu, odeslání výjimky ho odešle do stavu výjimky. Na získání chyby, která způsobila přechod *agenta* do tohoto stavu, lze použít *agent-error*. Všechny nové akce budou odeslány do fronty, dokud nebude vyvolán *restart-agent*.

Pokud je nakonfigurováno zpracování chyb, agent bude v režimu *:continue*. Když dojde k chybě, zavolá se obslužná rutina chyb a agent pracuje dále, jako by se nic nestalo. Obslužná rutina chyb je nakonfigurována tak, jak je uvedeno ve zdrojovém kódu 98.

```
1  (defn err-handler [agent error]
2    (println "Error" (.getMessage error)))
3  (def handle-agent
4    (agent 0
5     :validator #(= 0 (mod % 2))
6     :error-handler err-handler))
7  (send handle-agent inc)
8  => Error Invalid reference state
9  => #object[clojure.lang.Agent 0x7c4fc2bf {:status :ready, :val 0}]
10 (send handle-agent + 2)
11 => #object[clojure.lang.Agent 0x7c4fc2bf {:status :ready, :val 0}]
12 @handle-agent
13 => 2
```

Zdrojový kód 98: Obslužná rutina chyb *agenta*

4.3.4 Sjednocený model aktualizace

Sjednocený model aktualizace (*angl. – The Unified Update Model*) je jedním z ústředních konceptů v Clojure, podle něhož *refy*, *atomy* a *agenti* udržují funkce, které aktualizují jejich stav aplikací funkce na předchozí stav. Miller (2018) shrnuje všechny tyto funkce do tabulky, která se zde označuje jako tabulka 3.

Tabulka 3 znázorňuje rozdíl mezi referenčními typy. Protože pouze *ref* funguje s transakcemi, ve kterých lze koordinovat aktualizace několika *refů*, má pouze *ref* komutativní funkci, která optimalizuje tuto koordinaci. Pouze agent pracuje asynchronně, takže jedině on má neblokující funkci.

Tabulka 3: Funkce referenčních typů

Mechanismus aktualizace	**Funkce *refu***	**Funkce *atomu***	**Funkce *agenta***
Aplikace funkcí	*alter*	*swap!*	*send-off*
Funkce (komutativní)	*commute*	N/A	N/A
Funkce (neblokující)	N/A	N/A	*send*
Jednoduchý setter	*ref-set*	*reset!*	N/A

Mezi funkcemi z tabulky 3 ještě nebyla popsána *send-off*. Na rozdíl od *send* neposílá funkci aktualizace do fondu vláken, ale vytváří nové vlákno. Používá se, když funkce aktualizace může vlákno zablokovat nebo její dokončení trvá dlouho (Butcher, 2014, s. 99).

4.4 Souhrn

Clojure je docela elegantní jazyk a jeho použití přináší řadu výhod jak při programování obecně, tak zejména při psaní paralelních a souběžných programů. Clojure byl původně navržen jako jazyk vhodný pro souběžnost. Ve výchozím nastavení jsou data immutable a paralelní model není vytvořen kolem zámků. Clojure navíc není čistě funkcionální jazyk a podporuje mutující datové typy ze světa imperativních jazyků. Jak již bylo řečeno, tyto mutující datové typy jsou vhodné pro souběžnost. Byly navrženy tak, aby bylo vhodné je používat v souběžných programech bez obav, že dojde k deadlocku.

Pro programátory zvyklé na práci s jazyky jako Java, JavaScript, C# může být nevýhodou neobvyklá syntaxe Clojure. Clojure umožňuje psát krátké a efektivní programy, ale vyžaduje to jiný způsob myšlení. Implementace Fibonacciho čísla, uvedená v této kapitole ve zdrojových kódech 69 až 79, trvá přibližně pět řádků kódu. Totéž lze napsat pomocí rekurze (jak se to obvykle děje v jazycích jako je Java), ale kvůli tomu bude kód těžkopádný, zbytečně složitý a obtížně čitelný (Miller, 2018).

Clojure má elegantní syntaxi. Je vhodný pro souběžné programování. Díky tomu, že běží na JVM, je rychlejším jazykem než mnoho populárních funkcionálních jazyků a má těsnou integraci s Javou (Miller, 2018).

5 PROSTŘEDÍ PARALELISMU V JAZYCÍCH VYUŽÍVAJÍCÍCH TECHNIKU CSP NA PŘÍKLADU GO

Koncept *Communicating Sequential Processes*, neboli *CSP*, navrhl Charles Antony Richard Hoare v roce 1978 ve svém článku „Communicating Sequential Processes“. V tomto článku popsal koncept programovacího jazyka zaměřeného na vývoj paralelních programů.

K popisu modelu souběžnosti CSP lze použít porovnání s modelem souběžnosti „vlákna a zámky“. V modelu „vlákna a zámky“ vlákna soutěží o data. Aby bylo zajištěno, že ke stejným datům může současně přistupovat pouze jedno vlákno, používají se různé metody synchronizace, například semafory a mutexy. Naproti tomu v modelu CSP jsou data předávána mezi vlákny (nebo jinými entitami, které pracují s daty souběžně) prostřednictvím „cest“.

Jazyk Go implementuje tento model. Hlavní souběžnou entitou jazyka je takzvaná *gorutina (angl. – goroutine)*. Cesty, kterými se přenášejí data mezi gorutiny, jsou *kanály (angl. – channels)*. Navzdory své orientaci na CSP jazyk podporuje také primitiva souběžnosti *(angl. – concurrency primitives)*, které jsou nejužitečnější pro synchronizaci přístupu k paměti na nízké úrovni, jako jsou například mutexy. Tato primitiva jsou obsažena v balíčku *sync*. Díky tomu si vývojář Go může vybrat mezi modely souběžnosti.

5.1 Základy souběžnosti v Go

5.1.1 Gorutiny

„Gorutiny jsou funkce, které běží souběžně s jinými gorutiny, včetně vstupního bodu vašeho programu.“[32] (Kennedy, 2016, s. 3)

Běhové prostředí Go mapuje vlákna operačního systému na takzvané logické procesory. Každý logický procesor je individuálně vázán na jedno vlákno operačního systému. Gorutiny jsou prováděny v těchto logických procesorech. Gorutiny jsou velmi lehké: dokonce i v jediném logickém procesoru mohou souběžně běžet stovky tisíc gorutin s velkou efektivitou a výkonem (Kennedy, 2016, s. 129).

Provádění gorutin je naplánováno běhovým prostředím Go, stejně jako provádění procesů je naplánováno plánovačem operačního systému. To je podrobněji popsáno v podkapitole 5.3.

[32] Goroutines are functions that run concurrently with other goroutines, including the entry point of your program.

