Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Vizualizace algoritmů vyučovaných v předmětu Operační systémy

Bc. Jaroslav Janíček

Diplomová práce

2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jaroslav Janíček**
Osobní číslo:	**I16219**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vizualizace algoritmů vyučovaných v předmětu Operační systémy**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Budou popsány problémy a algoritmy pro jejich řešení vyučované na předmětu Operační systémy (viz níže). Vždy bude uvedeno několik ukázkových příkladů vhodně zvolených tak, aby demonstrovaly co nejvíce možných typů vstupů a výstupů.
Algoritmy budou vizualizovány či animovány pomocí programu nezávislého na platformně (minimálně pro Windows a Linux, nejlépe i Android). Preferováno je řešení pomocí webových technologií (HTML5 canvas, JavaScript, CSS, Node.js) s dodržením příslušných norem W3C. Pro řešení lze využít existujících knihoven pro grafiku a animaci, např. Paper.js, jQuery (přehled vybraných dostupných knihoven je ve zdroji *Datavisualization.ch Selected Tools*).
Program umožní též parametrizovat vstupy (s vhodně nastaveným omezením). U algoritmů řešících nějaký problém (např. obědvající filosofové) budou implementovány i neřešené či nedořešené varianty demonstrující selhání.
Výčet problémů a algoritmů:
– ukládání parametrů a lokálních proměnných na stack (simulace potřeby samostatného stacku pro každé vlákno),
– plánování a plánovací algoritmy (context switch, FIFO, round-robin, prioritní, EDF),
– řízení přístupu do KS (SW algoritmy včetně nefunkčních, HW algoritmy, problém CPU s pamětí cache, semafor, monitor),
– řešení synchronizace (semafor, bariéra, podmínková proměnná, monitor),
– kombinace KS a synchronizace (obědvající filosofové, producent/konzument),
– alokace paměti (pevné dělení, dynamické dělení: best-fit, first-fit, next-fit, worst-fit, setřesení),
– stránkování (alokace stránek, algoritmy výměny: LRU, FIFO + second chance, clock policy, working set clock policy, page buffering),
– alokace bloků souboru (souvislá, řetězená, indexová),
– zápis se zachováním konzistence souborového systému (žurnál, copy-on-write),
– bezpečnost (buffer overflow).

Rozsah grafických prací:	**10**
Rozsah pracovní zprávy:	**60**
Forma zpracování diplomové práce:	**tištěná**
Seznam odborné literatury:	**viz příloha**

Vedoucí diplomové práce:	**Mgr. Tomáš Hudec** Katedra informačních technologií
Datum zadání diplomové práce:	**30. října 2017**
Termín odevzdání diplomové práce:	**18. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

Univerzita Pardubice
Česká republika

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

V Pardubicích dne 15. listopadu 2017

Prohlašuji:

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

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

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

V Pardubicích dne

Jaroslav Janíček

PODĚKOVÁNÍ

Děkuji vedoucímu práce, Mgr. Tomáši Hudcovi, za poskytnutí příležitosti vypracovat práci v jazyce JavaScrpit, jelikož se o tento jazyk již delší dobu zajímám a neměl jsem příležitost v něm vytvořit větší projekt. Také samozřejmě děkuji za všechny konzultace, které byly velmi přínosné a jejich trvání se někdy značně protáhlo. Díky konzultacím bylo možné vytvořit potřebné scénáře k požadovaným vizualizacím.

ANOTACE

Práce se věnuje vizualizaci algoritmů a využívání nástrojů IPC v operačních systémech. Vizualizace byly vypracovány v jazyce JavaScript a zobrazují stěžejní situace usnadňující pochopení daného tématu. Vizualizace jsou buď statické, dynamické nebo interaktivní a zahrnují například plánovací algoritmy, algoritmy výměny stránek, algoritmy alokace paměti, nástroje IPC a další.

KLÍČOVÁ SLOVA

operační systémy, algoritmus, IPC, vizualizace, animace, JavaScript

TITLE

Visualization of Algorithms Taught in the course Operating SystemsCourse

ANNOTATION

The thesis is devoted to visualization of algorithms and usage of the IPC tools in operating systems. The visualizations were developed in the JavaScript language and show important situations that ease understanding given topics. Visualizations are either static, dynamic, or interactive and include scheduling algorithms, page replacement algorithms, memory allocation algorithms, the IPC tools and others.

KEYWORDS

operating systems, algorithm, IPC, visualization, animation, JavaScript

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM ZKRATEK

ECMA	European Computer Manufactures Association
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
W3C	World Wide Web Consortium
LESS	Leaner Style Sheets
IDE	Integrated Development Environment
UWP	Universal Windows Platform
T#	Task, Thread
CPU	Central Processing Unit
RAM	Random Access Memory
FIFO	First In First Out
IRQ	Interrupt ReQuest
EDF	Earliest Deadline First
IPC	Inter-Process Communication
LRU	Least Recently Used
GIF	Graphics Interchange Format
FPS	Frames Per Second
SW	Software
FEI	Fakulta Elektrotechniky a Informatiky
PC	Personal Computer

ÚVOD

Operační systémy jsou v dnešní době nedílnou součástí oboru informačních technologií. Rozsáhlost operačních systémů a fungování jednotlivých algoritmů nebo metod, které používají, mohou být velmi těžko pochopitelné, a proto může vizualizace (animace) jednotlivých algoritmů či metodik velmi usnadnit pochopení.

V rámci práce dochází ke stručnému vysvětlení algoritmů a metod, které používá operační systém například pro alokaci paměti, výměnu stránek, či jsou systémem nabízeny pro řešení vzájemného vylučování (kritické sekce) a synchronizace. Pro každý popsaný algoritmus byla vypracována vizualizace (animace), která graficky znázorňuje jeho průběh.

Pro každou vizualizaci byl zhotoven popis o průběhu animace, jak daná vizualizace vypadá a co reprezentují jednotlivé grafické objekty. Každá vizualizace se řídí buď předem určeným scénářem nebo je její průběh zcela dynamický nebo využívá interakci s uživatelem. Některé vizualizace obsahují více scénářů a mohou znázorňovat i nefunkční návrhy řešení.

Práce obsahuje návod pro vytvoření vizualizace pomocí šablony, která byla vytvořena v průběhu vypracovaní práce.

1 VÝBĚR TECHNOLOGIE

Při výběru technologií bylo třeba dodržet podmínky ze zadání diplomové práce, neboli technologie platformově nezávislé s preferencí webové technologie HTML, CSS a JavaScript. Jelikož hlavní náplní práce je grafická vizualizace, bylo zapotřebí zvolit vhodnou grafickou knihovnu pro jazyk JavaScript.

1.1 JavaScript

Jedná se o interpretovaný programovací jazyk, který v současné době patří mezi nejpopulárnější programovací jazyky na světě. Jeho využití je především ve webových aplikacích, především na front end s využitím technologie React. Díky jeho oblibě se dostal i na back end v podobě technologie Node.js, mezi oblíbené frameworky patří například Express.js nebo Meteor.js.

JavaScript však trpěl jedním problémem, a to různou interpretací tohoto jazyka v různých webových prohlížečích. Například, jedna totožná funkce nemusí fungovat stejně ve dvou různých prohlížečích nebo jeden z webových prohlížečů danou funkci nepodporuje.

Tento problém však řeší formální standardizace jazyka ECMA. V roce 2009 přichází standard ECMA 5, jenž přináší striktní režim a práci s datovým formátem JSON. Standard ECMA 5 je momentálně podporovaná všemi více rozšířenými webovými prohlížeči, jako je Google Chrome, Opera, Mozilla Firefox a Microsoft Edge. [1]

V roce 2018 ve zmíněných webových prohlížečích je již částečná podpora ECMA 6 a zároveň vznikla ECMA 7 [2]. Pokud bychom chtěli využít standard ECMA 6, bylo by zapotřebí použít Babel.js, jenž převede kód z ECMA 6 na požadovaný standard do ECMA 5.

Pro JavaScript existují různé nadstavby, knihovny či různé jazykové modifikace, jako je například knihovna jQuery, TypeScript, Vanila.js nebo PaperScript, jež jsou v jádru jsou stále JavaScript, ale navenek se většinou liší jejich syntaxe.

Jelikož mé osobní zkušenosti s programovacím jazykem JavaScript byly velmi slabé, došlo se po prostudování a testování tohoto jazyka k závěru, že bude v rámci sebevzdělání nejlepší využít standard ECMA 5, jenž je podporován všemi prohlížeči, a není nutné využívat Babel.js, popřípadě jinou externí knihovnu, jako je třeba jQuery.

1.1.1 Node.js

Díky společnosti Google, která v roce 2008 vytvořila vlastní webový prohlížeč Google Chrome a zároveň v tomto kroku vytvořila novou implementaci jazyka JavaScript s jádrem nazývaným V8, umožnila oživení myšlenky, že jazyk jako takový nemá důvod být používán jen v rámci klientského skriptování a lze jej využít pro programování serverového kódu. [1]

V roce 2009 vzniká projekt Node.js, který obsahuje sadu rozhraní a knihoven včetně implementace V8, a dovoluje tak jednoduše spouštět javascriptový kód i mimo webový prohlížeč [1]. Po instalaci Node.js je k dispozici balíčkovací systém npm pro instalaci javascriptových knihoven například: React.js, Express.js, bower atd.

Tato technologie byla brána v úvahu z důvodu rozšíření diplomové práce, a to o virtuální prostor, ve které by vyučující založil učebnu. Studenti by se do této virtuální učebny přihlásili a vyučující by jim na monitor jejich zařízení mohl promítat vytvořené vizualizace algoritmů a pouze on by mohl pozastavovat či spouštět danou vizualizaci. Dále by mohla být přidána funkcionalita kreslení přímo do přehrávané vizualizace, čímž by se dal upřesnit výklad dané látky.

V době, kdy byla prováděná analýza technologií, tedy v listopadu a prosinci roku 2017, bylo možné zvolenou grafickou knihovnu Paper.js sice doinstalovat do Node.js pomocí balíčkovacího systému npm, ale po instalaci a následném testování se ukázalo, že v dané době byla v Paper.js v distribuci pro Node.js kritická chyba, která zabraňovala vykreslovat. Z toho důvodu se od této technologie odstoupilo, ale nadále byl využit balíčkovací systém npm.

1.1.2 Knihovna Paper.js

Knihovna Paper.js je grafická vektorová knihovna, která umožňuje vytvářet grafická primitiva jako je čtverec, trojúhelník, kružnice apod. Dále umožňuje vytvářet interakce s grafickou komponentou na klik, pohyb myši nebo na vstup z klávesnice. Všechny grafické objekty jsou vykreslovány pomocí tzv. canvas, který je součástí HTML5.

Stránky Paper.js obsahují mnoho ukázkových aplikací, animací a dokonce i jednoduchou videohru. Balíček stažený přímo ze stránek *paperjs.org* obsahuje vše zmíněné včetně různých provedení a samotné grafické knihovny, jak v plném formátu, tak v minimalistickém, jenž neobsahuje bílé znaky, komentáře ani dokumentaci a tím je zmenšena velikost souboru.

Paper.js využívá PaperScript, neboli JavaScript, s přidanou podporou matematických operátorů (+ - * / %) pro objekty typu Point a Size. Kód PaperScript se automaticky provádí ve svém vlastním rozsahu (scope), který má stále přístup ke všem globálním objektům a funkcím prohlížeče, jako je document nebo window. Ve výchozím nastavení knihovna Paper.js exportuje pouze jeden objekt do globálního scope – objekt s názvem paper. [3]

Pokud však nechceme využít PaperScript, umožňuje Paper.js použít čistý JavaScript. V tomto případě je zapotřebí přidat alokaci canvas pro Paper.js a ke všem funkcím a objektům Paper.s přistupovat přes globální proměnnou s názvem paper. Přístupem přes globální proměnnou paper se vyhneme možným problémům v budoucnu.

Tato grafická knihovna byla vybrána kvůli velmi dobře provedené výukové stránce, dále kvůli práci s vektorovou grafikou umožňující základní transformace jako scale, rotate, kvůli možnosti vytvářet skupiny grafických objektů a v poslední řadě především kvůli velké podobnosti s technologií JavaFX, která se vyučuje na FEI. Pro řešení diplomové práce byly zváženy i jiné grafické knihovny, například: Pixi.js, nebo Two.js, ale oproti Paper.js se jeví zbytečně rozsáhlé či složité.

1.2 HTML5 a CSS3

HTML je značkovací jazyk určený pro vytváření webových stránek. Aktuální verze HTML5 je podporovaná všemi nejrozšířenějšími webovými prohlížeči. Mezi největší novinky HTML5 patří vkládání multimediálního obsahu a přidává také nové možnosti vstupních polí formuláře, jako je date, color, email, range, a navíc je tu novinka vykreslovat grafiku do canvas pomocí WebGL.

CSS je kaskádový jazyk, který je určený k popisování vzhledu elementů jazyka HTML. Cílem vzniku CSS bylo oddělení vzhledu webové stránky od jejího významu. Jazyk CSS navrhlo sdružení W3C. [4]

Aktuální verze CSS3 využívá nové jednotky závislé na velikosti okna – vh a vw – a zároveň umožňuje vytvářet responzivní web design pomocí výčtu omezujících pravidel s klíčovým slovem *@media* bez nutnosti použití jazyka JavaScript. Pro více odvážné je možné pomocí CSS3 vytvářet dokonce i grafiku a animace.

1.2.1 Less

Less je jazykové rozšíření pro CSS. Oproti CSS umožňuje definovat proměnné a tím usnadnit práci při tvorbě CSS. Less je součástí balíčkovacího systému *npm*, proto je nutné mít nainstalovaný Node.js, který tento balíčkovací systém obsahuje.

Less lze nainstalovat příkazem *npm install -g less*, kde parametr *-g* znamená globální instalaci. Po této instalaci lze využívat less bez nutnosti instalace pro každý projekt a lze jej využívat prostřednictvím konzole nebo pomocí vývojových IDE.

Po napsání skriptu less je zapotřebí tento skript překompilovat do standardního CSS pomocí příkazu *lessc*. V případě využití IDE, jako je na příklad WebStorm od společnosti JetBrains, lze vytvořit File Watcher, který tuto kompilaci provádí automaticky po uložení či změně ve zdrojovém souboru less.

