Komunikace mezi procesy a synchronizace procesů

Jozef Kováč

Bakalářská práce
2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jozef Kováč**
Osobní číslo:	**A17561**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Komunikace mezi procesy a synchronizace procesů**
Téma práce anglicky:	**InterProcess Communication and Process Synchronization**

Zásady pro vypracování

1. Nastudujte problematiku komunikace mezi procesy (IPC) a synchronizace procesů v operačním systému.
2. Vytvořte sadu ukázkových programů pro komunikaci procesů.
3. Vysvětlete problematiku souběhu (Race Conditions) a připravte sadu ukázkových programů včetně správného řešení.
4. Vypracujte sadu ukázkových programů včetně správného řešení problematiky uváznutí (Deadlock).
5. Vytvořené programy graficky vizualizujte, zdrojový kód srozumitelně okomentujte.

Rozsah bakalářské práce:
Rozsah příloh:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. DEITEL H. M., DEITEL P. J. & CHOFFNES D. R.: *Operating systems*. 3^{rd} ed., Pearson/Prentice Hall, 2004. ISBN 0131246968.
2. TANENBAUM A. S.: *Modern operating systems*. 4^{th} ed. Boston: Pearson, 2015. ISBN 0-13-359162-x.
3. SILBERSCHATZ A., GALVIN P. B. & GAGNE G.: *Operating system concepts*. 9^{th} ed. Hoboken, NJ: Wiley, 2013. ISBN 978-1-118-06333-0.
4. STALLINGS W.: *Operating Systems: Internals and Design Principles*. 8^{th} ed., Pearson Education Limited, 2014.
5. ARPACI-DUSSEAU, Remzi H. and Andrea C. ARPACI-DUSSEAU. *Operating Systems: Three Easy Pieces* [online]. Arpaci-Dusseau Books, 2018 [cit. 2019-11-06]. Dostupné z: http://www.ostep.org

Vedoucí bakalářské práce: **doc. Ing. Martin Sysel, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **28. listopadu 2019**
Termín odevzdání bakalářské práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 15.5.2020 Jozef Kováč, v. r.

ABSTRAKT

Bakalárska práca sa zaoberá problematikou komunikácie a synchronizácie procesov a implementáciou dostupných riešení v reálnych viacvláknových aplikáciách. Teoretická časť práce si kladie za cieľ objasniť základné pojmy skúmanej tematiky a ponúka náhľad do princípov fungovania bežne používaných mechanizmov. Praktické spracovanie je realizované pomocou série ukážkových programov, ktoré vizualizujú najčastejšie problémy a ich korektné riešenia na typových úlohách.

Klíčová slova: procesy, vlákna, komunikácia, synchronizácia, vizualizácia

ABSTRACT

The bachelor thesis is focused on the issue of communication and synchronisation of processes and implementation of available solutions in real multi-threaded applications. The theoretical part of thesis aims to clarify elementary concepts of subject matter and offers insight into principles of functionality of commonly used mechanisms. Practical part is implemented using a set of sample programs visualising the most common problems and their correct solutions on typical tasks.

Keywords: processes, threads, communication, synchronisation, visualisation

Ďakujem môjmu vedúcemu doc. Ing. Martinovi Syslovi Ph.D. za hodnotné rady, podnety, konzultácie, usmernenie a ústretovosť počas písania bakalárskej práce.

Ďalej ďakujem Ing. et Ing. Erikovi Královi Ph.D. za cenné informácie k implementácii praktickej časti práce.

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Počítače prešli od čias svojho vzniku ďalekú a tŕnitú cestu – a zďaleka sa nedá povedať, že by jej koniec bol na dosah. Tieto hnacie motory modernej, informačnej spoločnosti však už v minulosti viackrát narazili (a stále narážajú) na prekážky stojace v ceste nekonečnej honbe za vyšším výkonom. Kvalitnejšie materiály, vyššie takty, dômyselné dizajnové riešenia či optimalizácia a vývoj algoritmov – to všetko prispievalo a neustále prispieva k vytrvalému prekonávaniu hraníc a obchádzaniu fyzikálnych či iných bariér kladených na ďalší rozvoj počítačov. Spomedzi všetkých dômyselných mechanizmov ktoré umožnili stať sa počítačom tým, čím sú dnes zastáva špeciálne miesto paralelizmus.

Možnosť paralelného spracovania úloh a s tým súvisiace viacprocesorové systémy pri väčšine bežných použití súčasného sveta prinášajú potenciálne extrémne zrýchlenie a zefektívnenie výpočtov – Podobne ako u mnohých významných zlepšení, aj v tomto prípade je možné nájsť množstvo drobných problémov a špecifík súvisiacich s jeho využitím, pričom ich popis, kategorizácia a v neposlednom rade vizualizácia je cieľom tejto bakalárskej práce.

Teoretická časť práce si kladie za cieľ uviesť do pozornosti čitateľa pojmy a všeobecný prehľad z oblasti viacvláknového spracovania úloh. Poznatky v nej uvedené sú prevažne deskriptívneho a kategorizačného charakteru, pričom zvláštna pozornosť je venovaná problematike komunikácie medzi procesmi a problematike možných nežiadúcich dôsledkov špecifických pre paralelné spracovanie procesov, vrátane možností ich riešenia a prostriedkov k tomu bežne využívaných.

Praktická časť práce je následne zameraná na programové spracovanie typových úloh súvisiacich so skúmanou problematikou a vizualizáciu podkladovej logiky za pomoci grafického užívateľského rozhrania. Primárnou úlohou vytvorených aplikácií je slúžiť ako mnemotechnický prostriedok vedúci ku komplexnejšiemu pochopeniu mechanizmov a vzťahov medzi príčinou a dôsledkom v zobrazovaných situáciách prostredníctvom možnosti priamej interakcie s užívateľským rozhraním aplikácie.

I. TEORETICKÁ ČASŤ

1 PROBLEMATIKA

Moderný operačný systém je komplexný a sofistikovaný nástroj - aby však fungoval tak, ako po ňom požadujeme, nezaobíde sa bez mechanizmov zabezpečujúcich komunikáciu medzi procesmi či synchronizáciu akcií, ktoré vykonávajú. Aby sme boli schopní plne porozumieť tejto problematike, je nutné sa najskôr oboznámiť s niekoľkými základnými pojmami.

1.1 Procesy

Definícia a samotný koncept procesu sa môžu naprieč rôznymi zdrojmi výrazne líšiť, avšak najčastejšie sa proces chápe ako program v určitom štádiu svojho životného cyklu [1][2]. Každý proces má vyhradený vlastný kus pamäte pozostávajúci zo zásobníku, dátovej sekcie a (prípadne) haldy [2].

1.1.1 Process Control Block

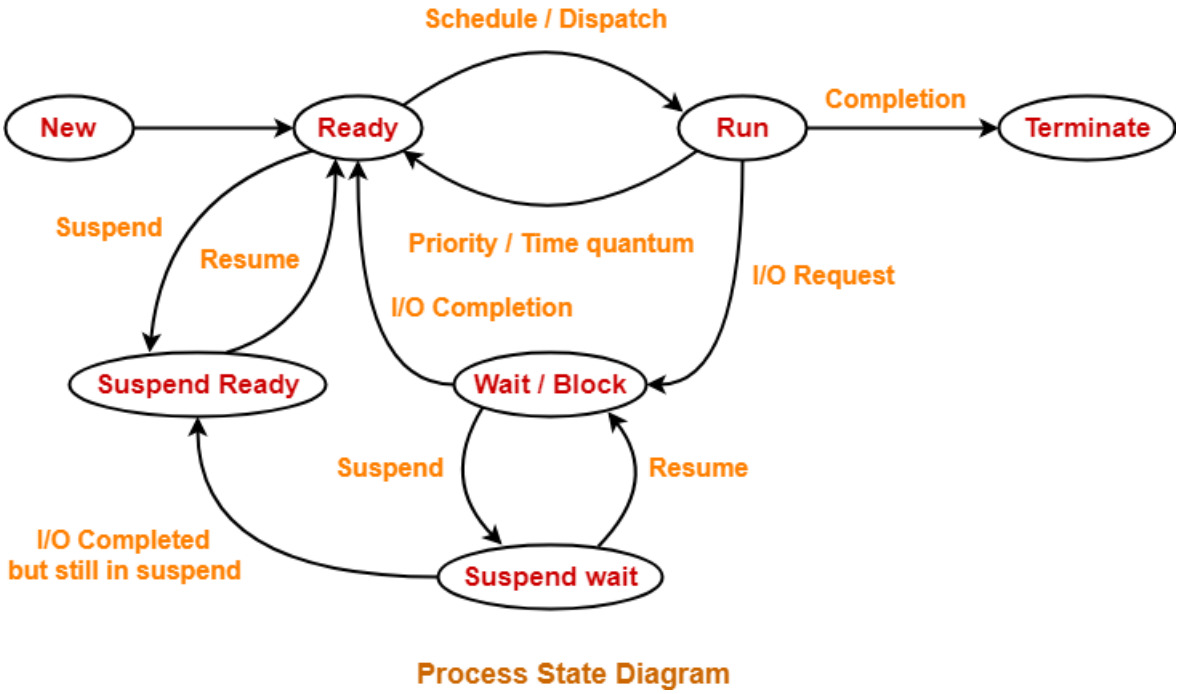
Každému procesu je pridelená jeho vlastná dátová štruktúra [1][2] - teda usporiadaná množina dát, ktorými je proces jednoznačne definovaný.

Táto dátová štruktúra sa nazýva kontrolným blokom procesu, alebo aj *Process Control Block* (PCB). PCB mimo iné obsahuje napríklad [3]:

- Unikátny identifikátor (*Process Identifier* - PID)
- Aktuálny stav v ktorom proces momentálne zotrváva
- Prioritu procesu (celočíselná hodnota)
- *Program Counter* (ukazovateľ do pamäte). Cieľová adresa závisí od konkrétnej architektúry procesoru – vo väčšine procesorov ukazuje na nasledujúcu inštrukciu ktorú by mal proces vykonávať [4]. V procesoroch, kde je navýšenie obsahu *Program Counter* registru vykonávané v inštrukčnom cykle skôr ako fáza *Fetch* teda ukazuje na práve vykonávanú inštrukciu.
- Procesorové registre pridelené procesu (akumulátory, index-registre..)
- *I/O* informácie (Údaje o požiadavkách vykonaných procesom na *I/O* operácie, priradené I/O zariadenia či zoznam procesom využívaných súborov)
- Účtovacie informácie (Po aký čas, resp. počet taktov bol procesu pridelený procesor, prípadné časové limity a pod.

1.1.2 Stavy procesov

Ďalšou charakteristickou vlastnosťou procesu je jeho stav. Stav procesu určuje, či je procesu práve pridelený procesor, čaká na nejakú udalosť, alebo naopak na pridelenie procesoru.



Obr. 1. Stavy procesu [5]

Pre bližšie informácie ohľadom prepojenia stavu procesu s problémami medziprocesovej synchronizácie sa odporúča nahliadnuť do kapitoly „Synchronizačné Problémy“.

1.2 Vlákna

Vlákna sú často zamieňané s procesmi (niekedy sa im dokonca hovorí “*lightweight process*”). Podobne ako procesy sú jednotkou symbolizujúcou práve prebiehajúci výkon programu. Z hľadiska nutnej réžie a použiteľnosti sú v skutočnosti medzi týmito dvomi výpočtovými konštruktmi pomerne výrazné rozdiely.

1.2.1 Vlákno vs. proces

Pri paralelizácii programov je efektívnejšie využívať väčšie množstvo vlákien v rámci jedného procesu ako viacero nezávislých procesov – oproti procesom vlákna ponúkajú niekoľko nezanedbateľných výhod [6].

Rozdiely a výhody použitia vlákien oproti procesom zachytáva nasledujúca tabuľka:

Tab. 1. Porovnanie rozdielov medzi procesmi a vláknami [7]

	PROCESY	**VLÁKNA**
Adresový priestor	Vlastný adresový priestor	Zdieľajú priestor pridelený rodičovskému procesu
Rýchlosť IPC	Pomalá (rozdielne pamäťové priestory procesov)	Rýchlejšia (zdieľajú pamäťový priestor)
Zmena kontextu	Náročnejšia	Jednoduchšia
Zdieľanie pamäte	Nezdieľajú pamäť s inými procesmi	Zdieľajú pamäť s ostatnými vláknami rodičovského procesu
Komplexita	Komplexné	Menej komplexné

Dôsledkom rozdelenia programu do viacerých výpočtových vlákien môžeme dosiahnuť značný nárast efektivity – jednotlivé vlákna môžu byť spracovávané nezávisle na sebe, nezriedka na rôznych procesoroch či jadrách. Schopnosť vykonávať viacero vlákien súbežne ale musí byť podporovaná operačným systémom.

1.3 Paralelizácia

S postupným zvyšovaním počtu jadier a vlákien na procesoroch sa zvyšujú aj nároky na mieru paralelizmu v samotných počítačových programoch, čo so sebou nutne nesie významné výhody, ale aj určité riziká.

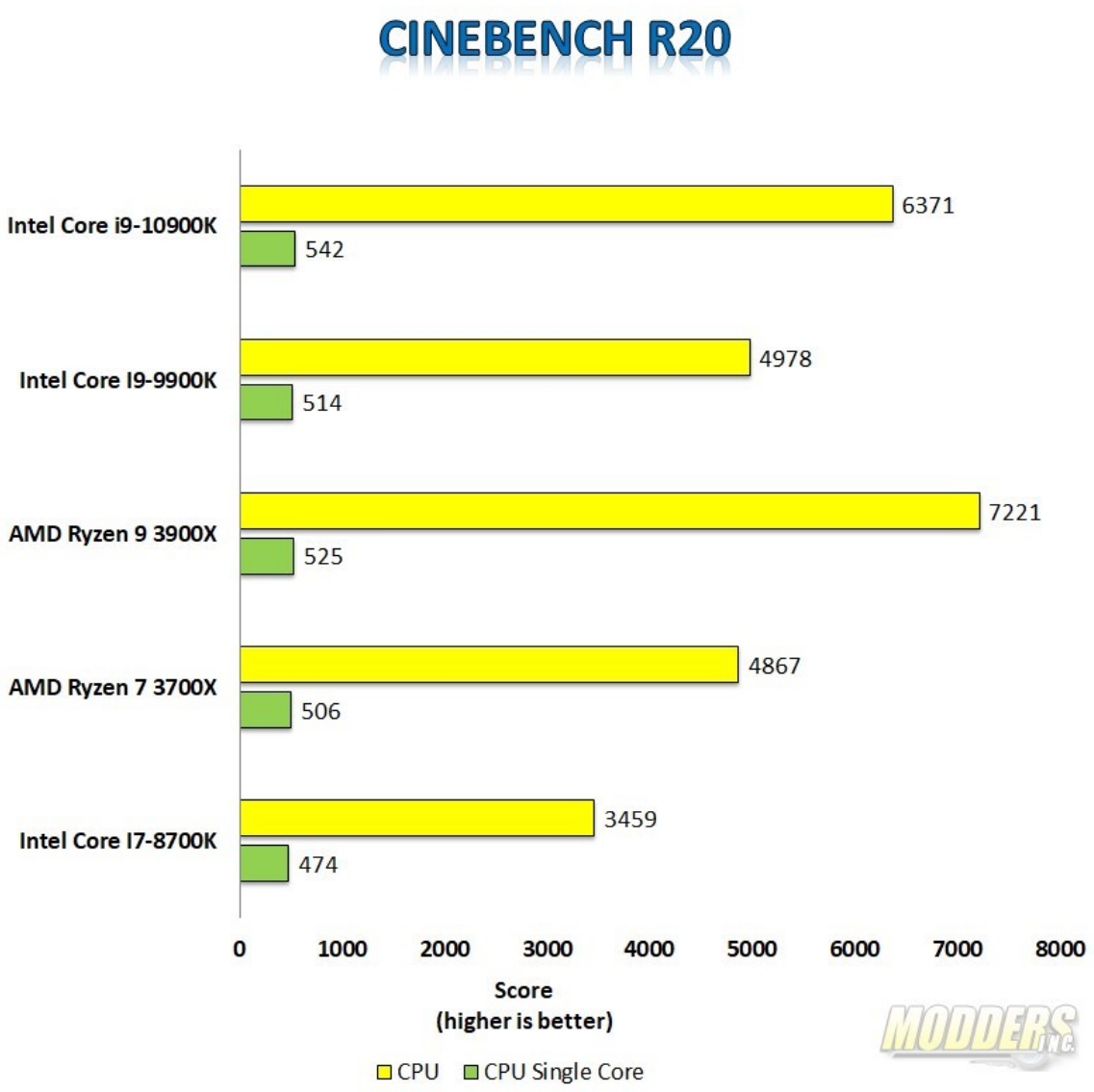
1.3.1 Multiprogramovanie a multitasking

Multiprogramovanie je spôsob, ako optimalizovať využitie procesoru a vstupno-výstupných zariadení. Operačný systém implementujúci multiprogramovanie rýchlo preraďuje procesor medzi viacerými procesmi, čím pre užívateľa vytvára ilúziu, že sú tieto procesy vykonávané súčasne [8].

Multitaskingom rozumieme skutočne súbežné vykonávanie viacerých procesov, pričom každému z nich je pridelený samostatný procesor či fyzické procesorové jadro. Využitie multitaskingu je teda pochopiteľne možné len na zariadeniach, ktoré oplývajú viac ako jedným procesorom [9].

1.3.2 Prínosy

Zvýšená miera paralelizmu pri použití na správnych miestach dokáže priniesť masívne zrýchlenie a zefektívnenie spracovania daného procesu. Pri dobre paralelizovateľných výpočtových problémoch (napríklad spracovanie obrazu) môže byť v závislosti na miere paralelizmu toto zrýchlenie až niekoľkonásobné [10].



Obr. 2. Porovnanie relatívneho výkonu moderných procesorov v teste „Cinebench R20“ pri využití jedného / všetkých dostupných jadier [11].

1.3.3 Úskalia použitia

Prínosy paralelne realizovaných programov sú nepopierateľné, avšak existuje aj množina problémov, ktorá s nimi úzko súvisí – prevažne sa jedná sa o problémy súvisiace s nutnosťou vymieňať informácie medzi existujúcimi procesmi, resp. zabraňovať nepriaznivým dôsledkom, ktoré môžu občas vzniknúť ako dôsledok súbežného prístupu viacerých procesov či vlákien k zdieľaným zdrojom (negatívne potenciálne dôsledky tejto situácie bližšie popisuje kapitola „Synchronizačné Problémy“). Týmito problémami sa zaoberá oblasť medziprocesovej komunikácie (*InterProcess Communication* - IPC) a synchronizácie procesov (*Process Synchronisation* – PS).

1.3.4 Amdahlov zákon

Miera prínosov plynúcich z paralelného spracovania má svoje obmedzenia. Vo väčšine prípadov platí, že nie je možné paralelizovať kompletne celý program, ale niektoré jeho časti je nutné vykonať sekvenčne – táto skutočnosť naznačuje existenciu koeficientu indikujúceho vzťah medzi rýchlosťou spracovania programu a počtom procesorov, na ktorých je spracúvaný (potenciálne až limitne do nekonečna).

Amdahlov zákon je vzorec použitý k určeniu teoretického maximálneho možného zrýchlenia programu dôsledkom využitia väčšieho množstva procesorov [12]. Pre aproximáciu celkového zrýchlenia výpočtu vplyvom paralelného spracovania uvažujeme vzorec:

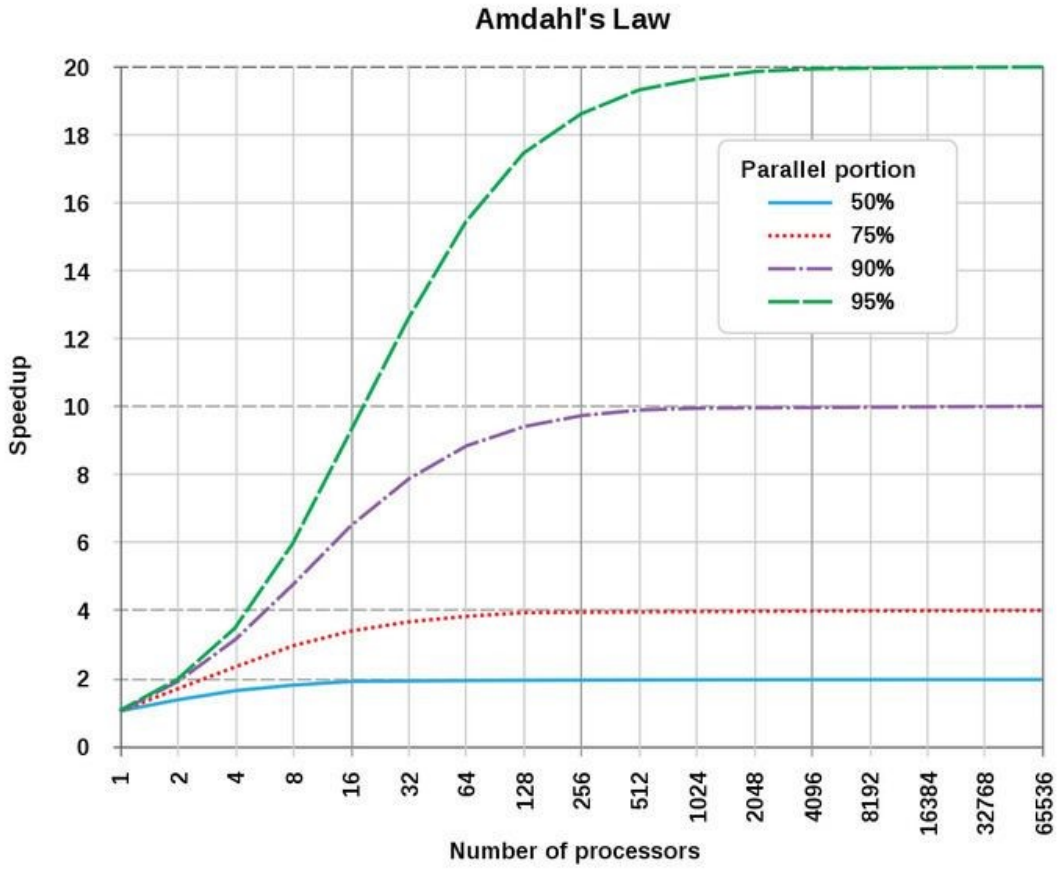
$$\boldsymbol{v} = \frac{\mathbf{1}}{(\mathbf{1} - \boldsymbol{P}) + \frac{\boldsymbol{P}}{\boldsymbol{S}}}$$

Pričom platí, že:

P = percentuálne vyjadrenie podielu výpočtového času, ktorý je možné zrýchliť

S = Maximálne možné zrýchlenie dosiahnuté zvýšením miery paralelizácie aplikovanej v zrýchliteľnej časti výpočtu (násobok)

Amdahlov zákon poskytuje pomôcku pri približnom určovaní benefitu zo zvýšenej miery paralelizácie algoritmov. Existujú však zdroje presvedčené o tom, že v dobe vzniku daného vzťahu neexistovali mnohé výrobné faktory a poznatky ovplyvňujúce výslednú efektivitu paralelizácie a preto sa v súčasnosti už môže jednať o ťažkopádnu až zastaralú metriku [13].



Obr. 3. Amdahlov zákon – vzťah medzi mierou zrýchlenia a mierou paralelizácie [14]

2 KOMUNIKÁCIA

2.1 Základné mechanizmy

Komunikácia medzi procesmi (*InterProcess Communication – IPC*) je jedným z elementárnych mechanizmov prevažne zastrešovaných operačným systémom. Motivácia pre implementáciu možnosti medziprocesovej komunikácie môže byť rôzna – často je dodaná možnosť kooperácie použitá ako prostriedok k prevencii opakovania znovupoužiteľného výpočtu iným (súvisiacim) procesom, k zvýšeniu výpočtovej efektivity či modularity jednotlivých zúčastnených procesov a k dôkladnejšej separácii práv a zodpovedností jednotlivých procesov [15]. Problematika medziprocesovej komunikácie zahŕňa široké spektrum samostatných a značne odlišných situácií – Do IPC spadá jednoduché zasielanie informácií medzi dvoma či viacerými procesmi bežiacimi v kontexte jedného zariadenia, rovnako ako sieťová komunikácia medzi veľkým množstvom rozdielnych počítačových systémov prepojených prostredníctvom siete či Internetu [16].

Základnými a najpoužívanejšími metódami pre realizáciu medziprocesovej komunikácie je využitie zdieľanej pamäte a tzv. zasielanie správ (*Message Passing*). Podpora jednotlivých mechanizmov závisí na OS, pričom niektoré OS podporujú oba zároveň a vhodne ich kombinujú [17].

2.1.1 Zdieľaná pamäť

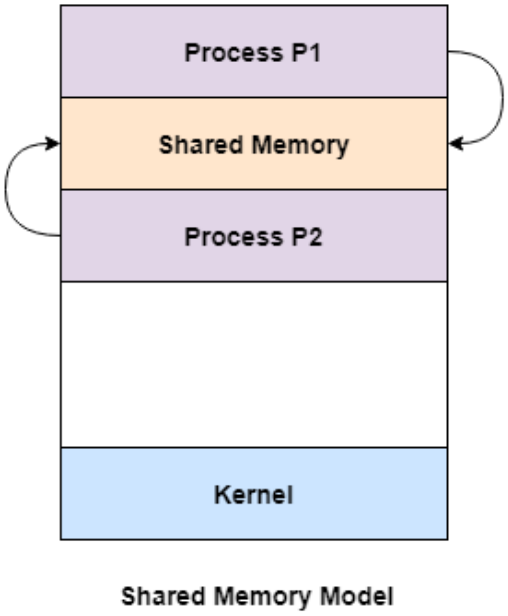
Zdieľanou pamäťou sa označuje taká pamäť, do ktorej má prístup viac procesov. Tieto procesy ju využívajú štandardne na komunikačné účely. Zdieľaná pamäť môže nadobúdať rôzne formy (premenná, objekt, súbor...).

Použiteľnosť medziprocesovej komunikácie výhradne za pomoci zdieľanej pamäte je však sporná, pričom by sa mali brať do úvahy klady aj zápory [2].

- **Výhody:**
    - Jednoduchá implementácia (spravidla zdieľané premenné chránené vhodnými synchronizačnými prostriedkami)
    - Výpočtová nenáročnosť (žiadna réžia zo strany OS)
    - Vysoká variabilita – forma zdieľanej pamäte sa môže meniť v závislosti na konkrétnom použití.

- **Nevýhody:**
    - Z bezpečnostného hľadiska nevhodná (globálne dáta sú prístupné viacerým procesom a citlivé informácie v nich uložené by mohli byť zneužité).
    - Neumožňuje sieťovú komunikáciu
    - Náchylná k synchronizačným problémom (napr. Producent-Konzument)

Hlavným úskalím komunikácie pomocou zdieľanej pamäte v rámci lokálneho prostredia je práve tendencia podliehať problémom spadajúcim do oblasti medziprocesovej synchronizácie (keďže zdieľaná pamäť je prístupná viacerým potenciálne súbežne vykonávaným procesom) [3]. V prípade využitia tejto komunikačnej metódy je teda nutné zabezpečiť dostatočné hardwarové či softwarové prostriedky ošetrujúce prístup k zdieľaným zdrojom (napríklad rôzne synchronizačné nástroje – Mutexy, Semafory, ktorým sa táto práca dôkladnejšie venuje v kapitole „Synchronizačné Prostriedky“).



Obr. 4. Medziprocesová komunikácia prostredníctvom zdieľanej pamäte [18]

2.1.2 Zasielanie správ

Zasielanie správ je oproti zdieľanej pamäti komplexnejšou metódou medziprocesovej komunikácie, ktorá sa principiálne zakladá na založení komunikačného spojenia a následného odosielania správ za využitia komunikačných primitív / funkcií (minimálne dve základné – *Send* a *Receive*) [2].