Vstupním bodem do libovolného programu Go je funkce *main()*. Je také hlavní gorutinou programu. Při ukončení *main()* je ukončen samotný program a všechny ostatní gorutiny. Klíčové slovo *go* se používá k vytvoření novou gorutiny. Ve zdrojovém kódu 99 je anonymní funkce spuštěna jako nová gorutina. V Go je kód rozdělen na balíčky, proto je na začátku každého souboru uveden balíček, do kterého patří (jako tady na řádku 1). V následujících příkladech bude tento řádek vynechán. Řádek 2 importuje standardní balíček *fmt,* obsahující funkci *Println(),* použitou v řádku 6. Řádky 5 až 7 jsou definicí a tělem anonymní funkce. Na řádku 7 jsou závorky hned za tělem funkce. Toto je volání anonymní funkce. Anonymní funkce běží jako samostatná gorutina díky klíčovému slovu *go*.

```
1 package main
2 import "fmt"
3
4 func main(){
5   go func () {
6     fmt.Println("This is concurrent goroutine")
7   }()
8 }
```

Zdrojový kód 99: Spuštění anonymní funkce jako gorutiny

Počínaje řádkem 5 obsahuje program ze zdrojového kódu 98 dvě gorutiny: hlavní a tu, která tiskne řetězce. Poté, co je vytvořena nová gorutina, je zařazena do fronty k logickému procesoru, který ji vytvořil. Hlavní gorutina pokračuje ve své práci a okamžitě ji dokončí, a proto celý program končí. Druhá gorutina tedy s největší pravděpodobností ani nebude mít čas začít pracovat dříve, než program skončí.

Souběžnost v Go funguje ve vzoru *fork/join*. V bodě *fork* se program rozdělí na více než jednu gorutinu a v bodě *join* se gorutiny spojí dohromady. Bod *fork* ve zdroji 98 je na řádku 5, vytvořený klíčovým slovem *go*. Bod join ve zdroji 99 chybí. Proto není známo, zda bude druhá gorutina někdy spuštěna.

K vytvoření bodu *join* lze použít primitiva souběžnosti i kanály.

5.1.2 Kanály

„Kanály jsou datové struktury, které umožňují bezpečnou datovou komunikaci mezi gorutinami.“[33] (Kennedy, 2016, s. 4)

Rozsah použití kanálů v Go je ve skutečnosti ještě širší. S jejich pomocí lze plně navázat komunikaci mezi gorutiny. Nejen přenos dat mezi nimi, ale také jejich synchronizace a správu.

Zdrojový kód 100 je upravený kód 99, který přidává bod *join* pomocí kanálu. Řádky 3 a 4 jsou definice a inicializace proměnné typu *chan* s názvem *joinChan*. Typ *chan* odpovídá kanálu. Při definici a inicializaci kanálu je za typem uveden typ dat, který kanál vysílá. V tomto případě to je *interface{}*. Jedná se o jedinečný datový typ, který vyhovuje všem datovým typům (jako je třída Object v Javě). Inicializace kanálu se vždy provádí pomocí vestavěné funkce *make()*.

```
1  //...
2  func main(){
3    var joinChan chan interface{}
4    joinChan = make(chan interface{})
5
6    go func () {
7      fmt.Println("This is concurrent goroutine")
8      joinChan <- "Done"
9    }()
10
11   fmt.Println(<-joinChan)
12 }
```

Zdrojový kód 100: Kanál jako vysílač dat a bod *join*

Na řádku 8 se odesílají data na kanál pomocí operátoru <-. Operátor je napravo od názvu kanálu. Na řádku 11 jsou data přijímána z kanálu pomocí stejného operátora nalevo od názvu kanálu. Hlavní gorutina tentokrát nekončí bez čekání na druhou gorutinu. Na řádku 11 je hlavní gorutina blokována a čeká na data z *joinChan*. Zdrojový kód 99 bude mít následující výstup:

```
This is concurrent goroutine
Done
```

[33] Channels are data structures that enable safe data communication between goroutines.

V případě, že druhá gorutina nikdy neposílá data do kanálu, například pokud není v kódu řádek 8, prostředí Go uvidí, že jsou všechny gorutiny blokovány, a ukončí program zprávou o deadlocku. Výstup bude takový:

```
It's concurrent goroutine
fatal error: all goroutines are asleep - deadlock!
```

Aby k tomu nedocházelo, lze zavřít kanál ihned poté, co gorutina dokončí svou práci. Tento vzor je zastoupen ve zdrojovém kódu 101. Na řádku 3 příkaz *defer* naplánuje spuštění funkce *close()* po dokončení funkce, ze které je volána. Spustí se, i když volající funkce končí chybou. Funkce *close()* kanál uzavře. Na uzavřený kanál nelze psát, ale stále z něj lze číst. Takový kanál vrací data, která jsou pro daný typ považována za výchozí. Pro *int* to je *0*, pro *string* je prázdný řetězec, pro struktury to je *nil* a tak dále. Hlavní gorutina bude tedy blokována, dokud nebude číst něco (v tomto případě *nil*) z kanálu *joinChan*.

```
1  //...
2  go func () {
3    defer close(joinChan)
4    fmt.Println("This is concurrent goroutine")
5  }()
6
7  fmt.Println(<-joinChan)
8  //...
```

Zdrojový kód 101: Zavření kanálu

```
1 func main(){
2   joinChan := make(chan interface{})
3
4   go func () {
5     defer close(joinChan)
6     fmt.Println("This is concurrent goroutine")
7   }()
8   <-joinChan
9 }
```

Zdrojový kód 102: Kanál jako bod *join*

Zdrojový kód 100 bude mít následující výstup:

```
This is concurrent goroutine
<nil>
```

Ve skutečnosti je výše popsaný vzor běžným způsobem, jak vytvořit bod *join* pomocí kanálu. Je vytvořen kanál, který nepřenáší žádná data, ale pouze vytváří synchronizační bod,

jak je uvedeno ve zdrojovém kódu 102. Operátor *:=* umožňuje kombinovat definici a inicializaci do jednoho řádku.

Existuje také opačná možnost: když hlavní gorutina ovládá podřízené gorutiny pomocí kanálu. Zdrojový kód 103 synchronizuje 10 podřízených gorutin pomocí kanálu *runChan*. Každá z nich se blokuje při čekání na data z tohoto kanálu a odblokuje se, když hlavní gorutina kanál uzavře. Tento program může mít následující výstup:

```
Unblocking goroutines...
Goroutine 0 is running
Goroutine 4 is running
Goroutine 1 is running
Goroutine 3 is running
Goroutine 2 is running
```

```
1  //...
2  func main(){
3    runChan := make(chan interface{})
4    var wg sync.WaitGroup
5    wg.Add(5)
6    for i := 0; i < 5; i++ {
7      go func(id int) {
8        defer wg.Done()
9        <-runChan
10       fmt.Printf("Goroutine %v is running\n", id)
11     }(i)
12   }
13   fmt.Println("Unblocking goroutines...")
14   close(runChan)
15   wg.Wait()
16 }
```

Zdrojový kód 103: Odblokování více gorutin pomocí kanálu

Při čtení z kanálu jsou vráceny dvě hodnoty: *value* a *ok*. *ok* je typu *bool*. Pokud *ok* je *false*, tak kanál je uzavřen. Zdrojový kód 104 ukazuje, jak získat druhou návratovou hodnotu kanálu.

```
1 value, ok := <- chanel
```

Zdrojový kód 104: Čtení kanálu vrátí dvě hodnoty

Tato podkapitola dosud popisovala kanál bez bufferu *(angl. – unbuffered channel)*. Když se gorutina pokusí odeslat data přes takový kanál, zablokuje se, dokud jiná gorutina nepřijme data z tohoto kanálu. Naopak gorutina, která přijímá data, je blokována, dokud jiná gorutina nepřenáší data tímto kanálem.