Alternativní možností je tento překlad provádět přímo v prohlížeči za pomoci knihovny less.js. Při zvolení této metody stačí do zdrojového dokumentu HTML přidat příslušný tag pro načtení skriptu.

1.2.2 Validační standardy W3C

World Wide Web Consortium neboli zkráceně W3C je mezinárodní konsorcium, které vyvíjí webové standardy pro World Wide Web. Pro validaci, zda jsou standardy dodrženy, lze využít onlinový nástroj W3C – validátor na stránkách *validator.w3.org*, kde lze zadat adresu stránky, nahrát zdrojový soubor nebo přímo vložit zdrojový kód do textového pole.

Po kontrole je předána informace, zda je námi kontrolovaný vstup validní či ne. Upozornění lze ignorovat, ale i přesto by bylo lepší se jim vyvarovat. Sám validátor vypisuje možné řešení problému. Nalezené chyby by měly být odstraněny, ale najdou se případy, kdy to není možné, například sdílení videa od služby YouTube. Pro sdílené video je vytvořen *iframe*, který svými atributy porušuje standardy W3C. Stejně tak *iframe* pro vložení mapy od Google Maps. Nevalidní atributy můžeme odstranit a doufat, že funkce daného tagu zůstane nezměněna. Popřípadě je nutno tyto standardy porušit, když jsme k tomu nuceni službami třetích stran.

1.3 Shrnutí

Pro vizualizaci algoritmů byly vybrány následující technologie: Programovací jazyk JavaScript ve standardu ECMA 5, grafická knihovna Paper.js bez využití PaperScript, HTML5 pro vykreslování v canvas, což obstarává právě knihovna Paper.js a less pro snazší tvorbu stylů CSS a jejich následnou úpravu.

Tento výběr technologií by měl umožnit výslednou vizualizaci bez větších problémů a přidat ji do již existující webové stránky nebo využít možnosti portace do UWP (Universal Windows Platform). Aplikace UWP lze spustit na zařízeních PC, Xbox, Hololens, jež obsahují operační systém Windows 10 od společnosti Microsoft.

2 ALGORITMY A JEJICH VIZUALIZACE

V této kapitole dojde k seznámení s algoritmy, které byly vybrány pro vizualizaci nebo s nimi úzce souvisejí. Je zde obsažena většina vyučovaných algoritmů v předmětu Operační systémy z bakalářského studia na Univerzitě Pardubice.

V první části jednotlivých podkapitol dojde k vysvětlení principu algoritmu a v druhé části bude vysvětlen průběh vypracované vizualizace.

2.1 Alokace paměti

Mezi techniky alokace paměti patří pevné a dynamické dělení paměti, stránkování a segmentace. Pevné dělení paměti rozdělí dostupnou paměť do oblastí (partitions) s pevnými hranicemi. Tyto oblasti mohou mít stejné nebo různé velikosti. Při použití pevného dělení pamětí vzniká vnitřní fragmentace. [5]

Při dynamickém dělení není paměť rozdělena do pevně rozdělených oblastí. Procesu se při startu přidělí tolik paměti, kolik potřebuje. Tím je odstraněna vnitřní fragmentace paměti. Vzniká však jiný problém, a to ten, že po ukončení procesu vznikne v paměti mezera, do které lze vložit stejně velký nebo menší proces. Z pravidla se do této mezery vejde pouze menší proces a tím se mezera postupně zmenšuje až do velikosti, kdy je obtížné ji zaplnit. Vzniká tak vnější fragmentace paměti. Vnější fragmentaci lze odstranit pomocí setřesení (compaction), kdy se všechny procesy seřadí za sebe a na konci paměti vzniká jedna velká oblast volné paměťi. [5]

Best-fit

„Algoritmus best-fit vybírá takovou nejmenší volnou oblast, do které se proces ještě vejde. Cílem je minimalizovat (vnější) fragmentaci. Důsledkem je však rychlé přibývání malých fragmentů, které lze jen velmi obtížně využít – nutnost častého provádění setřesení. Díky nutnosti prohledávat celou paměť (hledání nejmenšího volného bloku) je tato metoda nejméně výkonná.“ [5]

Worst-fit

„Algoritmus worst-fit vybírá největší volnou oblast, do které je proces následně umístěn. Má tendenci dělit velké oblasti paměti na menší, což v důsledku vede k nemožnosti přidělení paměti procesu s velkou paměťovou náročností. Tato metoda alokace má nejhorší využití paměti.“ [5]

„Jelikož je třeba stejně jako u algoritmu best-fit prohledat celou paměť, může se algoritmus modifikovat na variantu exact-or-worst-fit, kdy se proces přednostně umístí do přesně vyhovující oblasti, je-li taková nalezena.“ [5]

First-fit

„Algoritmus first-fit hledá první vyhovující oblast, do které se proces vejde. Paměť je prohledávána vždy od začátku. Protože není potřeba prohledávat celou paměť, je tato metoda rychlejší než předchozí. Prohledávání zpomaluje výskyt velkého počtu obsazených oblastí na začátku paměti, tato část paměti se vždy zbytečně prohledává. Vylepšením v tomto ohledu je následující algoritmus.“ [5]

Next-fit

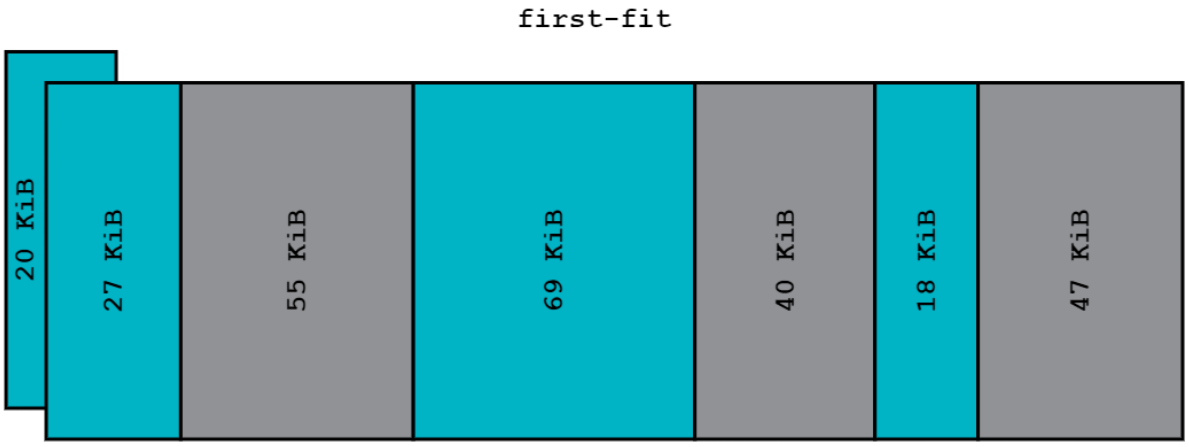
„Algoritmus next-fit stejně jako first-fit vybírá první volnou oblast, do které se proces vejde, ale paměť se prohledává vždy od oblasti, do které se naposledy umisťovalo. Je-li po umístění procesu do volné oblasti zbytek oblasti ještě dostatečně velký pro další proces, umístí tento sem. Algoritmus má tendenci umisťovat častěji ke konci paměti, neboť je tam obvykle nejvíce volného místa. Na rozdíl od first-fit má větší tendenci dělit velké oblasti paměti na menší. Jedná se o nejrychlejší metodu.“ [5]

2.1.1 Vizualizace alokace paměti

Pro vizualizaci paměti byla zhotovena plně dynamická animace, která umožňuje zobrazit princip umisťovacích algoritmů best-fit, worst-fit, first-fit a next-fit. Uživatel může mezi těmito algoritmy přepínat pomocí výběrového menu a také si může zvolit rychlost animace. Jelikož je vizualizace plně dynamická, není možno zcela předpovídat její průběh. Z tohoto důvodu byla pro všechny algoritmy vytvořena stejná výchozí situace. Díky tomu je výsledek umístění prvního procesu do paměti vždy stejný, následující velikosti procesů jsou již generovány náhodně.

Obrázek 1: Ovládací prvky vizualizace alokace paměti

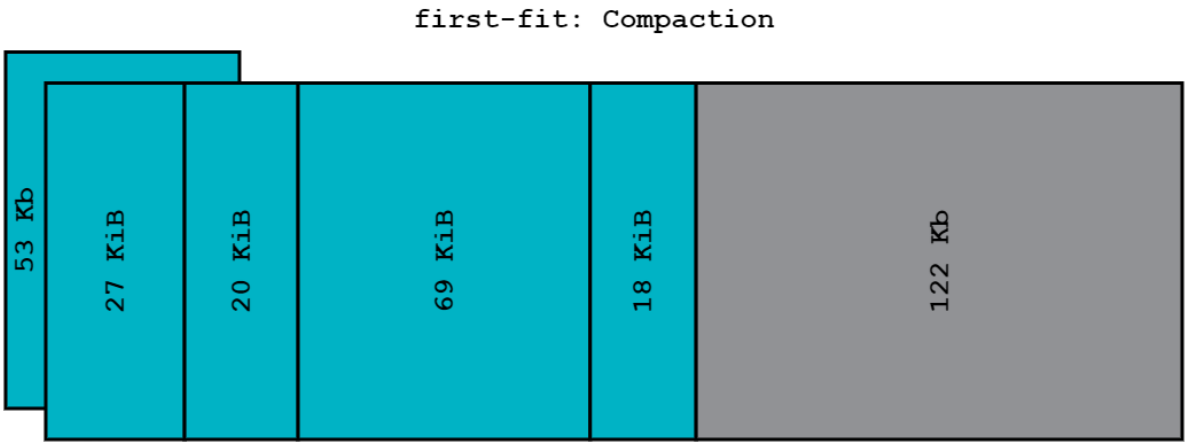
Postup vizualizace je následující. V prvním kroku je zobrazen výchozí stav paměti, který už obsahuje tři procesy a zároveň jsou mezi procesy volné bloky paměti. V levé části je zobrazen proces, který se má přidat do paměti. Po spuštění animace je proces přemístěn k prvnímu volnému bloku paměti. Následně je v titulku animace vypsána informace, zda volný blok vyhovuje pro umístění procesu. Pokud ano, je v případě algoritmů first-fit a next-fit tento proces přidán do této volné oblasti.



Obrázek 2: Výchozí stav vizualizace first-fit

Pokud volná oblast nesplňuje požadavky pro umístění procesu, pokračuje animace přemístěním procesu k dalšímu volnému bloku v pořadí. Tato animace se opakuje, dokud proces nedojde na konec paměti.

Pokud se dojde na konec paměti v případě first-fit a next-fit, znamená to, že paměť je buď plná, nebo je volná oblast rozdělená do příliš malých oblastí. Nastat mohou dva jevy, buď animace začne simulovat swap-out, neboli přesunutí procesu na disk, a tím uvolní paměť pro další procesy, nebo dojde k setřesení (compaction). Při setřesení postupně animace přesune všechny volné bloky na konec paměti, kde společně vytvoří jeden velký volný blok paměti.



Obrázek 3: First-fit po setřesení (compaction)

Pokud se dojde na konec paměti v případě u algoritmů best-fit a worst-fit, znamená to buď stejný případ jako u first-fit a next-fit nebo během prohledávání nebyl nalezen volný blok se stejnou velikostí. Pokud proces došel na konec a našel se vyhovující volný blok, je procesu tento blok přidělen.

Následně se v levé části animace vytvoří nový proces, který má být umístěn do paměti, a celý proces se opakuje, dokud uživatel animaci nepozastaví či nerestartuje.

2.2 Plánovací algoritmy

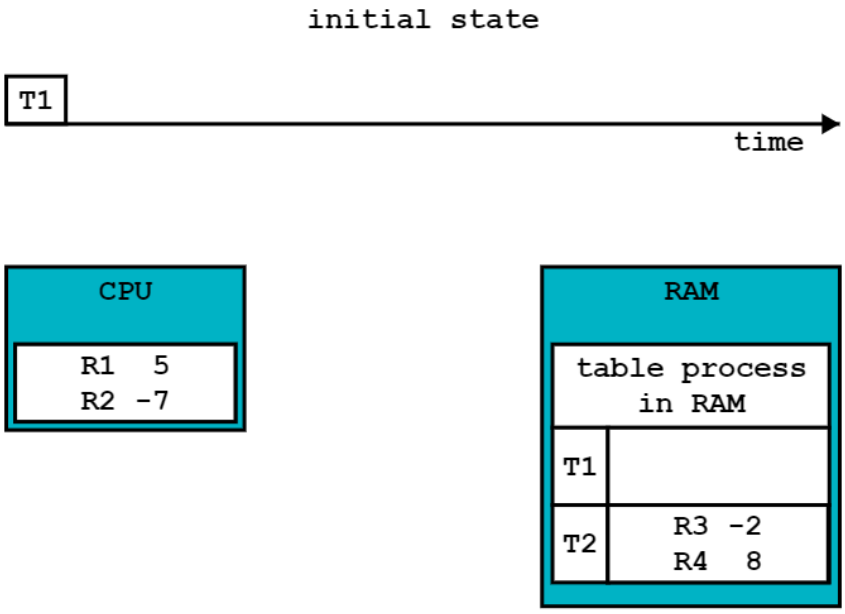
Cílem plánovacích algoritmů je rovnovážné zatížení systému a tím optimalizovat jeho výkon. Spravedlnost plánování zajistí, aby každý proces měl stejný přidělený čas a také by měla být dodržena zvolená strategii, která může být pro různé systémy jiná. [6]

Typicky lze rozdělit plánovací algoritmy na dvě kategorie preemptivní a nepreemptivní. Preemptivní plánovací algoritmy jsou takové algoritmy, které umožňují přerušit signálem přerušení právě běžící proces s pomocí časovače. Díky této vlastnosti plánovací algoritmus může efektivně plánovat procesy, které se dělí o procesor. Nepreemptivní plánovací algoritmy nemají možnost přerušit právě provádějící proces. Neboli proces se musí sám vzdát procesoru nebo blokovat. [6]

2.2.1 Context-switch

Context-switch je činnost systému, kdy se provádění jednoho procesu v CPU přepne na provádění jiného. Proces uloží aktuální stav (context) z procesoru do paměti a následně je načten kontext jiného procesu z paměti do procesoru. Rozhodnutí o tom, který proces má být načten z paměti, má na starosti právě plánovací algoritmus. [7]

Při vizualizaci context-switch je ve výchozím stavu uživateli zobrazena časová osa, která obsahuje pouze T1. Po spuštění animace se na časové ose zobrazí právě provádějící úkony.