Elementárne primitíva pre zasielanie správ:

- **Send:** Slúži k odoslaniu správy adresátovi, najčastejšie tvary sú:
    - *Send* (správa, príjemca): Odoslanie správy jednému adresátovi
    - *Send* (správa): *Broadcast* všetkým naslúchajúcim adresátom či odoslanie správy do pred-definovanej destinácie
- **Receive:** Slúži k prijatiu správy, najčastejšie tvary sú:
    - *Receive* (správa, odosielateľ): Prijatie správy od určitého odosielateľa
    - *Receive* (správa): Prijatie správy, kde na odosielateľovi nezáleží či prijatie správy pomocou tzv. „poštovej schránky“

K predávaniu správ sa štandardne používa dátová štruktúra typu FIFO (fronta) [8], ktorá uľahčuje odstránenie synchronizačných neduhov a slúži k tomu, aby do nej boli pomocou funkcie *Send* správy vkladané procesom - odosielateľom a pomocou primitíva *Receive* vyberané a následne spracúvané procesom - príjemcom.

Samotná implementácia mechanizmu zaisťujúceho medziprocesovú komunikáciu prostredníctvom zasielania správ môže byť značne členitá (v závislosti na konkrétnom použití a type OS). Rozdiely v implementácii zasielania správ môžeme rozdeliť do niekoľkých základných kategórií [1][6][19][20]:

- **Spôsob doručenia správy**
    - *Umiestnenie správy do zdieľanej pamäte*: Odosielajúci proces umiestni správu priamo do fronty správ v zdieľanej pamäti, z ktorej si ju prijímajúci proces je schopný sám vyzdvihnúť.
    - *Umiestnenie správy do pamäte kernel-u (jadra OS):* Odosielajúci proces umiestni správu do špeciálnej dátovej štruktúry spravovanej operačným systémom – OS samotný potom zabezpečí prenos správy do pamäťového priestoru dostupného príjemcovi.

- **Počet procesov využívajúcich jedno spojenie**
    - *Spojenie je využívané najviac 2 procesmi*
    - *Spojenie smie byť využívané viac ako 2 procesmi*
- **Počet spojení medzi procesmi**
    - *Medzi dvoma procesmi existuje vždy najviac 1 spojenie*
    - *Medzi dvoma procesmi smie existovať viac ako 1 spojenie*
- **Kapacita spojenia**

  Udáva, koľko zatiaľ nespracovaných správ sa môže naraz nachádzať v dátovej štruktúre, do ktorej sa tieto správy v rámci komunikačného spojenia ukladajú. Najčastejšie používané kapacity sú:

    - *Nulová kapacita* – Pri nulovej kapacite spojenia nie je možné realizovať uloženie správy na neskoršie spracovanie. V tomto prípade je nutné aby proces odosielajúci správu buď čakal, kým si bude prijímateľ schopný správu vyzdvihnúť, alebo nečaká a správa sa stráca (nežiadúca situácia).
    - *Obmedzená kapacita* - Dátová štruktúra určená ku skladovaniu správ má určitý maximálny počet správ, ktoré dokáže udržiavať. V prípade že je plná, musí proces odosielajúci správu čakať až dokým sa potrebné miesto neuvoľní.
    - *Neobmedzená kapacita* – Dátová štruktúra určená ku skladovaniu správ nemá obmedzenie počtu správ, ktoré dokáže súčasne udržiavať. Neobmedzená kapacita zaručuje, že odosielajúci proces nikdy nie je nútený čakať (žiadúca situácia, je však nutné vyriešiť hrozbu prípadného pamäťového zahltenia a prebytočnej réžie spojenej s príliš frekventovaným odosielaním).
- **Veľkosť správy**
    - Pevná veľkosť – Každá správa má pevne určenú dĺžku. Tento prístup zjednodušuje implementáciu aplikácií využívajúcich komunikáciu medzi procesmi pomocou zasielania správ, avšak zvyšuje réžiu zo strany OS (kontrola dĺžky a zarovnávanie správ).

- Variabilná veľkosť – Jednotlivé správy nemusia mať konštantnú dĺžku. Prístup odľahčuje operačnému systému, avšak v prípade implementácie komunikácie za pomoci správ vo vlastnej aplikácii je treba ošetriť prípadné nechcené dôsledky vyplývajúce z rozličnej dĺžky správ.

- **Smerovosť spojenia**
  - *Jednosmerné komunikačné spojenia* – Komunikačné spojenie medzi procesmi P1 a P2 umožňuje posielanie správ od procesu P1 k procesu P2, nie však naopak.
  - *Obojsmerné komunikačné spojenia* – Komunikačné spojenie medzi procesmi P1 A P2 umožňuje posielanie správ od procesu P1 k procesu P2 a zároveň od procesu P2 k procesu P1 (funguje teda v tzv. duplexnom režime).

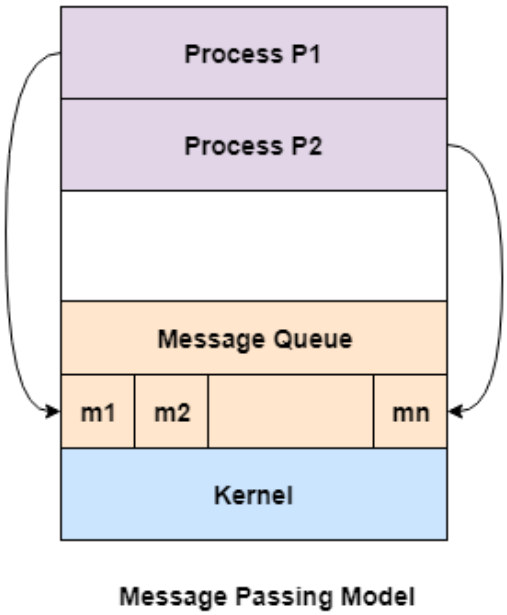
- **Synchrónnosť komunikácie**
  - *Synchrónna komunikácia:* Pri synchrónnej komunikácii odosielajúci proces čaká na odpoveď, resp. potvrdenie prijatia správy od procesu prijímajúceho (buď pomocou aktívneho čakania, alebo zablokovaním samého seba po odoslaní správy). Poskytuje lepšiu kontrolu nad komunikáciou za cenu potenciálnych strát procesorového času.
  - *Asynchrónna komunikácia:* Po odoslaní správy pokračuje proces-odosielateľ v činnosti bez prerušenia a nutnosti čakať na odpoveď príjemcu. Je rýchlejšia ako komunikácia synchrónna avšak náchylnejšia k chybovosti [viď. chyby pri zasielaní správ nižšie].

Súčasné operačné systémy implementujúce zasielanie správ využívajú neblokujúcu operáciu *Send()* v kombinácii s blokujúcou operáciou *Receive()* – tak sa zaručí minimálna časová strata zo strany odosielajúceho procesu a zároveň je zabezpečené „počkanie“ si na správu na strane procesu – príjemcu, keďže dáta obsiahnuté v správe sú v momente zavolania funkcie *Receive()* pravdepodobne kritické k pokračovaniu práce vykonávanej týmto procesom [20].

Popísané riešenie funguje v prípade, že odoslaná správa bude vždy doručená, čo pochopiteľne nie je vždy možné zaručiť – v prípade rizika náchylného na stratu správy je nutné zvážiť výhody nasledujúcich prístupov:

- Blokujúci *Send()* + Blokujúci *Receive()*
- Neblokujúci *Send()* + Neblokujúci *Receive()*

Chýb ku ktorým dochádza pri odosielaní správ existuje pomerne široké spektrum [21]. Patrí sem napr. duplikovanie správy (pri neodoslaní či strate potvrdenia o príjme), strata správy, ukončenie niektorého z komunikujúcich procesov či zmena obsahu správy (napr. chybou prenosu).



Obr. 5. Medziprocesová komunikácia prostredníctvom zasielania správ [22]

2.1.3 Rúry

Rúry (Pipes) sú jedným z najstarších komunikačných mechanizmov, do súčasnosti hojne využívané najmä v systémoch odvodených od OS UNIX, keďže možnosti rúr na OS Windows sú značne limitované [23]. Rozlišujeme dva základné typy rúr – pomenované a nepomenované.

- **Nepomenované rúry**

  Nepomenované rúry nemajú (na rozdiel od pomenovaných) silné prepojenie na súborový systém. Vznikajú ako dôsledok systémového volania (zvyčajne pomocou príkazu „*pipe*“ [24]). Operačný Systém zabezpečuje potrebnú réžiu vo forme priradenia tzv. *file descriptor*-u, ktorý umožňuje čítanie / zápis do rúry. Nepomenované rúry sú vždy jednosmerné – jeden koniec rúry je vždy otvorený pre čítanie a druhý pre zápis. Nepomenovaná rúra efektívne zaniká spolu so zánikom niektorého z procesov, ktoré ju využívali [25].

- **Pomenované rúry**

  Implementácia pomenovaných rúr je analogická k implementácii štandardných súborov v kontexte súborového systému [26] v tom zmysle, že jednotlivé procesy realizujú nad rúrou zápisy a čítania. Názov tohto typu rúry je použitý ako referencia na prúd (*stream*) bajtov predstavujúcich daný špeciálny súbor. Pomenovaná rúra má teoretickú životnosť dlhšiu ako proces ktorým bola vytvorená (resp. dokáže existovať aj po zániku rodičovského procesu ako každý iný konvenčný súbor [25]).

  Pomenované rúry môžeme nazývať aj FIFO (*First In, First Out*), podľa spôsobu akým sú do špeciálnych súborov jednotlivé bajty zapisované / čítané.

2.2 Architektúra klient-server

Komunikácia medzi procesmi neprebieha vždy len v rámci jedného zariadenia – vzhľadom na stále vyššie využitie počítačových sietí a rozšírenie Internetu rástla nutnosť zabezpečiť prostriedky pokrývajúce komunikačné potreby procesov bežiacich na rôznych systémoch, ktoré sú prostredníctvom týchto sietí prepojené.

2.2.1 Socket

Socket je najrozšírenejším prostriedkom pre zabezpečenie sieťovej komunikácie.

Komunikácia s využitím socketov je realizovaná zvyčajne podobne ako komunikácia prostredníctvom rúry, pričom socket samotný reprezentuje jeden z koncov rúry. Aby došlo k dátovému prenosu, musí dôjsť k spárovaniu dvoch alebo viacerých socketov [27] .

Samotná definícia pojmu socket a jeho implementácia sa môžu v rôznom kontexte významne líšiť – rozlišujeme veľké množstvo druhov socketov, pričom na operačných systémoch odvodených od OS UNIX ich rozdeľujeme do dvoch hlavných kategórií [27]:

- **Doména UNIX:** Sockety spadajúce do domény UNIX sa využívajú ku komunikácii procesov v rámci jedného zariadenia.
- **Doména INET:** Sockety spadajúce do tejto domény slúžia ku komunikácii procesov nachádzajúcich sa na rôznych fyzických zariadeniach.

3 SYNCHRONIZÁCIA

3.1 Kritická oblasť

Kritická oblasť je súhrnným pomenovaním pre množinu zdieľaných dát (premenné, objekty, súbory...), ku ktorým môže súčasne pristupovať viac ako jeden proces [28]. Manipuláciou kritickej oblasti viacerými procesmi zároveň dochádza k súbehu a jeho potenciálnym nežiadúcim následkom (pre viac informácií sa odporúča náhľad do kapitoly „Súbeh“). Pojem „Kritická Oblasť“ je často nesprávne zamieňaný s pojmom „Kritická sekcia“. Súvislosť medzi kritickou sekciou a kritickou oblasťou je popísaná v nasledujúcej podkapitole.

3.2 Kritická sekcia

Viacvláknové aplikácie s veľkou pravdepodobnosťou budú obsahovať aspoň jeden zdieľaný zdroj (kritickú oblasť), s ktorým smie pracovať viac vlákien zároveň. Takéto správanie so sebou nesie riziko súbehu (viď. kapitola „Súbeh“) a všetkých nepriaznivých následkov, ktoré sa s ním spájajú. Aby bolo možné k týmto problémom zmysluplne pristupovať a riešiť ich, je nutné definovať a kvantifikovať oblasť, v ktorej nastávajú - rovnako ako stanoviť podmienky či okolnosti, ktoré vedú k nežiadúcemu správaniu.

Kritická sekcia je súhrnným pomenovaním pre najmenšiu časť kódu, v ktorej proces (alebo vlákno) pristupuje ku zdieľaným prostriedkom (kritickej oblasti) [29].
Pokiaľ sa v kritickej sekcii nachádza v ľubovoľnom momente iba jeden proces, k žiadnemu súbehu (a z neho vyplývajúcim problémom) nedochádza.

Pokiaľ však smie do kritickej sekcie – a tým pádom k dátam kritickej oblasti – pristupovať viac procesov súbežne, dochádza k riziku nekorektného správania programu. Zmienený jav vzniká prevažne v dôsledku vykonávania sledu operácií nad zdieľanými dátami, ktoré by mali byť vykonané ako nedeľný celok, avšak v skutočnosti to tak nie je (jedná sa teda o neatomickú postupnosť príkazov). Riešeniu tohto typu problému sa bakalárska práca bližšie venuje v praktickej časti pri spracovaní príkladu „Populárny pekár“.

3.2.1 Problém kritickej sekcie

Problém kritickej sekcie je rozsiahly okruh zaoberajúci sa možnosťami synchronizácie procesov, ktoré ku zdieľaným dátam pristupujú prostredníctvom svojich kritických sekcií. Problematika kritickej sekcie sa zaoberá otázkou ako zaistiť, že kritickú sekciu bude zároveň

vykonávať vždy iba jeden proces zároveň [30] a ktoré prostriedky, či už softwarové alebo hardwarové, sú vhodné k zaisteniu tejto vzájomnej výlučnosti v konkrétnych situáciách.

Aby sa vyriešil problém kritickej sekcie, musí súčasne platiť niekoľko kľúčových podmienok. Konkrétna veľkosť podmienkovej množiny pritom nie je presne stanovená – v rôznych zdrojoch je možné nájsť rozdielne vyčíslenia (prevažne v závislosti na preferencii autora danej literatúry a veku publikácie). Podstata a hlavné myšlienky podmienok zabezpečujúcich riešenie problému kritickej sekcie však ostáva nezmenená (nech už je skondenzovaná do viac či menej bodov) [1][2][3].

Podmienky pre úspešné vyriešenie prístupu ku kritickej sekcii sú nasledovné:

- **Vzájomná výlučnosť (Mutual exclusion)**

  Podmienka vzájomnej výlučnosti značí, že ak proces vykonáva kód obsiahnutý vo svojej kritickej sekcii, nesmie byť počas tohto výkonu do kritickej sekcie vpustený žiaden iný proces, ktorého kritická sekcia operuje nad tým istým zdrojom (objektom, zdieľanými premennými...) [31].

- **Postup (Progress)**

  Ak žiaden proces svoju kritickú sekciu momentálne nevykonáva a existuje neprázdna množina procesov, ktoré sa o vstup do kritickej sekcie pokúšajú, musí byť jeden z týchto procesov do kritickej sekcie s určitosťou vpustený a jeho vstup nesmie byť odkladaný donekonečna [32].

- **Obmedzené čakanie (Bounded waiting)**

  Podmienka obmedzeného čakania stanovuje, že po žiadosti procesu o vstup do kritickej sekcie musí existovať určitá horná hranica či limit na počet vstupov udávajúci, koľko iných procesov ešte smie do kritickej sekcie vstúpiť pred udelením povolenia (t.j. „predbehnúť“ proces, ktorý práve o povolenie vstupu požiadal). Splnenie tejto podmienky zabezpečuje, že proces, ktorého priorita nie je príliš vysoká nebude nekonečne dlho ignorovaný v dôsledku neustáleho uprednostňovania dôležitejších procesov [33].

Povšimnutie by malo byť venované aj dodatočnému predpokladu pre úspešné vyriešenie problému kritickej sekcie (tento predpoklad je niekedy udávaný ako samostatná podmienka) [1].

Architektúrna neutralita (Architectural neutrality)

Platný predpoklad architektúrnej neutrality značí, že pri riešenie kritickej sekcie nie je viazané na konkrétnu hardwarovú implementáciu zariadenia či určitom predpoklade o rýchlosti a počte procesorov (t.j. riešenie, ktoré funguje len na určitom type procesoru nie je riešením).

Z povahy problému kritickej sekcie vyplýva, že neexistuje v podstate v architektúrach nepreemptívneho typu – t.j. v takých OS, v ktorých je v každom momente súčasne aktívny iba jeden proces. Tieto OS sa však používajú veľmi zriedka, keďže disponujú len obmedzenou schopnosťou responzivity a nedisponujú vlastnosťami nutnými pre možnosť tzv. „*real-time*“ programovania [3].

3.2.2 Riešenie kritickej sekcie

Snaha zabezpečiť, aby sa v kritickej sekcii zároveň nachádzal vždy iba jeden proces (vlákno) vyústila v niekoľko dnes známych a používaných riešení, ktorých vhodnosť, resp. užitočnosť sa rôzni v závislosti na konkrétnom probléme či implementačných obmedzeniach. Podľa spôsobu implementácie rozlišujeme prístupy k riešeniu problému kritickej sekcie na hardwarové a softwarové.

3.2.2.1 Hardwarové mechanizmy

- **Zakázanie prerušenia**

  Vypnutie prerušení je jednoduchým riešením – ak pred vstupom do kritickej sekcie proces zakáže prerušenie, bude celá kritická sekcia vykonaná bez rizika prerušenia a teda jej spracovanie bude možné vnímať ako atomickú operáciu [34].

  Riešenie, ktoré znie jednoducho a elegantne je ale v skutočnosti samo o sebe prakticky nepoužiteľné. Predstavuje mnoho nevýhod, pričom najvýraznejšou z nich je prílišná závislosť na dobrých úmysloch či kvalite implementácie jednotlivých procesov – môže dôjsť napr. k situácii, kedy proces vypne prerušenia a následne v ňom dôjde k chybe alebo vstúpi do nekonečného cyklu. V tomto prípade ani OS nedokáže prevziať kontrolu a jediným riešením je tvrdý reset (pokiaľ by daný proces bol napr. vírus, škody by mohli byť značné). Ďalšou poľutovaniahodnou skutočnosťou je, že na viacprocesorových systémoch vypnutie prerušenia problém kritickej sekcie nerieši – prerušenie sa síce zakáže v kontexte procesoru, avšak zvyšné procesory môžu do kritickej sekcie bez obmedzenia vstupovať [30].

- **Test-and-Set**

  Test-and-Set je hardwarová podpora synchronizácie procesov. Ide o špecializovanú inštrukciu procesora, ktorá (ako názov napovedá) dokáže vykonať operáciu prečítania a zároveň nastavenia hodnoty ako nedeliteľný, atomický celok.

  Inštrukcia *Test-and-Set* je v niektorých publikáciách označovaná pod názvom Test-*and-Set-Lock* a z neho odvodenou skratkou *TSL* – Mení sa však iba terminológia, ide o rovnaký mechanizmus pod rozdielnym názvom [35].

```
boolean TestAndSet( boolean *pTarget ) {
    boolean rv = *pTarget;
    *pTarget = TRUE;
    return rv;
}
```

Obr. 6. Interný mechanizmus inštrukcie Test-and-Set formou pseudokódu [36]

Využitie inštrukcie *Test-and-Set* je jednoduchým a efektívnym riešením, ale prináša so sebou určité negatíva. Najproblematickejším prípadom je situácia, keď sa o vstup do kritickej sekcie pokúša súčasne väčšie množstvo procesov - V tomto prípade všetky okrem procesu práve vykonávajúceho kód kritickej sekcie realizujú aktívne čakanie opakovaným testovaním hodnoty synchronizačnej premennej, a teda vyťažujú procesor bez reálneho prínosu [2].