Kanál s bufferem *(angl. – buffered channel)*, jak název napovídá, má k dispozici buffer. Vytvořen stejným způsobem jako kanál bez bufferu, ale s uvedením kapacity bufferu (zdrojový kód 105). Pokud chce gorutina poslat data přes takový kanál a jeho buffer není plný, gorutina přidá data do bufferu a není blokována. Pokud je buffer kanálu plný, je gorutina blokována. Gorutina, která se pokusí načíst data z takového kanálu (pokud buffer není prázdný), načte data a neblokuje. Pokud je buffer prázdný, gorutina je blokována.

```
1 bufferedChan := make(chan int, 10)
```

Zdrojový kód 105: Kanál s bufferem kapacitou 10

Kanály jsou také rozděleny na jednosměrné a obousměrné. Kanály popsané výše jsou obousměrné. Lze na ně zapisovat data a rovněž z nich data číst. Jednosměrné kanály mohou být „pouze pro čtení“ *(angl. – read-only)* (vytvoření je ve zdrojovém kódu 106), nebo „pouze pro zápis“ *(angl. – write-only)* (vytvoření je ve zdrojovém kódu 107).

```
1 readChan := make(<-chan int)
```

Zdrojový kód 106: Kanál pouze pro čtení

```
1 writeChan := make(chan<- int)
```

Zdrojový kód 107: Kanál pouze pro zápis

Zřídka se vytvářejí instance jednosměrných kanálů. Obvykle se používají jako parametry funkcí a návratové typy. To je možné, protože Go může implicitně převádět obousměrné kanály na jednosměrné kanály (Cox-Buday, 2017).

5.1.3 Balíček *sync*

Balíček *sync* obsahuje primitiva souběžnosti, která jsou nejužitečnější pro synchronizaci přístupu k paměti na nízké úrovni. Mezi nimi: *WaitGroup*, *Mutex*, *Once* a jiné. Tato primitiva byla přidána do Go, aby vývojářům poskytla více nástrojů pro práci. Nejčastěji se tyto operace používají v omezených oblastech kódu, například uvnitř struktur (Cox-Buday, 2017).

WaitGroup je další způsob, jak vytvořit bod *join* pro gorutiny. Typ synchronizace, který WaitGroup implementuje, se běžně označuje jako „bariéra“. Vzor tohoto je ukázán ve zdrojovém kódu 105. *WaitGroup* obsahuje uvnitř sebe čítač, jehož hodnota se zvyšuje funkcí *Add()*. Funkce *Done()* sníží hodnotu čítače o jednu. Funkce *Wait()* blokuje gorutinu, pokud

hodnota čítače není nula. Ve zdrojovém kódu 108 nejprve hlavní gorutina zvýší čítač o dva, poté vytvoří dvě nové gorutiny a zablokuje se. Až jedna z ostatních gorutin dokončí práci, je volána funkce *Done()*. Až obě gorutiny dokončí práci, hodnota čítače se změní na nulu a hlavní gorutina je odemčena.

```
1  //...
2  import "sync"
3
4  func main(){
5    var wg sync.WaitGroup
6    wg.Add(2)
7    go func () {
8      defer wg.Done()
9      fmt.Println("This is concurrent goroutine 1")
10   }()
11   go func () {
12     defer wg.Done()
13     fmt.Println("This is concurrent goroutine 2")
14   }()
15   wg.Wait()
16   fmt.Println("Done")
17 }
```

Zdrojový kód 108: Vytvoření bodu *join* pomocí *WaitGroup*

Mutex znamená „vzájemné vyloučení“ *(angl. – mutual exclusion)*. Používá se k ochraně kritické sekce. Kritická sekce je část kódu, ke které má přístup pouze jedna gorutina najednou. Tímto způsobem lze vytvořit atomické operace k aktualizaci sdílených dat.

Zdrojový kód 109 představuje program, který nesynchronizuje gorutiny, které pracují se sdílenými daty. Proměnná *count* je předána do funkce *inc()* odkazem, stejně jako v jazyce C. Proměnná *count* není inicializována, v takovém případě je automaticky přiřazena výchozí hodnota pro zadaný typ. Pro *int* je to *0*. Tady tisíc gorutin volá funkci *inc()* pro proměnnou *count*. Tento program by měl na konci vytisknout „1000“, ale po několika spuštěních dal následující výstupy:

```
989
995
993
998
```

To je způsobeno skutečností, že operace ++, kterou volá funkce *inc()*, není atomická, skládá se z několika příkazů. Při přepínání mezi gorutinami může tento příkaz přerušit, což vede k nepředvídatelným výsledkům.

Zdrojový kód 110 je stejný program jako zdrojový kód 109, ale používá *Mutex*. *Mutex* nastaví zámek na začátku funkce *inc()* (funkce *Lock()*) a odemkne ho, když *inc()* skončí (funkce *Unlock()*). Mezi funkcemi *Lock()* a *Unlock()* je chráněná kritická sekce. Kód v této sekci nemůže být přerušen. Výstup tohoto programu tedy bude vždy:

```
1000
```

```
//...
func inc(num *int){
  *num++
}

func main(){
  var wg sync.WaitGroup
  var count int

  wg.Add(1000)
  for i := 0; i < 1000; i++ {
    go func () {
      defer wg.Done()
      inc(&count)
    }()
  }
  wg.Wait()
  fmt.Println(count)
}
```

Zdrojový kód 109: Aktualizace sdílené proměnné s mnoha gorutinámi bez použití kritické sekce

Mutex umožňuje psát své vlastní atomické funkce. Nelze však nezmínit balíček sync/atomic, který obsahuje hotové atomické funkce.

```
//...
var mutex sync.Mutex

func inc(num *int){
  mutex.Lock()
  defer mutex.Unlock()
  *num++
}

func main(){
  var wg sync.WaitGroup
  var count int

  wg.Add(1000)
  for i := 0; i < 1000; i++ {
```

```
16      go func () {
17        defer wg.Done()
18        inc(&count)
19      }()
20    }
21    wg.Wait()
22    fmt.Println(count)
23  }
```

Zdrojový kód 110: Aktualizace sdílené proměnné s mnoha gorutinámi s použitím kritické sekce

```
1   //...
2   func inc(num *int){
3     *num++
4   }
5
6   func main(){
7     var wg sync.WaitGroup
8     var count int
9     var once sync.Once
10    countIncrement := func () {
11      inc(&count)
12    }
13
14    wg.Add(1000)
15    for i := 0; i < 1000; i++ {
16      go func () {
17        defer wg.Done()
18         once.Do(countIncrement)
19      }()
20    }
21    wg.Wait()
22    fmt.Println(count)
23  }
```

Zdrojový kód 111: *Once*

Funkce *Do()* objektu *Once* bude volána pouze jednou. Ve zdrojovém kódu 111 je funkce *countIncrement()* volána pomocí funkce *Do()*. Z tohoto důvodu, i když je tato funkce volána tisícem gorutin, je volána pouze jednou a výstup je:

```
1
```

I když se zdá divné, že je možné volat funkci jen jednou, může to být užitečné. I ve standardním balíčku Go se tento primitiv objeví 70krát (Cox-Buday, 2017).