Obrázek 4: Počáteční stav context-switch

Prováděné úkony jsou následující: *save*, *scheduling*, *restore*. Při provádění *save* se obsah registrů CPU bloku přesune do bloku RAM. Při provádění *scheduling* se zobrazí pouze blok na časové ose, jenž symbolizuje průběh plánovacího algoritmu. Během *restore* dochází přesunu T2 z RAM do CPU. Celá animace je následně ukončena zobrazením T2 na časové ose, která znázorňuje zpracování T2 procesorem.

2.2.2 FIFO

Plánovací algoritmus FIFO je nepreemptivní algoritmus. „Nové úlohy se zařadí do fronty a po ukončení aktuálního procesu se přidělí CPU proces, který čekal ve frontě nejdéle. Krátké procesy musejí zbytečně dlouho čekat. Zvýhodněny jsou výpočtově orientované procesy bez V/V

operací, které čekají pouze jednou. Procesy s mnoha V/V operacemi čekají při každém dokončení operace, než se jiný proces vzdá CPU.“ [6]

2.2.3 Round-Robin

„Round-Robin je založen na cyklické obsluze připravených procesů. Jedná se o preemptivní algoritmus, kdy plánovač je periodicky aktivován při obsluze přerušení časovače. Každý proces dostane přidělen časový interval pro využití CPU. Přepnutí na jiný proces (context switch) je prováděno při vypršení tohoto intervalu nebo při volání blokujícího systémového volání procesem, který má právě předělen procesor.“ [6]

2.2.4 Prioritní plánování

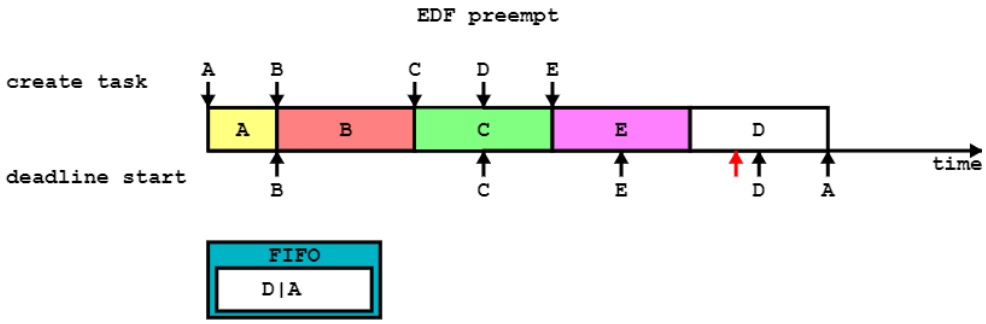
„Každý proces má přidělenou prioritu, což je obvykle celé číslo. Při výběru, který proces dostane CPU, se dává přednost procesu s vyšší prioritou. Obvykle je pro každou prioritu samostatná fronta. Nízká priorita může mít za následek vyhladovění procesu.“ [6]

2.2.5 EDF

EDF je plánovací algoritmus, jenž je využit v operačních systémech reálného času. Jedná se o termínové plánování, kdy dokončení všech úloh musí být ve stanoveném termínu. Ke spuštění je vybrána úloha s nebližším termínem zahájení (deadline). Tento algoritmus je optimální na jednoprocesorových systémech umožňujících preempci. Mezi nevýhody tohoto algoritmu patří nutnost znát termíny a doby zpracování všech úloh předem. [8]

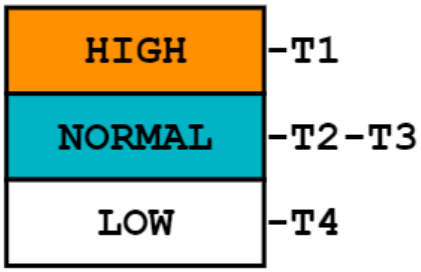
2.2.6 Vizualizace plánovacích algoritmů

Pro vizualizaci plánovacího algoritmu FIFO byla zhotovena jak nepreemptivní, tak i preemptivní varianta. Výchozí stav pro všechny plánovací algoritmy je téměř totožný. Uživateli je zobrazena prázdná časová osa. Výjimkou je algoritmus EDF, jenž na časové ose obsahuje již první proces a šipky znázorňující nejzažší dobu (deadline) pro zahájení daného procesu.



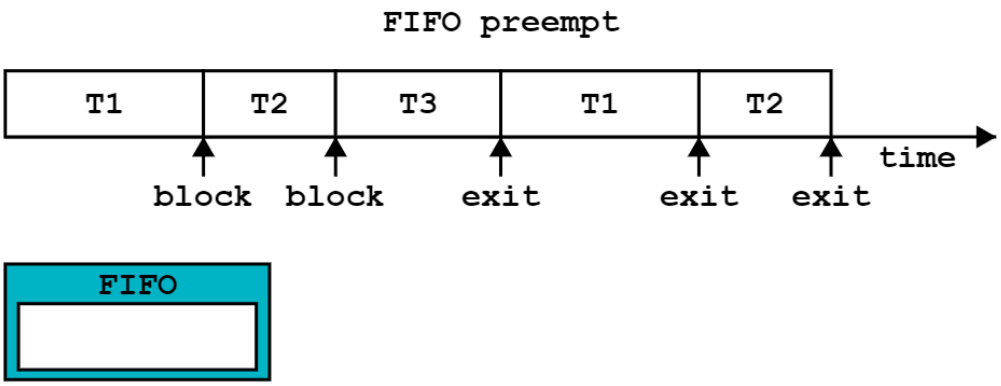
Obrázek 5: Vizualizace EDF v průběhu animace

V levém dolním rohu je zobrazena fronta znázorňující pořadí úloh (task) či vláken (thread), proto bylo zvolené obecné označení T#. V případě prioritního plánovaní jsou zde zobrazeny tři fronty pro priority high, normal a low.



Obrázek 6: Fronty pro prioritní plánování

Po spuštění animace je odebráno T# z příslušné fronty a zobrazeno na časové ose. Každé T# může být buď blokováno nebo ukončeno. Pokud je algoritmus preemptivní, může být přerušen s označením IRQ a přesunut zpátky do příslušné fronty. Animace je ukončena vyprázdněním fronty a ukončením všech T#, jež figurují v dané animaci.



Obrázek 7: Konečný stav vizualizace FIRO preempt

2.3 Kritická sekce a synchronizace

Kritická sekce je oblast, kde více procesů nebo vláken současně přistupuje ke sdíleným prostředkům. Přístup k těmto prostředkům musí být vzájemně výlučný, neboli v každém okamžiku smí být v kritické sekci pouze jediný proces/vlákno, jinak může dojít k nekonzistenci, k uváznutí (deadlock) nebo vyhladovění. Zdrojový kód, jenž obsahuje kritickou sekci, lze rozdělit na části: vstupní sekce, kritická sekce, výstupní sekce a zbytková sekce. [9]

Algoritmus pro řešení exkluzivního přístupu do kritické sekce by měl splňovat následující podmínky: vzájemné vylučování, pokrok v přidělování a omezené čekání. „Přístup do kritické sekce lze implementovat čistě softwarově, případně lze využít speciálních instrukcí procesoru, speciálních systémových volání operačního systému nebo speciálních funkcí a datových struktur programovacího jazyka.“ [9]

2.3.1 Ošetření kritické sekce pomocí SW řešení

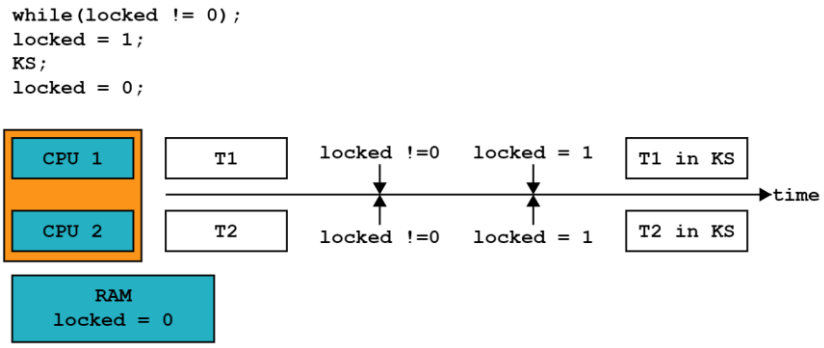
Pro ošetření kritické sekce využívá SW řešení algoritmus pro vstupní a výstupní sekci za pomocí sdílené proměnné například: locked, turn nebo flag[i] pro každý proces. Tato řešení využívají aktivního čekání, neboli procesy čekají na vstup do kritické sekce v cyklu, ve kterém kontrolují proměnné, čímž zbytečně spotřebovávají čas procesoru. [9]

Následující návrhy řešení jsou nefunkční algoritmy, které mají za cíl ukázat studentům, že problematika ošetření vstupu do kritické sekce je netriviální. Po těchto návrzích je uveden na jednoprocesorových systémech funkční Petersonův algoritmus.

Locked

Řešení kritické sekce pomocí proměnné (algoritmu) *locked* je čistě softwarová metoda bez použití IPC, kdy jedna proměnná (*locked*) vyjadřuje, zda lze vstoupit do kritické sekce. Tato metoda je zcela nefunkční, protože dochází k oddělenému čtení a zápisu proměnné bez exkluzivního přístupu. To umožňuje dvěma různým procesům přečíst stejnou hodnotu, kterou následně procesy vyhodnotí jako volnou kritickou sekci a vstoupí do ní současně.

Vizualizace pro řízení přístupu do kritické sekce je vytvořena pro dva procesory. Na časové ose je již dopředu vidět jaký vývoj animace bude následovat. V průběhu animace se posouvá oranžová čtecí hlava po časové ose. Čtecí hlava pouze znázorňuje, co oba procesory dělají ve stejném nebo téměř stejném čase. Celá animace je doprovázena textovým výkladem v horní části vizualizace.



Obrázek 8: Vizualizace kritické sekce za pomocí proměnné

Při prvním posunu čtecí hlavy je znázorněno, že na CPU1 běží T1 a na CPU2 běží T2. Při dalším posunu T1 a T2 vyhodnocují, že mohou vstoupit do kritické sekce a nastavují proměnou *locked* na hodnotu jedna a oba procesy vstupují do kritické sekce.

Turn

Řešení pomocí proměnné (algoritmu) *turn* je obdobné řešení jako řešení pomocí proměnné *locked*. Proměnná *turn* určuje, který proces může vstoupit do kritické sekce. Vizualizace znázorňuje situaci, kdy druhý proces chce vstoupit do kritické sekce a po jejím opuštění opětovně do ní vstoupit.

T1 v prvním kroku nastavuje proměnnou *turn* na hodnotu jedna a tím umožňuje T2 vstoupit do kritické sekce. T1 ukončuje svoji činnost a T2 vstupuje do kritické sekce. Po opuštění kritické

sekce nastavuje T2 proměnnou *turn* na hodnotu nula a následně čeká na vstup do kritické sekce, avšak se nikdy nedočká, jelikož proces T1 již ukončil svou činnost a tím pádem nemůže dojít k nastavení proměnné *turn* na hodnotu jedna.

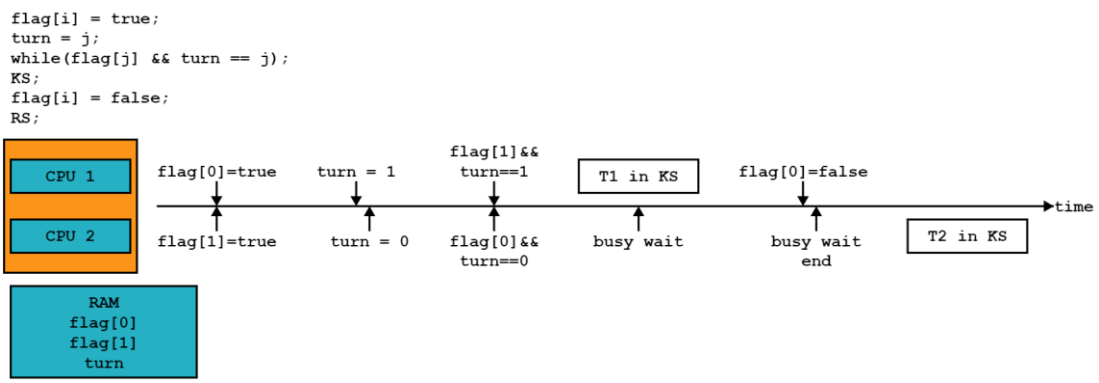
Flag

„Řešení pomocí zavedení sdílené proměnné *flag* pro každý proces *i*, tedy flag[i]. Tato proměnná signalizuje, zda proces požaduje vstup do kritické sekce.“ [9]

Vizualizace pro sdílenou proměnou flag znázorňuje situaci, kdy v prvním kroku T1 a T2 signalizují požadavek o vstup do kritické sekce současně nastavením hodnoty na příslušném indexu pole flag[i] a následně dojde k uváznutí, jelikož obě vlákna T1 a T2 čekají na uvolnění kritické sekce.

Petersonův algoritmus

Petersonův algoritmus je tvořen kombinací algoritmů využívající proměnou *flag* a *turn*. Jedná se o funkční řešení pro dva procesy za předpokladu, že není využita paměť cache nebo je použit jednoprocesorový systém.