- **Compare-and-Swap**

  Compare-and-Swap je podobne ako *Test-and-Set* hardwarovo implementovaná inštrukcia procesora využívajúca sa najmä k synchronizačným účelom. Priebeh inštrukcie je podmienený obsahom dvoch premenných, ktoré často nazývané zámok a kľúč (*lock*, *key*).

```
int compare_and_swap (int *word, int testval, int
newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Obr. 7. Interný mechanizmus inštrukcie Compare-And-Swap formou pseudokódu [37]

Iniciálne je v premennej *lock* priradená pravdivostná hodnota *false* a v premennej *lock* pravdivostná hodnota *true*. Ak je v premennej *lock* priradená hodnota *false*, proces smie vstúpiť do kritickej sekcie. Ak proces vstúpi do kritickej sekcie, prehodí pomocou inštrukcie *swap* pravdivostné hodnoty kľúča a zámku. *Swap* umožňuje porovnanie a nasledovnú zámenu hodnôt zámku a kľúča realizovať ako nedeliteľnú operáciu.

Je zrejmé, že použiteľnosť, výhody a nevýhody použitia inštrukcie *Compare-and-Swap* sú veľmi podobné ako u inštrukcie *Test-And-Set*, jej drobnou výhodou je mierne lepšia efektivita [38]. *Compare-And-Swap* mení hodnotu premennej len v prípade, že sa porovnávané hodnoty rovnajú, pričom *Test-And-Set* vykonáva prepísanie testovanej hodnoty vždy.

3.2.2.2 Softwarové mechanizmy

- **Petersonov algoritmus**

  Petersonov Algoritmus je korektným riešením problému kritickej sekcie. Demonštruje sa zvyčajne na dvoch procesoch – *P1*, *P2* za pomoci premenných *flag1*, *flag2* a *turn*.

  Premenná *flag* príslušného indexu nastaveného na pravdivú hodnotu značí, že proces je pripravený vstúpiť do svojej kritickej sekcie [1]. Pred vstupom však prepína príznak *turn* na druhý proces, čím mu dáva najavo, že smie vstúpiť do kritickej sekcie, pokiaľ si to želá (zjednodušene povedané - dáva druhému procesu prednosť pred vlastným vstupom). Pri výstupe z kritickej sekcie musí proces nastaviť svoj príznak *flag* na pravdivostnú hodnotu *false* – Samozrejme až do doby, kedy sa opäť nebude pokúšať o vstup.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Obr. 8. Ošetrenie kritickej sekcie pomocou Petersonovho riešenia [39]

Pri bližšom pohľade na kód implementácie Petersonovho algoritmu sú ale viditeľné nevýhody tohoto riešenia – okrem skutočnosti, že funguje maximálne pre dva procesy zároveň je nutné uvážiť, že pri nemožnosti vstúpiť do kritickej sekcie procesy opäť realizujú "*busy waiting*" cyklickým monitorovaním stavu príznakových premenných, čím plytvajú časom procesoru neproduktívnym spôsobom.

- **Synchronizačné primitíva**

  Synchronizačné primitíva sú súhrnným pomenovaním pre množinu mechanizmov zvyčajne zabezpečovaných operačným systémom a implementovaných (s drobnými odlišnosťami) konkrétnym programovacím jazykom. Tieto štruktúry či objekty môžu byť využité k softwarovému riešeniu kritickej sekcie – patria sem napr. rôzne typy zámkov, Spinlock, Mutexy, Semafory, či synchronizačné Udalosti. Každému typu sa budeme venovať podrobnejšie v nasledujúcej podkapitole.

3.3 Synchronizačné prostriedky

3.3.1 Spinlock

Rotačný Zámok (*Spinlock)* je jedným z najjednoduchších softwarových synchronizačných prostriedkov. V podstate sa jedná o príznak zvyčajne implementovaný ako posledný bit v celočíselnej premennej [40].

Zámok môže nadobúdať iba dva stavy – buď je zamknutý, alebo odomknutý. Proces vstupujúci do kritickej sekcie v prvom rade skontroluje, či je zámok zamknutý. Následne buď sám zmení stav zámku na zamknutý a začne vykonávať kód svojej kritickej sekcie, alebo realizuje aktívne čakanie, až dokiaľ zámok nie je odomknutý (preto názov „*spinlock*“, lebo čakajúce vlákna vykonávajú „*spinning*“ cyklickým čakaním na odomknutie zámku)[2].

Aktívne čakanie na odomknutie zámku so sebou opäť nesie už viackrát spomínaný problém plytvania procesorovým časom, ktorý by mohol byť využitý efektívnejšie. Použitie štruktúry spinlock ako primárneho synchronizačného mechanizmu sa odporúča v prípade, že predpokladáme len veľmi krátke čakania [1], keďže na rozdiel od mnohých riešení založených na hardwarových inštrukciách či na uspaní čakajúceho vlákna nie je nutná zmena kontextu (potenciálne náročná a relatívne dlhotrvajúca operácia).

Zámky kvôli svojej jednoduchosti ponúkajú lepší výkon ako zložitejšie synchronizačné primitíva (napr. semafory), avšak sú sužované viacerými obmedzeniami, pričom najzávažnejším je potenciálne dlhá čakacia doba (v prípade rozsiahlej či výpočtovo náročnej kritickej sekcie). Tento problém sa ďalej zväčšuje s rastúcim počtom vlákien žiadajúcimi o udelenie prístupu k danej kritickej sekcii.

3.3.2 Mutex

Mutex je názov poskladaný zo slovného spojenia “*mutual exclusion*” – a presne to je hlavnou úlohou mutexu, zabezpečiť vzájomnú výlučnosť v rámci kritickej sekcie. Mutex je často nesprávne zamieňaný za iné synchronizačné primitívum – binárny semafor. Medzi mutexom a semaforom ako takým existuje viacero rozdielov – niektoré z nich sú zrejmé na prvý pohľad, ostatné až po preskúmaní implementácie podrobného fungovania jednotlivých primitív [3].

3.3.2.1 Mutex vs. Semafor

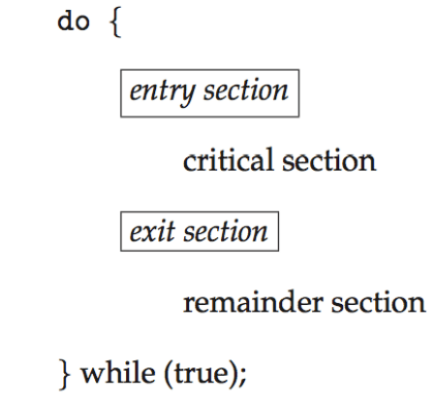
Tab. 2. Porovnanie charakteristických čŕt semaforu a mutexu [41]

	SEMAFOR	**MUTEX**
Účel	Signálny mechanizmus	Zámkový mechanizmus
Forma	Celočíselná premenná	Objekt / štruktúra
Funkcia	Pripustenie viacerých programových vlákien k ku konečnej množine zdrojov	Pripustenie viacerých programových vlákien k jedinému zdroju (nie však súčasne)
Vlastníctvo	Nezdieľajú pamäť s inými procesmi	Zdieľajú pamäť s ostatnými vláknami rodičovského procesu
Kategorizácia	Binárne semafory, Počítadlové semafory	Bez ďalšieho rozdelenia
Operácia	Hodnota semaforu je modifikovaná operáciami *Wait()* a *Signal()*	Objekt mutexu je uzamknutý/odomknutý procesom požadujúcim/uvoľňujúcim daný zdieľaný zdroj
Zahltenie zdrojov	Ak sa všetky zdroje práve používajú a proces požadujúci podklady zavolá *Wait(),* je zablokovaný až kým sa počítadlo semaforu nenavýši	Ak je mutex už uzamknutý, proces požadujúci prístup ku zdrojom čaká až kým nie je zámok uvoľnený

Principiálne sa *mutex* veľmi podobá klasickému zámku: implementuje dve operácie, pričom jedna z nich *mutex* „zamkne“ a druhá ho „odomkne“. Podobne ako u ostatných synchronizačných primitív, pokiaľ je *mutex* momentálne odomknutý, volajúce vlákno / proces dostane povolenie vstúpiť do svojej kritickej sekcie.

Diametrálna odlišnosť však nastáva v spôsobe reakcie na zlyhanie pri snahe vlákna získať mutex – na rozdiel od zámku sa nevykonáva aktívne čakanie, ale vlákno sa zablokuje (uspí), čím dostane plánovač OS možnosť využiť procesor efektívnejšie, ako opakovaným testovaním premennej [42]. Je teda možné skonštatovať, že mutex je oproti klasickým zámkom s aktívnym čakaním preferovanou možnosťou - pokiaľ nie je zaručené, že doba čakania bude dostatočne krátka [1].

Mutexy sú hojne využívané a zastúpené najmä v OS *Linux*, kde sú funkcie na manipuláciu s nimi súčasťou *API* výkonového modelu *POSIX Threads*. V OS *Windows* je mutex tiež prítomný vo forme tzv. „*Mutex Objects*“ [43]. Tieto objekty obsahujú vlastnú množinu funkcií (*CreateMutex(), OpenMutex(), ReleaseMutex()*) pokrývajúcich funkcionalitu približne ekvivalentnú ich *POSIX* variante. Mnohokrát je však pred objektom “*Mutex”* preferovaný objekt „*Critical Section*“ [44].



Obr. 9. Princíp ošetrenia kritickej sekcie prostredníctvom mutexu – bloky „entry section“ a „exit section“ sú implementované vo forme uzamknutia a uvoľnenia mutexu [2][45]

3.3.3 Semafor

Semafory sú návrhovo jednoduché, avšak funkčne bohaté – interne pozostávajú z celočíselnej premennej a dvoch operácií – *Wait()* a *Signal()*. Každá kritická sekcia (resp. zdieľaný zdroj) by mala mať pridelený samostatný semafor – následne každý proces usilujúci o vstup do kritickej sekcie skontroluje, či je číselná premenná tohto semaforu väčšia ako 0. Ak áno, dekrementuje túto hodnotu pomocou operácie *Wait()* a vstupuje do kritickej sekcie – ak nie, sám seba uspí a uvoľní procesor (podobne ako to je u mutexov) [8]. Pri opustení kritickej sekcie ošetrenej semaforom by malo výkonové vlákno zavolať operáciu *Signal(),* čím efektívne dôjde k navýšeniu číselnej premennej semaforu.

Hodnota celočíselnej premennej obsiahnutej v semafore je minimálne 0 (t.j. musí to byť vždy hodnota kladná alebo nulová).

Aby semafory fungovali podľa očakávaní, je nutné zabezpečiť, že operácie *Wait()* a *Signal()* sú vykonávané ako nedeliteľné – a teda nesmie pri prístupe k číselnej premennej semaforu nastať situácia typu Producent-Konzument, kedy by počas modifikácie stavu tejto premennej niektoré procesy mohli prečítať nesprávnu hodnotu. (Atomicita zvykne byť implementovaná pomocou využitia inštrukcie *Test-And-Set* v kombinácii s krátkym zakázaním prerušenia).

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

Obr. 10. Princíp operácií Wait() a Signal() vykonávaných semaforom [46]

Zakázanie prerušenia je využité z dôvodu minimálnej časovej straty vplyvom tohto zákazu – testovanie číselnej premennej na nulovosť, resp. nenulovosť a navýšenie celočíselnej premennej sú elementárne inštrukcie, ktoré procesor vykoná mimoriadne rýchlo a teda prerušenie nemusí byť zakázané po príliš dlhú dobu.

Ďalším dôvodom pre implementáciu internej logiky semaforu za pomoci zakázania prerušení je optimalizácia výkonu semaforu. Semafor často neovplyvňuje len jedno vlákno,

ale závisí na ňom rôzny počet vlákien bežiacich na rôznych procesoroch. Ak by teda došlo k prerušeniu napr. počas vykonávania inštrukcie *Test-And-Set* či pred navýšením čítača semaforu, potenciálne časové straty by mohli mnohonásobne presiahnuť časovú škodu spôsobenú jednoduchým zákazom prerušenia pred výkonom malého množstva operácií [47].

Navzdory častému nedorozumeniu, že semafory sú určené len k synchronizácii prístupu do kritickej sekcie, je ich využitie často omnoho širšie – vo všeobecnosti ich smieme použiť k ošetreniu prístupu k ľubovoľnému zdieľanému zdroju, nad ktorým chceme implementovať „limit“, napr. pri vytváraní nových okien aplikácie s definovaným maximálnym počtom [48].

Výhodami semaforov sú najmä možnosť povoliť prístup do kritickej sekcie viacerým vláknam súčasne (ak to je žiadúce), a jednoduchosť použitia (zvyčajne je API semaforu implementované pomocou troch základných operácií – *Create()*, *Wait()* a *Signal()*).

Nevýhodou semaforového riešenia je najmä nutnosť zvýšenej réžie (OS musí udržiavať všetky volania operácií nad semaforom z dôvodu efektívneho plánovania vlákien) a neoptimálne správanie v prípade inverzie priorít (*Priority Inversion*), čo je situácia, kedy je semafor držaný procesom s nízkou prioritou a proces s vyššou prioritou musí čakať na jeho uvoľnenie (viac sa dá o inverzii priorít dočítať v samostatnej kapitole „Inverzia Priorít“).
Naviac, pri zvolení nesprávneho poradia operácií *Wait()* a *Signal()* či pri nesprávnom iniciálnom nastavení semaforu môže dôjsť k situácii, kedy budú všetky vlákna čakať na uvoľnenie semaforu a dôjde k uviaznutiu (pre detailnejší popis ako k danej situácii môže dôjsť sa odporúča referovať ku kapitole „Uviaznutie“).

3.3.4 Event

Event (udalosť), ako názov napovedá, je synchronizačné primitívum označujúce vnútorný stav vo forme pravdivostnej hodnoty a množiny operácií, pomocou ktorých s týmto stavom smieme manipulovať. V zásade sa jedná o jednoduchý synchronizačný prostriedok, zvyčajne používaný ako vhodný spôsob napr. k tomu, aby sme vláknam čakajúcim na dokončenie určitej operácie oznámili, že skutočne dokončená bola.

Udalosť môže byť v závislosti na svojom vnútornom stave v dvoch stavoch – nastavená alebo nenastavená. Pokiaľ je nenastavená, prípadné procesy čakajúce na dokončenie tejto udalosti budú zobudené až v moment jej nastavenia. (Nie je možno jednoznačne stanoviť, či

procesy čakajúce na dokončenie udalosti procesy vykonávajú aktívne čakanie, alebo uspia. Toto správanie závisí na konkrétnej implementácii v rámci OS, či dokonca na úrovni programovacie jazyka, keďže udalosti sú jazykovo / systémovo špecifické konštrukty). Alternatívne sa dajú synchronizačné udalosti použiť vo forme bariéry – tzv. „*Countdown Event*“ - teda taká udalosť, ktorá sa prepína do „zopnutého“ stavu až v momente, keď zaregistruje určité množstvo signálov.

Udalosti sa za účelom synchronizácie používajú najmä v OS Windows [49] – v OS Linux však existuje možnosť ich implementovať za pomoci kombinácie mutexu a pravdivostnej premennej [50].

3.3.5 Monitor

Využitie nízkoúrovňových synchronizačných primitív môže byť často náročné či programátorsky nepríjemné – napr. nesprávne využitie semaforu nezriedka vedie k uviaznutiu. Z toho dôvodu bol popísaný koncept dôkladne uzavretej štruktúry či objektu nazývaného *Monitor*, ktorý slúži ako komplexnejší synchronizačný prostriedok – ponúka funkcionalitu ekvivalentnú semaforu, avšak jednoduchšiu na použitie a s nižšou náchylnosťou k programátorským chybám. Monitor je vo svojej podstate značne abstraktný pojem. Tvorí ho sústava premenných, dátových štruktúr a procedúr ktoré sú zapuzdrené – teda viditeľné iba v rámci monitoru samotného, pričom zvonku je prístup k nim obmedzený na volania prostredníctvom presne definovaného aplikačného rozhrania [3].

Základným parametrom monitorov je to, že v ľubovoľnom čase dokáže byť v monitore aktívny iba jeden proces zároveň. Táto vlastnosť ja zabezpečovaná kompilátorom, resp. interpreterom jazyka, v ktorom je monitor implementovaný. Aj keď interná implementácia monitoru zabezpečuje vzájomnú výlučnosť, je nutné zabezpečiť, že v prípade nemožnosti pokračovať do kritickej sekcie bude proces zablokovaný – K tomu sa využívajú tzv. *Condition Variables*. Štandardne sú súčasťou *API* Monitoru dve hlavné operácie – *Wait()* , ktorá umožňuje procesu zablokovať sa a „čakať“ na určitú *Condition Variable*, a *Signal(),* ktorá umožňuje prebudiť vlákna čakajúce na tejto premennej v rámci operácie *Wait()* [8].

Monitory sú abstraktným konštruktom a aby bolo možné ich použiť, musia byť implementované v rámci daného programovacieho jazyka, aby kompilátor (prípadne interpreter) vedel zaručiť vzájomnú výlučnosť pri volaní procedúr monitoru [1].

SYNCHRONIZAČNÉ PROBLÉMY

3.4 Súbeh

Súbeh (*Race Condition*) je jav, ku ktorému môže dochádzať, ak má viac procesov (alebo vlákien) povolený prístup k určitému zdieľanému zdroju [1][2]. Zdieľaným zdrojom môže byť globálna premenná, súbor, pamäť...

Dôsledkom súbehu môže byť nekorektné správanie procesov pristupujúcich k zdieľaným zdrojom (a to aj za predpokladu, že sa každý zo zúčastnených procesov sám o sebe správa korektne). Príčinou tohto fenoménu je komplexita operácií nad zdieľanými dátami, u ktorých sa predpokladalo, že budú operáciami atomickými (t.j. realizované v jedinom kroku) [1].

Súčasný prístup viacerých procesov k dátam niekedy vedie k nekonzistencii, ktorá sa vyznačuje vysokou mierou náhodnosti – chyba môže a nemusí vzniknúť. Záleží iba na tom, v akom poradí bola sekvencia príkazov od pristupujúcich procesov vykonaná, resp. či bol niektorý počas výkonu prerušený operačným systémom a v ktorej časti výkonu sa nachádzal, keď k tomuto prerušeniu došlo [6].

V moderných operačných systémoch, ktorých neodmysliteľnou súčasťou je neustále sa zvyšujúca miera paralelizmu predstavuje možnosť súbehu problém, ktorý je nutné adresovať.

Typickým prípadom súbehu je tzv. problém „*Producent-Konzument*“ (úloha producenta a konzumenta bola spracovaná aj v praktickej časti práce, pričom jej je venovaná kapitola „Producent / Konzument - Synchronizácia“).

3.5 Uviaznutie

Uviaznutie (*Deadlock*) je situácia, ktorá nastáva v dôsledku súbehu za predpokladu, že rozdeľovanie zdieľaných zdrojov nie je korektne ošetrené (napr. pomocou supervízie OS). K uviaznutiu môže dôjsť, ak existuje množina taká množina procesov $M = \{P_1, P_2, \ldots P_n\}$, že každý jeden z procesov v množine M potrebuje ku svojmu pokračovaniu zdroj, ktorý je momentálne blokovaný niektorým zo zvyšných zdrojov nachádzajúcich sa v M [30].

Uviaznutie je nežiadúcim javom – pokiaľ k nemu dôjde, procesy $P_1 \ldots P_n$ nedokážu pokračovať vo svojej činnosti bez intervencie od vyššej autority (napr. OS).

3.5.1 Podmienky uviaznutia

Existuje pevne definovaná štvorica podmienok určujúca, či môže dôjsť k uviaznutiu. Súčasná pravdivosť všetkých týchto podmienok určuje, že uviaznutie je skutočne možné (ale v žiadnom prípade nie isté) [51].

Coffmanove podmienky pre vznik uviaznutia [52]:

- **Podmienka Vzájomnej výlučnosti** (*Mutual Exclusion Condition*)

  Ľubovoľný zdroj je v každom časovom momente je buď priradený určitému procesu, alebo voľne dostupný.

- **Podmienka „Drž-A-Čakaj“** (*Hold And Wait Condition*)

  Procesy vlastniace zdroje, ktoré im boli pridelené v minulosti, môžu žiadať o pridelenie ďalších zdrojov bez nutnosti uvoľniť práve vlastnené.

- **Podmienka Nepreemptívneho prístupu** (*No-Preemption Condition*). Pridelené prostriedky nie je možné procesu násilne odobrať a jediný spôsob ako ich získať je vyčkať, kým ich proces dobrovoľne uvoľní.

- **Podmienka Cyklického čakania** (*Cyclic Wait Condition*). Existuje cyklický zoznam procesov $Z = \{P_1, P_2, \ldots P_n\}$ obsahujúci dva alebo viac procesov, pričom každý proces P_i v zozname čaká na uvoľnenie zdroja vlastneného nasledujúcim procesom P_{i+1}, a P_n čaká na zdroj vlastnený procesom P_1.

Systémy, ktoré neimplementujú žiadnu formu paralelizmu (teda v nich v ľubovoľnom časovom okamihu beží zároveň iba jeden proces) sa nemusia obávať konvenčného uviaznutia, avšak stále podliehajú riziku uviaznutia vo forme nekonečného „spánku“ vplyvom cyklického čakania, napr. pokiaľ proces očakáva výsledok I/O operácie, pričom I/O operácia očakáva dodatočný signál od tohto blokovaného procesu [3].

3.5.2 Riešenie uviaznutia

3.5.2.1 Predchádzanie uviaznutiu

Uviaznutie môže nastať jedine za predpokladu súčasného splnenia podmienok jeho výskytu – je teda možné mu efektívne predchádzať narušením pravdivosti jednej z daných podmienok. Cieliť je pritom možné prakticky na ktorúkoľvek podmienku:

- **Vzájomná výlučnosť:**

  Narušenie podmienky vzájomnej výlučnosti je dosiahnuteľné pomocou metódy nazývanej *Spooling*. *Spooling* sa zakladá na úvahe, že zdroje určené iba k čítaniu nemôžu viesť k uviaznutiu [1] – preto by sa všetky zdroje mali virtualizovať, t.j. zaobaliť do dodatočnej vrstvy abstrakcie s využitím súborov určených iba na čítanie.

  Príklad použitia techniky *Spooling*:

  - Pri vstupných zariadeniach (napr. čítačka kariet) OS načíta požadované dáta do súborov určených iba k čítaniu, z ktorých ku nim potom pristupujú jednotlivé procesy.
  - Pri výstupných zariadeniach (napr. tlačiareň), naopak, jednotlivé procesy zapisujú výstup nie priamo na výstupné zariadenie, ale do súborov. Nad týmito súbormi manipuluje singulárny proces zodpovedný za realizáciu výstupu.

  Bohužiaľ, dokonca aj s využitím *spooling*-u môže uviaznutie nastať (na niektoré zdroje totiž nie je možné *spooling* vôbec aplikovať).

  Algoritmy pokúšajúce sa o prevenciu uviaznutia pomocou prelomenia podmienky vzájomnej výlučnosti nazývame neblokujúce synchronizačné algoritmy [53].

- **„Drž-A-Čakaj“:**

  Hold-And-Wait podmienka je prelomiteľná postupnou alokáciou zdrojov procesom, pričom zdroje rozdeľuje nejaká vyššia autorita, spravidla OS.

  Každý proces ešte pred svojím štartom musí požiadať o všetky zdroje, ktoré bude potrebovať k úspešnému dokončeniu. O zdroje smie žiadať iba vtedy, ak žiadne zdroje nevlastní. Alternatívne musí pred požiadaním o ďalšie zdroje odovzdať všetky momentálne vlastnené.

Tento prístup vedie k efektívnemu odstráneniu uviaznutia, ale obsahuje niekoľko zásadných slabín.

Najvýznamnejším problémom je skutočnosť, že zdroje sú dynamicky alokované a veľmi málo procesov pri svojom spustení presne vie, aké zdroje (a v akej kvantite) bude potrebovať k vlastnému dokončeniu.

Ďalším problémom sú spôsobené dôsledky – „vyhladovanie“ procesov, ktoré potrebujú k dokončeniu viacero často používaných zdrojov [54]. Tieto procesy môžu byť ignorované po veľmi dlhý čas - šanca, že budú súčasne dostupné všetky zdroje ktoré takto náročný proces potrebuje je pomerne nízka.

Tento prístup prevencie uviaznutia naviac vedie k nízkej efektivite využitia procesov (proces po celú dobu svojho behu blokuje zdroje, ktoré potrebuje len na chvíľu).

- **Nepreemptívny prístup:**

Prelomenie podmienky nepreemptívneho prístupu vyžaduje intervenciu od vyššej autority, správcu (napr. OS). Pokiaľ dôjde k prerušeniu procesu, správca odoberá procesu prostriedky a prerozdeľuje ich iným procesom. Kvôli nutnosti uschovať a obnoviť stav prostriedku nie je tento postup možný u všetkých typov zdrojov – aplikovateľné napr. pri CPU, RAM [52], problematické napr. pri tlačiarni.

- **Cyklické čakanie:**

Na prelomenie podmienky cyklického čakania existujú dva možné prístupy:

  - Pridelenie vždy iba jedného prostriedku zároveň. Tento prostriedok musí byť vrátený vždy, ak proces žiada o prostriedok ďalší. Viacero procesov ale potrebuje na svoje úspešné dokončenie niekoľko prostriedkov (nezriedka súčasne), čím sa tento prístup stáva v praxi nepoužiteľným.
  - Číslovanie zdrojov: Každý prostriedok má pridelené nejaké číslo. Proces smie žiadať iba o prostriedky s číslom vyšším ako je najvyššie číslovaný zdroj z množiny zdrojov, ktoré už vlastní. Proces vyžadujúci viacero prostriedkov s rovnakým číslom musí o tieto zdroje žiadať zároveň.

    Číslovanie zdrojov je zaujímavý prístup, ktorý je však obtiažne realizovateľný – väčšinou platí, že prostriedkov na prideľovanie je veľké množstvo. Vyhľadanie takého očíslovania prostriedkov, ktoré k uviaznutiu

nepovedie je výpočtovo náročná úloha, pričom pri skutočne veľkom množstve zdrojov takéto usporiadanie vôbec nemusí existovať [1].

3.5.2.2 Vyhýbanie sa uviaznutiu

Vyhýbanie uviaznutiu je realizované pomocou jednoduchej myšlienky - každý proces ešte pred svojim spustením požiada o maximálny počet zdrojov o ktorých je známe, že by ich mohol počas svojho behu potrebovať.

OS musí monitorovať, či pridelenie daných zdrojov procesu nemôže potenciálne viesť k uviaznutiu (t.j. pred pridelením každého prostriedku zistiť, či je možné bezpečne dokončiť všetky procesy). Ak to možné je, prostriedky sú procesu pridelené.

Existuje väčšie množstvo známych algoritmov, ktoré sa danou problematikou zaoberajú – viď. napr. riešenie nazývané „Bankárov algoritmus“ [54][55].

3.5.2.3 Detekcia a zotavenie

Operačný systém povoľuje, aby uviaznutie nastalo. Z toho dôvodu sú zdroje jednotlivým procesom prideľované pomerne liberálne. V určitom časovom intervale sa pravidelne spúšťa algoritmus detekcie uviaznutia.

OS v tomto prípade musí udržiavať aktuálne rozdelenie pridelených prostriedkov vo forme grafu. Pokiaľ sa v grafe nachádza cyklus, došlo k uviaznutiu. Samotné riešenie môže nadobúdať jednu z troch foriem:

- Preempcia: Zdroje sú niektorému u uviaznutých procesov dočasne odobrané a priradené inému uviaznutému procesu. Ak získanie týchto prostriedkov vedie k prelomeniu uviaznutia, sú mu neskôr navrátené.
- Zabitie procesov: Uviaznuté procesy sú násilne ukončené a nimi vlastnené prostriedky uvoľnené. Takéto riešenie však vedie k strate práce, ktorú dané procesy vykonávali.
- Rollback: Prístup *Rollback* sa zakladá na úvahe, že pokiaľ má OS k dispozícii graf pridelených zdrojov, dokáže zvrátiť všetky vykonané zmeny, ktoré k uviaznutiu potenciálne viedli. Rollback je väčšinou výhodnejší ako tvrdé zabitie procesov, aj tak ale dochádza k strate časti práce vykonanej danými procesmi odkedy došlo k uviaznutiu.

3.5.2.4 Ignorovanie uviaznutia

Alebo aj „Pštrosí Algoritmus“ (*Ostrich Algorithm*). Ako názov napovedá, nie je v skutočnosti riešením. Myšlienkou algoritmu je predstieranie, že situácia uviaznutia neexistuje a nikdy k nej nemôže dôjsť. Navzdory jednoduchosti a zdanlivej naivite tohto riešenia je to preferovaný prístup v mnohých dnešných OS. Dôvodom je mimoriadna vzácnosť uviaznutia - Mechanizmus, ktorý by periodicky zabezpečoval ich riešenie či prevenciu vyžaduje dodatočnú réžiu, čo by sa mohlo negatívne prejaviť na celkovom výkone systému [56].

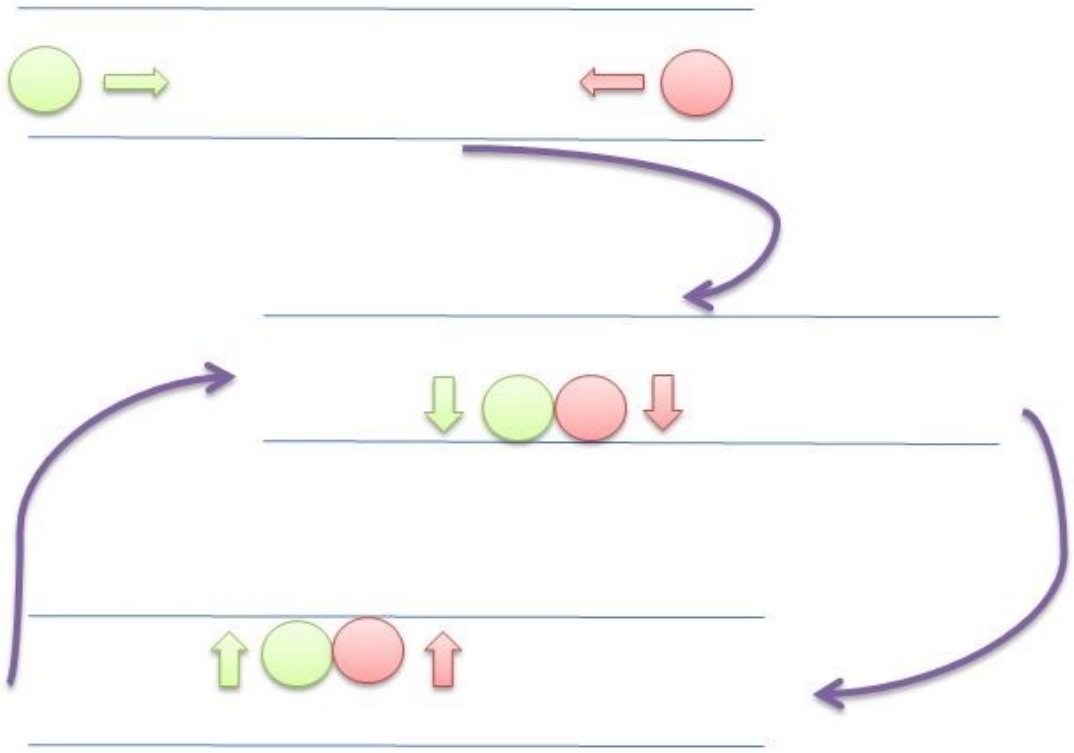
Aby sa v tomto prípade systém zotavil, môže byť nutný manuálny zásah užívateľa, poprípade násilné ukončenie (zabitie) niekoľkých procesov za účelom odobratia pridelených prostriedkov.

3.5.3 LiveLock

LiveLock je nežiadúca situácia, ktorá občas vzniká najmä u algoritmov slúžiacich k riešeniu (prevencii) uviaznutia – paradoxne je LiveLock svojim efektom na exekúciu zúčastnených procesov uviaznutiu značne podobný.

Na rozdiel od uviaznutia, procesy (či vlákna) ktorých sa LiveLock týka nie sú blokované – sú jednoducho príliš vyťažené vzájomným reagovaním na svoje akcie v snahe zabrániť uviaznutiu bez toho, aby dosiahli skutočného progresu [57]. LiveLock nastáva, pokiaľ oba procesy opakujú určitú postupnosť interakcií (a tento sled akcií a reakcií má cyklický charakter) [3].

Analógiou z reálneho sveta k tomuto typu správania je stret dvoch vzájomne oproti idúcich chodcov v úzkom koridore [29]. Aby sa vyhli potenciálnej kolízii, dajú si navzájom prednosť – a obaja uhnú do rovnakej strany. Následne toto konanie nastáva opakovane a chodci si navzájom uhýbajú bez možnosti dôjsť do cieľa svojej cesty.

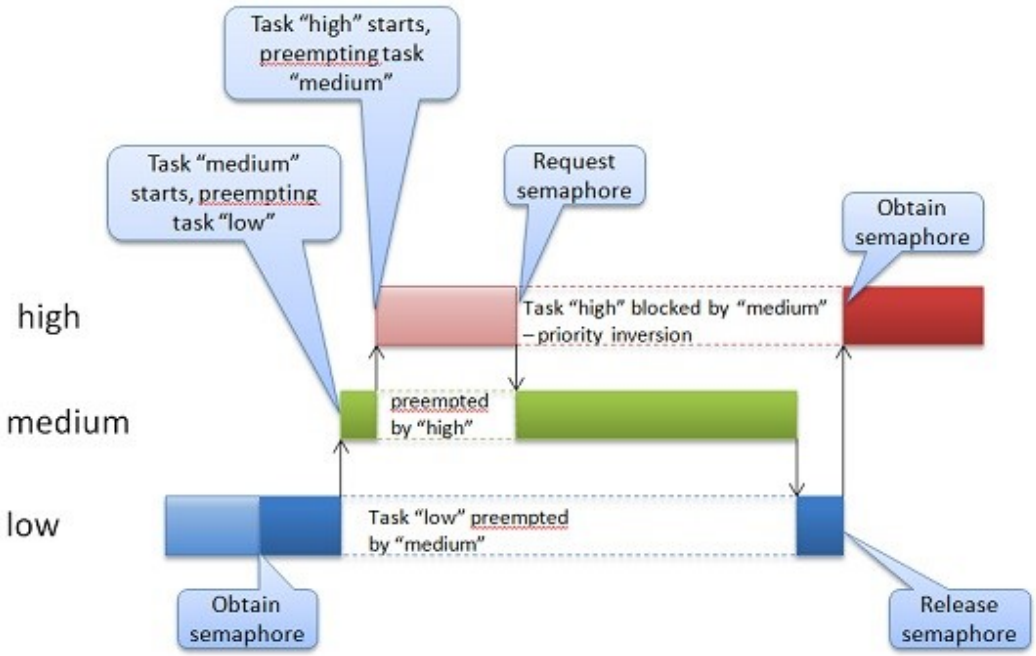


Obr. 11. Livelock – grafické znázornenie situácie [58]

3.5.4 Inverzia priorít

Inverzia Priorít (*Priority Inversion*) je zriedkavý fenomén spájaný s využitím synchronizačných prostriedkov, resp. povahou ich implementácie a spôsobom, akým procesy pristupujú k ich funkcionalitám [viď. semafor].

Inverzia Priorít značí, že v rámci systému preemptívneho charakteru proces s nízkou prioritou vlastní zdroje vyžadované procesom s vyššou prioritou, čím je vysokoprioritný proces zablokovaný. Následne proces s prioritou strednou ktorý daný prostriedok nepotrebuje v dôsledku preemptivity systému zabraňuje nízkoprioritnému procesu pokračovať v činnosti a tým prostriedok uvoľniť – čo vedie k situácii, že práve vykonávaný proces strednej priority blokuje beh dôležitejšieho procesu [29].



Obr. 12. Inverzia priorít – grafické znázornenie situácie [59]

II. PRAKTICKÁ ČASŤ

4 TECHNOLÓGIE

4.1 .NET Core

Ukážkové aplikácie implementované v praktickej časti tejto práce boli realizované pomocou grafického subsystému WPF, za ktorým beží kód v jazyku C#. Pre túto kombináciu technológií bol kvôli rozsiahlym výhodám oproti svojmu predchodcovi [60] logickou voľbou *.NET Core Framework* vo verzii *3.xx* (xx stojí za číslice označujúce konkrétnu pod-verziu, keďže *.NET Core Framework* sa vyvíja pomerne rýchlo a jednotlivé ukážkové programy môžu byť s postupujúcim časom založené na novších mutáciách verzie *3*).

Využitie framework-u *.NET Core* pre vývoj našich ukážkových programov so sebou nesie určité výhody, pričom najvýznamnejšími sú:

- Open-Source framework, udržiavaný vo viacerých repozitároch siete GitHub (dostupný skrz https://github.com/dotnet), väčšinou spravovaných pod licenciami *MIT* / *Apache 2* / *Creative Commons Attribution 4.0* [61]
- Veľké množstvo dostupných knižníc a rozšírení
- Vysoká rýchlosť
- Podpora moderných programátorských princípov (*Containers, Dependency Injection, Inversion of Control*)

.NET Core sám o sebe je naviac multiplatformný – avšak v prípade ukážkových programov je nutné brať do úvahy ich previazanosť s technológiou WPF, ktorá multiplatformná nie je [62]. Z toho dôvodu je cieľovým OS implementovaných aplikácií OS Windows.

4.2 WPF

WPF – *Windows Presentation Foundation.*

WPF je grafický subsystém vyvíjaný firmou Microsoft. Slúži k vykresľovaniu užívateľských rozhraní na ňom založených a tvorbu tzv. *Windows-Based Applications*, alebo „oknových aplikácií“.

Vykresľovanie (*rendering*) v technológii WPF je hardwarovo akcelerované. WPF nahrádza staršiu technológiu *WinForms*, oproti ktorým ponúka lepší výkon, škálovateľnosť, širšie zabezpečenie responzivity, väčší výber kontrolných prvkov a rozsiahlejšie možnosti ich prispôsobenia (štýly, umiestnenie, rozloženie ...).

Každá WPF aplikácia sa skladá z dvoch elementárnych častí:

- XAML Layout
- Code Behind

XAML Layout: Ide o dokument využívajúci deklaratívny značkovací jazyk XAML. Takýto dokument má príponu .xaml a umožňuje vytváranie a štýlovanie UI elementov [63]. XAML layout sa všeobecnou štruktúrou značne podobá ostatným známym značkovacím jazykom, napr. jazyku HTML. Umožňuje jednoducho vytvárať elementy užívateľského rozhrania a pomocou tzv. atribút určiť ich ďalšie vlastnosti – ako sú napr. výška, šírka, farba pozadia, umiestnenie v rámci rodičovskej mriežky a pod.