5.2 Techniky souběžného programování v Go

5.2.1 Skrytá nebezpečí při používání kanálů a jak se jim vyhnout

Výchozí hodnota kanálu (hodnota, která je nastavena, pokud kanál není inicializován) je *nil*. Pokus o provedení operací zápisu nebo čtení na takovém kanálu gorutinu zablokuje. Pokus o uzavření kanálu vede k panice (stručně řečeno, panika v Go je obdobou výjimky v Javě). Při pokusu o čtení z kanálu pouze pro zápis nebo o zápis do kanálu pouze pro čtení dojde k chybě kompilace. Aby nedošlo ke zmatení, Cox-Buday (2017) poskytuje tabulku, která popisuje výsledek použití různých operací na kanály v různých stavech (tabulka 4). Poznámka: kanál bez bufferu má kapacitu nula, a proto je považován za plný až do prvního zápisu.

Tabulka 4: Výsledek operací kanálu vzhledem ke stavu kanálu

Operace	**Stav kanálu**	**Výsledek**
Čtení	*nil*	Blokování
	Otevřený a není prázdný	Hodnota
	Otevřený a prázdný	Blokování
	Zavřený	*<výchozí hodnota>, false*
	Pouze pro zápis	Chyba kompilace
Zápis	*nil*	Blokování
	Otevřený a plný	Blokování
	Otevřený a není plný	Zápis hodnoty
	Zavřený	*Panika*
	Pouze pro čtení	Chyba kompilace
Uzavření	*nil*	*Panika*
	Otevřený a není prázdný	Zavře kanál; čtení proběhne úspěšně, dokud není kanál vyčerpán, pak čtení vytvoří výchozí hodnotu
	Otevřený a prázdný	Zavře kanál; čtení vytvoří výchozí hodnotu
	Zavřený	*Panika*
	Pouze pro čtení	Chyba kompilace

Aby nedošlo k deadlocku a panice, je nutné oddělit operace mezi gorutinami. Jedna z gorutin je považována za *vlastníka* kanálu. Vlastník vytvoří, zapisuje a zavírá kanál. Zbytek gorutin lze nazvat *uživateli* kanálu. Uživatel má přístup pouze ke čtení kanálu (Cox-Buday, 2017).

Zdrojový kód 112 demonstruje použití takového principu. Návratová hodnota funkce je uvedena za seznamem argumentů, jak je vidět na řádku 2. Funkce *createChan()* vrací kanál pouze pro čtení. Volající tedy nebude moci na tento kanál zapisovat ani jej zavřít. Funkce *createChan()* zapouzdřuje celý životní cyklus kanálu: inicializuje (řádek 3), zapisuje data (řádek 7) a zavírá je (řádek 5). Proto existuje jistota, že na žádném jiném místě v programu nebude kanál, který není inicializován, uzavřen.

Řádky 15–16 zdroje 112 obsahují speciální formu smyčky. Tato smyčka je speciálně navržena pro čtení z kanálu. Program ukončí takovou smyčku, když je uzavřen kanál, ze kterého se čte.

```
1  //...
2  func createChan() <-chan int {
3    resultChan := make(chan int, 5)
4    go func() {
5      defer close(resultChan)
6      for i := 0; i <= 5; i++ {
7        resultChan <- i
8      }
9    }()
10   return resultChan
11 }
12
13 func main(){
14   resultChan := createChan()
15   for result := range resultChan {
16     fmt.Printf("Received: %d\n", result)
17   }
18   fmt.Println("Done receiving!")
19 }
```

Zdrojový kód 112: Vlastnictví kanálu

Program ze zdrojového kódu 112 bude mít následující výstup:

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Done receiving!
```

„Pokud je gorutina odpovědná za vytvoření gorutiny, je také zodpovědná za zajištění, že může gorutinu zastavit.“[34] (Cox-Buday, 2017)

5.2.2 Příkaz *select*

Podle Cox-Buday (2017) jsou příkazy select spolu se souběžností jednou z nejdůležitějších věcí v programu Go. Příkazy *select* spojují dohromady kanály lokálně, v rámci jedné funkce nebo typu a také globálně, na křižovatce dvou nebo více komponent v systému. Kromě toho příkazy *select* mohou pomoci bezpečně spojit kanály s koncepty, jako jsou zrušení *(angl. – cancellations)*, časové limity *(angl. – timeouts)*, čekání *(angl. – waiting)* a výchozí hodnoty *(angl. – default values)*.

Zdrojový kód 113 demonstruje strukturu příkazu *select*. Vypadá v podstatě stejně jako blok *switch-case*. Příkaz *select* kontroluje, zda je některý z kanálů připraven ke čtení/zápisu. Pokud žádný z nich není připraven, je blokován celý blok *select*. Jakmile je jeden z kanálů připraven, provede se odpovídající kód a blok *select* dokončí svou práci. Pokud je připraveno několik kanálů současně, příkaz *select* z nich vybírá pseudonáhodně se stejnou šancí pro každý kanál. Důvodem je, že Go nezná účel konkrétního příkazu *select* v kódu, ale v průměru vybere správný kanál (Cox-Buday, 2017).

```
1 var chan1, chan2 <-chan interface{}
2 var chan3 chan<- interface{}
3 select {
4   case <- chan1:
5   case <- chan2:
6   case chan3 <- struct{}{}:
7 }
```

Zdrojový kód 113: Příkaz *select*

Jedním z nejběžnějších vzorů souběžného programování v Go, který používá příkaz *select*, je takzvaná *smyčka for-select (angl. – for-select Loop)*. Implementuje se umístěním příkazu *select* do smyčky. Používá se, když je nutné odeslat iterovatelná data kanálem (zdrojový kód 114) nebo v nekonečných smyčkách, které čekají na signál k dokončení (zdrojový kód 115).

Zdrojový kód 114 používá smyčku pro kolekce. Je to podobné jako smyčka pro kanály, ale výrazu *range* předchází proměnná, která je zodpovědná za číslo iterace. Protože se tato

[34] If a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine.

proměnná v tomto případě nepoužívá, je nahrazena podtržítkem (kompilátor Go neumožňuje vytváření nepoužívaných proměnných).

Zdrojové kódy 114 a 115 používají kanál *done* jako signál k dokončení. Když se zavře, smyčka se přeruší.