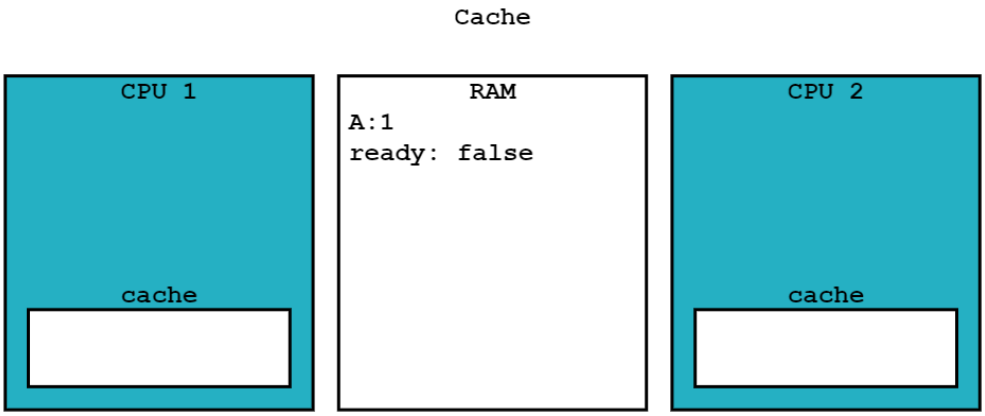
Obrázek 9: Vizualizace Petersonův algoritmus

2.3.2 Problémy s cache

Jelikož je procesor mnohem rychlejší než operační paměť, využívá procesor velmi rychlou mezipaměť zvanou cache. Do této mezipaměti se ukládá část aktuálně prováděného kódu procesu a část jeho dat. Procesor pak nemusí čekat na pomalejší operační paměť. Přitom se využívá

toho, že proces má tendenci přistupovat ke stejné části paměti jako v bezprostředně předcházejícím čase. Tato tendence se označuje jako princip lokality odkazů. [10]

Vizualizace s problémy s cache ve výchozím stavu zobrazuje paměť RAM a dva CPU s vlastní pamětí cache. V bloku RAM jsou dvě proměnné. Proměnná A obsahující číselnou hodnotu a boolovská proměnná *ready*, která udává, zda výpočet na proměnné A byl již proveden.



Obrázek 10: Výchozí stav vizualizace problému s cache

Po spuštění animace CPU2 načte obě proměnné do své paměti cache. Zkontroluje, zda proměnná *ready* je true. Jestliže je hodnota *ready* false, CPU2 aktivně čeká na změnu proměnné *ready*. Načítá proměnou *ready* opakovaně do paměti cache a kontroluje její stav. V dalším kroku načte CPU1 obě proměnné do vlastní paměti *cache* a následně modifikuje proměnou A. Jestliže simulovaný výpočet proměnné A je ukončen, nastaví CPU1 proměnnou *ready* na true.

Následně CPU1 aktualizuje hodnoty proměnných v paměti RAM. CPU2 zaznamená změnu proměnné *ready* na true a ukončí aktivní čekání a následně přečte hodnotu proměnné A. Může však dojít k chybě, jelikož v rámci úspory CPU2 aktualizuje pouze hodnotu proměnné *ready* a proměnou A nechává v paměti cache bez aktualizace, takže CPU2 přečte neaktuální hodnotu proměnné A.

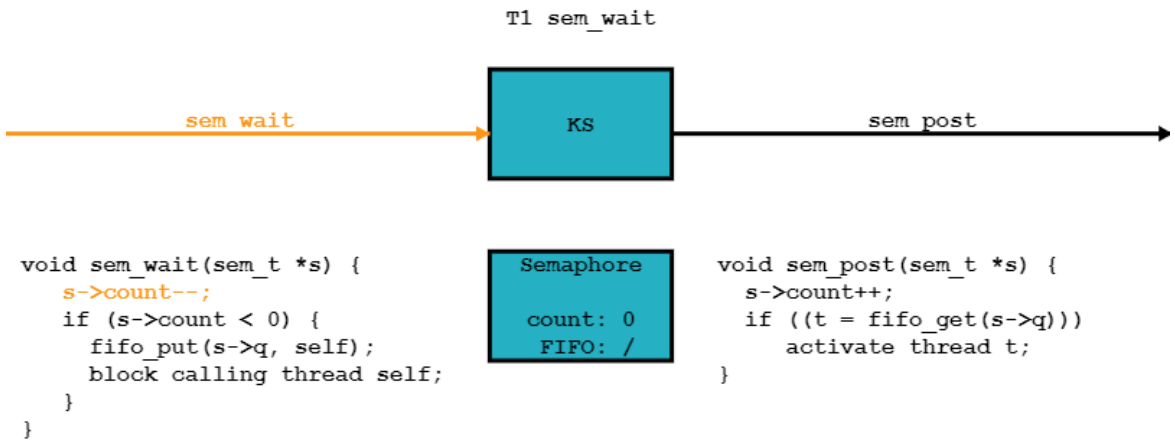
2.3.3 Semafor

„Semafor je nástroj IPC pro řešení problémů synchronizace a kritické sekce. Jedná se o datovou strukturu s celočíselným čítačem, frontou procesů a operacemi inicializace (*init*), snižování (*wait*) a zvyšování čítače (*signal*). K čítači se přistupuje výhradně pomocí atomických operací

init, *wait* a *signal*. Nezáporná hodnota čítače udává, kolikrát lze volat operaci wait bez blokování. Absolutní hodnota záporného čítače udává, kolik procesů čeká ve frontě.“ [11]

Pro semafor byly vytvořeny dvě vizualizace. První znázorňuje fungování semaforu a příslušných funkcí *wait* a *post* (*signal*). Výchozí stav zobrazuje zdrojový kód funkcí, kritickou sekci a samotný semafor s čítačem. V první části je provedena inicializace čítače semaforu, následně T1 volá funkci *wait*. Tím začíná animace průchodem zdrojového kódu funkce *wait*. Jelikož je hodnota čítače nezáporná, smí vstoupit T1 do kritické sekce. Následně volají funkci *wait* vlákna T2 a T3. Hodnota čítače je záporná, tudíž vlákna T2 a T3 jsou přidána do fronty v semaforu a blokována.

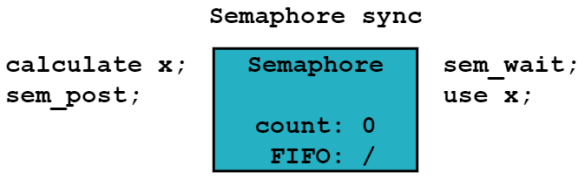
Druhá část vizualizace se skládá z následujících kroků: T1 opouští kritickou sekci a volá funkci *post*. Proběhne animace průchodu kódem funkce *post* a dochází k uvolnění vlákna T2, které vstupuje do kritické sekce. Tento postup se opakuje i pro T3 a tím celá vizualizace pro semafor končí.



Obrázek 11: Vizualizace semaforu animace průchodem zdrojového kódu funkce wait

Semafor lze využít i pro synchronizaci, z toho důvodu byla vytvořena druhá vizualizace pro semafor, která má dva různé scénáře. Scénář si může uživatel libovolně přepnout.

Vizualizace pro synchronizaci simuluje použití vypočítané proměnné *x*. První scénář znázorňuje situaci, kdy T1 vypočítá hodnotu proměnné *x* a následně zavolá funkci *post*. Tím je nastaven čítač semaforu na hodnotu jedna. T2 zavolá funkci *wait* a sníží hodnotu čítače na nulu. Hodnota čítače není záporná a T2 smí použít proměnou *x*.

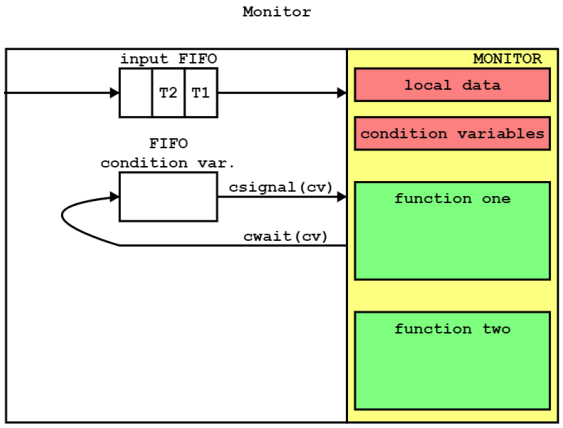


Obrázek 12: Vizualizace semaforu při synchronizaci

Druhý scénář znázorňuje stejnou situaci, ale v opačném pořadí plánování vláken. T1 volá funkci *wait*, tím snižuje hodnotu čítače na mínus jedna a je tedy přidáno do fronty semaforu a blokováno. V dalším kroku T2 vypočítá hodnotu proměnné *x* a volá funkci *wait*. Hodnota čítače se zvyšuje na hodnotu nula. Dochází k odebrání T1 z fronty semaforu a aktivaci T1. T1 nyní smí použít vypočítanou proměnou *x*.

2.3.4 Monitor

„Jedná se o konstrukci programovacího jazyka velmi podobnou třídě. Všechny lokální proměnné jsou přístupné pouze pomocí funkcí monitoru. Nejdůležitější vlastností monitoru je, že uvnitř monitoru smí být v každém okamžiku nejvýše jedno vlákno (proces). Tím je zaručeno vzájemné vylučování. Synchronizaci lze řešit pomocí podmínkových proměnných monitoru. S každou podmínkovou proměnnou je vytvořena fronta pro vlákna čekající na danou podmínku. Nad podmínkovými proměnnými jsou implementovány operace *condition_wait* a *condition_signal*.“ [12]



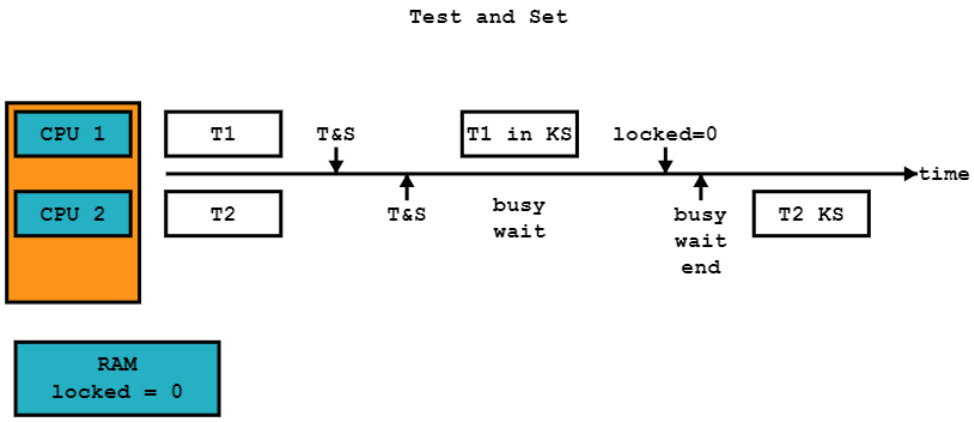
Obrázek 13: Vizualizace monitoru

Vizualizace monitoru probíhá následovně. Vstupní fronta obsahuje T1 a T2. T1 vstupuje do function one a T2 se posouvá ve vstupní frontě na první pozici. T1 volá *condition_wait* a přesouvá se do fronty podmínkových proměnných (FIFO condition *var*.). T2 vstupuje do function two a volá *condition_singnal* a spouští vizualizaci. T1 tedy může opustit frontu podmínkových proměnných a přesouvá se do function one.

2.3.5 Test-and-set

Jedná se o hardwarové řešení vstupu do kritické sekce. K přerušení procesu může dojít pouze na hranicích instrukcí, tedy po dokončení jedné instrukce a před začátkem další. Test-and-set v jedné atomické instrukci přečte příznak a současně jej nastaví. Je-li příznak již nastaven, tak nové nastavení nic nezmění. Jelikož je instrukce nepřerušitelná, jsou obě operace provedeny atomicky. Toto řešení lze využít pro přístup do více kritických sekcí s využitím vytvoření nové proměnné pro každou kritickou sekci. [9]

Řešení s využitím test-and-set je vizualizováno pro dva procesory. Na časové ose je již dopředu vidět jaký vývoj animace bude následovat. V průběhu animace se posouvá oranžová čtecí hlava po časové ose, podobně jako výše u SW algoritmů.



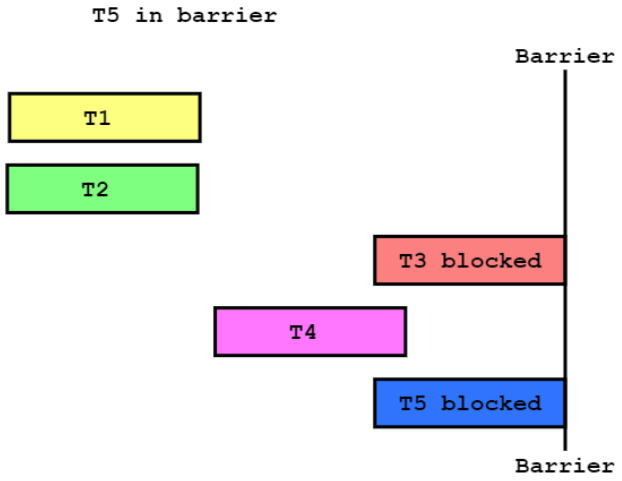
Obrázek 14: Vizualizace test-and-set

Při prvním posunu čtecí hlavy je znázorněno, že na CPU1 běží T1 a na CPU2 běží T2. V druhém posunu T1 zavolá test-and-set a hned vzápětí zavolá test-and-set T2, ale příznak je již nastaven, takže musí aktivně čekat. Ve třetím posunu vstupuje T1 do kritické sekce a T2 aktivně čeká. Po opuštění T1 z kritické sekce T2 ukončuje aktivní čekání a vstupuje do kritické sekce.

2.3.6 Bariéra

Bariéra slouží k synchronizaci skupiny vláken. Bariéra je inicializovaná na určitý počet vláken. Jakmile vlákno dorazí k bariéře, je blokováno až do okamžiku, než k bariéře dorazí inicializovaný počet vláken. [7]

Při vizualizaci bariéry jde především o zdůraznění toho, že bariéra je pomyslná zastávka, ve které se zastaví všechna vlákna T# a čekají na ostatní T#. T# jsou rozložena náhodně v levé části vizualizace a buď mají nastavenou stejnou rychlost, nebo každé T má náhodně vygenerovanou rychlost, která znázorňuje, že každé vlákno může postupovat programem k bariéře jinou rychlostí. Uživatel může zvolit, zda T# mají náhodně generované rychlosti nebo mají všechny stejnou rychlost. Po spuštění animace se T# začnou přibližovat k bariéře, když T dosáhne bariéry, změní se jeho stav na blokovaný. Jakmile všechna T# dosáhnou bariéry, je tato bariéra uvolněna.