```
<Controls:MetroProgressBar
    Grid.Row="0" Grid.Column="2" Margin="0, 5, 10, 5"
    Minimum="0"
    Maximum="100"
    Value="{Binding Philosophers[0].Progress}"
    Name="philosopher_I_Status_ProgressBar">
</Controls:MetroProgressBar>
```

Obr. 13. XAML ProgressBar element vrátane atribútov a jednoduchého previazania zobrazovanej hodnoty

Code Behind: Je to kód na pozadí XAML Layout-u. Používa sa k obsluhe rôznych udalostí (napr. kliknutie na tlačidlo, listovanie v zozname a pod.). Z dôvodu využitia architektonického vzoru MVVM je žiadúce, aby bol Code Behind čo najmenší – a teda neobsahoval kód, ktorý by mal správne obsahovať *ViewModel*, *Model*, či príslušná *Service*. Preto bol *Code Behind* v ukážkových projektoch využitý najmä ako prostriedok k obsluhe a reagovaniu na udalosti GUI vyvolávajúce zmenu štýlu či správania jednotlivých elementov, nie k implementácii samotnej logiky vyvolanej týmito akciami.

```
/// <summary> Event-induced function used to keep DataGrid scrolled to bottom (u ...
1 reference
private void DataGrid_ScrollChanged(object sender, ScrollChangedEventArgs e)
{
    if (e.ExtentHeight < e.ViewportHeight)
        return;
    if (LoggerGrid.Items.Count <= 0)
        return;
    if (e.ExtentHeightChange == 0.0 && e.ViewportHeightChange == 0.0)
        return;

    var oldExtentHeight = e.ExtentHeight - e.ExtentHeightChange;
    var oldVerticalOffset = e.VerticalOffset - e.VerticalChange;
    var oldViewportHeight = e.ViewportHeight - e.ViewportHeightChange;
    if (oldVerticalOffset + oldViewportHeight + 5 >= oldExtentHeight)
    {
        LoggerGrid.ScrollIntoView(LoggerGrid.Items[^1]);
    }
}
```

Obr. 14. Typická ukážka funkcie umiestnenej do Code Behind, ktorá zabezpečuje automatické posúvanie pohľadu mriežky s logmi na poslednú položku

XAML layout má ešte jednu kľúčovú schopnosť, ktorou je tzv. *Data Binding* (detailnejší popis mechanizmu *Binding* je poskytnutý v rovnomennej kapitole tejto práce).

4.3 Nástroje

Ukážkové programy, z ktorých pozostáva praktická časť tejto práce boli skonštruované za pomoci viacerých softwarových nástrojov a knižníc (výpis jednotlivých nástrojov viď. nižšie). Či už sa jednalo o oficiálne balíčky a rozšírenia poskytované priamo firmou Microsoft, alebo voľne dostupné produkty tretích strán a nezávislých vývojárov, je vhodné uviesť aspoň tie, ktoré prácu významne urýchlili či uľahčili:

- **MahApps.Metro**

  Komunitný projekt, inštalovaný vo forme *NuGet* balíčku. Jedná sa o UI vývojovú sadu obsahujúcu pôsobivé množstvo nástrojov, štýlov, paliet, ovládacích prvkov a iných zdrojov [64]. Pomáha prekryť niektoré obmedzenia WPF a zjednodušuje tvorbu čisto a moderne pôsobiaceho GUI.

- **Ninject**

  Ninject je *open-source*, "light-weight" nástroj slúžiaci k implementácii princípu Dependency Injection [65]. Inštalovaný vo forme *NuGet* balíčku.

- **Visual Studio**

  Vývojové prostredie (*IDE*) použité k vývoju aplikácií praktickej časti. Solídny softwarový nástroj značne uľahčujúci vývoj vďaka integrovanému *WPF dizajnéru* [66], Doplňaciemu nástroju *Intellisense* [67], podpore rýchlych rozšírení v podobe *NuGet* balíčkov a sofistikovanému súboru rýchlych akcií.

  Visual Studio bolo za účelom zhotovenia praktickej časti práce získané a použité vo verzii *Community* prostredníctvom oficiálneho portálu určeného k jeho distribúcii na stránke https://visualstudio.microsoft.com/cs/downloads ako súčasť balíčku Visual Studio Dev Essentials. Získanie vývojového balíčku prebehlo v súlade s podmienkami študentského distribučného programu tohto softvéru [68]. Software bol následne registrovaný s využitím študentského Microsoft účtu poskytovaného univerzitou.

5 ARCHITEKTÚRA M-V-VM

Model – View – ViewModel je konkrétnym architektonickým vzorom a teda špecifikuje spôsob, ktorým je možné pristupovať k návrhu aplikácie z hľadiska štruktúry, členenia a logického rozdelenia. Zvyčajne používaný v počítačových a mobilných aplikáciách, MVVM ponúka rozumné oddelenie zobrazovacej a aplikačnej logiky. Architektúra MVVM sa naviac postupne vyvíjala okolo myšlienky *Data Binding*-u v technológii WPF [69], čím sa stala vhodným kandidátom na použitie pri implementácii praktickej časti práce.

5.1 Model

Modely popisujú formu dát, s ktorými aplikácia pracuje. Väčšinou sa jedná o triedu udržiavajúcu určitý stav. Definuje vlastnosti, z ktorých niektoré môžu (a nemusia) byť zobrazované pomocou prístupu skrz ViewModel. Pokiaľ je žiadúce využitie databázy, tak sú modely väčšinou triedami priamo mapovanými na jej tabuľky. Modelové triedy samotné nesmú mať priamu závislosť na UI a nemali by nič vedieť o stave ovládacích prvkov.

V čistom MVVM návrhovom vzore modely udržiavajú Business logiku aplikácie. V mutácii použitej aplikáciami vyvinutými pri praktickú časť tejto práce však používame tzv. „skinny models“ – Modely ktoré sú iba šablónou určujúcou tvar dát a neobsahujú žiadnu logiku. Business logiku a tým pádom aj prepojenie medzi logickými celkami *Model* - *ViewModel* v tomto prípade zabezpečujú služby (*Services*), ktoré budú podrobnejšie popísané v samostatnej podkapitole.

5.2 View

Grafický subsystém WPF implementuje *Views* pomocou už spomínaného XAML jazyka. *Views* sú zodpovedné za vizuálnu a štylistickú stránku aplikácie. Umožňujú skladať užívateľské rozhranie z dostupných vizualizačných a ovládacích prvkov. Zároveň zaručujú správne zobrazenie dát k tomu určených. Ak je žiadúce, aby View dynamicky reagovalo na zmeny dát v podkladových triedach, vyvstáva nutnosť špecifikovať tzv. Binding Context.

Ďalšie informácie o bohatých možnostiach prepojenia zobrazenia a dát na pozadí je možné nájsť na oficiálnej dokumentácii firmy Microsoft [70].

5.3 ViewModel

ViewModel je najdôležitejšou súčasťou MVVM Architektúry. Udržiava vizualizačný stav aplikácie – štruktúrované dáta, ktoré sú zobrazované prostredníctvom *View* (užívateľského rozhrania). *View* je na *ViewModel* napojené pomocou mechanizmu Viazania (tzv. *Binding*). Čo je ešte dôležitejšie, udržiava dáta v štruktúrach, ktoré umožňujú vyvolanie udalosti pri zmene a tým signalizujú grafickému rozhraniu potrebu na túto zmenu reagovať. *ViewModel* ďalej slúži ako prepojenie medzi dátami samotnými (*Model*) a zobrazením (*View*) pričom platí, že tieto dve prvky by nemali byť schopné vzájomnej modifikácie a komunikácie bez použitia *ViewModel*-u ako medzistupňa.

5.4 Modifikácie

Základný architektonický vzor *Model-View-ViewModel* bol pre potreby vypracovania praktickej časti tejto práce rozšírený o dodatočnú zložku, ktorou sú Služby (*Services*).

Služby majú v architektúre MVVM značne voľnú interpretáciu a použitie, avšak platí, že ich účel je vykonávať úzku časť funkcionality nad špecifickou množinou dát. Ak služba potrebuje vykonať niečo čo nespadá do jej kompetencie, je vhodné túto situáciu vyriešiť pomocou referencie na inú službu [71].

V kontexte ukážkových príkladov sú služby princípom *Dependency Injection* vkladané do *ViewModel*-ov či iných služieb – vykonávajú operácie nad ich dátami a posielajú *ViewModel*-om upozornenie, ak boli dáta modifikované. Slúžia teda ako dodatočná vrstva, manipulujúca s dátami *ViewModel*-ov, ktorú však *ViewModel* smie konzumovať iba prostredníctvom nami definovaného rozhrania.

Popisovaný spôsob využitia predstavuje odchýlku od konvenčného použitia služieb, kedy *Service* manipuluje primárne nad triedami Modelov (nie *ViewModel*-ov). V našom prípade je však tento prístup vhodnejší z viacerých dôvodov. Najvýznamnejším z nich je „skinny“ prístup k modelom a fakt, že drvivá väčšina logiky implementovaných programov je logikou vizualizačného charakteru – a teda by sa správne mala odohrávať nad dátami *ViewModelov*.

```
0 references
public WindowViewModel(IPhilosopherService philosopherService,
    ILoggingService loggingService, SimulationSettingsViewModel simulationSettings)
{
    _philosopherService = philosopherService;
    _loggingService = loggingService;

    Settings = simulationSettings;
    Philosophers = _philosopherService.GetPhilosophers();
    Forks = _philosopherService.GetForks();
    Logs = _loggingService.GetLogs();

    InitCommands();
}
```

Obr. 15. Konštruktor hlavného ViewModelu ukladajúci referencie na Service triedy poskytnuté prostredníctvom mechanizmu Dependency Injection

6 VŠEOBECNÁ ŠTRUKTÚRA

6.1 Adresárová štruktúra

6.1.1 Projekty

Konzistentná súborová štruktúra bola v projektoch praktickej časti zavedená z nutnosti čo najdôkladnejšieho rozdelenia logických celkov jednotlivých programov.

Projekty sú umiestnené v spoločnej zložke a interne sa na seba odkazujú pomocou referencií. Štruktúra rodičovskej zložky projektov je nasledovná:

Projekt IPCS: Prázdny rodičovský projekt (*Solution*), slúži ako prostriedok k jednoduchému súčasnému otvoreniu a managementu všetkých projektov podriadených

- **Zložka src**. Obsahom sú jednotlivé projekty reprezentujúce implementované ukážkové aplikácie.
    - **Projekt IPCS.Deadlock_Core**

      Obsahuje pomocný projekt so zdieľanou funkcionalitou.
    - **Projekt IPCS.Deadlock_Philosophers**

      Obsahuje ukážkový projekt „Stolujúci filozofi“.
    - **Projekt IPCS.Synchronisation_Bakery**

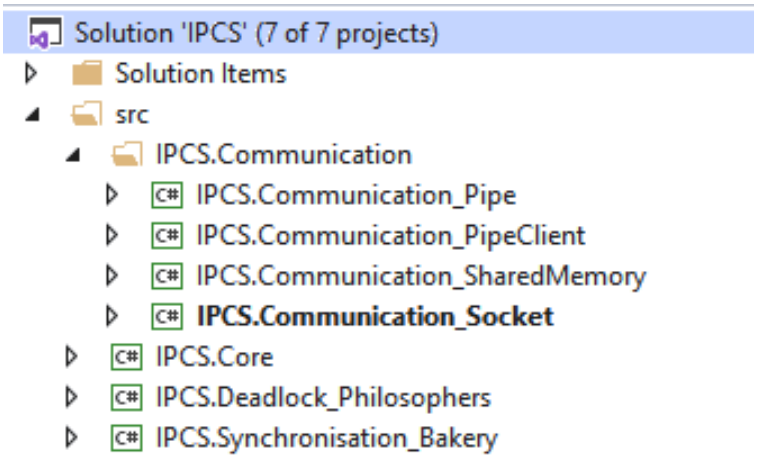
      Obsahuje ukážkový projekt „Producent - Konzument (pekáreň)“.
    - **Zložka IPCS.Communication**

      Obsahuje ukážkové projekty obsahujúce programy vypracované k demonštrácii použitia prostriedkov medziprocesovej komunikácie.
        - **Projekt IPCS.Communication_Pipe**

          Obsahuje projekt zachytávajúci komunikáciu prostredníctvom rúry.
        - **Projekt IPCS.Communication_SharedMemory**

          Obsahuje projekt zachytávajúci komunikáciu prostredníctvom zdieľanej pamäte
        - **Projekt IPCS.Communication.Socket**

          Obsahuje projekt zobrazujúci komunikáciu prostredníctvom socketov

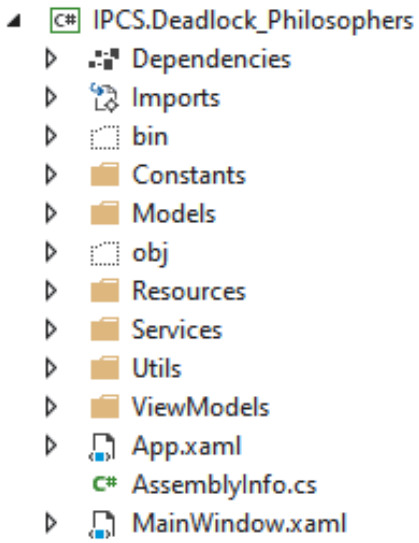


Obr. 16. Vonkajšia adresárová štruktúra

6.1.2 Štruktúra aplikácií

Súborová štruktúra v kontexte individuálnych projektov je vzhľadom na nutnosť vizualizácie činnosti programov a zvolenú architektúru aplikácie koncipovaná nasledovne:

- **Zložka Constants** : Obsahuje statické triedy. V nich sa nachádzajú nemenné konštanty používané naprieč aplikáciou – môže sa jednať o konštanty časové, číselné, ale napríklad aj o reťazce či slovníky, slúžiace primárne na spracovanie diania v bežiacom programe do užívateľsky čitateľnej formy z dôvodu vizualizácie.
- **Zložka Models**: Táto zložka v súlade s princípmi zvolenej architektúry MVVM obsahuje triedy Modelov.
- **Zložka Resources**: Obsahuje statické zdroje aplikácie. Prevažne sa jedná o obrázky rôznych formátov alebo pomocné súbory.
- **Zložka Services**: Nájdeme tu služby starajúce sa o primárnu logiku aplikácie spolu s nimi užívanou abstrakciou vo forme rozhrania.
- **Zložka Utils:** Obsahuje pomocné objekty, ktoré sa zaoberajú sekundárnou funkcionalitou programu – patria sem prevodníky, mapovacie triedy a profily objektov, či programová úprava a formátovanie textu.
- **Zložka Views**: Táto zložka v súlade s princípmi zvolenej architektúry MVVM obsahuje triedy *Views*.
- **Zložka ViewModels**: Táto zložka v súlade s princípmi zvolenej architektúry MVVM obsahuje triedy *ViewModel*ov.



Obr. 17. Vnútorná adresárová štruktúra projektu IPCS.Deadlock_Philosophers

6.1.3 Projekt Core

Priečinok obsahujúci projekt *IPCS.Core* (projekt typu *Class Library*) sa nachádza v odovzdaných zdrojových kódoch praktickej časti práce, na rovnakej úrovni ako priečinky určené jednotlivým ukážkovým aplikáciám. Projekt *IPCS.Core* neobsahuje žiadnu ucelenú funkcionalitu ani vizualizáciu – slúži ako priestor vyhradený pre triedy, konštanty, súbory a pomôcky, ktoré sa opakovane používajú naprieč viacerými implementovanými programami a teda by ich prítomnosť v každom projekte zvlášť viedla k zbytočnej repetícii už existujúceho kódu.

Primárny obsah projektu *IPCS.Core*:

- **ViewModelBase** a **ViewModelBaseDispatcher**: Abstraktné triedy implementujúce jednoduchú logiku súvisiacu s upovedomením *GUI* o zmene určitej vlastnosti, na ktorú sú jeho prvky naviazané (verzia bez a s využitím objektu *Dispatcher,* pre viac informácií viď. kapitola “Interaktivita”). Dedia od nich prakticky všetky ViewModely.
- **RelayCommand**: Trieda implementujúca rozhranie *ICommand*. Umožňuje naviazať na príkaz typu *RelayCommand* ľubovoľnú metódu bez nutnosti vytvárať pre každý príkaz vždy novú špecifickú triedu [72].

- **LoggingService**: Trieda *LoggingService* zodpovedá za dokumentáciu akcií prebiehajúcich v jednotlivých ukážkových programoch. Logy sú realizované pomocou správy kolekcie objektov typu Log – trieda samotná dokáže logy pridávať a čistiť. Z dôvodu vizualizácie kolekcie (vrátane nutnosti reagovať na jej zmenu) realizujeme operácie nad touto dátovou štruktúrou pomocou asynchrónneho volania k hlavnému vláknu, ktoré musí zmenu v kolekcii vykresliť do GUI.

```
88 references
public void WriteLog(string title, Enum action, string argument = null)
{
    var log = new Log()
    {
        Title = title,
        Action = new Tuple<Enum, string>(action, argument),
        StartTime = DateTime.Now
    };
    Application.Current.Dispatcher.BeginInvoke(new Action(() => Logs.Add(log)));
}
```

Obr. 18. Funkcia WriteLog() nachádzajúca sa v tele LoggingService, používa sa k pridaniu nového logu do kolekcie

6.2 Organizačné prostriedky

6.2.1 Menné priestory

Menné priestory (*Namespaces*) - za predpokladu, že sú správne využité, zvyšujú mieru zachovania prehľadnosti aj v rámci rozsiahlejších projektov. Logika tvorenia názvov menných priestorov je jednoduchá – všetky začínajú prefixom „IPCS“, ktorý značí názov spoločného rodičovského projektu (*solution*) všetkých implementovaných ukážok. Nasleduje bodka a za ňou reťazec jednoznačne identifikujúci projekt – teda napr. u stolujúcich filozofov „Deadlock_Philosophers“. Nasledujú bodkami oddelené vrstvy, ktorých názvy kopírujú jednotlivé úrovne zložiek, z ktorých pozostáva adresárová štruktúra projektu. Zmysluplné určenie pravidiel pre tvorenie a údržbu menných priestorov uľahčuje dôkladné rozdelenie častí projektu a umožňuje jednoduchú orientáciu napr. pri pridávaní referencie pomocou direktívy „*using*“.

```
1   using IPCS.Core.Infrastructure.Commands;
2   using IPCS.Core.Services.Logging;
3   using IPCS.Synchronisation_Bakery.Services.Bakery;
4   using IPCS.Synchronisation_Bakery.Services.Consumer;
5   using IPCS.Synchronisation_Bakery.Services.Producer;
6   using IPCS.Synchronisation_Bakery.SharedResources;
7   using IPCS.Synchronisation_Bakery.ViewModels;
8   using IPCS.Synchronisation_Bakery.ViewModels.Synchronisation;
9   using Ninject;
10  using System.Windows;
11  using System.Windows.Input;
```

Obr. 19. Odkazovanie menných priestorov na začiatku súboru pomocou direktívy „using“

6.3 Dependency Injection

Návrhový vzor *Dependency Injection* je implementáciou myšlienky *Inversion of Control* (*IoC*) [73]. Umožňuje poskytovať programovej triede inštancie ňou využívaných objektov (*dependencies*), štandardne pomocou konštruktoru už pri jej vytvorení.

Využitie *Dependency Injection* zvyčajne vedie k vizuálne čistejšiemu kódu a drobnej pamäťovej optimalizácii v dôsledku toho, že prakticky odpadá nutnosť (potenciálne opakovane) vytvárať triedou opätovne používané objekty priamo v kóde, keď môžeme využiť injektovanú premennú (vo forme *Property*).

```
0 references
public BakeryService(
    IProducerService producerService,
    IConsumerService consumerService,
    ILoggingService loggingService,
    SynchronisationToolViewModel synchronisationTool,
    SharedResource sharedResource)
{
    _producerService = producerService;
    _consumerService = consumerService;
    _loggingService = loggingService;
    SynchronisationTool = synchronisationTool;
    SharedResources = sharedResource;

    Running = CleanupRunning = false;
}
```

Obr. 20. Konštruktor triedy BakeryService. Všetky parametre konštruktoru sú poskytované prostredníctvom mechanizmu Dependency Injection

6.4 Rozhrania

Súbory projektu obsahujú mnoho tried (súborov s príponou „.cs“), ktorých jediným obsahom je *Interface* – rozhranie. Názvy týchto rozhraní (a zároveň im patriacich .cs súborov) v súlade s programátorskými dobrými návykmi začínajú predponou „I“ [74], teda napr. „ISomeService“.

Z definície je rozhranie súborom navzájom súvisiacich funkcionalít, ktoré by mala pokrývať každá neabstraktná trieda či štruktúra, ktorá toto rozhranie implementuje. Dôvodov, prečo používať rozhrania je mnoho [75]. Z hľadiska aplikácií implementovaných v praktickej časti rozhrania využívame kvôli nasledovným výhodám:

- Rozhrania poskytujú dodatočnú vrstvu abstrakcie a generický základ pre odvodené triedy. Táto vlastnosť je užitočná v prípadoch, keď existuje viac tried odvodených od určitého rozhrania, nad ktorými by sme mali byť schopní vykonávať danú sadu operácií definovaných rozhraním bez ohľadu na internú implementáciu triedy samotnej.
- Pristupovaním k funkcionalite triedy pomocou generiky poskytovanej rozhraním umožňujeme kvalitnejšie zapuzdrenie internej logiky danej triedy a znižujeme riziko zásahu do vlastností triedy nepredpokladaným či nechceným spôsobom.

7 TYPOVÉ ÚLOHY

7.1 Stolujúci filozofovia – uviaznutie

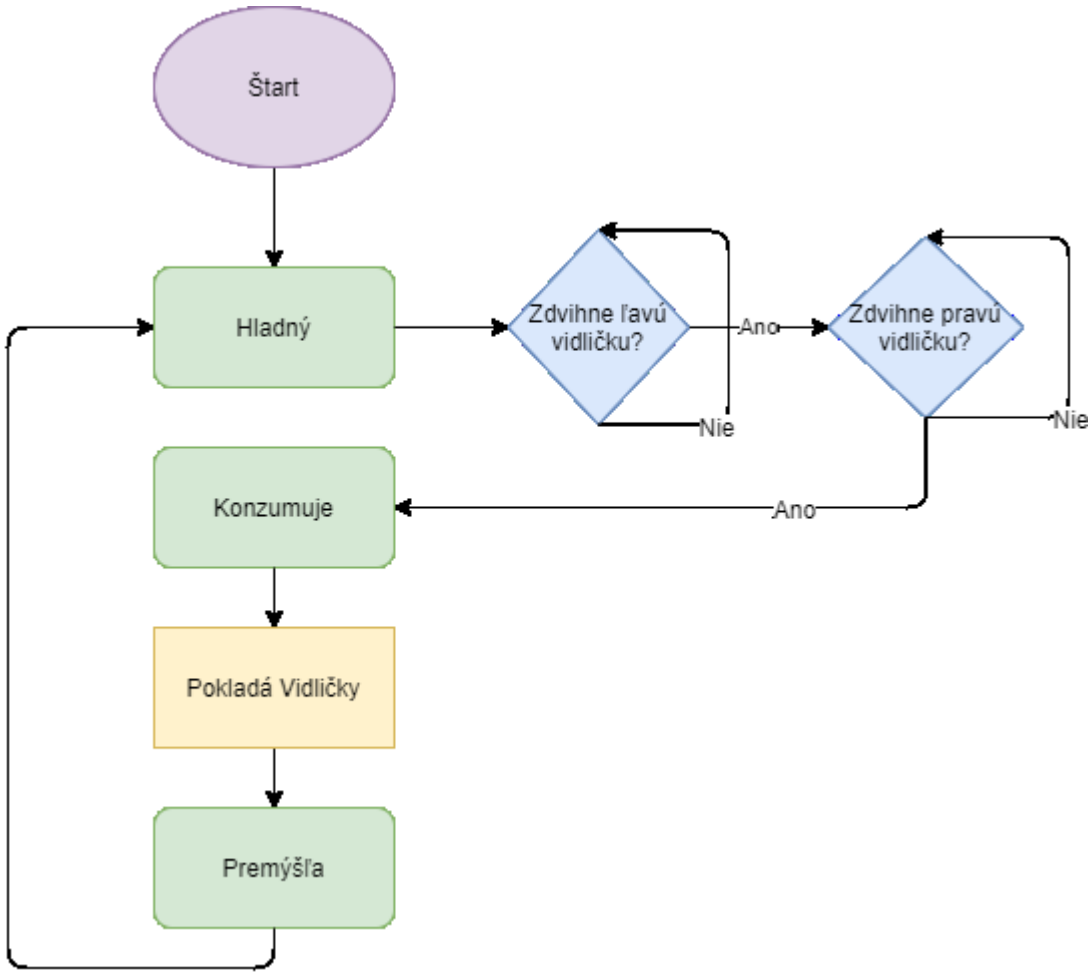
7.1.1 Popis

Ukážkový program "Stolujúci filozofovia" sa pokúša o vizualizáciu rovnomenného problému zameraného na úskalia súbehu viacerých nezávislých procesov, resp. vlákien pri prístupe k zdieľanému zdroju.

Myšlienka problému je pomerne jednoduchá: Dejiskom problému je okrúhly jedálenský stôl, za ktorým je usadených 5 filozofov. Každý z týchto filozofov má pred sebou tanier s jedlom a medzi každou dvojicou filozofov je umiestnená jedna vidlička (vzhľadom na okrúhly tvar stola má každý filozof jednu vidličku po ľavej aj po pravej ruke a na stole sa teda celkovo nachádza rovnaký počet vidličiek, ako filozofov).

Každý filozof v ľubovoľnom momente vykonáva jednu z troch činností – buď konzumuje, hladuje alebo premýšľa. Hladujúci filozof sa snaží najesť - aby však mohol začať konzumovať, musí súčasne držať obe vidličky v jeho dosahu. Zdvihnutie vidličiek je akcia sekvenčnej povahy s pevne daným poradím – filozof sa najskôr pokúsi zdvihnúť ľavú vidličku a pokiaľ uspeje, pokúsi sa zdvihnúť pravú vidličku. Filozof počas zdvíhania pravej vidličky stále drží vidličku ľavú – a nepustí ju ani v prípade, že pokus o zdvihnutie vidličky po pravej ruke skončí neúspechom a bude nútený ho opakovať.
Neúspešný pokus o zdvihnutie vidličky (napr. v prípade, ak ju už vlastní iný filozof) vyústi do krátkeho čakania a následného opakovanie tohto pokusu. Filozof začne konzumovať až keď obe vidličky úspešne zdvihne.

Konzumácia prebieha istý čas, potom filozof prestane jesť, položí vidličky a začne premýšľať. Fáza premýšľania taktiež netrvá večne – filozof po čase opäť vyhladne a celý proces sa opakuje. Pre potreby ukážkovej aplikácie predpokladáme, že tento cyklus je nekonečný.



Obr. 21. Vývojový diagram algoritmu akcií filozofov

Implementovaný program demonštruje, ako môže nevhodná súhra okolností (v našom prípade načasovania) viesť k uviaznutiu (pre lepšie pochopenie príčin a dôsledkov uviaznutia slúži kapitola „Uviaznutie. Procesy sú analogicky reprezentované množinou bežiacich rutín jednotlivých filozofov. Zdieľaný zdroj o ktorý súperia sú vidličky rozmiestnené medzi jednotlivými filozofmi.

7.1.2 Implementácia

Dáta na hlavnú obrazovku aplikácie sú premietané za pomoci triedy *WindowViewModel*. Táto trieda nie je sama o sebe „čistý“ *ViewModel*, slúži však ako kontajner obsahujúci ostatné *ViewModel*y a kolekcie, ktoré následne môžeme pomocou mechanizmu *Data Binding* jednoducho vizualizovať (Pre objasnenie podrobností o tomto mechanizme slúži kapitola „Binding“). *WindowViewModel* taktiež obsahuje príkazy (*Commands*), ktoré umožňujú ovládanie internej logiky aplikácie pomocou tlačidiel grafického rozhrania.