```
1 for _, i := range []int{1, 2, 3} {
2   select {
3     case <-done:
4       return
5     case intChan <- i:
6   }
7 }
```

Zdrojový kód 114: Vzor s*myčka for-select*. První varianta

```
1 for {
2   select {
3     case <-done:
4       return
5     default:
6   }
7   // Do something
8 }
```

Zdrojový kód 115: Vzor *smyčka for-select*. Druhá varianta

Smyčka for-select se používá k zabránění úniku gorutin. I když jsou gorutiny levné a snadno se vytvářejí, zabírají prostředky a běhové prostředí Go neshromažďuje nepoužívané gorutiny.

Zdrojový kód 116 ukazuje příklad úniku gorutiny. Funkce *doSomeWork()* je předán *nil*, což způsobí zablokování gorutiny při pokusu o čtení z kanálu *integers*. Ve velkých programech může trvat poměrně dlouho, než hlavní gorutina dosáhne bodu *join* (v tomto případě kanálu *terminated*). Gorutina tedy může zabírat zdroje po celou dobu životního cyklu programu.

```
1  //...
2  func doSomeWork(integers <-chan int) <-chan interface{} {
3    terminated := make(chan interface{})
4    go func() {
5      defer fmt.Println("doSomeWork exited.")
6      defer close(terminated)
7      for i := range integers {
8        fmt.Println(i)
9      }
```

```
10   }()
11   return terminated
12 }
13
14 func main(){
15   terminated := doSomeWork(nil)
16   // Do somthing
17   <-terminated
18   fmt.Println("Done.")
19 }
```

Zdrojový kód 116: Únik gorutiny

Proto nadřazené gorutiny vyžadují mechanismus k dokončení podřízených gorutin, které již nejsou potřeba, ale nedokončily svou práci, protože byly například blokovány.

Pro tyto účely je zaveden signál, který umožňuje nadřazené gorutině dokončit podřízené gorutiny. Podle konvence je tento signál kanálem pouze pro čtení s názvem *done* (jako ve zdrojových kódech 114 a 115).

Zdrojový kód 117 vylepšuje kód 116 pomocí kanálu *done*. Kanál *done* je nyní předán jako první parametr *doSomeWork()*. Čtení z tohoto kanálu je běžně zabudováno do vzoru *smyčka for-select*. Když je kanál *done* uzavřen, gorutina se skončí. Tímto způsobem má nadřazená gorutina kontrolu nad podřízenou gorutinou.

```
1  //...
2  func doSomeWork(
3    done <-chan interface{},
4    integers <-chan int,
5    ) <-chan interface{} {
6      terminated := make(chan interface{})
7      go func() {
8        defer close(terminated)
9        for {
10         select {
11           case i := <-integers:
12             fmt.Println(i)
13           case <-done:
14             return
15         }
16       }
17     }()
18     return terminated
19 }
20
21 func main(){
```

```
22   done := make(chan interface{})
23   terminated := doSomeWork(done, nil)
24   go func() {
25     close(done)
26   }()
27   <-terminated
28   fmt.Println("Done.")
29 }
```

Zdrojový kód 117: Použití kanálu *done* a vzoru *Smyčka for-select*

5.2.3 Pipeline

Pipeline v Go je řetězec funkcí, kterými proudí data. Každá taková funkce se nazývá fáze pipelinu. Fáze pipelinu nutně přijímá a vrací data stejného typu. Použití kanálů při stavbě pipelinu umožňuje souběžnou práci různých fází. Ve struktuře a chování se pipeliny z Go podobají *Stream API* z Javy a funkcím kolekcí z Clojure.

```
1  func add(
2    done <-chan interface{},
3    intChan <-chan int,
4    additive int,
5    ) <-chan int {
6      addedChan := make(chan int)
7      go func() {
8        defer close(addedChan)
9        for i := range intChan {
10         select {
11           case <-done:
12             return
13           case addedChan <- i+additive:
14         }
15       }
16     }()
17     return addedChan
18 }
```

Zdrojový kód 118: Funkce *add()* jako fáze pipelinu

Zdrojový kód 118 ukazuje příklad funkce *add()*, která může být fází souběžného pipelinu. Přidává hodnotu *additive* ke každé hodnotě vrácené kanálem *intChan*. Všechny fáze daného pipelinu budou mít jako první dva argumenty kanál *done* a kanál typu *int*, zbytek argumentů se liší v závislosti na účelu funkce (řádky 2–4). Každá fáze tohoto pipelinu musí vrátit kanál typu *int* (řádek 5). Řádky 9–14 implementují vzor *smyčka for-select* popsaný v předchozí

podkapitole: data jsou zpracována a odeslána do další fáze, dokud není kanál *done* nebo kanál *intChan* uzavřen. Řádek 17 vrací kanál, který další fáze použije k příjmu dat.

Pro úplnější demonstraci pipelinů zdrojový kód 119 zavádí funkci *multiply()*. Násobí hodnotu *multiplier* každou hodnotou vrácenou kanálem *intChan*. Jinak funguje stejně jako funkce *add()*.

```
1  func multiply(
2    done <-chan interface{},
3    intChan <-chan int,
4    multiplier int,
5    ) <-chan int {
6      multipliedChan := make(chan int)
7      go func() {
8        defer close(multipliedChan)
9        for i := range intChan {
10         select {
11           case <-done:
12             return
13           case multipliedChan <- i*multiplier:
14         }
15       }
16     }()
17     return multipliedChan
18 }
```

Zdrojový kód 119: Funkce *multiply()* jako fáze pipelinu

Protože funkce fáze přijímá kanály jako argument, je potřeba funkci, která převádí data na kanál. K tomu se používají funkce generátoru. Taková funkce je uvedena ve zdrojovém kódu 120. Funkce generátoru přijímá kanál *done* a pole celých čísel jako argumenty. Vrátí kanál typu *int*. Vnitřní struktura je téměř stejná jako u funkcí fází, ale odesílá čísla do kanálu beze změny.