Obrázek 15: Vizualizace bariéry

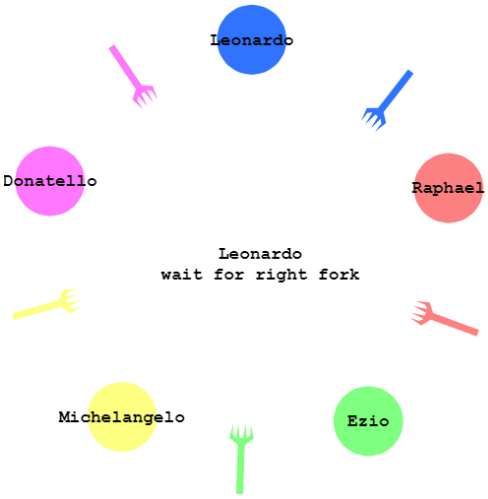
2.3.7 Obědvající filozofové

Jedná se o další příklad problému souběhu. Filozofové sedí u stolu a buď přemýšlí, nebo jedí. Filozof k jídlu potřebuje levou a pravou vidličku. Vidliček je však málo, a proto se musí filozofové o tyto vidličky dělit. Rozhodne-li se filozof najíst, zvedne prvně levou vidličku, následně pravou vidličku a začne jíst. Potom obě vidličky postupně uvolní. [7]

Problematické je používání sdílených vidliček, na které je třeba zavést kritické sekce. K problému dochází také v případě, kdy se rozhodnou všichni filozofové najíst. Zvednou levou vidličku, avšak po jejich pravici je vidlička již zabrána jiným filozofem, a tudíž každý čeká na její uvolnění – dochází k deadlocku. Řešení tohohle problému je prosté, stačí, aby současně jedlo pouze *n - 1* filozofů. [7]

Vizualizace filozofů je odlišná od všech předchozích, jelikož se nejedná o předem danou animaci, ale o interaktivní minihru. Uživateli je zobrazeno pět filozofů, Leonardo, Donatello, Raphael, Michelangelo a Ezio, odlišujících se barvou talíře. Po kliknutí na příslušný talíř provede filozof jeden z pěti úkonů dle svého stavu: zvednutí levé vidličky, zvednutí pravé vidličky, jedení, položení pravé vidličky a na závěr položení levé vidličky. Zvedat lze pouze volné vidličky.

V základním nastavení se uživatel nemůže dostat právě do již zmiňovaného deadlocku, kdy všichni filozofové čekají na uvolnění vidličky. Uživatel však může toto nastavení změnit pomocí checkboxu a tím se dostat do dedalocku v průběhu klikání na jednotlivé filozofy.



Obrázek 16: Vizualizace objedvajících filozofů ve stavu deadlock

2.3.8 Producenti a konzumenti

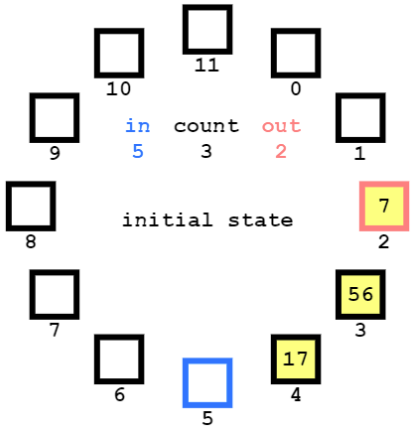
Jedná se o situaci, kdy producenti a konzumenti sdílí stejný úložný prostor, zpravidla kruhový buffer. Producenti vkládají položky do bufferu na volná místa a konzumenti je odebírají. V případě, že je buffer plný, dochází k uspání producenta, který je probuzen v případě, kdy konzument odebere z bufferu jednu nebo více položek. Stejným způsobem lze řešit případ, kdy je buffer prázdný – dojde k uspání konzumenta, který se probudí, až v případě, kdy producent vloží do bufferu. [7]

Na první pohled se může zdát problematika jednoduchá, nesmí se však zapomenout na to, že producentů a konzumentů může být více a může dojít k zapsání producentem do části bufferu, která je již obsazená. Exkluzivní přístup do paměti lze zajistit pomocí mutexu a pro synchronizaci obsazenosti bufferu lze využít semafor. [7]

Producenti a konzumenti mají vytvořenou vizualizaci se čtyřmi scénáři. První znázorňuje situaci, kdy do bufferu vkládají dva producenti na stejnou pozici (index *in*) a následně posunou hodnotu indexu *in* o dvě pozice (každý o jednu), čímže se přeskočená pozice stává neinicializovanou.

Druhý scénář znázorňuje podobnou situaci, kdy dva konzumenti čtou (odebírají) ze stejné pozice v bufferu (index *out*) a následně posunou hodnotu indexu *out* o dvě pozice, čímž nedojde ke zpracování položky na přeskočené pozici.

Třetí scénář znázorňuje situaci, kdy producent zapisuje do plně zaplněného bufferu, takže přepíše existující položku.



Obrázek 17: Vizualizace producent konzument

Čtvrtý scénář znázorňuje správnou obsluhu bufferu. Zápis do bufferu může v jeden okamžik provést pouze jeden producent, a číst (odebírat) smí pouze jeden konzument. Po spuštění animace dojde k nastavení výchozího stav u bufferu a následně je třikrát vkládáno do bufferu producentem. Poté čte konzument dvě položky z bufferu a následně je opět vloženo konzumentem do bufferu a tím celá animace končí. Po každém vložení a čtení z bufferu, jsou aktualizovány hodnoty indexů *in*, *out* a počtu položek v bufferu *count*.

2.4 Algoritmy výměny stránek

V případě, že nastane page fault při zaplněné paměti RAM, má operační sytém na výběr ze dvou možností. Buď odstraní stránku, která byla modifikována, v tom případě před nahrazením stránky musí dojít k zápisu této stránky na disk, nebo nahradí stránku, která nebyla modifikována, tudíž není nutné se zdržovat zápisem na disk a lze rovnou danou stránku nahradit. [7]

Na první pohled se může zdát celá problematika snadná, opak je však pravdou. V první řadě se musí určit, která stránka má být přepsána. Stránku určenou pro nahrazení lze vybrat pomocí náhodného výběru. Tento způsob má však velké nedostatky. Může dojít k odstranění stránky, která vzápětí je vyžadována a musí se tak čekat na její opětovné načtení do paměti. Nebo dojde k častému výběru stránky, která byla modifikována a musí být uložena na disk, což je vzhledem k řádově pomalejší sekundární paměti velmi časově náročné. [7]

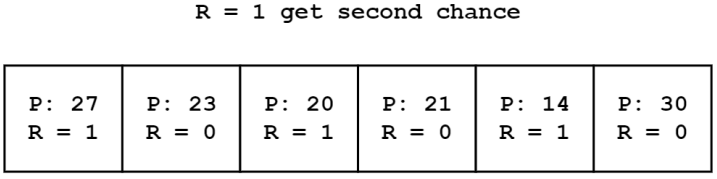
O efektivní výběr stránky k nahrazení v případě page fault se právě snaží strategie zvaná replacement policy. Replacement policy vyhledává optimálně takové stránky, které není nutno zapisovat na disk a zároveň se nepředpokládá brzký přístup k právě odstraňované stránce.

2.4.1 FIFO a Second-Chance

Replacement policy FIFO funguje na běžném principu fronty, do které se stránky řadí při zavedení do paměti. Při page fault se odebere stránka z první pozice a nová stránka je přidána na konec fronty. Tento algoritmus není vhodný, jelikož odebírá stránky bez ohledu na to, zda jsou používány. [7]

Second-Chance je modifikace strategie FIFO, která omezuje problém odstraňování používaných stránek pomocí příznaku používání (bit R – referenced). Je-li příznak R nulový, znamená to, že příslušná stránka nebyla delší dobu použita, a může být nahrazena. Pokud je přáznak R nastaven, znamená to, že stránka byla v nedávné době použita. V takovém případě dostává druhou šanci v podobě vynulování R a zařazením na konec fronty. [7]

Vizualizace obsahuje možnost zobrazit průběh algoritmu FIFO i FIFO s modifikací second chance. Pomocí checkboxu lze zvolit, zda chce uživatel zobrazit variantu s modifikací či nikoliv.



Obrázek 18: Vizualizace FIFO second chance

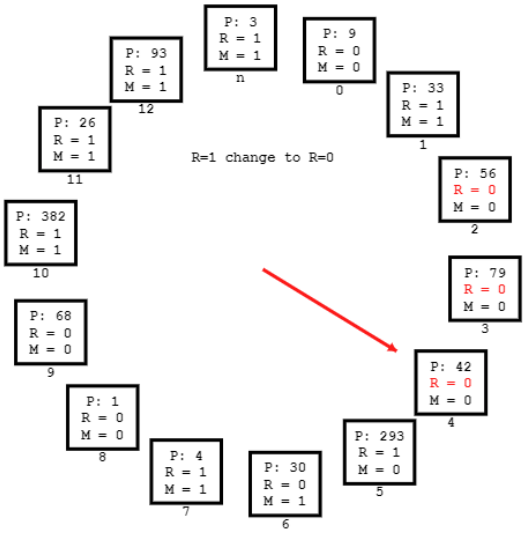
Na počátku je zobrazena fronta obsahující stránky a jejich číslo. Při vizualizaci modifikace je ke každé stránce přidána hodnota příznaku R. Po spuštění animace dochází k upozornění, že začíná nahrazení stránky. Při variantě bez modifikace je odebrána stránka ze začátku fronty a následně přidána nová stránka na konec. Opět je zobrazeno upozornění o ukončení nahrazení stránky. Tento postup je následně opakován, aby bylo zřejmé, jak daný algoritmus funguje.

V případě zapnuté modifikace second chance je postup obdobný s tím rozdílem, že při druhém nahrazování dostává stránka druhou šanci, když je příznak R nastaven (je roven jedné).

2.4.2 Clock Policy a modifikace Working Set

Metoda FIFO s modifikací second chance je sice přínosná, ale po každém resetování příznaku R dochází k přesunutí stránky na konec fronty. Implementace Clock Policy proto uchovává stránky v kruhovém seznamu, který připomíná hodiny. Ručička ukazuje na příslušnou pozici v seznamu. V případě, že nastane page fault, na pozici ručičky se začne provádět inspekce příznaku R. Je-li hodnota R jedna, dojde k resetování R bitu na hodnotu nula a posunutí na následující pozici. Tento úkon se opakuje, dokud se nenalezne příznak R s hodnotou nula. Pak dojde k nahrazení stránky na dané pozici a hned vzápětí se ručička posune na následující pozici a vyčkává na další page fault. [7]

Working Set Clock Policy je modifikace, kdy se využívá kromě příznaku R nově příznak M (modified). Tento příznak je nastaven, pokud byla stránka od posledního zápisu na disk změněná, a je tedy nutné ji před jejím nahrazením uložit na disk. Algoritmus je totožný s Clock Policy s tím rozdílem, že se porovnává kromě příznaku R zároveň příznak M. V případě nalezení nepoužívané stránky (R = 0) s nastaveným příznakem M dojde k záznamu o tom, že se daná stránka má uložit na disk, a pokračuje se dál do doby, než je nalezena stránka s oběma příznaky R a M nulovými. [7]



Obrázek 19: Vizualizace working set clock

Vizualizace Clock Policy zobrazuje kruhový seznam stránek s příslušnými příznaky a posunující se ručičku při hledání stránky k výměně. Průběh animace je z výukových důvodů vždy

stejný. Uživatel si může přepínat mezi variantami Clock Policy a Working Set pomocí checkboxu. Po spuštění animace se v informačním panelu vypisuje průběh algoritmu. Při nulovaní příznaku R se pro lepší názornost mění také jeho barva. Po nalezení příslušné pozice pro náhradu je stránka nahrazena, je změněna barva jejího pozadí a animace končí.

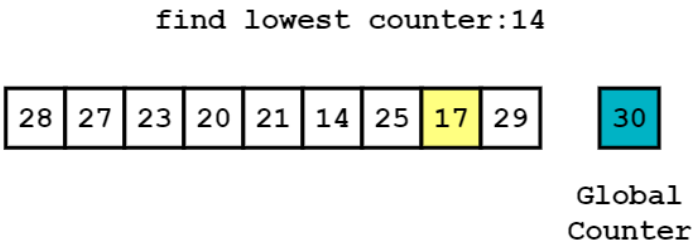
2.4.3 LRU

Least Recently Used je algoritmus, který vybírá nejstarší stránku, neboli stránku, která nebyla použita nejdelší dobu. V případě, že stránka nebyla delší dobu používána, předpokládá se, že nebude používána ani v budoucnu. Informaci o stáří stránky lze reprezentovat číselnou hodnotou pomocí globálního čítače, jenž se inkrementuje po každém přístupu ke stránce a následně je jeho hodnota přidělena k příslušné stránce. Nejnižší číselná hodnota značí nejstarší stránku. [7]

Implementace LRU může být provedena na uspořádaném seznamu, kdy po každém přístupu ke stránce je globální čítač inkrementován a jeho hodnota je následně přidělena k právě přistupované stránce. Následně dochází k zařazení stránky v seznamu na patřičnou pozici. Jelikož je seznam uspořádaný, dojde k nahrazení vždy první nebo poslední stránky podle zvoleného řazení. Tento způsob je však nevýhodný, jelikož po každém přístupu ke stránce je zapotřebí tento seznam seřadit nebo znova vybudovat dle číselných hodnot čítače stránky. [7]

Dalším způsobem implementace může být neuspořádaný seznam. Opět se stránkám přiděluje hodnota čítače při přístupu ke stránce. Ale při vyhledávání stránky k nahrazení musí dojít k prohledání celého seznamu za účelem nalézt stránku s nejnižší číselnou hodnotou (čítače). Jedná se sice o značné vylepšení, ale v případě častého výskytu page fault je zapotřebí pokaždé projít celý seznam, což opět není optimální.