```
#region Properties

///<value>[I_PHILOSOPHER_SERVICE] - Service used for main philosopher simulation </value>
private readonly IPhilosopherService _philosopherService;

///<value>[I_LOGGING_SERVICE] - Service used for creating and manipulation of log collection</value>
private readonly ILoggingService _loggingService;

///<value>[OBSERVABLE_COLLECTION<PHILOSOPHER_VIEW_MODEL>] - Collection holding ViewModels of philosophers </value>
1 reference
public ObservableCollection<PhilosopherViewModel> Philosophers { get; private set; }

///<value>[OBSERVABLE_COLLECTION<FORK_VIEW_MODEL>] - Collection holding ViewModels of forks</value>
1 reference
public ObservableCollection<ForkViewModel> Forks { get; private set; }

///<value>[OBSERVABLE_COLLECTION<LOG>] - Collection holding current Logs</value>
1 reference
public ObservableCollection<Log> Logs { get; private set; }

///<value>[SIMULATION_SETTINGS_VIEW_MODEL] - ViewModel of general settings concerning simulation process and state</value>
3 references
public SimulationSettingsViewModel Settings { get; private set; }

///<value>[I_COMMAND] - Command bound to certain View control - used to start or stop simulation </value>
1 reference
public ICommand StartStopCommand { get; private set; }

///<value>[I_COMMAND] - Command bound to certain View control - used to pause or resume running simulation </value>
1 reference
public ICommand PauseResumeCommand { get; private set; }

///<value>[I_COMMAND] - Command bound to certain View control - used to start all threads at once </value>
1 reference
public ICommand StartAllCommand { get; private set; }

#endregion
```

Obr. 22. Vlastnosti triedy WindowViewModel – určené k vizualizácii pomocou previazania na prvky grafického užívateľské rozhrania a k delegovaniu funkcionality na príslušné služby

O pokrytie hlbšej logiky sa starajú tzv. *Services* – v tomto programe využívame iba dve dodatočné services – *PhilosopherService* a *LoggingService*, pričom implementácia *LoggingService* je už poskytovaná v rámci zdieľaného projektu *IPCS.Core*.

7.1.2.1 PhilosopherService

Trieda *PhilosopherService* má na starosti rutinu filozofov, ich interakciu s vidličkami a operácie nad vláknami reprezentujúcimi jednotlivé bežiace rutiny filozofov.

O spustenie vlákien filozofov sa starajú funkcie „*Run()*“ a „*RunAll()*“. Je medzi nimi jediný rozdiel – „*RunAll()*“ spôsobuje uviaznutie. Obidve vyvolajú určité inicializačné nastavenia a pokračujú spustením rutiny filozofov.

Spustenie rutiny zabezpečuje funkcia „*PhilosophersStartRoutine()*“ s určujúcim, či majú byť vytvorené podmienky pre uviaznutie (k uviaznutiu môže dôjsť aj za predpokladu, že je tento parameter nepravdivý, avšak pravdepodobnosť že sa tak stane je omnoho nižšia).

```
2 references
private void PhilosophersStartRoutine(bool simultaneous = false)
{
    foreach (var philosopher in Philosophers)
    {
        var task = new Task(() => { PhilosopherRoutine(philosopher, simultaneous); });
        PhilosopherTasks.Add(task);
    }
    foreach (var task in PhilosopherTasks)
    {
        task.Start();
    }
}
```

Obr. 23. Vytvorenie a spustenie činnosti filozofov vo forme asynchrónnych operácií s využitím triedy Task

Nasleduje spustenie samotnej rutiny filozofa v samostatnom vlákne. Kód bol štruktúrovaný a rozdelený tak, aby čo najvernejšie kopíroval slovný popis algoritmu (viď. vyššie).

```
1 reference
private void PhilosopherRoutine(PhilosopherViewModel philosopher, bool simultaneous)
{
    _loggingService.WriteLog(philosopher.Name, ActionType.STARTING);
    if (simultaneous){ WaitForAction(false, false); }
    else { WaitForAction(true, true); }

    while (!_token.IsCancellationRequested)
    {
        switch (philosopher.State)
        {
            case PhilosopherState.HUNGRY:
                PickForks(philosopher);
                break;

            case PhilosopherState.EATING:
                Eat(philosopher);
                PutForks(philosopher);
                break;

            case PhilosopherState.THINKING:
                Think(philosopher);
                break;
        }
    }
    _loggingService.WriteLog(philosopher.Name, ActionType.FINISHING);
}
```

Obr. 24. Rutina vykonávaný všetkými vláknami reprezentujúcimi jednotlivých filozofov

Vyvolanie uviaznutia:

Uviaznutia docielime podobne, ako pri štandardnom spustení – riešenie vo vzorovom programe sa zakladá na predpoklade, že keď je odobratá akákoľvek náhodnosť z čakacej doby po spustení filozofovej rutiny, budú všetky ľavé vidličky zdvihnuté približne v rovnakom čase. Po vykonaní tejto akcie filozofovia (ako aj pri klasickom "postupnom" spustení) istý čas čakajú, kým sa pokúsia zdvihnúť ďalšiu vidličku – tým sa zaručí, že nedôjde k situácii, kedy by jeden z nich stihol zdvihnúť ľavú aj pravú vidličku.

Vlákna samotné vykonávajú rutinu až do bodu, kedy nie je signalizované ich ukončenie prostredníctvom štruktúry nazývanej *Cancellation Token*. Musia teda periodicky kontrolovať, či k tomuto ukončeniu nedošlo.

Popísaná simulácia uviaznutia nie je perfektná, ale ide o stvárnenie veľmi podobné realite, kedy sa požiadavky jednotlivých procesov o pridelenie zdieľaného zdroja musia stretnúť v úzkom časovom intervale.

Riešenie uviaznutia:

Prístupov vhodných k riešeniu uviaznutia je mnoho (podrobnejšie sú popísané v kapitole tejto práce s názvom „Uviaznutie"). V priloženej ukážkovej aplikácii je riešenie implementované formou zotavenia z uviaznutia za pomoci intervencie od užívateľa programu (stisnutím tlačidla „*Stop*"). Zotavenie samotné je realizované pomerne jednoduchým, avšak veľmi dobre zobraziteľným spôsobom – násilným ukončením vlákien oprávnenou autoritou a uvoľnením zdrojov, ktoré im boli pridelené. Programovo ukončenie vlákien implementujeme funkciou „*Stop All()*".

```
/// <summary>
/// Function serving to Stop all currently executed philosopher tasks and perform cleanup
/// </summary>
1 reference
public void StopAll()
{
    CleanupRunning = true;
    _tokenSource.Cancel();
    Task.WhenAll(PhilosopherTasks.ToArray()).ContinueWith((antecedent) => { Cleanup(); },
        TaskContinuationOptions.OnlyOnRanToCompletion);
    CleanupRunning = false;
}
```

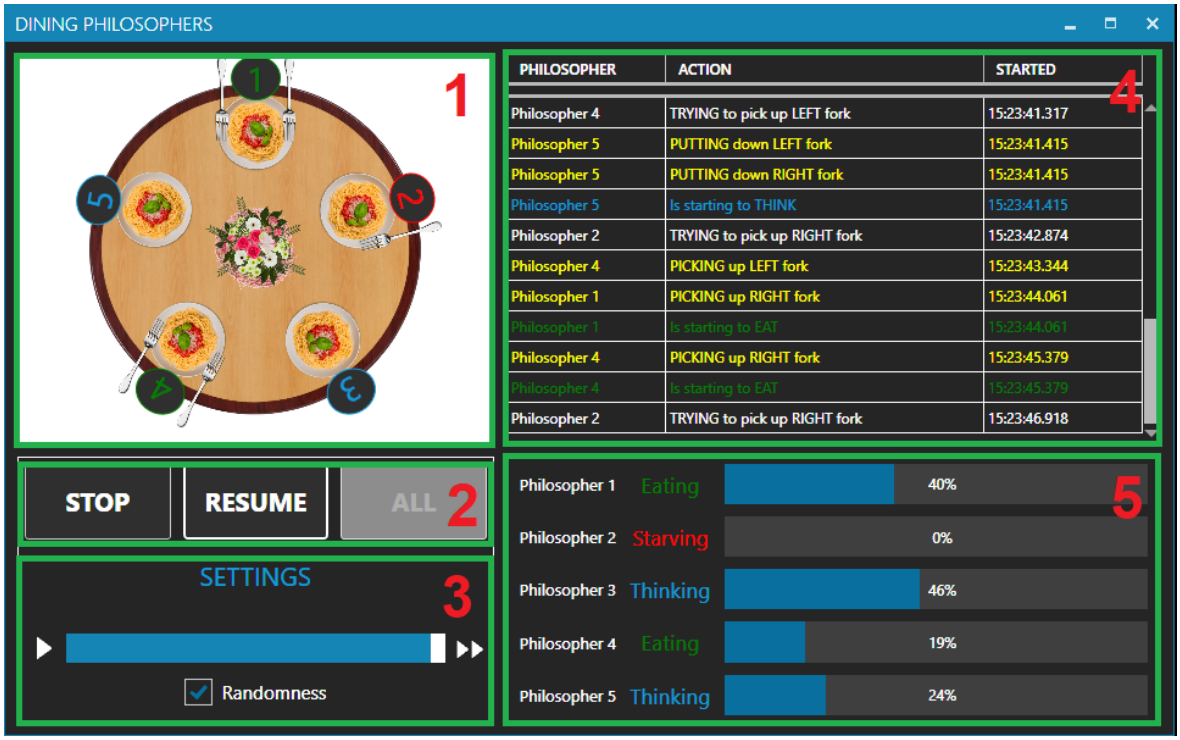
Obr. 25. Funkcia „StopAll()" zodpovedná za ukončenie činnosti všetkých spustených vlákien filozofov a management čistenia stavu aplikácie.

V tele funkcie je vyžiadaná zmena stavu štruktúry *Cancellation Token* na ukončený stav. Následné čakanie a ukončenie všetkých spustených vlákien pre rutiny filozofov je zavŕšené čistkou stavu aplikácie (progres filozofov, vymazanie logov, príznaky atď.).

Okrem vyššie spomenutých mechanizmov a tried projekt obsahuje veľké množstvo pomocných tried a konštruktov zabezpečujúcich hladký priebeh vizualizácie – jedná sa prevažne o konvertory, súbory s konštantami a slovníkmi pre prevod rôznych dátových typov na ľudsky čitateľný tvar.

7.1.3 Vizualizácia

Vizualizácia je realizovaná pomocou jediného GUI okna rozdeleného na niekoľko hlavných sekcií:



Obr. 26. Grafické užívateľské rozhranie programu zachytávajúceho stolujúcich filozofov

1) **Situácia pri stole:**

   Grafické znázornenie piatich filozofov a ich pozícii pri jedálenskom stole vrátane vizualizácie momentálneho rozdelenia vidličiek medzi jednotlivých filozofov.

2) **Ovládací panel**:

   Ovládací panel obsahuje tri tlačidlá slúžiace pre ovládanie aplikácie. Tlačidlá „*Start*“ pre spustenie v klasickom režime, tlačidlo „*All*“ pre simultánne spustenie (vyvolá uviaznutie) a tlačidlo „*Stop*“ pre ukončenie činnosti vlákien na pozadí.

3) **Ovládacie prvky**:

   Posuvník pre rýchlosť aplikácie (ovplyvňuje rýchlosť vykonávania akcií filozofov a zároveň dĺžku čakacích intervalov medzi pokusom o zmenu stavu, pokiaľ sa filozof pokúša o zdvihnutie vidličiek) a zaškrtávacie pole pre povolenie „náhodného prvku“ v rýchlostiach (pričítanie náhodnej percentáže trvania nasledujúcej akcie z intervalu {-35% ; 35%}).

4) **Výpis logov:**

   Zaznamenáva akcie jednotlivých filozofov. Zobrazované parametre sú meno filozofa, akcia ktorú vykonáva (alebo sa pokúša ju vykonať) a čas vo formáte „HH:mm:ss.FFF“.

5) **Prehľad aktivít:**

   Aktivita, ktorú každý z filozofov momentálne vykonáva vrátane kontrolky typu *ProgressBar* znázorňujúcej, aký je jeho percentuálny postup v porovnaní s celkovým časom určeným na túto aktivitu.

7.1.4 Zhrnutie úlohy

Program vizualizujúci problém stolujúcich filozofov dobre uchopiteľným a názorným spôsobom demonštruje okolnosti a dôsledky uviaznutia s možnosťou jednoduchej analógie vykresľovanej metaforickej situácie k skutočným príčinám / aktérom v kontexte informačných technológií. Program umožňuje užívateľovi zasiahnuť do simulácie a zrýchliť / pozastaviť / prerušiť jej chod prostredníctvom sady funkcionalít zasadenej do zrozumiteľného grafického užívateľského rozhrania. Cieľom aplikácie je poskytnúť čo najvernejšiu interaktívnu reprezentáciu uviaznutia vrátane možnosti riešiť situáciu „hrubou silou“, a síce pomocou resetu zúčastnených aktérov.

7.2 Producent / Konzument - synchronizácia

7.2.1 Popis

Problém producenta a konzumenta (alebo aj tzv. *Bounded Buffer*) je klasickým príkladom demonštrujúcim potenciálne dôsledky neošetreného súbežného prístupu viacerých výpočtových vlákien k jedinému zdieľanému zdroju a možnosti, ako danej situácii predchádzať.

Súhrnný názov Producent-Konzument pomenúva samotný typ a podstatu problému. Existuje veľké množstvo analógií z reálneho sveta, ktoré stavajú na problematike producenta a konzumenta, avšak približujú ju pomocou priemetu do dobre uchopiteľnej oblasti bežného života – takýmito úlohami sú napr. „Populárny pekár“, „Spiaci holič“ [76], či priemet problému do oblasti streamovania multimediálneho obsahu [77]. Pre implementovaný ukážkový program bola použitá šablóna úlohy „Populárny pekár“.

Zhotovená aplikácia slúži k vizualizácii synchronizačného problému, ktorý bol popísaný (mimo iné) ako analógia práve k obehovému mechanizmu pečiva v pekárni, pričom producentom je pekár a konzumentom zákazník. Buffer je v tomto prípade reprezentovaný naskladnenými zásobami hotového pečiva.

Myšlienka naivného algoritmu je nasledovná [78]:

- Producent opakovanie produkuje nové položky a vkladá ich do bufferu
- Konzument opakovane položky z bufferu opakovane vyberá a postupne spracúva

Nedostatok naivného algoritmu spočíva v skutočnosti, že sled operácií vykonávaných nad bufferom (postupnosť čítanie počtu predmetov –> vklad / výber predmetu –> zvyšovanie / znižovanie počtu predmetov) v žiadnom prípade nie je atomický a kedykoľvek môže dôjsť k prerušeniu procesu vykonávajúceho túto postupnosť aktivít ešte pred jej dokončením [2]. Prehádzanie poradia jednotlivých operácií vykonávaných producentom / konzumentom napr. vplyvom prerušenia jedného z procesov môže mať rôzne následky – napr. výber predmetu z prázdneho bufferu konzumentom, či snahu o pridanie do plného bufferu producentom. Prerušenie (v potenciálne rizikových momentoch) je možné nasimulovať pomocou ovládacích prvkov umiestnených v GUI aplikácie (jednotlivé ovládacie prvky sú bližšie popísané v sekcii „Vizualizácia“ priradenej k práve diskutovanej aplikácii).

Implementovaný program umožňuje navodiť situáciu, v ktorej dôjde k nepredpokladanému správaniu naivného algoritmu vplyvom neočakávanej zmeny poradia jednotlivých akcií vykonávaných konzumentom / producentom. Vzorový program zároveň obsahuje alternatívnu formu algoritmov pre producenta a konzumenta využívajúcu synchronizačné prostriedky (Semafor, Monitor) schopné slúžiť ako prevencia nechcenej situácie.

7.2.2 Implementácia

Podobne ako v predchádzajúcom programe (viď. kapitola „Stolujúci filozofovia - Uviaznutie“) prebieha previazanie vizualizácie a podkladovej logiky aplikácie pomocou triedy *WindowViewModel*. Táto trieda je zodpovedná za udržiavanie samotného buffer-u (chlebníka) a referencií na všetky dáta a podkladové triedy, ktoré musíme vizualizovať, či k nim prostredníctvom GUI chceme pristupovať. *WindowViewModel* ďalej obsahuje objekty implementujúce rozhranie *ICommand* slúžiace primárne k manipulácii behu aplikácie a riadeniu akcií konzumenta / producenta.

7.2.2.1 BakeryService

BakeryService je trieda implementujúca rozhranie *IBakeryService*. Trieda zodpovedá za sprostredkovanie operácií nad producentom a konzumentom a prepojenie hlbšej logiky na štruktúry obsiahnuté v triede *WindowViewModel*.

BakeryService je štrukturálne jednoduchá „sprostredkovateľská“ trieda – najdôležitejšie v nej obsiahnuté metódy sú „*Run()“* na spustenie rutiny producenta a konzumenta, „*StopExecution()“* na jej pozastavenie a „*Cleanup()“* na obnovenie premenných a prostredia do pôvodného stavu (napr. po úspešnom volaní „*StopExecution()“).*

```
#region Commands

/// <summary>
/// Command-induced function - Runs new simulation
/// </summary>
/// <param name="param">RelayCommand-supplied optional parameter</param>
2 references
public void Run(object param)...

/// <summary>
/// Command-induced function - Stops current simulation
/// </summary>
/// <param name="param">RelayCommand-supplied optional parameter</param>
2 references
public void StopExecution(object param)...

/// <summary>
/// Command-induced function - Pauses / Resumes consumer activity
/// </summary>
/// <param name="obj">RelayCommand-supplied optional parameter</param>
2 references
public void ChangeInterruptStateConsumer(object obj) => _consumerService.ChangeInterruptState();

/// <summary>
/// Command-induced function - Pauses / Resumes producer activity
/// </summary>
/// <param name="obj">RelayCommand-supplied optional parameter</param>
2 references
public void ChangeInterruptStateProducer(object obj) => _producerService.ChangeInterruptState();

#endregion
```

Obr. 27. Hlavná funkcionalita sprostredkovaná metódami triedy „BakeryService“

7.2.2.2 ConsumerService

ConsumerService je trieda implementujúca rozhranie *IConsumerService*. Jedna z dvoch hlavných tried programu pekárne, obsahuje táto trieda logiku rutiny vykonávanej konzumentom a implementáciu všetkých podporovaných variant riešenia.

Funkciami triedy *ConsumerService* s najvyššou váhou sú metódy „*StartConsuming()* „ a „*StopConsuming()“,* ktoré zabezpečujú spustenie / ukončenie rutiny konzumenta.

Algoritmus realizovaný konzumentom a jeho forma sa nachádza v metódach „*ConsumingNaive()“, „ConsumingSemaphore()“* a „*ConsumingMonitor()“,* pričom výber korektného prebehne pri spustení rutiny konzumenta na základe zvolenej (resp. nezvolenej) metódy synchronizácie.

7.2.2.3 ProducerService

ProducerService je trieda implementujúca rozhranie *IProducerService*. Obsahom triedy je logika riadiaca činnosť producenta, vrátane implementácie všetkých implementovaných variant riešenia.

Analogicky ku *ConsumerService*, nájdeme aj v *ProducerService* funkcie riadiace samotný beh producenta – „*ProducingNaive()*“, „*ProducingSemaphore()*“ a „*ProducingMonitor()*“. Výber vhodnej funkcie je realizovaný taktiež zhodným spôsobom – podľa zvoleného synchronizačného prostriedku.

Synchronizácia

Naivný algoritmus riadiaci beh konzumenta a producenta (aj keď na prvý pohľad korektný) nemusí vždy pracovať podľa očakávaní – ak uvažujeme pekáreň ako analógiu k výpočtovým vláknam procesoru s možnosťou zdieľať určitý zdroj (buffer), je nutné vziať do úvahy možnosť prerušenia vlákna operačným systémom. Možnosť prerušenia je v ukážkovom programe poskytnutá vo forme tlačidla na pozastavenie konzumenta / producenta, pričom správanie danej entity po stisku tlačidla je porovnateľné so správaním prerušeného výpočtového vlákna - t.j. dokončenie práve prebiehajúcej atomickej operácie a následné pozastavenie činnosti až do momentu, kedy nebude činnosť vlákna znovu obnovená.

Naivná implementácia algoritmu konzumpcie a produkcie je nasledovná:

```
1 reference
public void ConsumingNaive(SynchronisationToolViewModel synchronisationTool)
{
    _loggingService.WriteLog(LogConstants.ConsumerName, ActionType.BEGINS_NAIVE_ALGORITHM_CONSUMING);

    while (!_token.IsCancellationRequested)
    {
        if (synchronisationTool.Naive.ConsumerWoke)
        {
            if (ReadBufferCount() == 0)
            {
                ConsumerSleep(synchronisationTool);
                continue;
            }

            TakeItemFromBuffer();
            DecrementCounter();

            if (ReadBufferCount() == FunctionalityRelatedConstants.BufferSize - 1)
            {
                WakeUpProducer(synchronisationTool);
            }
            ConsumeItem();
        }
    }
    _loggingService.WriteLog(LogConstants.ConsumerName, ActionType.ENDING);
}
```

Obr. 28. Algoritmus vykonávaný konzumentom pri naivnej forme riešenia

```
1 reference
private void ProducingNaive(SynchronisationToolViewModel synchronisationTool)
{
    _loggingService.WriteLog(LogConstants.ProducerName, ActionType.BEGINS_NAIVE_ALGORITHM_PRODUCING);
    while (!_token.IsCancellationRequested)
    {
        if (synchronisationTool.Naive.ProducerWoke)
        {
            var newBread = ProduceItem();
            if (ReadBufferCount() == FunctionalityRelatedConstants.BufferSize)
            {
                ProducerSleep(synchronisationTool);
                continue;
            }

            AddItemToBuffer(newBread);
            IncrementCounter();

            if (ReadBufferCount() == 1)
            {
                WakeUpConsumer(synchronisationTool);
            }
        }
    }
    _loggingService.WriteLog(LogConstants.ProducerName, ActionType.ENDING);
}
```

Obr. 29. Algoritmus vykonávaný producentom pri naivnej forme riešenia

Problémovou situáciou je pri využití naivného algoritmu napríklad nasledujúca sekvencia akcií, ktorá vedie k súčasnému a definitívnemu uspaniu producenta aj konzumenta [1]:

- 1) Konzument prečítal počet položiek v bufferi
- 2) Konzument je prerušený (ešte predtým, ako sa stihol sám uspať)
- 3) Producent vytvorí predmet a vloží ho do Bufferu a počet predmetov v bufferi
- 4) Producent budí konzumenta (volanie sa stratí, keďže konzument nespí)
- 5) Producent pokračuje v plnení buffer-u a po jeho naplnení zaspáva
- 6) Konzument je opäť spustený a keďže jeho poslednou činnosťou bolo prečítanie počtu položiek v bufferi (a keďže v dobe čítania hodnoty bol buffer prázdny), uspáva sám seba a už nikdy sa nezobudí.

Vzorový program implementuje riešenie problémovej situácie pomocou určenia rozsahu kritickej sekcie a následným využitím pred-implementovaných synchronizačných prostriedkov (Semafor, Monitor) takým spôsobom, aby sa v kritickej sekcii nikdy nevyskytoval producent aj konzument zároveň (a to ani v prípade neočakávaného prerušenia), čím zaručíme vzájomnú výlučnosť v rámci kritickej sekcie a teda jej vykonanie formou veľkej atomickej operácie bez rizika poškodenia logickej integrity zdieľaných dát

vplyvom súbežného prístupu (dôvody, prečo je nutné zabezpečiť atomicitu operácií popisuje kapitola „Kritická Sekcia“) [79].