```
1  func generator(
2    done <-chan interface{},
3    integers ...int) <-chan int {
4      intChan := make(chan int)
5      go func() {
6        defer close(intChan)
7        for _, i := range integers {
8          select {
9            case <-done:
10             return
11           case intChan <- i:
```

```
12          }
13        }
14      }()
15      return intChan
16  }
```

Zdrojový kód 120: Funkce *generator()*

Nyní je vše připraveno k vybudování pipelinu. Zdrojový kód 121 shromažďuje všechny výše uvedené do pipelinu. Řádek 2 vytvoří kanál done, který se zavře, když program skončí. *intChan* je kanál, na který bude generátor posílat čísla (řádek 4). Řádek 5 vytváří samotný pipeline. Do tohoto kanálu se předávají čísla 1, 2, 3, 4, 5. Procházejí funkcí *multiply()*, která je vynásobí o 2, poté k nim *add()* přidá 2, pak je znovu *multiply()* vynásobí o 3. Smyčka na řádcích 6–8 vytiskne výsledná čísla:

```
12
18
24
30
36
```

```
1 func main(){
2   done := make(chan interface{})
3   defer close(done)
4   intChan := generator(done, 1, 2, 3, 4, 5)
5   pipeline := multiply(done, add(done, multiply(done, intChan, 2), 2), 3)
6   for v := range pipeline {
7     fmt.Println(v)
8   }
9 }
```

Zdrojový kód 121: Pipeline

Díky struktuře všech fází pipelinu, včetně generátoru, je každé číslo zpracováváno paralelně v různých rutinách. Jinými slovy, zatímco poslední *multiply()* zpracovává pátý prvek, první *multiply()* zpracovává třetí prvek. Protože každá fáze pipelinu je blokována jak při příjmu, tak při přenosu kanály.

Generátory se mohou lišit. Například zdrojový kód 122 zobrazuje generátor *infinite()*, který nekonečně generuje vzestupná celá čísla. Bylo by samozřejmě nerozumné jakýmkoli způsobem používat nekonečnou sekvenci. Proto je potřeba funkce, která ji omezuje. Například funkce *take()*, která přebírá prvních *n* čísel z kanálu (zdrojový kód 123). Vytvořením kanálu z těchto dvou funkcí generuje zdrojový kód 124 celá čísla od nuly do devíti.

```
1  func infinite(
2    done <-chan interface{},
3    ) <-chan interface{} {
4      intChan := make(chan interface{})
5      go func() {
6        defer close(intChan)
7        var i int
8        for {
9          select {
10           case <-done:
11             return
12           case intChan <- i:
13             i++
14         }
15       }
16     }()
17     return intChan
18 }
```

Zdrojový kód 122: Generátor *infinite()*

```
1  func take(
2    done <-chan interface{},
3    valueChan <-chan interface{},
4    n int,
5    ) <-chan interface{} {
6      takeChan := make(chan interface{})
7       go func() {
8        defer close(takeChan)
9        for i := 0; i < n; i++ {
10         select {
11           case <-done:
12             return
13           case takeChan <- <-valueChan:
14         }
15       }
16     }()
17     return takeChan
18 }
```

Zdrojový kód 123: Funkce *take()*

Pipeliny tedy implementují „lazy“ chování jako líné sekvence v Clojure nebo steamy v Javě. Kromě toho se výše uvedené funkce *infinite()* a *take()* chovají doslova stejně jako analogické funkce v Clojure a *Stream API*.

```
1 func main(){
2   done := make(chan interface{})
3   defer close(done)
4   for num := range take(done, infinite(done), 10) {
5     fmt.Printf("%v ", num)
6   }
7 }
```

Zdrojový kód 124: Generování prvních deseti celých čísel

Někdy se stává, že výpočty jedné z fází trvají příliš dlouho a následující fáze jsou kvůli tomu blokované a nečinné. Tuto fázi lze paralelizovat. Vzor, který slouží tomuto účelu, se nazývá *fan-out, fan-in*. „*Fan-out je termín, který popisuje proces spouštění více gorutin pro zpracování vstupu z pipelinu, a fan-in je termín, který popisuje proces kombinování více výsledků do jednoho kanálu*“[35] (Cox-Buday, 2017). Tento vzorec je samozřejmě použitelný pouze pro fáze, které se nespoléhají na dříve vypočítané hodnoty.

```
1  //...
2  import "time"
3  //...
4  func main(){
5    done := make(chan interface{})
6    defer close(done)
7    start := time.Now()
8    intChan := generator(done, 1, 2, 3, 4, 5)
9    pipeline := multiply(done, add(done, multiply(done, intChan, 2), 2), 3)
10   for v := range pipeline{
11     fmt.Println(v)
12   }
13   fmt.Printf("Time: %v", time.Since(start))
14 }
```

Zdrojový kód 125: Měření doby provedení

Pro demonstraci je vybrána funkce *add()*. Nyní před provedením operace jednu sekundu spí. Proto má zdrojový kód 125 následující výstup:

```
12
18
24
30
36
```

[35] Fan-out is a term to describe the process of starting multiple goroutines to handle input from the pipeline, and fan-in is a term to describe the process of combining multiple results into one channel.

```
Time: 5.0559331s
```

Balíček *time* umožňuje měřit čas provádění programu. Nyní je pipeline dokončeno za 5 sekund.

K implementaci *fan-out* stačí spustit několik kopií fáze *add()* (zdrojový kód 126). V tomto případě je spuštěno tolik kopií, kolik je v prostředí logických procesorů. *Runtime.NumCPU()* vrací počet logických procesorů (řádek 6). Proměnná *stageAdd* je pole kanálů typu *int* (řádek 7). Řádky 8–10 několikrát spustí fázi *add()*.

```
1  //...
2  import "runtime"
3  //...
4  intChan := generator(done, 1, 2, 3, 4, 5)
5  stageMultiplyOne := multiply(done, intChan, 2)
6  numCPUs := runtime.NumCPU()
7  stageAdd := make([]<-chan int, numCPUs)
8  for i := 0; i < numCPUs; i++{
9    stageAdd[i] = add(done, stageMultiplyOne, 2)
10 }
11 //...
```

Zdrojový kód 126: *Fan-out*

Nyní je nutné shromáždit výsledky z více kanálů *stageAdd* do jednoho kanálu. To provádí funkce fáze *fanIn()*, která přijímá pole kanálů a odesílá jejich výsledky do jednoho kanálu (zdrojový kód 127). Ačkoli funkce vypadá složité, je poměrně jednoduchá. Interní funkce *multiplex()* (řádky 8–17) přijímá kanál a předává z něj data do *multiplexedChan* (kombinujícího kanálu). Tato funkce je volána pro každý kanál v samostatných gorutinách (řádky 18–20). Další rutina čeká na dokončení zbývajících rutin a zavírá *multiplexedChan*.