Výhodnou implementací může být uchování informace o stáří pomocí matice. Pro n stránek s počtem n je vytvořena matice o velikosti $n \times n$. V případě přístupu ke stránce číslo k jsou všechny hodnoty řádku s indexem k nastavené na hodnotu jedna a následně všechny hodnoty sloupce s indexem k nastavené na hodnotu nula. Čísla v řádku matice se interpretují jako binární hodnota, která vyjadřuje stáří příslušné stránky (dle řádku). [7]



Obrázek 20: Vizualizace LRU při vyhledávání nejmenšího čítače

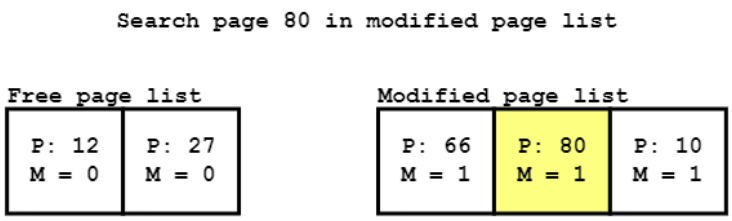
Algoritmus LRU je vizualizován pomocí metody neutříděného seznamu. Na scéně je zobrazen neuspořádaný seznam s číselnými hodnotami čítače stránek a globální čítač. Po spuštění animace dojde k vyhledání nejnižší hodnoty v seznamu. Animace provádí procházení seznamu zleva doprava a znázorňuje změnou barvy pozadí právě porovnávanou pozici. V titulku animace se zobrazuje stav o porovnání hodnot, případně o nalezení příslušné stránky k nahrazení. Po nalezení stránky k nahrazení dojde k odložení stránky na disk (swap-out), dále k inkrementaci globálního čítače a k načtení nové stránky do paměti na právě uvolněnou pozici. Následně je simulován náhodný přístup ke stránkám, kdy dochází k inkrementaci globálního čítače a přiřazení jeho hodnoty k právě přistupované stránce. V posledním kroku je opět zobrazena animace k vyhledání nejnižší hodnoty v seznamu.

2.4.4 Page buffering

V případě, že algoritmus provádějící nahrazování vybere stránku k nahrazení, zařadí tuto stránku na konec do jednoho ze dvou seznamů a označí ji jako odstraněnou z RAM. První seznam obsahuje volné stránky (free page list), které nebyly modifikovány, takže není zapotřebí je před jejich přepsáním v RAM ukládat na disk. Druhý seznam obsahuje stránky, které byly modifikované (modified page list), neboli před přepsáním stránky v paměti je zapotřebí je uložit na disk. [13]

Nastane-li page fault, dojde k prohledání obou seznamů. Pokud je stránka nalezena, nastaví se zpět bit přítomnosti v paměti a stránka je z příslušného seznamu odstraněna. V případě, že stránka není nalezena ani v jednom seznamu a seznam volných stránek není prázdný, dojde k načtení stránky do rámce ze začátku seznamu volných stránek a následně je stránka z tohoto seznamu odstraněna. [13]

V případě, že se vyprázdní seznam volných stránek, dojde k uložení modifikovaných stránek na disk a ze seznamu modifikovaných stránek se stane seznam volných stránek. [13]



Obrázek 21: Vizualizace page buffering

Vizualizace page buffering je reprezentována dvěma seznamy: free page list a modified page list. Vizualizace bere v úvahu všechny výše popsané situace při prohledávání obou seznamů.

Po spuštění animace dochází k vyhledání stránky, která je obsažena v seznamu free page list a dochází tak k jejímu odstranění ze seznamu. Druhá vyhledávaná stránka je nalezena v seznamu modified page list a je ze seznamu odebrána. Třetí vyhledávaná stránka není nalezena ani v jednom seznamu a seznam free page list není prázdný. Může tedy dojít k nahrazení stránky z tohoto seznamu.

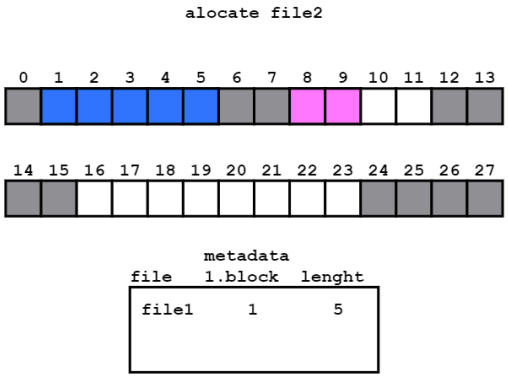
Další kroky vedou k postupnému vyprázdnění seznamu free page list, které vedou k situaci, kdy vyhledávaná stránka není ani v jednom seznamu a musí tedy dojít k uložení seznamu modfied page list na disk. Poté jsou stránkám změněny hodnoty příznaku M na nulu a jsou přesunuty do seznamu free page list. Dochází tak k postupnému vyprazdňování seznamu free page list.

2.5 Alokace prostoru na médiu

„Souborové systémy alokují souborům prostor po alokačních blocích (používá se též pojem cluster), které mají velikost několik (obvykle mocnina dvou, tj. 1, 2, 4, 8, 16, …) sektorů, přičemž sektor je nejmenší alokovatelná jednotka na médiu (typicky 512, 2 048, případně 4 096 bajtů). Díky alokaci prostoru po blocích zůstává poslední blok často nevyužit zcela – vzniká vnitřní fragmentace.“ [14]

Souvislá alokace

„Souboru jsou vždy přiděleny po sobě jdoucí bloky. V metadatech souboru se eviduje adresa prvního (počátečního) bloku a velikost souboru (v alokačních blocích). Při alokaci prostoru na médiu dochází k vnější fragmentaci – vznikají díry, které je obtížné využít. Soubor tak nemůže (bez přemístění) ani růst nad limit volného prostoru za posledním blokem.“ [14]

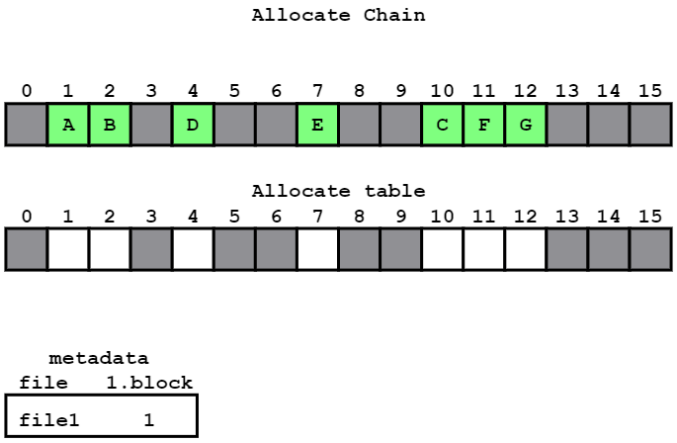


Obrázek 22: Vizualizace souvislé alokace souboru v průběhu alokace file2

Vizualizace souvislé alokace znázorňuje průběh alokace tří souborů. V počátečním stavu je zobrazena paměť s indexy nad každým blokem. Prázdné bloky jsou znázorněny bílým obdélníkem a plné šedým. Po spuštění animace dojde k alokaci prvního souboru (file1). Volné bloky se začnou postupně zaplňovat. Po zápisu celého souboru dojde k aktualizaci metadat. Následně začne animace pro soubor druhý (file2) a třetí (file3). Soubory jsou pro přehlednost znázorněny různými barvami.

Řetězená alokace

„Pro soubory se alokují jednotlivé bloky. Alokační tabulku tvoří zřetězené seznamy bloků. Každý blok obsahuje odkaz na následující blok nebo informaci o konci řetězu. Tato metoda alokace odstraňuje problém vnější fragmentace, neboť souboru lze přidělit libovolný volný blok. Logicky sousedící bloky tak mohou být na různých místech na médiu, čímž vzniká datová fragmentace. Jelikož se datová fragmentace typicky zvyšuje s tím, jak se souborový systém používá (vznikají a zanikají soubory, mění se jejich velikosti), používá se také termín stárnutí – file system aging.“ [14]

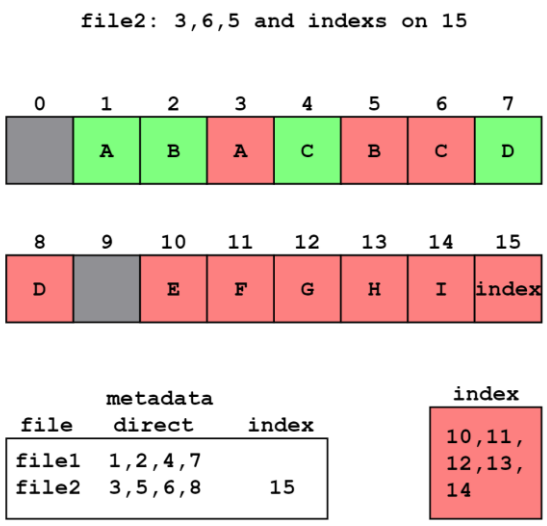


Obrázek 23: Vizualizace řetězové alokace

Pro vizualizaci řetězové alokace byla zvolena trochu jiná forma pomocí lehkého kvízu. Po spuštění animace je zobrazen soubor s alokovanými bloky a je položena otázka, jak bude vypadat alokační tabulka. V tuto chvíli může uživatel či vyučující animaci pozastavit a počkat na odpověď studentů. Pro ověření lze animaci opět spustit, čímž dojde k vyplnění alokační tabulky blok po bloku.

Indexová alokace

„Metadata souboru obsahují index, který je tvořen seznamem alokovaných bloků souboru. Ani tento způsob alokace netrpí problémem vnější fragmentace. Index může být také nepřímý (víceúrovňový), tj. může obsahovat i odkaz na blok, který je opět indexem. Velikost bloku může být různá (cílem je redukovat vnitřní fragmentaci).“ [14]



Obrázek 24: Vizualizace indexová alokace po ukončení animace

Pro indexovou alokaci je vytvořena vizualizace, kdy je na počátku zobrazena plná paměť, ale s prázdnými metadaty. Po spuštění animace dojde právě k naplnění metadat. Soubor file1 je tvořen pouze z indexů na přímé bloky, file2 je tvořen jak z odkazů na přímé bloky, tak na nepřímé, a proto je při vyplnění metadat u file2 zobrazen detail bloku index v pravém dolním rohu.

2.6 Konzistence souborového systému

Zápis dat do souboru znamená provedení několika operací na různých místech na médiu.

a) Metadata souborového systému,
b) datová část souborového systému – zápis nových dat souboru,
c) metadata souboru – aktualizace přidělených bloků, velikosti, času změny apod.

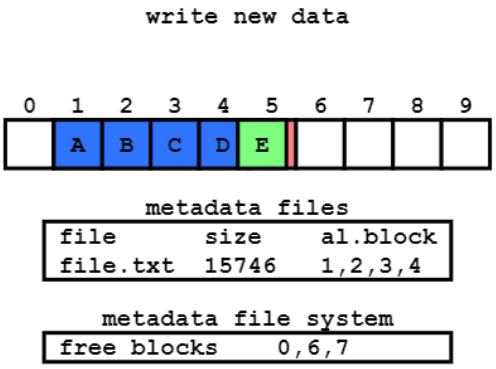
Tyto změny probíhají postupně, čímž nutně vznikají stavy nekonzistence. Po pádu systému je třeba provést kontrolu stavu souborových systémů a případné nekonzistence odstranit. [14]

Žurnálování

„Pro zachování konzistence dat a metadat souborového systému lze zavést transakční log nazývaný žurnál. Jedná se v podstatě o kruhový buffer, do kterého se zapisují prováděné změny metadat (případně i dat).

Teprve po potvrzení zápisu do žurnálu se provede patřičná změna souborového systému. Při startu systému po pádu tak není třeba kontrolovat konzistenci celého souborového systému, ale

pouze těch míst, na kterých docházelo ke změnám bezprostředně před pádem. Tato místa se zjistí právě ze žurnálu a podle něj se případně opraví struktury souborového systému, takže ten je opět konzistentní. Výsledkem je mnohonásobné zrychlení kontroly konzistence souborového systému.“ [14]

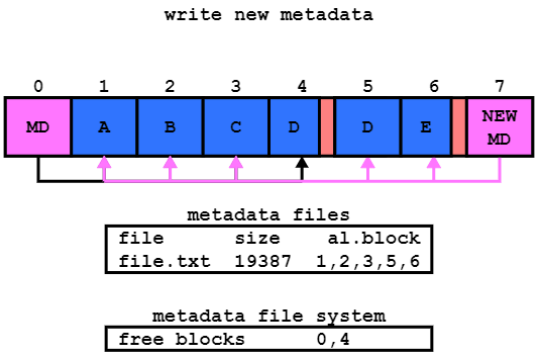


Obrázek 25: Vizualizace žurnálu

Celý zmíněný postup je prováděn při vizualizaci. Na scéně je zobrazena paměť, metadata souboru a metadata souborového systému. Průběh vizualizace probíhá v následujících krocích: Alokace nového bloku, aktualizace metadat souborového systému, zapsání nových dat do paměti, aktualizace alokovaných bloků v tabulce metadat souboru a následně změna velikosti souboru v metadatech. Tímto končí celá transakce a vizualizace je u konce.