```
1 reference
private void ConsumingSemaphore(SynchronisationToolViewModel synchronisationTool)
{
    _loggingService.WriteLog(LogConstants.ConsumerName, ActionType.BEGINS_SEMAPHORE_ALGORITHM_CONSUMING);
    while (!_token.IsCancellationRequested)
    {
        SemaphoreWait(synchronisationTool.Semaphores.FillCount);
        SemaphoreWait(synchronisationTool.Semaphores.Mutex, true);

        TakeItemFromBuffer();
        DecrementCounter();

        SemaphoreSignal(synchronisationTool.Semaphores.Mutex);
        SemaphoreSignal(synchronisationTool.Semaphores.EmptyCount);
        ConsumeItem();
    }
    _loggingService.WriteLog(LogConstants.ConsumerName, ActionType.ENDING);
}
```

Obr. 30. Algoritmus vykonávaný konzumentom pri forme riešenia synchronizovanej prostredníctvom semaforov

```
1 reference
private void ProducingSemaphore(SynchronisationToolViewModel synchronisationTool)
{
    _loggingService.WriteLog(LogConstants.ProducerName, ActionType.BEGINS_SEMAPHORE_ALGORITHM_PRODUCING);
    while (!_token.IsCancellationRequested)
    {
        var newBread = ProduceItem();

        SemaphoreWait(synchronisationTool.Semaphores.EmptyCount);
        SemaphoreWait(synchronisationTool.Semaphores.Mutex, true);

        AddItemToBuffer(newBread);
        IncrementCounter();

        SemaphoreSignal(synchronisationTool.Semaphores.Mutex);
        SemaphoreSignal(synchronisationTool.Semaphores.FillCount);

    }
    _loggingService.WriteLog(LogConstants.ProducerName, ActionType.ENDING);
}
```

Obr. 31. Algoritmus vykonávaný producentom pri forme riešenia synchronizovanej prostredníctvom semaforov

7.2.2.4 Synchronizačné mechanizmy v .NET Core

Ukážkový program “Populárny pekár” zhotovený v rámci praktickej časti tejto práce demonštruje využitie dvoch typov synchronizačných mechanizmov ktoré poskytuje *.NET Core Framework.*

- **Monitor**

  Monitory v *.NET Core* sú implementované vo forme kombinácie viacerých jazykových špecifík. Konštrukt monitoru je realizovaný pomocou statickej triedy *Monitor*.
  Samotné *Condition Variables* je možné vytvoriť ako premenné základného typu *object*, ktorými synchronizujeme prístup do kritickej sekcie. Tieto premenné môžu byť uzamknuté pomocou kľúčového slova *lock* [80], čo vedie k vzniku synchronizovaných blokov kódu. Z týchto blokov je následne bezpečné volať statické funkcie triedy Monitor.

  Implementácia statickej triedy Monitoru ponúka (mimo iné) nasledujúce metódy [81]:

  - *Enter()*

    Pomocou operácie *Enter()* volajúce vlákno získava zámok na objekte a pokúša sa o vstup do kritickej sekcie. Po úspešnom vstupe do kritickej sekcie pomocou volania *Enter()* nesmie do kritickej sekcie vstúpiť žiadne iné vlákno až do doby, kým nebude kritická sekcia opäť uvoľnená.

  - *Wait()*

    Uvoľní zámok na objekte a dovolí ostatným vláknam pristupovať k zdieľaným dátam objektu, pričom vlákno z ktorého bola metóda *Wait()* zavolaná čaká. Na prebudenie čakajúcich vlákien sa používa funkcia *Pulse()*, ktorá by ich správne mala notifikovať o zmene stavu objektu.

  - *Pulse()*

    Posiela signál vláknam očakávajúcim uvoľnenie zámku na objekte. Operácia *Pulse()* po správnosti znamená, že stav objektu bol nejakým spôsobom modifikovaný a vlákno momentálne držiace zámok je pripravené ho uvoľniť.

  - *Exit()*

    Uvoľňuje zámok na objekte a naznačuje koniec kritickej sekcie chránenej týmto zámkom.

- **Semafor**

  Semafor ako objekt v *.NET Core* sa riadi princípmi semaforu bližšie popísanými v teoretickej časti práce (kapitola „Synchronizačné Prostriedky“).

  API Semaforu definuje mimo iné niekoľko základných operácií [82]:

  - *Semaphore(int, int):*

    Konštruktor, pomocou celočíselných argumentov umožňuje špecifikovať počiatočný stav semaforu (t.j. počet signálov uložených v „zásobe“) a maximálny počet vlákien súbežne sa nachádzajúcich v kritickej sekcii.

  - *WaitOne():*

    Zablokuje vlákno pomocou ktorého bola metóda zavolaná až do doby, kedy semafor neobdrží príslušný signál (štandardne dovtedy, kým nejaké iné vlákno neuvoľní semafor).

  - *Release():*

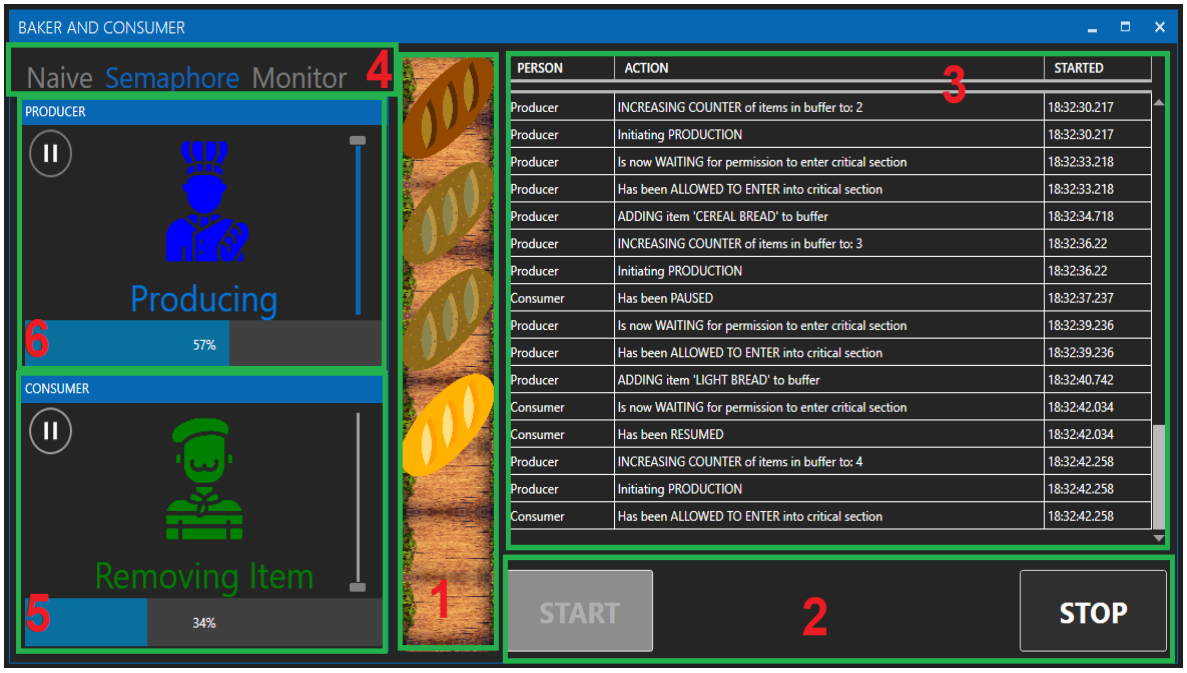
    Uvoľňuje semafor a navyšuje počet interného čítača (pokiaľ na semafore čakajú ďalšie vlákna, je táto operácia analogická s „vpustením“ niektorého z čakajúcich vlákien do kritickej sekcie chránenej semaforom).

Všetky zmieňované metódy semaforu majú pochopiteľne ešte ďalšie verzie (kontrola stavu semaforu bez nutnosti blokovania procesu, špecifikácia maximálnej čakacej doby a pod., avšak tieto modifikácie neboli pri vytváraní ukážkového programu použité.

Pre viac informácií ohľadom využitých synchronizačných primitív a ich použitia najlepšie poslúži oficiálna dokumentácia firmy Microsoft [82].

7.2.3 Vizualizácia

Vizuálna realizácia ukážkového príkladu Populárny pekár je zhotovená pomocou grafického užívateľského rozhrania obsahujúceho pevné zobrazovacie prvky (buffer – „chlebáreň“, výpis logov...), rovnako ako prepínače či elementy slúžiace k manipulácii programu za behu (tlačidlá „*Start*“ / „*Stop*“, posuvníky na zmenu rýchlosti činnosti, prepínače aktuálneho spôsobu riešenia...).



Obr. 32. Grafické užívateľské rozhranie programu zachytávajúceho problém producenta a konzumenta

1) **Buffer:**

   Buffer slúži k ukladaniu bochníkov chleba, ktoré producent – pekár do bufferu vkladá a konzument – zákazník z bufferu vyberá. Pozor - Počet objektov v bufferi ukazuje, koľko objektov sa vo fronte skutočne nachádza (nemusí nutne korešpondovať s množstvom, nad ktorým momentálne operuje producent / konzument.

2) **Ovládací panel:**

   Jednoduchý ovládací panel pozostáva z tlačidiel „*Start*“ a „*Stop*“, ovládajúcich beh aplikácie. Tlačidlo „*Start*“ spustí program s využitím momentálne zvoleného synchronizačného prostriedku (viď. „Prepínač Synchronizačných Prostriedkov“), tlačidlo „*Stop*“ bežiaci program preruší a resetuje interné mechanizmy do počiatočného stavu.

3) **Výpis logov:**

   Zaznamenáva akcie producenta a konzumenta. Zobrazované parametre sú meno osoby ktorá akciu vykonala, popis akcie ktorú vykonáva a čas vo formáte „HH:mm:ss.FFF“.

4) **Prepínač synchronizačných prostriedkov:**

   Implementovaný pomocou kontrolky *TabControl*, slúži k zmene synchronizačného mechanizmu ktorý bude použitý pri nasledujúcom stisku tlačidla „*Start*“. (t.j. aplikácia musí byť zastavená a znovu spustená pomocou sekvencie tlačidiel „Stop“ + „*Start*“, aby sa zmeny v synchronizačnom mechanizme programu uplatnili).

5) **Okno konzumenta:**

   Okno popisujúce konzumenta (zákazníka). Zobrazuje momentálne vykonávanú aktivitu a súvisiacu kontrolku typu *ProgressBar* znázorňujúcu percentuálnu mieru finalizácie tejto činnosti. Posuvník v okne konzumenta umožňuje manipuláciu rýchlosťou akcií, ktoré je schopný vykonávať. Tlačidlo vo forme ikony so symbolom signalizujúcim operáciu „*Pause*“ / „*Resume*“ slúžiace k pozastaveniu konzumenta (pozastavenie prebehne po dokončení práve vykonávanej atomickej operácie).

6) **Okno producenta:**

   Okno popisujúce producenta (pekára). Od okna konzumenta sa líši iba rozdielnymi typmi akcií, ktoré je producent schopný vykonávať (akcie sa zobrazujú nad zobrazovacím prvkom typu *Progress Bar*).

7.2.4 Zhrnutie úlohy

Cieľom programu zachytávajúceho aktivitu hypotetickej pekárne je vizualizovať problém producenta a konzumenta pristupujúcich k zdieľanému zdroju. Program obsahuje možnosť osobitnej úpravy rýchlostí či pozastavenia producenta / konzumenta. Zároveň užívateľovi poskytuje korektné riešenie nežiadúcej situácie vyplývajúcej zo súbehu prostredníctvom ponúkanej voľby riešenia ošetrenej softwarovým synchronizačným prostriedkom (monitor, semafor). Zrozumiteľná vizualizácia vo forme širokého spektra textových výpisov a stavu bufferu zachyteného obrázkovou formou a aktualizovaného v reálnom čase počas priebehu simulácie zabezpečujú možnosť prispôsobiť simuláciu potrebám a preferenciám užívateľa za zachovania hlavnej myšlienky a zrozumiteľnosti.

7.3 Komunikačné prostriedky

7.3.1 Popis

Aplikácie vypracované v rámci praktickej časti bakalárskej práce v súvislosti s komunikačnými mechanizmami využívanými k IPC sa od predchádzajúcich ukážkových aplikácií mierne líšia – či už rozsahovo (menej komplexné čo do implementovanej funkcionality), tak myšlienkou – nedemonštrujú niektorý zo známych problémov, iba znázorňujú korektné použitie niektorého z dostupných komunikačných prostriedkov na dosiahnutie spojenia medzi rôznymi procesmi. Aplikácie sú celkovo tri, pričom tematicky ich môžeme rozdeliť v závislosti na forme komunikácie implementovanej každou z nich:

- **Zdieľaná pamäť:**

  Program obsahujúci komunikáciu pomocou zdieľanej pamäte (resp. pomocou tzv. *MemoryMappedFiles*) sa zakladá na možnosti realizovať zápis či čítanie bloku neperzistentnej pamäte určitého názvu, či už z jedného okna aplikácie, alebo (pri viacnásobnom spustení) z okien viacerých (čím je možné realizovať pravú komunikáciu medzi rôznymi procesmi, nielen medzi rôznymi vláknami jedného procesu).

- **Socket:**

  Program, ktorý implementuje komunikáciu založenú na socketoch umožňuje využitie socketov v roli serveru či klienta s možnosťou špecifikácie portu v rámci lokálneho zariadenia – v prípade ponechania aplikácie v režime jedného okna sú úlohy klient / server spúšťané v rozdielnych vláknach, v prípade viacnásobného otvorenia je možné pozorovať pravú medziprocesovú komunikáciu.

- **Rúra:**

  Ukážková aplikácia implementujúca medziprocesovú komunikáciu s využitím anonymnej rúry obsahuje iba jedno okno reprezentujúce proces odosielajúci správu. Keďže anonymná rúra je určená primárne ku komunikácii medzi rodičovským a odvodeným procesom, pri vynútení odoslania správy je spustená konzolová aplikácia prijímajúca prostredníctvom rúry správu od rodičovského procesu.

Keďže vizualizácia komunikačných metód je náročnejšia ako vizualizácia problémov synchronizačnej povahy, je nutné zabezpečiť, aby bolo prípadnému užívateľovi aplikácií

zrejmé dianie na pozadí – z toho dôvodu väčšina logov vypisovaných do logovacej mriežky v jednotlivých aplikáciách špecifikuje zdrojový proces prostredníctvom unikátnych identifikátorov procesu, vlákna a prípadne dodatočných údajov (názov procesu, adresa socketu ...).

7.3.2 Implementácia

Vytvorené komunikačné utility zdieľajú hlavné myšlienky, premisu a z veľkej časti implementujú analogické užívateľské rozhranie (a v dôsledku aj drvivú väčšinu infraštruktúry jednotlivých projektov). Podobne ako v ostatných spracovaných aplikáciách je hlavné okno vo všetkých prípadoch naviazané na *WindowViewModel* obsahujúci zobrazované kolekcie, jednotlivé *ViewModel*y nižšej úrovne a objekty implementujúce rozhranie *ICommand* spúšťajúce podkladové funkcionality programov. Hlavná funkcionalita programov je obsiahnutá v triedach s príponou *Service*, pričom *Services* v nasledujúcich podkapitolách je možné radiť podľa príslušnosti k jednotlivým komunikačným utilitám.

7.3.2.1 Zdieľaná pamäť

- **MemoryWriteService**

  MemoryWriteService je trieda implementujúca rozhranie *IMemoryWriteService*, ktorá obsahuje všetky funkcie potrebné k úspešnému zápisu do zdieľanej pamäte – jedná sa napr. o metódu „*Write()*“, ktorá spustí nové vlákno – pisateľa a následne hlavnú metódu „*WriteOperation()*“ obsahujúcu celú logiku zápisu do zdieľaného pamäťového priestoru daného názvu (vrátane prípadnej nutnosti inicializácie nového pamäťového bloku a konverzie správy v textovom formáte na pole bajtov).

```
using (var sharedMemory = MemoryMappedFile.CreateOrOpen(MemoryWriter.SharedMemoryName,
    FunctionalityRelatedConstants.ByteArraySize))
{
    var byteArray = Encoding.UTF8.GetBytes(MemoryWriter.Message);
    using (var writer = sharedMemory.CreateViewAccessor
        (FunctionalityRelatedConstants.ByteArrayStartPosition,
        FunctionalityRelatedConstants.ByteArraySize))
    {
        writer.WriteArray(FunctionalityRelatedConstants.ByteArrayStartPosition,
            byteArray, 0, byteArray.Length);
```

Obr. 33. Obsah MemoryWriteService – Prístup k bloku zdieľanej pamäte a zápis poľa bajtov

- **MemoryReadService**

  Trieda *MemoryReadService* je protipólom k *MemoryWriteService*. Implementuje rozhranie „*IMemoryReadService*“ a obsahuje metódu „*Read()*“ slúžiacu k vytvoreniu vlákna – čitateľa, ktoré následne pomocou funkcie „*ReadOperation()*“ prečíta obsah zdieľaného pamäťového bloku na základe špecifikovaného názvu a zabezpečí jeho konverziu do ľudsky čitateľnej textovej formy.

```
using var sharedMemory = MemoryMappedFile.OpenExisting(MemoryReader.SharedMemoryName);
using var reader = sharedMemory.CreateViewAccessor(0, FunctionalityRelatedConstants.ByteArraySize);

var byteArray = new byte[FunctionalityRelatedConstants.ByteArraySize];
reader.ReadArray(FunctionalityRelatedConstants.ByteArrayStartPosition, byteArray, 0, byteArray.Length);

MemoryReader.Message = Encoding.UTF8.GetString(byteArray);
```

Obr. 34. Obsah MemoryReadService – Prístup k bloku zdieľanej pamäte a konverzia obsahu do textového formátu

7.3.2.2 Sockety

- **ClientService**

  Služba pokrývajúca operácie vykonávané v súvislosti s klientskou časťou komunikačného spojenia implementujúca rozhranie *IClientService*. Najdôležitejšími obsiahnutými metódami sú „*RunOrStop()*” zabezpečujúca zavolanie obslužnej rutiny pre spustenie a pripojenie nového socketu klienta na špecifikovaný port či prerušenie činnosti tohto socketu, ak v danej inštancii aplikácie už bol spustený. Hlavná funkcionalita je lokalizovaná v metóde „*StartClient()*”, ktorá obsluhuje životný cyklus klientského socketu – od jeho vytvorenia a pripojenia na server až po spracovanie požiadaviek na odoslanie správy či príjem odpovede prostredníctvom volania asynchrónnych *Callback* funkcií.

```
if (Client.MessageSendRequested)
{
    Send(ClientSocket, $"{Client.Message}{FunctionalityRelatedConstants.EOFSymbol}");
    var sent = false;
    while (!sent && !CancellationToken.IsCancellationRequested)
    {
        sent = SendDoneEvent.WaitOne(FunctionalityRelatedConstants.CheckActionIntervalMillis);
    }

    Receive(ClientSocket);
    var received = false;
    while (!received && !CancellationToken.IsCancellationRequested)
    {
        received = ReceiveDoneEvent.WaitOne(FunctionalityRelatedConstants.CheckActionIntervalMillis);
    }
    _loggingService.WriteLog(Client.ClientIdentifier,
        ActionType.CLIEND_RECEIVING_MESSAGE,
        Client.StateObject.StringBuilder.ToString()
        .Replace(FunctionalityRelatedConstants.EOFSymbol, string.Empty));

    ClientSocket.Shutdown(SocketShutdown.Both);
    ClientSocket.Close();
    break;
}
```

Obr. 35. Obsah ClientService – hlavná funkcionalita klientského socketu pozostávajúca z odoslania správy serveru a obdržania odpovede

- **ServerService**

  Podobne ako v prípade *ClientService*, *ServerService* zodpovedá za druhú polovinu funkcionality programu – a teda obsluhu serveru, s ním súvisiaceho socketu a obsluhu pripojení klientov. Podobne ako u *ClientService*, aj *ServerService* obsahuje metódu „*StartOrStop()*", ktorá vytvorí serverový socket naslúchajúci na zadanom porte, resp. jeho beh preruší ak už socket na danom porte v danej inštancii aplikácie bol vytvorený. Hlavnou metódou je „*StartListening()*" vykonávaná serverovým vláknom. Telo metódy obsahuje cyklicky vykonávaný algoritmus, ktorý pokrýva príjem a spracovanie prichádzajúcich spojení, spracovanie obdržanej správy a zabezpečenie odpovede prostredníctvom asynchrónnych *Callback* funkcií.

```
while (!CancellationToken.IsCancellationRequested)
{
    AllDoneEvent.Reset();
    ServerSocket.BeginAccept(new AsyncCallback(AcceptCallback), ServerSocket);

    var allDone = false;
    while (!allDone && !CancellationToken.IsCancellationRequested)
    {
        allDone = AllDoneEvent.WaitOne
            (FunctionalityRelatedConstants.CheckActionIntervalMillis);
    }
}
```

Obr. 36. Obsah ServerService – hlavná funkcionalita serveru zabezpečujúca pripojenie klientov a ich asynchrónnu obsluhu

7.3.2.3 Rúry

- **PipeWriteService**

  Na rozdiel od ostatku komunikačných utilít, program monitorujúci komunikáciu prostredníctvom rúry obsahuje iba jednu triedu typu *Service* – je ňou *PipeWriteService* implementujúca rozhranie *IPipeWriteService*. Vstupnou, odkrytou metódou je funkcia „*RunWriter()*", zodpovedná za spustenie vlákna vykonávajúceho proces zápisu do nepomenovanej rúry. Samotný zápis je vykonávaný prostredníctvom súkromnej metódy „*Write()*" a zahŕňa spustenie procesu - príjemcu, inicializáciu objektu reprezentujúceho nepomenovanú rúru a samotné zaslanie správy.

```
using (AnonymousPipeServerStream pipeServer = new AnonymousPipeServerStream
    (PipeDirection.Out, HandleInheritability.Inheritable))
{
    pipeClient.StartInfo.Arguments = $"{pipeServer.GetClientHandleAsString()} {serverProcess.Id}";
    pipeClient.StartInfo.UseShellExecute = false;
    pipeClient.Start();

    pipeClientTitle = string.Format(LogConstants.ProcessTemplate, pipeClient.Id, pipeClient.ProcessName);
    _loggingService.WriteLog(serverProcessTitle, ActionType.CLIENT_STARTED, pipeClientTitle);
    Thread.Sleep(TimeSpan.FromSeconds(PipeWriter.TimeBetweenActions));

    pipeServer.DisposeLocalCopyOfClientHandle();

    try
    {
        using (var streamWriter = new StreamWriter(pipeServer))
        {
            streamWriter.AutoFlush = true;
            streamWriter.WriteLine(FunctionalityRelatedConstants.SynchronisationMessage);
            _loggingService.WriteLog(serverProcessTitle,
                ActionType.SENDING_SYNC_MESSAGE, FunctionalityRelatedConstants.SynchronisationMessage);
            Thread.Sleep(TimeSpan.FromSeconds(PipeWriter.TimeBetweenActions));

            pipeServer.WaitForPipeDrain();