```
1  func fanIn(
2    done <-chan interface{},
3    channels ...<-chan int,
4    ) <-chan int {
5      var wg sync.WaitGroup
6      wg.Add(len(channels))
7      multiplexedChan := make(chan int)
8      multiplex := func(c <-chan int) {
9        defer wg.Done()
10       for i := range c {
11         select {
12           case <-done:
13             return
14           case multiplexedChan <- i:
```

```
15          }
16        }
17      }
18      for _, c := range channels {
19        go multiplex(c)
20      }
21      go func() {
22        wg.Wait()
23        close(multiplexedChan)
24      }()
25      return multiplexedChan
26  }
```

Zdrojový kód 127: *Fan-in*

```
1   func main(){
2     done := make(chan interface{})
3     defer close(done)
4     start := time.Now()
5     intChan := generator(done, 1, 2, 3, 4, 5)
6     stageMultiplyOne := multiply(done, intChan, 2)
7     numCPUs := runtime.NumCPU()
8     stageAdd := make([]<-chan int, numCPUs)
9     for i := 0; i < numCPUs; i++{
10      stageAdd[i] = add(done, stageMultiplyOne, 2)
11    }
12    for v := range multiply(done, fanIn(done, stageAdd...), 3) {
13      fmt.Println(v)
14    }
15    fmt.Printf("Time: %v", time.Since(start))
16  }
```

Zdrojový kód 128: *Fan-out* a *fan-in*

Fan-out a *fan-in* jsou zobrazeny společně ve zdrojovém kódu 128. Tento kód může mít následující výstup (pořadí čísel je nedefinováno):