Copy-on-write

„Při nutnosti přepsat některý blok novými informacemi se modifikuje (v paměti) kopie původního bloku a tato kopie se zapíše do nově alokovaného prostoru. Stejným způsobem se aktualizují také metadata. Po dokončení zápisu dat a metadat (do nových bloků) se atomicky provede zneplatnění původních dat a metadat a potvrdí se platnost nových. Souborový systém je tedy v každém okamžiku konzistentní. Navíc není třeba dvojího zápisu jako u metody žurnálování.“ [14]



Obrázek 26: Vizualizace copy-on-write při vytvoření nových metadat

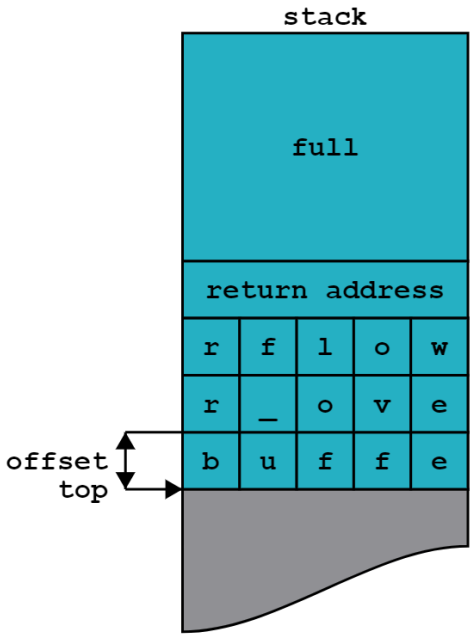
Copy-on-write při vizualizaci probíhá následovně. Scéna obsahuje paměť s daty a s jedním blokem metadat. Při prvním kroku dochází k zobrazení pomocí černých šipek, jaké bloky patří k metadatům. Následuje zápis nových dat a k zápisu kopie bloku D. Nová data jsou odlišená zelenou barvou. Na řadu přichází vytvoření nových metadat a pomocí fialových šipek dochází k zobrazení bloků, které k nim patří, a na závěr dochází k přepnutí ze starých metadat na nová.

2.7 Přetečení zásobníku

Jedná se o problém bezpečnosti paměti, kdy například programovací jazyk C nehlídá délku pole a umožňuje zapisovat i mimo jeho rozsah. Pokud programátor tuto skutečnost neřeší, může dojít k situaci, kdy je očekáván řetězec od uživatele o omezené délce, např. pěti znaků, ale uživatel zadá řetězec o délce deseti znaků. Program bez kontroly zapíše do pole zadaný řetězec od uživatele a tím dochází k přetečení paměti vyhrazené pro řetězec (pole) a přepsání sousedního úseku paměti, který nebyl určen pro pole.

Důsledkem je chybné nebo nepředvídatelné chování programu v závislosti na přepsaných datech. Neošetření přetečení lze využít k napadení programu, jeho ovlivnění, nebo dokonce k umožnění spuštění útočníkova kódu.

Pro buffer overflow byla vypracována vizualizace, která umožňuje spustit předem připravenou animaci (scénář) nebo lze zaplnit pamět pomocí zadaného řetězce uživatelem. Pokud uživatel zadá řetězec o větší délce než pět znaků, dochází k přepsání bloku s parametry, s návratovou adresou, popřípadě další paměti.



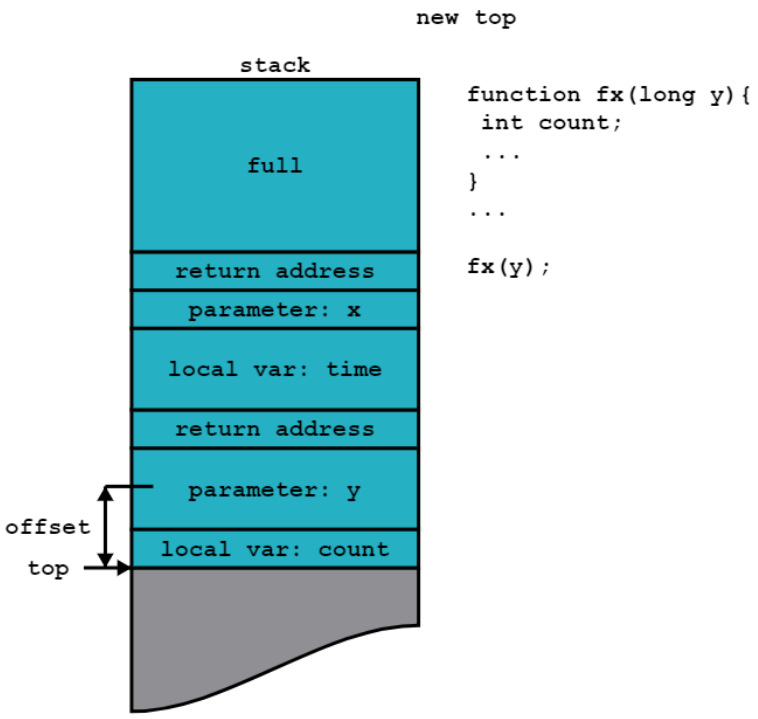
Obrázek 27: Vizualizace buffer overflow

Jelikož řetězec pro vizualizaci může zadat uživatel, je tento řetězec z bezpečnostních a vizualizačních důvodů filtrován od všech speciálních znaků včetně znaků s diakritikou.

2.8 Zásobník pro vlákno

Tato vizualizace demonstruje, proč musí mít každé vlákno vlastní zásobník (stack). Pro tuto vizualizaci byly vytvořeny dva scénáře.

První scénář popisuje, jak bude probíhat uložení hodnot do zásobníku při volání funkce. Počáteční stav zobrazuje pouze zásobník. Po spuštění animace dochází k zobrazení zdrojového kódu funkce *f*. Po zavolání této funkce dochází k rozšíření zásobníku o hodnoty návratové adresy, parametrů funkce a lokální proměnné funkce. Animace pokračuje zobrazením vrcholu (top) zásobníku a offsetu. Offset vyjadřuje relativní adresaci proměnných vzhledem k vrcholu zásobníku.



Obrázek 28: Vizualizace zásobníku pro vlákno konečný stav

Druhý scénář pokračuje tam, kde první končí. Animace probíhá téměř identicky, ale místo volání funkce *f* se volá funkce *fx* z jiného vlákna, která používá lokální proměnou a parametry lišící se především v datovém typu. Jakmile animace dojde k místu, kde se animuje nová pozice vrcholu zásobníku, dojde ke znázornění, že offset používaný pro adresaci parametru ve funkci *f* v prvním vlákně nyní ukazuje špatně do středu bloku s parametry funkce *fx*.

3 IMPLEMENTACE VIZUALIZACE

V následujících kapitolách dojde k seznámení s postupem vytváření vizualizace. Jelikož popis každé vizualizace by byl velice zdlouhavý, nepřehledný a ve velké míře by docházelo k opakování postupu, bude zde popsána šablona, která byla vytvořena postupně při tvorbě jednotlivých vizualizací a po každé vizualizaci byla patřičně upravena a zdokonalena.

3.1 Základy animace

Pro správnou vizualizaci je zapotřebí vědět něco o animaci a počítačové grafice. Počítačová animace se nejčastěji řadí do dvou kategorií, 2D a 3D animace. Pro tvorbu animace lze využít speciální programy. Například pro tvorbu 2D animace lze použít Adobe Animate (Flash studio), Adobe After Effect, pro 3D animace Blender, Autodesk Maya a mnoho dalších.

Při tvorbě animace je zapotřebí se rozhodnout, zda se bude jednat o interaktivní animaci či nikoliv. Při vytváření animace bez interaktivních prvků lze animaci exportovat jako video či animovaný obrázek (GIF).

Při tvorbě interaktivní animace je již zapotřebí využít program nebo skript, který umožňuje interakci s uživatelem nebo prostředím a spouštět předem vytvořené animace nebo je sám vytvářet na základě klíčových snímků či vstupu uživatele.

3.1.1 Klíčové snímky

Klíčové snímky reprezentují stavy na časové ose pro určenou animaci. Mohou reprezentovat konečnou podobou scény, změnu objektu či začátek a konec provádění effektu nebo pohybu objektu. Jednotlivé snímky mezi klíčovými snímky jsou dopočítávány. Tuto metodu využívá mnoho programů pro tvorbu videa či animací.

Ve vytvořených vizualizacích není využita metoda klíčových snímků, ale na základě této metody byl inspirován postup pro tvorbu scénářů. Scénář reprezentuje posloupnost akcí ve scéně, která se má provést. Některé akce můžou „vyskočit“ z právě prováděného scénáře a provedou vlastní animaci na základě stavu animace. Pak se vrátí k provádění scénáře.

3.1.2 Animační cyklus

S pojmem animační cyklus se setkáváme především při vytváření (programování) videoher. Jedná se o cyklus, který provádí vykreslování scény, aktualizování hodnot objektů a odposlech vstupu od uživatele. Ve většině programovacích jazyků se animační cyklus implementuje na nekonečném cyklu while, který je většinou ukončen zároveň s programem.

Prvním krokem většinou bývá nastavit interval vykreslování neboli FPS (frames per second). FPS vyjadřuje, kolikrát došlo k vykreslení scény za jednu sekundu. Pro animaci je hraniční hodnota 24 FPS, filmy většinou používají 24 FPS, ale každý frame bývá zobrazen dvakrát nebo i třikrát. Takže finální FPS je 48 nebo 72. Současné hry podporují vykreslování 60 FPS, ale pro úsporu výkonu většina konzolí a her umožňuje režim ve 30 FPS. [15]

```
while(running){
    input();     // vstup uživatele
    update();    // aktualizování hodnot
    render();    // vytvoření grafických objektů
    draw();      // vykreslit
    timeDiff = Time.NOW - beforeTime;
    sleepTime = period - timeDiff;  // period odpovídá FPS
    beforeTime = Time.NOW
    Thread.sleep(sleepTime);
}
```

Pseudokód výše popisuje, jak by mohl vypadat animační cyklus. Nejdůležitější je poslední řádek, kdy dochází k uspání vlákna. Tím je zaručeno, že dojde k vykreslování v požadované frakvenci FPS a ostatní vlákna dostanou možnost se projevit. V případě snížení FPS z důvodu hardwarové zátěže může dojít k velkému zpomalení animace či trhání.

Jelikož o vykreslování vizualizací této diplomové práce se stará knihovna Paper.js, nemusíme většinu problémů spojených s animacemi řešit. Pro animaci má Paper.js implementovanou vlastní metodu obstarávající animační cyklus s názvem onFrame. V žádné vizualizaci není využit vstup (klávesnice) od uživatele, takže funkcí input se není nutné zabývat. Funkce update je implementována trochu odlišně a bude popsána v následující podkapitole. Update však není součástí metody onFrame.

Render obstarává opět knihovna Paper.js a funkce draw není zapotřebí, jelikož Paper.js v základním nastavení automaticky po vytvoření každého grafického objektu tento objekt neustále vykresluje. Je však zapotřebí aktualizovat pozici a po případě grafickou reprezentaci objektů ve scéně. Z toho důvodu má každá vizualizace metodu draw, která je volána uvnitř metody onFrame.

3.2 Postup při tvorbě vizualizace

Zde bude popsán obecný postup při tvorbě vizualizace. Dojde k popisu vytvoření základního HTML dokumentu a popis šablony pro tvorbu vizualizace. Je zcela zásadní si před vytvořením jakékoliv vizualizace (animace) uspořádat myšlenky a nápady, aby nedocházelo k častému překreslování scény a tím pádem ke komplikacím při její realizaci

Každá vizualizace, která byla vytvořena v rámci diplomové práce, byla konzultována a navržena tak, aby vyhovovala pro výukové účely vedoucího práce.

3.2.1 Vytvoření grafické šablony pro vizualizace

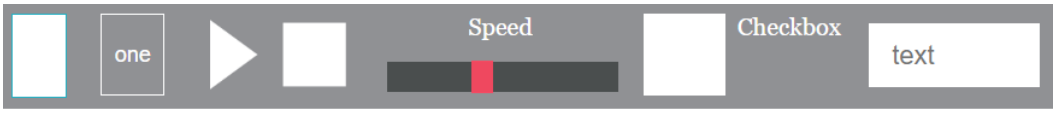
Prvním krokem je navržení scény. Při návrhu scény byl kladen důraz na to, aby všechny vizualizace vypadaly podobně. Scéna vizualizace se skládá ze tří základních prvků. Nadpisem scény, který slouží zároveň jako informační panel, prostorem pro grafické objekty a ohraničení obsahující celou grafickou scénu. Jelikož většina zařízení má širokoúhlý display (monitor), jsou vizualizace navrženy na šířku.

```
<canvas class="example-canvas" id="example-canvas" resize>
    Browser not supported HTML5 Canvas
</canvas>
```

Scéna je vykreslována pomocí canvas v HTML5. Canvas je roztažen přes celé okno webového prohlížeče. Canvas mění svou velikost závisle na změně velikosti okna prohlížeče. Aby došlo ke korektní změně velikosti scény, dochází k úmyslnému porušení validace W3C, jelikož Paper.js vyžaduje u tagu canvas atribut resize, který není součástí HTML5. Pokud tento atribut chybí, dochází k deformaci vykreslených objektů.

Obrázek 29: Rozložení scény vizualizace

Pro ovládání animace byl vytvořen panel nástrojů. Tento panel může obsahovat kterákoliv vizualizace. Ovládací panel obsahuje nástroje odpovídající dané vizualizaci podle její potřeby. Předefinovány jsou následující nástroje.



Obrázek 30: Přehled panelu nástrojů color, select, button, range, checkbox

Barevná paleta (color) pro výběr barvy. Nástroj pro výběr (select) scénáře, pokud vizualizace má více průběhů. Tlačítka stop a play. Play může sloužit i jako pauza za použití patřičné funkce. Nástroj range pro zvolení intenzity (velikosti) od minimální do maximální hodnoty. Vypínač (checkbox), který mění barvu podle svého stavu. Bílý checkbox značí, že je daná funkce vypnutá, modrá indikuje pravý opak. Input pro zadání textu uživatelem.

Kompletní přehled, jak vypadá základní HTML pro vizualizaci, lze nalézt v příloze A.

3.2.2 Vytvoření vizualizace

Při vytváření vizualizace (animace) je zapotřebí dopředu vědět průběh (scénář) dané vizualizace. Každý objekt musí mít určenou pozici ve scéně nebo svojí funkci a grafickou reprezentaci. Rozdělit objekty ve scéně lze na statické a dynamické. Statické objekty jsou vytvořeny a umístěny ve scéně a nadále se s nimi nějak nepracuje. Dynamické objekty v průběhu animace provádějí nějaké úkony jako na příklad: pohyb ve scéně, změna barvy, rotace, transformace objektu a podobně.