            streamWriter.WriteLine(PipeWriter.Message);
            _loggingService.WriteLog(serverProcessTitle, ActionType.SENDING_MESSAGE, PipeWriter.Message);
            Thread.Sleep(TimeSpan.FromSeconds(PipeWriter.TimeBetweenActions));
        }
    }
```

Obr. 37. Obsah PipeWriteService – vytvorenie komunikačného spojenia a postupné posielanie správ vytvorenému procesu

Protipólom k *PipeWriteService* je samotný „detský“ proces implementovaný vo forme konzolovej aplikácie. Tento proces vykonáva pevne stanovenú rutinu pozostávajúcu z reakcií na priebeh komunikácie, ktorú zobrazuje vo forme kolorovaného textového výstupu.

7.3.2.4 Komunikačné mechanizmy v .NET Core

Ukážkové programy zhotovené pre demonštráciu použitia možností medziprocesovej komunikácie v rámci praktickej časti tejto práce využívajú niekoľko dôležitých typov komunikačných prostriedkov implementovaných zväčša formou objektov v *.NET Core Framework*.

- **Rúry:**

  Rúry sú v *.NET Core* rozdelené na pomenované (*Named Pipes*) a nepomenované (*Anonymous Pipes*). Medzi týmito druhmi existuje viac rozdielov:

- **Anonymous Pipes:**

  Nepomenované rúry sú jednosmerné, používajú sa na komunikáciu medzi rodičovským (*parent*) a odvodeným (*child*) procesom, neumožňujú komunikáciu mimo lokálne zariadenie [83], avšak sú jednoduchšie a vyžadujú menej réžie.

- **Named Pipes:**

  Implementácia pomenovaných rúr sa nápadne ponáša na implementáciu komunikácie s využitím socketov [84]. Umožňujú komunikáciu po sieti, väčšie množstvo klientov súčasne a obojsmerné zasielanie správ.

V ukážkovom programe zobrazujúcom využitie rúry k medziprocesovej komunikácii boli po zohľadnení intuitívnejšieho použitia (a skutočnosti, že komunikácia s využitím socketu značne podobná implementácii pomenovanej rúry už má vytvorený vlastný program) použité práve rúry nepomenované.

Komunikáciu skrz nepomenovanú rúru umožňujú dva základné objekty

- *AnonymousPipeClientStream*
- *AnonymousPipeServerStream*

Ako samotné názvy napovedajú, tieto objekty umožňujú každému procesu reprezentovať jeden koniec rúry, pričom objekt *AnonymousPipeServerStream* poskytuje spustenému odvodenému procesu tzv. *handle* (teda akúsi formu odkazu) na proces rodičovský prostredníctvom textového argumentu.

Následne zápis do / čítanie z rúry prebieha analogicky k ľubovoľnému zápisu či čítaniu nad súborom v rámci súborového systému, a síce prostredníctvom niektorého zo širokého spektra ponúkaných objektov typu *Writer* v závislosti na konkrétnej forme zapisovaného obsahu.

- **Socket:**

  Sockety ponúkajú široké spektrum metód, funkcií a prostriedkov na komunikáciu či už v rámci lokálneho počítača alebo po sieti. V *.NET Core Framework* samotná inštancia základnej triedy *Socket* nemá pevne stanovenú úlohu v zmysle klient / server – tá je daná až postupnosťou akcií, ktoré je nad týmto objektom vykonávaná [85].

K implementácii medziprocesovej komunikácie prostredníctvom socketov je možné využiť synchrónny alebo asynchrónny prístup. Pri synchrónnom prístupe je nutné brať do úvahy, že ľubovoľné zlyhanie na ktorejkoľvek strane komunikačného kanála môže spôsobiť nekonečné blokovanie opačnej strany (napr. čakanie na odpoveď serveru, ktorý už dávno skončil a tým pádom odpoveď nikdy neodošle) [86].

Asynchrónna implementácia umožňuje prijímať a odosielať správy či očakávať spojenie (na strane serveru) neblokujúcim spôsobom, pričom následné spracovanie zabezpečia tzv. *CallBack* funkcie, spúšťané nastavením určitej udalosti [87].

Pre potreby ukážkovej aplikácie zaoberajúcej sa medziprocesovou komunikáciou prostredníctvom socketu bola využitá asynchrónna forma spojenia.

Najdôležitejšie funkcie kľúčové pre úspešné použitie socketov sú:

- *Send() / BeginSend():*

  Umožňuje synchrónne / asynchrónne zaslanie správy (zvyčajne vo forme poľa bajtov) na druhý koniec komunikačného spojenia. Pri asynchrónnej variante je nutné špecifikovať obslužnú *Callback* funkciu.

- *Receive() / BeginReceive():*

  Umožňuje synchrónne / asynchrónne prijatie správy z druhého konca komunikačného spojenia. Pri asynchrónnej variante je nutné špecifikovať obslužnú *Callback* funkciu.

- *Bind():*

  Previaže (napojí) inštanciu triedy Socket na konkrétnu adresu a port.

- *Connect / BeginConnect():*

  Funkcie zabezpečujú synchrónne / asynchrónne napojenie klientského socketu na aktívny server. Pri asynchrónnej variante je opäť žiadúce špecifikovať obslužnú rutinu.

- *Accept / BeginAccept():*

  Sprostredkúva u socketu slúžiaceho ako server synchrónne / asynchrónne prijatie žiadostí o pripojenie od socketov – klientov. V prípade asynchrónnej varianty je nutné špecifikovať obsluhu pripojenia.

- **Zdieľaná pamäť:**

  .NET Framework použitý k tvorbe aplikácií praktickej časti tejto práce umožňuje implementáciu zdieľanej pamäte (resp. mechanizmu, ktorý k nej má principiálne najbližšie) pomocou špeciálnych súborov, tzv. *Memory-Mapped Files*, ku ktorým pristupujeme prostredníctvom rovnomennej triedy [88].

  Memory-Mapped Files rozdeľujeme na dva druhy:

  - **Persisted Memory-Mapped Files:**

    Perzistentné súbory mapované do pamäte spojené s reálnym zdrojovým súborom na disku – Uchovávajú zmeny aj po tom, čo posledný proces ukončí operácie nad danou pamäťovou sekciou. Používané najmä na prácu s veľkými súbormi.

  - **Non-persisted Memory-Mapped Files:**

    Neperzistentné súbory mapované do pamäte nemajú väzbu na súborový systém – existujú iba v logickom pamäťovom priestore vyhradenom danej aplikácii. Keď posledný proces ukončí prácu s daným kusom pamäte, dáta sú stratené a súbor spracuje tzv. *Garbage Collector.*

Vzhľadom na absenciu väzby na súborový systém, samočistiacej povahe neperzistentného typu *Memory-Mapped Files* a skutočnosti, že ich primárnym účelom je slúžiť ako prostriedok pre sprostredkovanie medziprocesovej komunikácie (vďaka možnosti pomenovania pamäťovej sekcie) bola pre implementáciu programu demonštrujúceho komunikáciu s využitím zdieľanej pamäte zvolená práve neperzistentná forma *Memory-Mapped File.*

Pre účely ukážkovej aplikácie boli použité najmä nasledujúce funkcie poskytované pred-implementovaným API [88]:

- *OpenExisting(string):*

  Otvorí existujúci pamäťový blok na základe názvu poskytnutého vo forme parametru (ak neexistuje, je vyhodená výnimka)

- *CreateOrOpen(string, int):*

  Otvorí existujúci pamäťový blok – v prípade jeho neexistencie vytvorí nový s menom a kapacitou špecifikovanou parametrami funkcie.

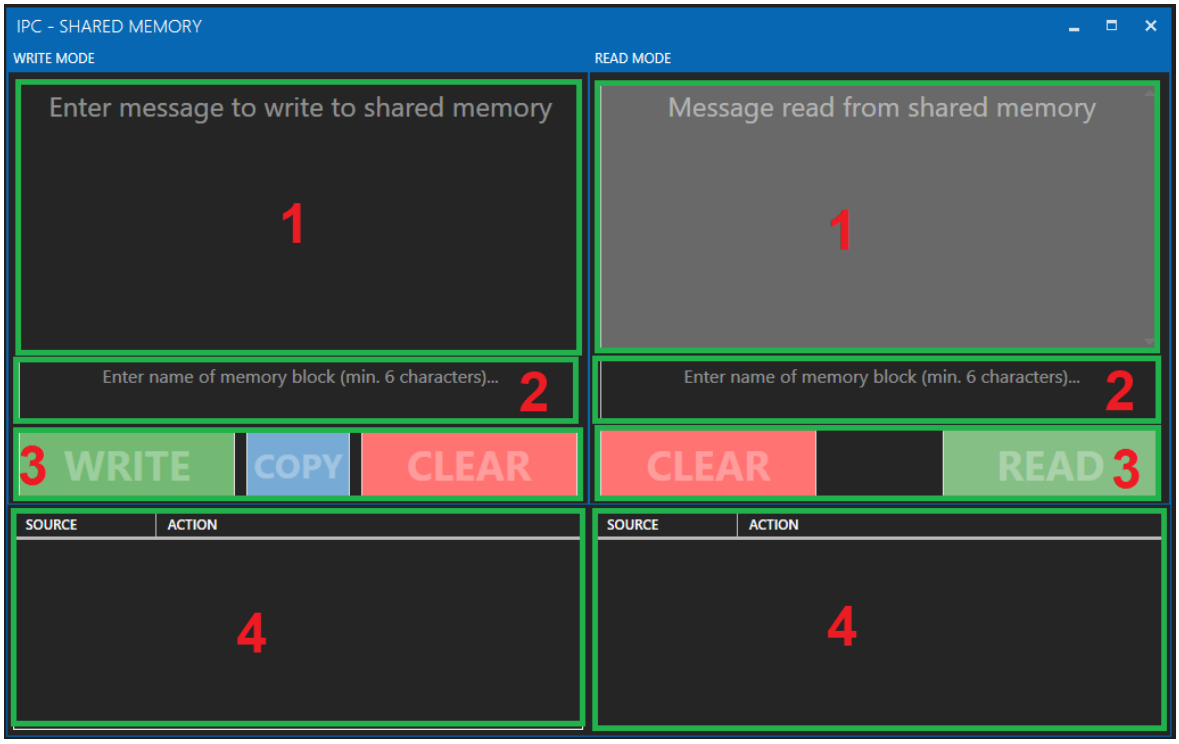
- *CreateViewAccessor(int, int):*

  Vytvorí objekt typu *MemoryMappedViewAccessor*, ktorý tvorí prístupový bod k pamäťovému bloku, pričom parametre špecifikujú odsadenie od začiatku bloku a veľkosť mapovaného úseku v bajtoch. Pomocou tohto objektu je možné taktiež realizovať zápis poľa bajtov do pamäte spôsobom podobným práci so štandardnými súbormi.

7.3.3 Vizualizácia

7.3.3.1 Zdieľaná pamäť

Vizualizácia v ukážkovom príklade pracujúcom so zdieľanou pamäťou je realizovaná prostredníctvom dvoch hlavných okien (jedno slúžiace pre zápis do zdieľanej pamäte, druhé pre čítanie), pričom obe okná obsahujú značne podobné ovládacie a zobrazovacie prvky, preto je možné ich popísať spoločne.



Obr. 38. Grafické užívateľské rozhranie programu zachytávajúceho medziprocesovú komunikáciu prostredníctvom zdieľanej pamäte

1) **Okno správy:**

   Okno vyhradené pre správu v prípade sekcie programu určenej k zápisu do pamäte slúži na užívateľský vstup textu, ktorý bude pri najbližšom úspešnom zápise do

zdieľanej pamäte umiestnený. V prípade sekcie programu určenej k čítaniu z pamäte nie je možné manuálne modifikovať jeho hodnotu – v tomto prípade slúži iba k zobrazeniu textovej reprezentácie poslednej správy, ktorá bola zo zdieľanej pamäte úspešne prečítaná.

2) **Pamäťová sekcia:**

   Textové pole určené pre užívateľský vstup určujúci názov bloku zdieľanej pamäte, do ktorého má byť realizovaný zápis / čítanie (v závislosti od sekcie programu, v ktorej sa toto pole nachádza).

3) **Výpis logov:**

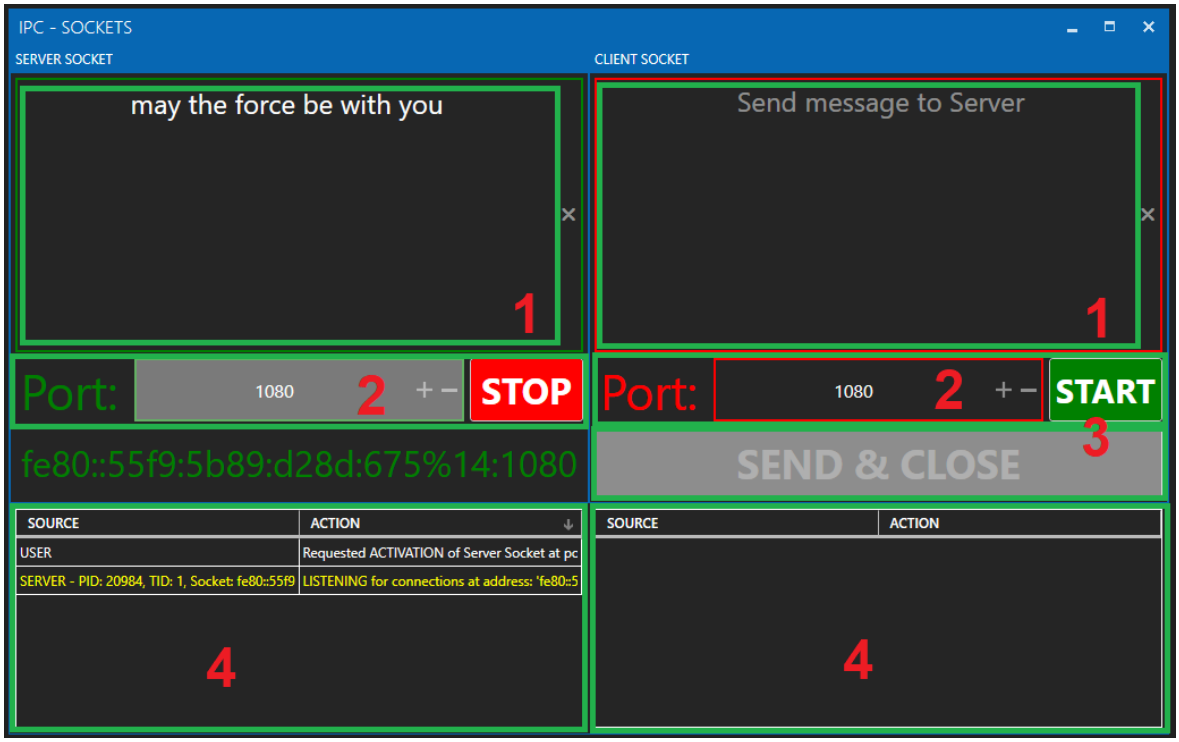
   Logovacia mriežka prítomná vo dvoch inštanciách – jedna zachytáva logy popisujúce priebeh zápisu do pamäte, druhá (podobne ako u ostatných ovládacích prvkov), priebeh čítania zo špecifikovaného pamäťového bloku vrátane špecifikácie, v ktorom procese (prípadne asynchrónnom konštrukte – *Task*).

4) **Ovládací panel:**

   Obe sekcie obsiahnuté v GUI implementujú sadu tlačidiel sprostredkujúcich funkcionalitu programu. Patrí tam tlačidlo „*Clear*“ na vyčistenie vstupných polí príslušnej sekcie, tlačidlo „*Read*“ / „*Write*“ pre čítanie, resp. zápis do zdieľanej pamäte a tlačidlo „*Copy*“, ktoré sa nachádza výhradne v časti programu určenej pre zápis a umožňuje prekopírovať zadaný názov pamäťovej sekcie do ekvivalentného vstupného poľa v sekcii čitateľa.

7.3.3.2 Sockety

Aplikácia implementujúca komunikáciu založenú na princípe socketov je (podobne ako program využívajúci zdieľanú pamäť) rozdelená na dva logické celky – vzhľadom na princíp použitia a terminológiu súvisiacu s daným typom komunikácie jedna strana reprezentuje klienta a druhá server. Drvivá väčšina ovládacích prvkov je opäť pre obe strany zhodná:



Obr. 39. Grafické užívateľské rozhranie programu zachytávajúceho medziprocesovú komunikáciu prostredníctvom socketov

1) **Okno správy:**

   Okná určené pre užívateľský vstup za účelom zadania správy, ktorá bude (v závislosti na umiestnení okna v kontexte role klient / server) odoslaná z klienta na server, resp. použitá ako odpoveď serveru na pripojenie klienta.

2) **Nastavenie portu:**

   Číselný vstup, umožňuje špecifikovať port ku ktorému bude pripojený nový socket slúžiaci (či špecifikujeme port pre socket slúžiaci ako klient / server je opäť určené konkrétnym umiestnením v rámci okna aplikácie)

3) **Tlačidlá:**

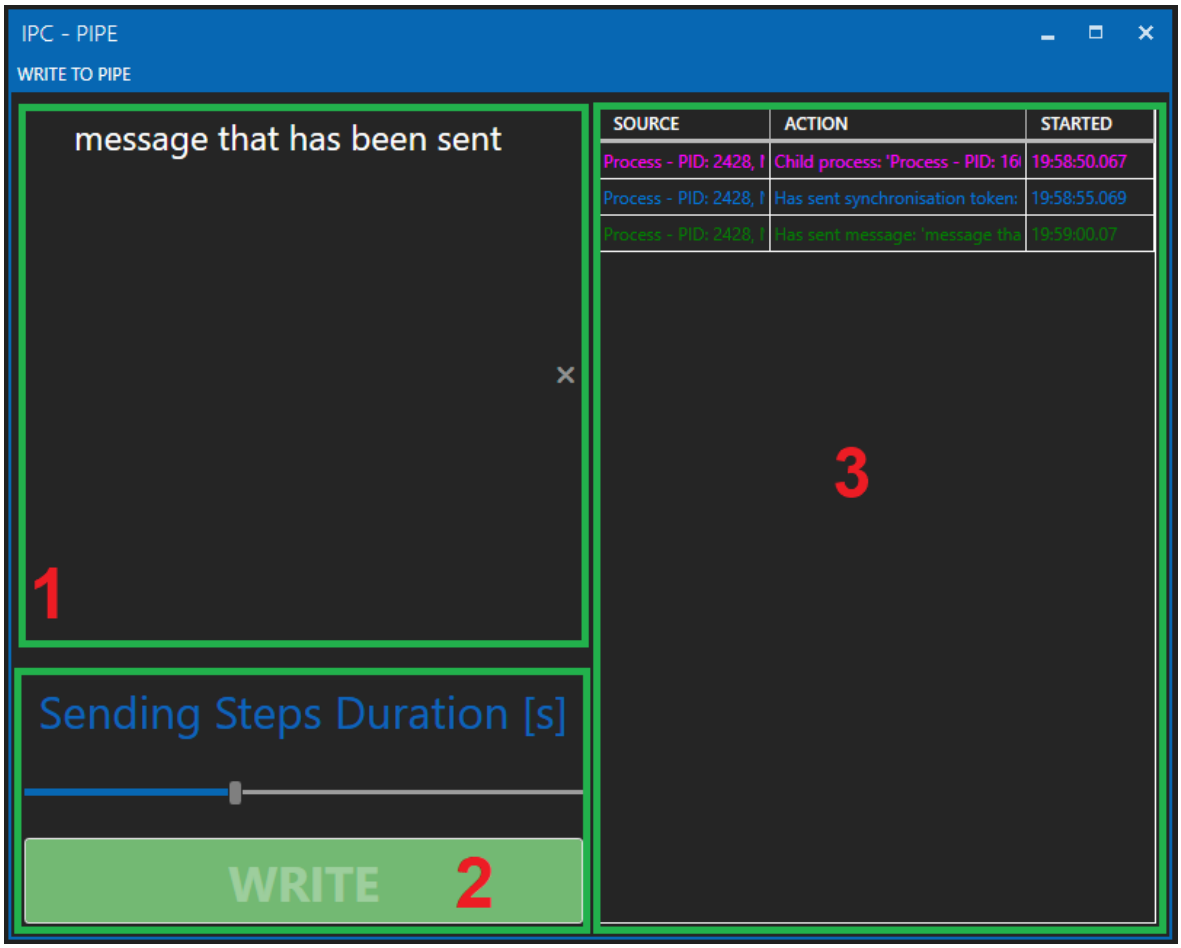
   Sekcia serveru / klienta implementuje vlastné tlačidlo umožňujúce spustenie socketu so zadaným nastavením, prípadne prerušenie prebiehajúcej činnosti už bežiaceho socketu. Časť rozhrania venovaná klientovi naviac implementuje tlačidlo „*Send & Close*“, ktoré slúži k nadviazaniu spojenia so serverom, odoslaniu správy a následnému prijatiu prípadnej odpovede, vrátane uzatvorenia komunikačného kanálu.

4) **Výpis logov:**

Podobne ako väčšina ostatných aplikácií implementovaných v rámci praktickej časti práce, aj tu je prítomná logovacia mriežka v dvoch inštanciách – poskytuje prehľad o akciách vykonávaných na strane klienta / serveru, vrátane detailných identifikátorov zúčastnených procesov.

7.3.3.3 Rúry

Grafické užívateľské rozhranie zhotovené pre program zachytávajúci komunikáciu s využitím rúry je v porovnaní s ostatnými rozhraniami vytvorených komunikačných utilít jednoduchšie – obsahuje iba jedno okno obsahujúce nasledujúce logické celky:



Obr. 40. Grafické užívateľské rozhranie programu zachytávajúceho medziprocesovú komunikáciu prostredníctvom nepomenovanej rúry

1) **Okno správy:**

Okno vyhradené pre správu, ktorá bude odosielaná odvodenému procesu prostredníctvom nepomenovanej rúry. Obsahuje natívnu možnosť vyčistenia obsahu.

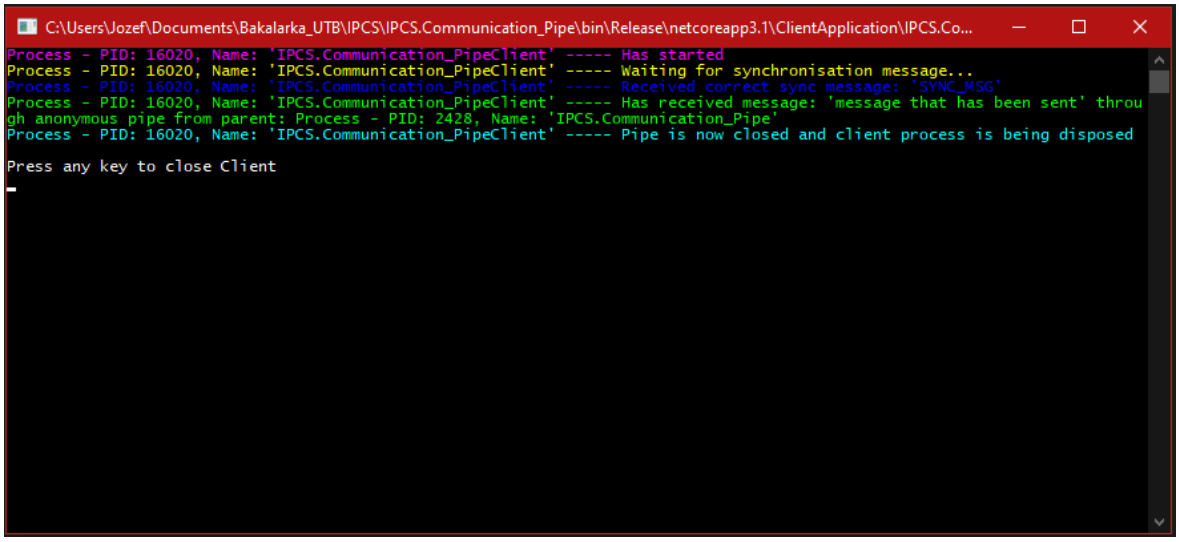
2) **Ovládací panel:**

   Poskytnuté ovládacie prostriedky zahŕňajú posuvník umožňujúci úpravu rýchlosti jednotlivých krokov zahŕňajúcich vytvorenie procesu – recipienta a odoslanie vloženej správy. Pod posuvníkom je umiestnené tlačidlo spúšťajúce samotný proces zápisu do nepomenovanej rúry.

3) **Výpis logov:**

   Logovacia mriežka zachytáva vo forme štruktúrovaného výpisu všetky akcie, ktoré je rodičovský proces schopný zaznamenať vrátane identifikátorov zúčastnených procesov.

Pri úspešnom odoslaní správy je následne vytvorený nový proces - príjemca s vlastným užívateľským rozhraním. Toto rozhranie pozostáva iba z okna konzoly schopnej zobraziť práve vykonávanú akciu prostredníctvom textového výpisu (vo formáte podobnom logovacej mriežke rodičovského procesu).



Obr. 41. Konzolová aplikácia popisujúca aktivitu príjemcu pri programe zachytávajúceho medziprocesovú komunikáciu prostredníctvom nepomenovanej rúry

7.3.4 Zhrnutie úlohy

Sada vypracovaných komunikačných utilít si kladie za cieľ jednoduchou formou poskytnúť riešenia používané ku korektnému zabezpečeniu medziprocesovej komunikácie prostredníctvom troch rôznych a nezávislých komunikačných mechanizmov. Zhotovené programy umožňujú samostatné ovládanie časti užívateľského rozhrania zodpovednej za zápis / odosielanie správy a časti zodpovednej za jej čítanie / príjem, vrátane možnosti

ovládania jednotlivých častí z rozdielnych inštancií spustenej aplikácie. Súčasťou logovacieho systému komunikačných programov je rozšírená identifikácia zúčastnených procesov. Príležitostné vstupné polia obsiahnuté v grafickom rozhraní aplikácie umožňujú užívateľovi na základe vlastných vstupov pozorovať správanie a vlastnosti jednotlivých komunikačných metód, rovnako ako príležitostné chybové stavy.

8 VIZUALIZÁCIA

8.1 Zobrazovacie prvky

Grafické rozhranie WPF ponúka širokú škálu zobrazovacích prvkov, ktorými umožňuje vizualizovať rôzne typy dát. Cieľom zobrazovacích prvkov je vhodným spôsobom interpretovať hodnoty uložené v premenných či objektoch, pričom prepojenie medzi zobrazovacím prvkom a zobrazovanými dátami sa zvyčajne realizuje pomocou mechanizmu *Data Binding* (ktorému je v tejto práci vyhradená samostatná kapitolka – viď. nižšie).
V programoch vypracovaných ako súčasť praktickej časti boli použité nasledujúce prvky [89][90][91][92]:

- **TextBlock:** Jednoduché textové pole, zvyčajne určené pre krátke pomenovanie, názov či popis.
- **ProgressBar:** Alebo aj *Loading Bar*, zobrazuje percentuálne vyjadrenie postupu určitej aktivity formou postupne vypĺňaného obdĺžniku.
- **GridView:** Umožňuje štruktúrované zobrazenie zložitejších dát v textovej podobe vo forme výpisu do tabuľky.
- **Image:** Statický obrázok v niektorom z konvenčných formátov (.bmp, .jpeg, .png).

8.2 Ovládacie prvky

Ovládacie prvky poskytované grafickým subsystémom WPF je možné vnímať ako prvky UI, pomocou ktorých spúšťame príkazy či pristupujeme k logike komplexnejšej ako jednoduché zobrazenie. Pre potreby zhotovených ukážkových programov sa väčšinou jedná o prvky typu *Button* [93].

Jednotlivé tlačidlá sú pomocou mechanizmu *Binding* naviazané na príkazy (tzv. *Commands*). V našom kóde sú všetky príkazy realizované triedou *RelayCommand*, ktorá dedí od rozhrania *ICommand* a vďaka svojej internej implementácii zaručuje vysokú flexibilitu a opakovanú použiteľnosť, keďže odstraňuje nutnosť vytvárať pre každý nový príkaz samostatnú triedu. [72].

8.3 Binding

8.3.1 Binding vo WPF

Data Binding je proces, ktorý vytvára prepojenie medzi prvkom užívateľského rozhrania (*View*) a dátami ktoré zobrazuje, modifikuje, alebo k nim pristupuje (*ViewModel*) [94]. Týmto spôsobom môžeme napr. naviazať element *Button* na obslužnú rutinu, alebo *ListView* na zoznam prvkov, ktoré má zobrazovať.

V kontexte WPF je možné *Binding Context* (trieda, na ktorú prvky viažeme) špecifikovať buď v *Code Behind*, alebo vo XAML súbore samotnom.

```
<Viewbox Margin="10, 10, 10, 10" Grid.Row="4" Grid.Column="1">
    <TextBlock Text="{Binding Philosophers[4].State,
        Converter={StaticResource PhilosopherStateToStringConverter}}"
              Foreground="{Binding Philosophers[4].State,
        Converter={StaticResource PhilosopherStateToColorConverter}}">
    </TextBlock>
</Viewbox>
```

Obr. 42. Previazanie hodnôt atribút elementu na vlastnosti ViewModelu pomocou mechanizmu Binding vrátane konvertora, nastavené v .XAML

Ak je naviac poskytnutá dodatočná programová obsluha vykonávaná pri zmene dát s ktorými GUI preväzujeme (napr. pomocou implementácie rozhrania *INotifyPropertyChanged*), budú prvky GUI automaticky aktualizované novou hodnotou [95].

8.3.2 INotifyPropertyChanged

INotifyPropertyChanged je rozhranie (teda určitá forma abstrakcie, od ktorej môže nami vytvorená trieda dediť). Zabezpečuje upovedomenie klientov („klientov“ v zmysle elementov, ktoré sú na dané dáta naviazané) o tom, že došlo k zmene týchto dát. [95]. Pri zmene vlastnosti objektu ktorého trieda implementuje *INotifyPropertyChanged* dôjde k vyvolaniu udalosti *PropertyChanged*.

8.4 Interaktivita

Grafická vizualizácia ľubovoľnej aplikácie vyžaduje správne uchopenie odozvy – teda to, aby užívateľ na svoje akcie pozoroval vhodnú a včasnú reakciu. Zvolené technológie

ponúkajú mnoho spôsobov, ako tohto cieľa dosiahnuť, avšak viacvláknová povaha implementovaných programov všetko mierne komplikuje.

Jednovláknové aplikácie zvyčajne implementujú model, v ktorom sa užívateľské rozhranie informuje o zmene dát pomocou implementácie rozhrania *INotifyPropertyChanged* v triedach slúžiacich ako *ViewModel*, prípadne pomocou špeciálnej pozorovateľnej dátovej štruktúry, tzv. *ObservableCollection*.

Mnohovláknové aplikácie však vyvolávajú vizualizačné komplikácie – je potrebné si uvedomiť, že udalosť existuje iba v kontexte vlákna, ktorým bola vyvolaná [96]. Preto nie je možné aktualizovať GUI z viacerých vlákien zároveň pomocou naivnej implementácie vyššie spomenutých mechanizmov.

Popísaná prekážka bola v ukážkových programoch prekonaná. Architektúra zhotovených aplikácií je navrhnutá tým spôsobom, že obsahujú jedno vlákno hlavné – teda vlákno obsluhujúce GUI a zároveň spúšťajúce vedľajšie vlákna zodpovedné za samotnú funkcionalitu, keďže v nich bežia procesy vizualizované práve za pomoci vlákna hlavného. Komunikáciu medzi hlavným a vedľajšími vláknami môžeme zabezpečiť viacerými spôsobmi:

- **Komunikačný kanál:**

  Vytvorenie dátovej štruktúry umožňujúcej bezpečný prístup z viacerých vlákien – tzv. „*thread-safe*“ štruktúry.(najpoužívanejšia býva fronta), na ktorú má hlavné vlákno aj vedľajšie vlákna spoločnú referenciu.

  Solídne riešenie, ale komplikovanejšia implementácia. Frontu je potrebné na strane GUI vlákna pravidelne vyprázdňovať. Ak vyprázdňovanie nemá byť realizované vo forme aktívneho čakania, je nutné vytvoriť časovač urgujúci k vyprázdneniu fronty v pravidelných intervaloch, pričom tento prístup zvyšuje reakčnú dobu GUI na zmenu dát.

- **Dispatcher:**

  Notifikácia hlavného vlákna pri zmene vizualizovaných dát takým spôsobom, aby túto zmenu vykonalo hlavné vlákno samotné [97] – tým pádom všetky udalosti slúžiace k aktualizácii zobrazovacích prvkov vzniknú a prebehnú v samotnom hlavnom vlákne.

Výhodami sú jednoduchšia implementácia a lepšia čitateľnosť, avšak prístup je nevhodný v prípade, že vedľajšie vlákna vykonávajú príliš mnoho zmien v príliš krátkom časovom intervale (ak by k tomu došlo, hlavné vlákno by nestíhalo obsluhu vizualizácie a pravdepodobne by skôr či neskôr nasledoval pád aplikácie).