```
30
18
24
36
12
Time: 1.0202676s
```

Podle očekávání bylo každé číslo zpracováno samostatným logickým procesorem, a proto spuštění celého programu trvalo jen sekundu.

5.3 Plánování gorutin. Algoritmus „Work stealing“

Při plánování gorutin implementuje prostředí Go algoritmus „Work stealing“ popsaný dříve v sekci 2.2.3. Podle tohoto algoritmu si vlákna navzájem „kradou“ úkoly. Každý logický procesor v Go má své vlastní vlákno a oboustrannou frontu, nazývanou také „deque“ (Cox-Buday, 2017).

Podle Cox-Buday (2017) obsahuje implementace algoritmu „Work stealing“ v Go následující kroky:

1. v bodu *fork* přidat úkoly na konec deque přidruženého k vláknu;
2. pokud je vlákno nečinné, ukradnout práci z hlavy deque spojené s jiným náhodným vláknem;
3. v bodu *join*, který ještě nelze realizovat (tj. gorutina, se kterou je synchronizován, ještě nebyla dokončena), vzít práci z konce vlastního deque vlákna;
4. pokud je vlákno prázdné:
   a) buď zastavit a počkat na připojení;
   b) nebo ukrást práci z hlavy deque přidruženého k náhodnému vláknu.

Důležitým detailem tohoto algoritmu je, že vlákno vždy dává nové úkoly na konci své fronty. Dále bere úkol také z konce své fronty, ale krade úkoly z hlavy fronty jiného vlákna. Ve skutečnosti není správné říkat, že vlákno dává do fronty nebo bere z fronty *úkol*. Bylo by správnější nazývat to *pokračováním*.

Gorutina, která vytvořila novou gorutinu, se pravděpodobně brzy zastaví v bodu *join*, aby čekala na připojení k této nové gorutině. Tato nová gorutina je považována za *pokračování*, nikoli za *úkol*. Aby první gorutina (*úkol*) nebyla dlouho blokována, je logické dokončit práci druhé gorutiny (*pokračování*) co nejdříve. Vzhledem k tomu, že *pokračování* je na konci fronty, když vlákno zastaví *úkol*, může přeskočit rovnou na práci s *pokračováním*. Skutečnost, že další vlákna kradou z hlavy, snižuje pravděpodobnost, že *pokračování* bude ukradeno (Cox-Buday, 2017).

Krádež pokračování má následující výhody: omezená velikost fronty, postupné pořadí provádění, vlákno není v bodu *join* nečinné (Cox-Buday, 2017).

Kromě krádeží úkolů provádí prostředí Go spoustu dalších optimalizací. Například pokud gorutina provede blokovací systémové volání, jako je otevření souboru, vlákno a gorutina se odpojí od logického procesoru. Vlákno pokračuje v blokování a logický procesor v tuto chvíli zůstává bez vlákna. Plánovač poté vytvoří nové vlákno a připojí ho k logickému procesoru.

Když je gorutina odemčena, je zařazena do fronty a odpojené vlákno je odloženo pro budoucí použití (Kennedy, 2016, s. 131).

5.4 Souhrn

Nepochybnou výhodou Go jako jazyka pro vytváření souběžných aplikací, je jeho zaměření na souběžné programování. Prostředí Go provádí mnoho optimalizací, aby souběžné programování bylo co nejvýkonnější a nejefektivnější. Rutiny jsou velmi levné a snadno se vytvářejí. Go umožňuje spravovat je na poměrně nízké úrovni.

Zároveň je při plánování architektury nutné brát v úvahu možná rizika a vývojář musí být schopen se jim vyhnout (například pomocí konceptu vlastnictví kanálu). Kromě toho je žádoucí znát Go vzory pro efektivní programování.

Pipeliny v Go jsou mocným nástrojem pro implementaci i nekonečných líných sekvencí a funkcí pro práci s nimi. To však bude vyžadovat psaní velkého množství kódu (knihovny třetích stran se neuvažují), zatímco v jiných jazycích (Java *Stream API*, Clojure) jsou tyto sekvence a funkce uživateli nativně k dispozici.

ZÁVĚR

Pro účely této práce byly vybrány čtyři jazyky představující různá paradigmata programování. To jsou *Java* jako OOP jazyk, *JavaScript* jako skriptovací jazyk, *Clojure* jako funkcionální jazyk a *Go* v jazycích využívajících techniku CSP. Všechny byly posouzeny z hlediska technologií paralelismu, asynchronismu a souběhu, které poskytují uživateli.

Java *Stream API* bylo navrženo tak, aby usnadňovalo práci s kolekcemi. Pro *Stream API* bylo zjevně mnoho principů převzato z funkcionálního programování. *Stream API* má mnoho společného s dalším jazykem, o kterém pojednává tato práce, Clojure. Streamy nemění původní kolekci, v Clojure jsou kolekce ve výchozím nastavení neměnné. Jak streamy, tak kolekce v Clojure implementují „lazy“ chování. Jak streamy, tak kolekce v Clojure mohou být nekonečné. *Stream API* i Clojure používají k paralelizaci úloh framework Java *fork/join*. Rozhraní Java *Stream API* a jazyk Clojure poskytují uživateli snadno čitelnou syntaxi, která před uživatelem skrývá práci s vlákny, aby se mohl soustředit na přímé řešení svých problémů.

Jak již bylo zmíněno, kolekce v Clojure jsou ve výchozím nastavení neměnné. Neměnnost dat je způsob, jakým Clojure chrání data, když jsou k dispozici pro více vláken současně. Clojure však poskytuje také měnné datové typy. Protože byl Clojure vytvořen jako jazyk zaměřený na paralelismus, jsou tato data také chráněna před změnami jinými vlákny za běhu. Každý z těchto typů se používá v různých případech v závislosti na požadované úrovni ochrany. Atomické datové typy lze v jiných jazycích považovat za analogické.

Množina jazyků používá *future/promise* koncepce při práci s asynchronním voláním. Jedná se o kontejner, který v okamžiku vytvoření nemá hodnotu, ale obdrží ji, až se asynchronní volání dokončí. K implementaci tohoto konceptu Java používá třídu *CompletableFuture*, JavaScript používá objekty *Promise* a Clojure používá objekty *promise* i *future*.

Jedním z hlavních cílů *CompletableFuture* v Javě a *Promise* v JavaScriptu je vyhnout se blokování vlákna při čekání na návrat asynchronních volání. Za tímto účelem se funkce, které používají výsledek asynchronního volání, nevolají dále v kódu, ale jako funkce zpětného volání funkcí, jako je *thenAccept()* pro *CompletableFuture* nebo *then()* pro *Promise*.

V Clojure fungují *promise* a *future* trochu jinak. *Future* se používá ke spuštění kódu na jiném vlákně. Při pokusu o získání hodnoty vrácené tímto kódem se vlákno zablokuje, dokud se kód nevrátí. *Promise* je kontejner, kterému pak lze přiřadit hodnotu. *Promise* a *future* se často používají společně.

JavaScript věnuje vývoji asynchronních volání velkou pozornost. Po *Promise* se v JavaScriptu objevily *generátory*. JavaScript nyní poskytuje technologii, která je zvláštním případem *generátoru* s názvem *async/await*. *Async/await* výrazně usnadňuje práci s asynchronními voláními. Tato technologie umožňuje psát asynchronní kód, jako by byl sekvenční. Díky tomu je práce s asynchronními voláními jednoduchá a intuitivní.

Paralelismus v JavaScriptu je obtížněji realizovatelný. Protože je JavaScript jazyk s jedním vláknem, byly v něm paralelní technologie vyvinuty teprve nedávno. Relativně nová technologie *„Web Workers“* umožňuje JavaScriptu přístup k vláknům operačního systému z prohlížeče. Bohužel, pro paralelizaci úloh je potřeba implementovat algoritmus *fork/join* ručně.

Hlavní charakteristikou jazyka Go je technologie CSP. Hlavními prvky jazyka jsou gorutiny a kanály. Data jsou mezi gorutinami přenášena prostřednictvím kanálů. Běhové prostředí Go provádí množinu optimalizací, aby byl paralelismus co nejefektivnější. Syntaxe jazyka umožňuje doladit řízení souběžnosti a paralelismu v programu. Kanály a syntaxe jazyka umožňují implementovat pipeline, který napodobuje chování streamů ze *Stream API* nebo kolekcí z Clojure. To však vyžaduje hodně kódu, zatímco *Stream API* a Clojure tuto funkcionalitu poskytují nativně.

Z výše uvedeného se syntaxe funkcionálních jazyků dobře hodí k paralelismu. Svědčí o tom fakt, že *Stream API* používá syntaxi podobnou syntaxi funkcionálních jazyků a jeden ze vzorů jazyka Go (pipeline) umožňuje tuto syntaxi emulovat. *Async/await* je bezpochyby nejpohodlnější technologií pro asynchronní volání, protože kód napsaný s její pomocí vypadá jako sekvenční, to znamená, že je snadno pochopitelný a analyzovatelný. Samotný jazyk Go je zaměřen na paralelní zpracování, je rychlý a umožňuje přesnější ovládání paralelních procesů, které se v programu vyskytují.

Navzdory skutečnosti, že jazyky uvažované v této práci patří do různých paradigmat, obsahují implementace stejných konceptů (*promise*/*future*, pipeliny „lazy“ kolekcí). Technologie *async/await* se používá nejen ve skriptovacím JavaScriptu, ale v OOP jazyce C#. Víceméně unikátní technologií ve srovnání s ostatními zvažovanými jazyky je implementace mutable proměnných v Clojure a CSP v Go.

POUŽITÁ LITERATURA

BADEWA, Kayode, 2019. *Concurrency Won't solve Your CPU problem, but it can help with your IO Problem.* In: *Medium* [online]. Medium [cit. 2021-07-31]. Dostupné z: https://kayodeb.medium.com/concurrency-wont-solve-your-cpu-problem-but-it-can-help-with-your-io-problem-b100d3d53f72

BODUCH, Adam, 2015. *JavaScript Concurrency*. Vyd. 1. Birmingham, UK: Packt Publishing, 292 s. ISBN 978-1-78588-923-3.

BONÉR, Jonas, Dave FARLEY, Roland KUHN a Martin THOMPSON, © 2014. Reaktivní Manifest. *The Reactive Manifesto* [online]. [cit. 2021-05-16].

BUTCHER, Paul, 2014. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Vyd. 1. Dallas, Texas: Pragmatic Bookshelf, 300 s. ISBN 978-1937785659.

Clojure [online], ©2008-2021. Rich Hickey [cit. 2021-06-28]. Dostupné z: clojure.org

COX-BUDAY, Katherine, 2017. *Concurrency in Go: Tools and Techniques for Developers*. Vyd. 1. Sebastopol, CA: O'Reilly Media, 238 s. ISBN 9781491941195.

GOETZ, Brian, Tim PEIERLS, Joshua BLOCH, Joseph BOWBEER, David HOLMES a Doug LEA, 2006. *Java Concurrency in Practice*. Vyd. 1. Upper Saddle River, NJ: Addison-Wesley Professional, 424 s. ISBN 0-321-34960-1.

HAVERBEKE, Marijn, 2018. *Eloquent JavaScript: A Modern Introduction to Programming*. 3rd ed. US: No Starch Press, 472 s. ISBN 1593279507.

HORSTMANN, Cay S., 2014. *Java SE8 for the Really Impatient: A Short Course on the Basics*. Vyd. 1. Upper Saddle River, NJ: Addison-Wesley Professional, 238 s. ISBN 978-0-321-92776-7.

KENNEDY, William, Brian KETELSEN a Erik St. MARTIN, 2016. *Go in Action*. Vyd. 1. Shelter Island, NY: Manning Publications, 260 s. ISBN 9781617291784.

MCCOOL, Michael, Arch D. ROBISON a James REINDERS, 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Vyd. 1. Waltham, MA: Elsevier, 432 s. ISBN 978-0-12-415993-8.

MDN Web Docs: moz://a [online], © 2005-2021. San Francisco: Mozilla [cit. 2021-06-05]. Dostupné z: developer.mozilla.org

MILLER, Alex, Stuart HALLOWAY a Aaron BEDRA, 2018. *Programming Clojure*. 3rd ed. Dallas, Texas: The Pragmatic Bookshelf, 300 s. ISBN 1680502468.

ORACLE, ©1993, 2021. *Documentation* [online]. Austin, TX: Oracle [cit. 2021-05-05]. Dostupné z: https://docs.oracle.com/

SIMPSON, Kyle, 2015. *You Don't Know JS: Async & Performance*. Vyd. 1. Sebastopol, CA: O'Reilly Media, 296 s. ISBN 9781491904220.

URMA, Raoul-Gabriel, Mario FUSCO a Alan MYCROFT, 2019. *Modern Java in Action: Lambdas, streams, functional and reactive programming*. Vyd. 1. Shelter Island, NY: Manning Publications Co, 592 s. ISBN 9781617293566.