```
/// <summary>
/// Base View Model class using Dispatcher implementing INotifyPropertyChanged
/// Updates Bound Elements even when properties are not modified from main thread
/// </summary>
8 references
public abstract class ViewModelBaseDispatcher : INotifyPropertyChanged
{

    public event PropertyChangedEventHandler PropertyChanged;

    32 references
    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        Dispatcher.CurrentDispatcher.Invoke(()
            => { PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));});
    }
}
```

Obr. 43. Implementácia abstraktnej triedy ViewModelBase s využitím možností objektu Dispatcher

ZÁVER

Výstupom teoretickej časti bakalárskej práce je jednotná množina informácií týkajúca sa problematiky paralelného spracovania úloh v dnešných počítačoch – poskytuje prehľad súčasných poznatkov o výhodách a nevýhodách viacvláknového prístupu k výpočtom a implikuje potrebu tieto nevýhody vhodným spôsobom adresovať. Poznatky využité pri tvorbe bakalárskej práce boli získané prostredníctvom prieskumu viacerých literárnych zdrojov rôzneho veku a snažia sa o čo najdôkladnejšie pokrytie skúmanej oblasti so zohľadnením rozdielností v zdrojových literárnych materiáloch.

Teoretická časť práce sa ďalej venovala problematike komunikácie procesov a uvedeniu v súčasnosti známych prostriedkov na to používaných, ich vlastnostiam, výhodám a nevýhodám. V súlade s týmto cieľom bola zhotovená sada troch ukážkových aplikácií, pričom každá z nich prostredníctvom interaktívneho užívateľského rozhrania a vizualizácie aktivity na pozadí prostredníctvom textových výpisov umožňuje sledovať činnosť a použitie prostriedkov medziprocesovej komunikácie ako sú sockety, zdieľaná pamäť či rúry, vrátane možnosti zadať ako posielanú správu užívateľský vstup a upresniť prípadné dodatočné parametre súvisiace s daným typom medziprocesovej komunikácie.

Problematike súbehu a k nej náležiacim poznatkom vrátane možných príčin, dopadov a spôsobov riešenia (či už softwarovými alebo hardwarovými prostriedkami) bola venovaná prevažná väčšina teoretickej časti tejto práce. V nadväznosti na formulované znalosti a prehľad o tematike bol vytvorený program založený na probléme typu Producent-Konzument, ktorý dáva užívateľovi možnosť priamo sa podieľať na priebehu daného algoritmu, pozorovať jeho chod a prostredníctvom kontroliek grafického užívateľského rozhrania doňho zasahovať a (pokiaľ si to želá), usmerniť jeho chod smerom k popísanej chybovej situácii. Program zároveň implementuje softwarové metódy riešenia nežiadúceho súbehu prostredníctvom možnosti využitia synchronizačných prostriedkov semaforu a monitoru.

Uviaznutie, jeho charakteristiky, príčiny a dôsledky teoretická časť práce popisuje v úzkej logickej súvislosti s problematikou synchronizácie v procesov. Demonštráciu typovej situácie uviaznutia znázorňuje vypracovaná ukážková aplikácia zachytávajúca typovú úlohu Stolujúcich filozofov implementujúca viacero funkcionalít vrátane režimu spustenia, ktorý zaručene povedie k uviaznutiu a možnosti túto situáciu riešiť pomocou zabitia zúčastnených procesov a násilného odobratia nimi vlastnených zdrojov

V súlade s priebežnými konzultáciami s vedúcim práce došlo k uzneseniu, že sadu ukážkových programov špecifikovaných zadaním je v prípade programov zobrazujúcich uviaznutie a synchronizačné prostriedky možné zastúpiť obsahovo bohatšou aplikáciou s rozšírenou funkcionalitou, čo je práve cieľom aplikácií implementovaných k týmto tematikám.

Primárnym cieľom bakalárskej práce bolo rozšíriť podvedomie o potenciálnych výhodách paralelného programovania a pôsobiť ako stručný a zrozumiteľný prehľad foriem a možností medziprocesovej komunikácie, synchronizačných problémov a prostriedkov využívaných k ich úspešnému riešeniu. Vypracované ukážkové aplikácie jednak formou interaktívnej vizualizácie sprostredkúvajú podstatu problému či mechanizmu na ktorý sú obsahovo viazané, jednak zdrojové kódy samotné ponúkajú náhľad do spôsobu riešenia analogických problémov v reálnych situáciách softwarového vývoja.

Výsledkom práce je teda dokument a programy, ktoré obsahovo uspokojujú požiadavky stanovené v zadaní bakalárskej práce a vďaka aplikáciám implementovaným v rámci praktickej časti práce umožňujú prípadnému čitateľovi vstrebanie relatívne náročnej a nepríliš obľúbenej problematiky zrozumiteľným a názorným spôsobom.

ZOZNAM POUŽITEJ LITERATÚRY

[1] TANENBAUM, Andrew S. a Herbert BOS. *Modern Operating Systems*. 4th ed. Boston: Pearson, 2014. Fourth Edition. ISBN 978-0-13-359162-0.

[2] SILBERSCHATZ, Abraham, Peter BAER GALVIN a Greg GAGNE. *Operating System Concepts*. 9th ed. Hoboken: NJ: Wiley, 2013. Ninth Edition. ISBN 978-1-118-12938-8.

[3] STALLINGS, William. *Operating Systems, Internals and Design Principles*. 7th ed. Prentice Hall: Prentice Hall, Pearson, 2012. Seventh Edition. ISBN 978-0-13-230998-1.

[4] GODSE, Atul P. a Deepali A. GODSE. *Computer Organisation and Architecture*. Technical Publications, Prune, 2006. ISBN 9788189411565.

[5] Process States in Operating Systems. In: *GateVidyalay* [online]. Gate Vidyalay [cit.2020-07-24]. Dostupné z: https://www.gatevidyalay.com/process-states-in-operating-system/

[6] DHAMDERE, Dhannajay M. *Operating Systems: A Concept-Based Approach*. New York: McGraw-Hill Education, c2009. ISBN 978–0–07–295769–3.

[7] BAUER, Roderick. *What's the Diff: Programs, Processes, and Threads* [online]. August 16, 2017 [cit. 2020-07-24]. Dostupné z: https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

[8] DEITEL, Harvey M., Paul DEITEL a David R. CHOFFNES. *Operating Systems, 3rd Edition*. 3rd ed. Prentice Hall: Pearson, 2004. ISBN 978-0131828278.

[9] KHILLAR, Sagar. Difference Between Multiprogramming and Multitasking in Operating System. *DifferenceBetween.net* [online]. DifferenceBetween.net, 2019, August 16, 2019 [cit. 2020-07-24]. Dostupné z: http://www.differencebetween.net/technology/difference-between-multiprogramming-and-multitasking-in-operating-system/

[10] BLAISE, Barney. Introduction to Parallel Computing. *Computing* [online]. Lawrence Livermore National Laboratory [cit. 2020-07-24]. Dostupné z: https://computing.llnl.gov/tutorials/parallel_comp/

[11] BROKAW, Tom. Intel Core i9-10900K CPU Review: Is this the CPU you're looking for? In: *Void Your Warranty Modders INC* [online]. c2006-2020, May 20, 2020 [cit. 2020-07-24]. Dostupné z: https://www.modders-inc.com/wp-content/uploads/image//2020/05/10900k_CinebenchR20.jpg

[12] Amdahl's Law. *Techopedia* [online]. c2020, June 30, 2020 [cit. 2020-07-24]. Dostupné z: https://www.techopedia.com/definition/17035/amdahls-law

[13] WOLFE, Michael. Compilers and More: Is Amdahl's Law Still Relevant? *HPCwire* [online]. January 22, 2015 [cit. 2020-07-24]. Dostupné z: https://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/

[14] GHOSH, Supriya. Amdahl's law. In: *Alchetron* [online]. c2020, Jul 23, 2018 [cit. 2020-07-24]. Dostupné z: https://alchetron.com/cdn/amdahls-law-174f6f14-eba9-4bce-8695-5db9460e17d-resize-750.jpg

[15] Inter Process Communication (IPC). *Guru99* [online]. c2020 [cit. 2020-07-24]. Dostupné z: https://www.guru99.com/inter-process-communication-ipc.html

[16] DUSEY, Aniket. Methods in Interprocess Communication. *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-24]. Dostupné z: https://www.geeksforgeeks.org/methods-in-interprocess-communication/

[17] PANDEY, Durgesh. Inter Process Communication (IPC). *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-24]. Dostupné z: https://www.geeksforgeeks.org/inter-process-communication-ipc/

[18] ONSMAN, Alex. Message Passing vs Shared Memory Process communication Models. In: *Tutorials Point* [online]. Hyderabad, Telangana, c2020, 10 Oct 2018 [cit. 2020-07-24]. Dostupné z: https://www.tutorialspoint.com/assets/questions/media/12672/Message%20passing%20model%20in%20OS.PNG

[19] SHENE, Ching-Kuang. Basic Concept. *Https://pages.mtu.edu/* [online]. c2001-2014, August 20, 2001 [cit. 2020-07-25]. Dostupné z: https://pages.mtu.edu/~shene/NSF-3/e-Book/CH/basics.html

[20] CHATHURANGA, Buddhika. Inter-Process Communication. *Medium* [online]. c2020, Apr 21, 2020 [cit. 2020-07-25]. Dostupné z: https://medium.com/runtimeerror/inter-process-communication-1c97f5798ea

[21] PEDERSEN, Jan B. *Classification of Programming Errors in Parallel Message Passing Systems*. University of Nevada, 4505 Maryland Parkway, Las Vegas, Nevada, 89154, USA, 2006. Konferenčný výzkum. University of Nevada, Las Vegas.

[22] ONSMAN, Alex. Message Passing vs Shared Memory Process communication Models. In: *Tutorials Point* [online]. Hyderabad, Telangana, c2020, 10 Oct 2018 [cit. 2020-07-25]. Dostupné z: https://www.tutorialspoint.com/assets/questions/media/12672/Shared%20Memory%20Model%20in%20OS.PNG

[23] Unix Functionality Vs Windows Functionality Discussion. In: *Wiki.C2* [online]. 2014, August 3, 2013 [cit. 2020-07-25]. Dostupné z: https://wiki.c2.com/?UnixFunctionalityVsWindowsFunctionalityDiscussion

[24] Pipes and FIFOs. *GNU: Operating System* [online]. c1996-2020 [cit. 2020-07-25]. Dostupné z: https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html

[25] BHARGAVA, Paarmita. Named Pipe or FIFO with example C program. *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-25]. Dostupné z: https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/

[26] VAUGHT, Andy. Introduction to Named Pipes. *Linux Journal* [online]. September 1, 1997, **2019** [cit. 2020-07-25]. Dostupné z: https://www.linuxjournal.com/article/2156

[27] A Socket-based IPC Tutorial. *BlackBerry QNX* [online]. c2020 [cit. 2020-07-25]. Dostupné z: http://www.qnx.com/developers/docs/qnx_4.25_docs/tcpip50/prog_guide/sock_ipc_tut.html

[28] KLIMEŠ, Cyril. *Principy výstavby počítačů a operačních systémů*. Kovosil, 2007. ISBN 8090369412.

[29] LAŽANSKÝ, Jiří. *Operační systémy a databáze: Synchronizace procesů a problém uváznutí*. České vysoké učení technické v Praze, Technická 1902/2, Dejvice-Praha 6, 2014. Prezentácie k výuke predmetu A3B33OSD. České Vysoké Učení Technické, Fakulta elektrotechnická.

[30] HICKS, Michael. *Operating Systems: Implementing Synchronization*. College Park, MD 20742, United States, 2007. Prezentácie k výuke predmetu CMSC 412. University of Maryland, Dept. of Computer Science.

[31] RANJAN, Amish, Vaishnavi PANDEY a Shreya SHAGRAWAL. Introduction of Process Synchronization. *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-25]. Dostupné z: https://www.geeksforgeeks.org/introduction-of-process-synchronization/

[32] The Critical Section Problem: Requirements of Synchronization mechanisms. *Java T Point: The Best Portal to Learn Technologies* [online]. Noida, UP, India, c2011-2018 [cit. 2020-07-25]. Dostupné z: https://www.javatpoint.com/os-critical-section-problem

[33] SHENE, Ching-Kuang. *Synchronization: Critical Section and Mutual Exclusion*. Houghton, Michigan, USA, 2015. Prezentácie k výuke predmetu CS3331. Michigan Technological University.

[34] ARPACI-DUSSEAU, Andrea C. a Remzi H. ARPACI-DUSSEAU. *Operating Systems: Three Easy Pieces* [online]. 2018 [cit. 2020-07-25]. ISBN 9781985086593. Dostupné z: http://pages.cs.wisc.edu/~remzi/OSTEP/

[35] KENDALL, Graham. *Operating Systems: Test and Set Lock*. Nottingham, United Kingdom, 2002. Prezentácie k výuke predmetu G53OPS - Operating Systems. University of Nottingham.

[36] DAILEY, Matthew. *Operating System Lecture Notes: Synchronization*. 2002. Dostupné také z: https://slideplayer.com/slide/9227835/. Prezentácia vychádzajúca z poznatkov autorov Silberschatz, Galvin, Gagne.

[37] CHOI, Lynn. *Operating System: Chapter 5. Concurrency: Mutual Exclusion and Synchronization*. Seongbuk-gu, Seoul, South Korea. Prezentácie k

výuke. Korea University, College of Engineering, School of Electrical Engineering.

[38] LYADVINSKY, Kirill V. Compare and swap vs test and set. In: *Stack Overflow: For developers, by developers* [online]. c2020, Sep 7, 2010 [cit. 2020-07-25]. Dostupné z: https://stackoverflow.com/questions/3659336/compare-and-swap-vs-test-and-set

[39] CHAKRABORTY, Arnab. Peterson's Problem. In: *Tutorials Point* [online]. Hyderabad, Telangana, c2020, 17 Oct. 2019 [cit. 2020-07-25]. Dostupné z: https://www.tutorialspoint.com/peterson-s-problem

[40] SHULYUPIN, Constantine. Spinlocks: Embedded Linux Software, Device Drivers. *Make Linux: Embedded Linux Software, Device Drivers* [online]. Tel Aviv, Israel [cit. 2020-07-25]. Dostupné z: http://www.makelinux.net/ldd3/chp-5-sect-5.shtml

[41] Difference Between Semaphore and Mutex. *Tech Differences* [online]. c2020, December 21, 2016 [cit. 2020-07-25]. Dostupné z: https://techdifferences.com/difference-between-semaphore-and-mutex.html

[42] ATTRACTIVECHAOS. Multi-threaded programming: efficiency of locking. In: *Attractive Chaos: Just another WordPress.com weblog* [online]. October 6, 2011 [cit. 2020-07-25]. Dostupné z: https://attractivechaos.wordpress.com/2011/10/06/multi-threaded-programming-efficiency-of-locking/

[43] MICROSOFT. Mutex Objects. *Microsoft Docs* [online]. c2020, 05/31/2018 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/sync/mutex-objects. Oficiálna dokumentácia firmy Microsoft Corporation.

[44] AURIGA, Eduard Trunov. Thread Synchronization in Linux and Windows Systems, Part 3: Multithreaded applications parallelism. *Embedded Computing* [online]. OpenSystemsMedia, July 29, 2019 [cit. 2020-07-25]. Dostupné z: https://www.embedded-computing.com/guest-blogs/thread-synchronization-in-linux-and-windows-systems-part-3

[45] ZITOC. Critical Section Problem. In: *Zitoc: Zillion Topics On Concerns* [online]. Collage Road, Layyah, c2014-2024, February 22, 2019 [cit. 2020-07-25]. Dostupné z: https://i1.wp.com/zitoc.com/wp-content/uploads/2019/02/Critical-Section-Problem.png?w=449&ssl=1

[46] Process Synchronization: Part 6 (Semaphores-1). In: *GATE Computer Science Ideas* [online]. December 15, 2018 [cit. 2020-07-25]. Dostupné z: http://gatecsideas.blogspot.com/2018/12/process-synchronization-part-6.html

[47] SOLMSSEN, Anne. Semaphore implementation : why is disabling interrupts required along with test-and-set? In: *Stack Overflow: For developers, by developers* [online]. c2020, Feb 17, 2015 [cit. 2020-07-25]. Dostupné z: https://stackoverflow.com/questions/27561084/semaphore-implementation-why-is-disabling-interrupts-required-along-with-test

[48] MICROSOFT. Semaphore Objects. *Microsoft Docs* [online]. c2020, 05/31/2018 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects. Oficiálna dokumentácia firmy Microsoft Corporation.

[49] MICROSOFT. Event Objects. *Microsoft Docs* [online]. c2020, 05/31/2018 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/sync/event-objects. Oficiálna dokumentácia firmy Microsoft Corporation.

[50] Obscure Synchronization Primitives. *Lockless Inc* [online]. Seattle, USA, c2020 [cit. 2020-07-25]. Dostupné z: https://locklessinc.com/articles/obscure_synch/

[51] COFFMAN, E. G., M. ELPHICK a A. SHOSHANI. System Deadlocks. *ACM Computing Surveys (CSUR)* [online]. 1971, **3**(2), 67-78 [cit. 2020-07-25]. DOI: 10.1145/356586.356588. ISSN 0360-0300. Dostupné z: http://dl.acm.org/doi/10.1145/356586.356588

[52] HICKS, Michael. *Operating Systems: Deadlock*. College Park, MD 20742, United States, 2007. Prezentácie k výuke predmetu CMSC 412. University of Maryland, Dept. of Computer Science.

[53] KALÉ, Laxmikant V., Abhinav BHATELE, Eric J. BOHM, et al. Non-Blocking Algorithms. *Encyclopedia of Parallel Computing* [online]. Boston, MA: Springer US, 2011, 2011, , 1321-1329 [cit. 2020-07-25]. DOI: 10.1007/978-0-387-09766-4_185. ISBN 978-0-387-09765-7. Dostupné z: http://link.springer.com/10.1007/978-0-387-09766-4_185

[54] STAUDEK, Jan. *Uváznutí*. Botanická 68A, Brno-Královo Pole-Ponava, 2013. Prezentácie k výuke predmetu PB152. Masarykova Univerzita, Fakulta Informatiky.

[55] KUMAR, Vikash. Banker's Algorithm in Operating System. *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-25]. Dostupné z: https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/

[56] INGALLS, Robert P. *Deadlock* [online]. Troy, New York [cit. 2020-07-25]. Dostupné z: http://www.cs.rpi.edu/academics/courses/fall04/os/c10/. Výukové materiály k predmetu CSCI.4210 Operating Systems. Rensselaer Polytechnic Institute.

[57] BHARDWAJ, Rashi. Deadlock, Starvation, and Livelock. *Geeks for Geeks: A computer science portal for geeks* [online]. Noida, Uttar Pradesh [cit. 2020-07-25]. Dostupné z: https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/

[58] BURU. What's the difference between deadlock and livelock? In: *Stack Overflow* [online]. c2020, Jan 17, 2015 [cit. 2020-07-25]. Dostupné z: https://i.stack.imgur.com/OAda8.jpg

[59] JONES, Mike. What really happened to the software on the Mars Pathfinder spacecraft? In: *Rapita Systems: A Danlaw Company* [online]. York, c2020, 04 Jul 2013 [cit. 2020-07-25]. Dostupné z: https://www.rapitasystems.com/system/files/priority-inversion1.jpg

[60] CHAND, Mahesh. Difference Between .NET and .NET Core. *C# Corner: .NET Core Advantages* [online]. c2020, Jul 05, 2020 [cit. 2020-07-25]. Dostupné z: https://www.c-sharpcorner.com/article/difference-between-net-framework-and-net-core/

[61] MICROSOFT. .NET is open-source. *.NET: Free. Cross-platform. Open source.* [online]. [cit. 2020-07-25]. Dostupné z: https://dotnet.microsoft.com/platform/open-source. Oficiálna dokumentácia firmy Microsoft Corporation.

[62] MICROSOFT. Introduction to WPF in Visual Studio. *Microsoft Docs* [online]. c2020, 01/26/2018 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/introduction-to-wpf-in-vs. Oficiálna dokumentácia firmy Microsoft Corporation.

[63] MICROSOFT. XAML overview in WPF. *Microsoft Docs* [online]. c2020, 08/08/2019 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/desktop-wpf/fundamentals/xaml. Oficiálna dokumentácia firmy Microsoft Corporation.

[64] *MahApps Metro: What is MahApps.Metro?* [online]. .NET Foundation [cit. 2020-07-25]. Dostupné z: https://mahapps.com/

[65] MISHLER, Tony. Ninject. In: *GitHub: The world's leading software development platform* [online]. c2020, May 5, 2018 [cit. 2020-07-25]. Dostupné z: https://github.com/ninject/ninject/wiki

[66] MICROSOFT. Create a UI by using XAML Designer. *Microsoft Docs* [online]. c2020, 03/03/2020 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/xaml-tools/creating-a-ui-by-using-xaml-designer-in-visual-studio?view=vs-2019. Oficiálna dokumentácia firmy Microsoft Corporation.

[67] MICROSOFT. IntelliSense in Visual Studio. *Microsoft Docs* [online]. c2020, 05/25/2018 [cit. 2020-07-25]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2019. Oficiálna dokumentácia firmy Microsoft Corporation.

[68] MICROSOFT. Vývojářské nástroje sady Visual Studio pro studenty. *Visual Studio: Nejlepší nástroje pro každého vývojáře* [online]. c2020 [cit. 2020-07-25]. Dostupné z: https://visualstudio.microsoft.com/cs/students/. Marketingové dokumenty firmy Microsoft Corporation.

[69] BOST, Kevin. Getting started with Model-View-ViewModel (MVVM) pattern using Windows Presentation Framework (WPF). *IntelliTect* [online]. Spokane, Washington, c2020, September 19, 2017 [cit. 2020-07-26]. Dostupné z: https://intellitect.com/getting-started-model-view-viewmodel-mvvm-pattern-using-windows-presentation-framework-wpf/

[70] MICROSOFT. Data binding overview in WPF. *Microsoft Docs* [online]. c2020, 09/19/2019 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/desktop-wpf/data/data-binding-overview. Oficiálna dokumentácia firmy Microsoft Corporation.

[71] WEKEMPF. Services – Your ViewModel Death Star. *Digital Tapestry* [online]. July 21, 2009 [cit. 2020-07-26]. Dostupné z: https://digitaltapestry.wordpress.com/2009/07/21/services-%e2%80%93-your-viewmodel-death-star/

[72] TOSKERSCORNER. C# WPF - Relay/Delegate Commands [Part 2]. In: *YouTube* [online]. 13.12.2016 [cit. 2020-07-26]. Dostupné z: https://www.youtube.com/watch?v=8WfD2cFRymM

[73] KOIRALA, Shivprasad. Dependency Injection (DI) vs. Inversion of Control (IOC). In: *Code Project: For those who code* [online]. c1999-2020, 31 Dec 2013 [cit. 2020-07-26]. Dostupné z: https://www.codeproject.com/articles/592372/dependency-injection-di-vs-inversion-of-control-io

[74] C# Programming/Interfaces. In: *Wiki Books: Open books for an open world* [online]. 16 April 2020 [cit. 2020-07-26]. Dostupné z: https://en.wikibooks.org/wiki/C_Sharp_Programming/Interfaces

[75] WALPOLE, Sam. Top 3 Reasons to use Interfaces in Your Code. *Dev* [online]. c2016-2020, 13.4.2020 [cit. 2020-07-26]. Dostupné z: https://dev.to/walpolesj/top-3-reasons-to-use-interfaces-in-your-code-4kol

[76] HUO, Yumei. *Operating System Design* [online]. Staten Island, New York, USA, 2006 [cit. 2020-07-26]. Dostupné z: http://www.cs.csi.cuny.edu/~yumei/csc718/homework2/homework2solution.pdf. Materiály k výuke predmetu CSC 718. College of Staten Island, The City University of New York.

[77] Classic Synchronisation Problems. In: *COEP Wiki* [online]. Pune, 7 March 2017n. l., 17 April 2015 [cit. 2020-07-26]. Dostupné z: https://foss.coep.org.in/coepwiki/index.php/Classic_Synchronisation_Problems#Producer_Consumer_Problem

[78] HICKS, Michael. *Operating Systems: Concurrency and Synchronization*. College Park, MD 20742, United States, 2007. Prezentácie k výuke predmetu CMSC 412. University of Maryland, Dept. of Computer Science.

[79] *The classic Producer-Consumer or "bounded buffer" problem* [online]. Boston [cit. 2020-07-26]. Dostupné z: https://www.cs.umb.edu/cs444/class13.html. Materiály k výuke predmetu CS444. UMass Boston Computer Science, College of Science and Mathematics.

[80] MICROSOFT. Lock statement (C# reference). *Microsoft Docs* [online]. c2020, 04/02/2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement. Oficiálna dokumentácia firmy Microsoft Corporation.

[81] MICROSOFT. Monitor Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.threading.monitor?view=netframework-4.8. Oficiálna dokumentácia firmy Microsoft Corporation.

[82] MICROSOFT. Semaphore Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore?view=netframework-4.8. Oficiálna dokumentácia firmy Microsoft Corporation.

[83] MICROSOFT. AnonymousPipeClientStream Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.io.pipes.anonymouspipeclientstream?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[84] MICROSOFT. How to: Use Named Pipes for Network Interprocess Communication. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné

z: https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication. Oficiálna dokumentácia firmy Microsoft Corporation.

[85] MICROSOFT. Socket Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[86] MICROSOFT. Synchronous Server Socket Example. *Microsoft Docs* [online]. c2020, 03/30/2017 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/synchronous-server-socket-example. Oficiálna dokumentácia firmy Microsoft Corporation.

[87] MICROSOFT. Asynchronous Server Socket Example. *Microsoft Docs* [online]. c2020, 03/30/2017 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-server-socket-example. Oficiálna dokumentácia firmy Microsoft Corporation.

[88] MICROSOFT. Memory-Mapped Files. *Microsoft Docs* [online]. c2020, 03/30/2017 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files. Oficiálna dokumentácia firmy Microsoft Corporation.

[89] MICROSOFT. TextBlock Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.textblock?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[90] MICROSOFT. ProgressBar Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.progressbar?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[91] MICROSOFT. GridView Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-

us/dotnet/api/system.windows.controls.gridview?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[92] MICROSOFT. Image Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.image?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[93] MICROSOFT. Button Class. *Microsoft Docs* [online]. c2020 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.button?view=netcore-3.1. Oficiálna dokumentácia firmy Microsoft Corporation.

[94] MICROSOFT. Data binding overview in WPF. *Microsoft Docs* [online]. c2020, 09/19/2019 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/desktop-wpf/data/data-binding-overview. Oficiálna dokumentácia firmy Microsoft Corporation.

[95] MICHELS, Bruno Leonardo. Data Binding in WPF. *C# Corner* [online]. c2020, May 28, 2015 [cit. 2020-07-26]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/20c06b/data-binding-in-wpf/

[96] INotifyPropertyChanged and Cross-thread exceptions. *ILoggable: A place to keep my thoughts on programming* [online]. July 8, 2007 [cit. 2020-07-26]. Dostupné z: http://www.claassen.net/geek/blog/2007/07/inotifypropertychanged-and-cross-thread-exceptions.html

[97] MICROSOFT. Threading Model. *Microsoft Docs* [online]. c2020, 03/30/2017 [cit. 2020-07-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/threading-model. Oficiálna dokumentácia firmy Microsoft Corporation.

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

API	Application Programming Interface
DI	Dependency Injection
FIFO	First In, First Out
GUI	Graphical User Interface
HTML	Hypertext Markup Language
I / O	Input / Output
IDE	Integrated Development Environment
IPC	Inter-Process Communication
IPCS	Inter-Process Communication and Synchronisation
IoC	Inversion of Control
MVVM	Model – View - ViewModel
OS	Operating System
PCB	Process Control Block
PID	Process Identifier
PS	Process Synchronization
TID	Thread / Task Identifier
UI	User Interface
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

Príloha P I: CD s bakalárskou prácou vrátane zdrojových kódov aplikácií

PRÍLOHA P I: CD

Priložené CD obsahuje:

- Bakalársku prácu vo formáte .docx: BP_JozefKovac_A17561.docx
- Bakalársku prácu vo formáte .pdf: BP_JozefKovac_A17561.pdf
- Zdrojové kódy aplikácií: BP_PRACT_JozefKovac_A17561.zip