Dalším rozhodnutím, které je nutno při vytváření vizualizace udělat je, zda vizualizace bude mít pevně daný průběh (scénář) nebo se animace bude sama vytvářet, neboli simulovat, nebo

jestli bude interaktivní a vizualizace bude reagovat na podměty od uživatele. Všechny metody vizualizace lze kombinovat, ale dochází tak prudkému navýšení obtížnosti při implementaci.

Vytvořená šablona pro vytváření vizualizací v základu řeší většinu stejných charakteristik, jako je změna velikosti, spuštění, pozastavení či resetování animace včetně zápis scénářů, jelikož jedna vizualizace může mít více průběhů.

Začneme tedy s popisem šablony, která je obsažena v příloze B. Vysvětlením všech metod a vlastností a jak je lze popřípadě modifikovat pro výslednou vizualizaci.

```
var AlgorithmOS = AlgorithmOS || {};

AlgorithmOS.TEMPLATE = {
    paperScope: null,
    sceneGroup: null,
    infoText: null,
    screenplays: [],
    screenplay: 'screenplayName',

    // _texts:  AlgorithmOS.Texts.eng.PlanningAlgorithms.TEMPLATE,
    objects:[],
```

JavaScript umožňuje vytvářet tak zvaný jmenný prostor v podobě tečkové notace, na příklad: MujProjekt.MujObjekt.DalsiObjekt. Proto v prvním řádku zdrojového kódu uvedeného výše, je vyžadován rodičovský objekt znázorňující název projektu, aby se k němu mohl přidat nově vytvářený objekt reprezentující naší vizualizaci.

Pokud však objekt AlogrithmOS není nalezen, je vrácen zcela prázdný objekt. Vytvoření objektu dojde pomocí literátu {}, který je doporučován pro vytváření objektu, jelikož literát nelze přepsat. Pokud by se vytvářel objekt pomocí new Objekt, mohlo by dojít k situaci, kdy jiná aplikace nebo modul v JavaScriptu vytvoří vlastní objekt s názvem Objekt a tím modifikuje originální a může dojít k narušení aplikace nebo dokonce k jejímu napadení.

Následně můžeme přidat objekt do rodičovského objektu pomocí tečkové notace. AlgorithmOS.TEMPLATE, název TEMPLATE nahraďte názvem reprezentující danou vizualizaci. Při tom by mělo být dodržováno pravidlo, že názvy objektů začínají velkým písmenem. Jelikož je popisován postup při tvorbě vizualizace na základě šablony, bude ponechán název TEMPLATE pro přehlednost.

Následně jsou uvedeny klíčové vlastnosti objektu TEMPLATE. Vlastnost paperScope uchovává referenci na objekt PaperScope, kterému je mu přiřazena vlastnost až při provázání skriptu s HTML stránkou. Tato reference je zde z důvodu, kdyby uživatel chtěl více vizualizací na

jedné stránce. Při každém vytvoření nové instance Paper.js na canvas, je přepsána globální proměnná paper na právě vytvořenou instanci. To může vyústit v to, že dojde k vytvoření více instancí na různé canvas, ale vše se bude promítat a animovat v posledním canvas. Pokud však objekt vizualizace uchovává referenci na svůj canvas v podobě paperScope, stačí před každým přidáním grafického objektu či jeho odebráním nebo modifikaci aktivovat právě paperScope. Tím dojde k přepnutí globální proměnné paper na příslušný canvas.

Vlastnost sceneGroup bude po zavolání metody build reprezentovat objekt Group, který obsahuje všechny grafické objekty a skupiny vytvořené pro reprezentaci scény. Stejně tak vlastnost infoText bude po zavolání metody obsahovat objekt PointText, reprezentující titulek a zároveň informační panel vizualizace.

Vlastnosti screenplay a screenplays reprezentují pole vytvořených scénářů a názvy scénářů, které se mají přehrát. Pro přepínání scénářů je nutno implementovat patřičnou metodu, kterou má každá vizualizace vlastní podle toho, zda obsahuje více scénářů.

Vlastnost texts je zde z důvodu možného rozšíření, jež je popsáno v další kapitole. Poslední klíčovou vlastnost reprezentuje vlastnost s názvem objects, která obsahuje všechny grafické objekty určené k animaci či jiné grafické transformaci.

3.2.3 Akce animace

Dále objekt pro vizualizaci obsahuje objekt AnimateAction reprezentující akce v animaci. V jiných programovacích jazycích by se dal tento objekt reprezentovat jako enum. Každá vizualizace obsahuje vlastní akce. V případě, že je animace řízená scénářem, musí scénář končit akcí END. Akce NEXT provede posun na další krok ve scénáři. Pro změnu informačního panelu (titulku) slouží akce SET_TEXT. Pro pozastavení animace je zde akce SET_WAIT.

```
AnimateAction: {
    NEXT: 'NEXT',
    WAIT: 'WAIT',
    SET_WAIT: 'SET_WAIT',
    SET_TEXT: 'SET_TEXT',
    SHOW: 'SHOW',
    END: 'END'
},
```

3.2.4 Objekt animace

Každá vizualizace obsahuje privátní objekt _animate, který uchovává stav animace. Vlastnost state reprezentuje stav, kdy je animace spuštěna, pozastavena nebo zastavena. V případě, že je animace pozastavena scénářem, je zde udána hodnota o tom, kolik zbývá k ukončení pozastavení prostřednictvím vlastnosti waitTime.

```
_animate: {
    state: 'STOP',
    waitTime: 120,
    speed: 2,
    next: {action: 'NEXT'},
    index: 0,
    setWait: function (waitTime) {
        this.next = {action: 'WAIT'};
        this.waitTime = waitTime;
    },
    reset: function () {
        this.state = 'STOP';
        this.index = 0;
        this.next = {action: 'NEXT'}
    },
    isRunning: function () {
        return this.state == AlgorithmOS.AnimateState.PLAY;
    }
}
```

Za nejdůležitější vlastnosti objektu _animate lze vyjmenovat objekt next a číselnou proměnou index. Objekt next uchovává objekt vytvořený scénářem v aktuálním kroku. Jelikož je vlastnost next objekt, je možné ho nahradit jakýmkoliv objektem. Tato vlastnost jazyka je využita scénářem, který je reprezentován jako pole objektů. Přičemž každý objekt obsahuje vlastnost action. Ostatní vlastnosti objektu mohou být zcela odlišné.

Funkce setWait v objektu _animate slouží k nastavení časovače při vyvolání akce SET_WAIT. V případě, že je vizualizace řízena simulací jako v případě alokačních algoritmů, je tato metoda modifikována o parametr afterWait. Po vypršení času je vyvolána akce NEXT, v případě modifikace s afterWait je vyvolána akce odpovídající parametru afterWait.

3.2.5 Vytvoření scény (build)

Přichází zde nejdůležitější metoda s názvem build. Tato metoda slouží pro vytvoření scény a scénářů. Bez zavolání této metody nelze zobrazit danou vizualizaci. Metoda build funguje korektně pouze v případě, že je objektu vizualizace přidělena reference paperScoupe. V prvním

kroku jsou vytvořeny nezbytné grafické objekty a to sceneGroup, _infoText a border. Následně jsou objekty _infoText a border přidány do hlavní skupiny scneGroup.

```
AlgorithmOS.TEMPLATE.build = function () {
    if (this.paperScope == null) {
        return;
    }

    if (this.sceneGroup == null) {
        this.paperScope.activate();
        this.sceneGroup = new paper.Group();

        var border = new paper.Path.Rectangle({
            from: [0, 0],
            to: [700, 500],
            strokeColor: "red"
        });

        this._infoText = new paper.PointText({
            content: '',
            justification: 'center',
            fillColor: AlgorithmOS.defaultStyle.mainFont.fillColor,
            fontSize: AlgorithmOS.defaultStyle.mainFont.fontSize,
            fontFamily: AlgorithmOS.defaultStyle.mainFont.fontFamily,
            fontWeight: AlgorithmOS.defaultStyle.mainFont.fontWeight
        });

        this.sceneGroup.addChildren([border, this._infoText]);
        this._infoText.position.x = this.sceneGroup.bounds.center.x;
    }

    this._reset(true);
    this.update();
};
```

Objekt border, reprezentovaný jako červený obdélník, má hlavní funkci ohraničit všechny objekty. Jeho význam přichází při změně velikosti, jelikož dochází k přepočtu a následnému změnění velikosti všech objektů obsažených ve sceneGroup. Skupiny (Group) vytvořené pomocí Paper.js vytvářejí vlastní ohraničení (bounds) objektů uvnitř skupiny, ale pouze pro zobrazené objekty. Pokud by scéna obsahovala grafické objekty, které jsou skryté, došlo by ke změně velikosti ve špatném poměru. To by mělo za následek špatné umístění při zviditelnění skrytích objektů. Tyto objekty by mohly být zobrazeny mimo scénu.

```
this.sceneGroup.children[0].bounds.topLeft;
```

Ohraničení pomocí objektu border má další výhodu. Jelikož je objekt border vždy prvním přidaným objektem do skupiny sceneGroup, můžou vytvářené objekty být umístěny k tomuto ohraničení bez potřeby výpočtu, neboť každý objekt od Paper.js obsahuje objekt popisující ohraničení (bounds).

Samozřejmě vizualizace, která obsahuje nevzhledný červený rám, není moc dobrá vizitka. Řešení je však jednoduší, než se zdá. Stačí pouze odebrat vlastnost strokeColor při vytváření objektu border. Objekt bude vykreslován s potřebnou definovanou velikostí, ale jelikož nemá definovanou barvu ohraničení ani výplně, bude se vykreslovat jako průhledný objekt.

Sama metoda build může vytvářet grafické objekty ve scéně, ale lepší je zavolat metody, které obstarají vytvoření a umístění grafických objektů ve scéně.

```
var bg = new paper.Path.Rectangle({
    height: AlgorithmOS.defaultStyle.mainFont.fontSize * 2,
    width: 200,
    fillColor: colorBg,
    strokeColor: AlgorithmOS.defaultStyle.main.strokeColor,
    strokeWidth: AlgorithmOS.defaultStyle.main.strokeWidth,
});

this.sceneGroup.addChildren([
    bg
]);
```

Většina grafických objektů je vytvářena pomocí předání objektu, viz zdrojový kód výše. Paper.js právě dovoluje vytvořit většinu vlastních objektů pomocí parametru objekt. Předávaný parametr obsahuje vlastnosti právě vytvářeného objektu. Tyto vlastnosti je samozřejmě možné kdykoliv změnit, buď jednu vlastnost po druhé, nebo zase pomocí objektu při využití metody set.

Každý vytvořený objekt musí být přidán do hlavní skupiny sceneGroup, jinak by došlo ke špatnému umístění ve scéně, jelikož objekty ve skupině jsou umisťovány relativně vůči vytvořené skupině. V případě, že se jedná o objekt, který bude využíván animací, je vhodné si na tento objekt pamatovat referenci, aby se nemusela procházet celá scéna, kterou uchovává Paper.js ve vlastnosti activeLayer.

Proto se tyto objety přidávají buď do obyčejného, nebo asociativního pole (object). Kdykoliv v průběhu vizualizace či při vytváření objektů umožňuje JavaScript vytvořit vlastnost objektu, která bude dostupná všem metodám. Tato vlastnost je využita při implementaci vizualizace alokace souborů, kdy při vytváření bloků souborů dochází k vytvoření seznamu pro každý soubor zvlášť a tím je usnadněn přístup k požadovanému objektu při animaci.

Každá vizualizace obsahuje metodu _reset, která vrátí scénu do původního stavu. Tuto metodu využívá metoda stop, která je provázána s tlačítkem stop v HTML dokumentu.

3.2.6 Změna velikosti scény (update)

Metoda update je volána metodou Paper.js onResize, která je definována v koncovém dokumentu HTML. Metoda update mění velikost všech objektů ve scéně v závislosti na velikosti okna. Právě z toho důvodu jsou všechny objekty obsaženy ve skupině sceneGroup. Díky tomu není zapotřebí procházet všechny objekty ve scéně a měnit jim velikost, ale stačí pouze použít transformační funkci scale na sceneGroup. Tím dojde ke změně velikosti všech objektů. Při změně velikosti je kolem sceneGroup vytvořené odsazení (ofset), aby grafické objekty nekončili na hraně okna.

```
AlgorithmOS.Barrier.update = function () {
    if (this.sceneGroup == null && this.paperScope != null) {
        return;
    }
    this.paperScope.activate();

    var offsetX = paper.view.size.width * 0.065;
    var offsetY = paper.view.size.height * 0.025;

    var height = paper.view.size.height - 2 * offsetY;
    var width = paper.view.size.width - 2 * offsetX;
    var scale;

    if (paper.view.size.width < paper.view.size.height) {
        // scale by width
        scale = width / this.sceneGroup.bounds.width;
    } else {
        // scale by height
        scale = height / this.sceneGroup.bounds.height;
    }
    this.sceneGroup.scale(scale);

    if (this.sceneGroup.bounds.width > width) {
        offsetX = paper.view.size.width * 0.065;
        width = paper.view.size.width - 2 * offsetX;
        scale = width / this.sceneGroup.bounds.width;
        this.sceneGroup.scale(scale);
    }

    this._infoText.position.x = this.sceneGroup.bounds.center.x;

    this.sceneGroup.position = paper.view.bounds.center;
};
```

Nevýhodou tohoto řešení spočívá v tom, že dochází ke změně velikosti všech textů ve scéně. Dochází tak k nečitelnosti textu při velkém zmenšení. Z toho důvodu byly pomocí CSS definovány požadavky na minimální velikost canvas. Tím není problém vyřešen, ale je alespoň částečně omezen.

Druhou možností updatu je nevyužívat metodu, kdy jsou všechny grafické objekty součástí jedné skupiny (scneGroup), ale jsou uchovány v datových strukturách nebo pouze pomocí struktury Paper.js ve vrstvách. Při každé změně velikosti okna musí dojít k průchodu všech objektů ve scéně, z důvodu přepočtu velikosti pro každý objekt. Díky tomu je celý proces více pod kontrolou a lze vyřešit problém s velikostí textu nebo při přidávání nového grafického objektu známe jeho velikost. Tato metoda byla využita při vizualizaci dynamické alokace paměti, zároveň je tato metoda velmi obtížná na implementaci, a proto se v dalších vizualizacích nevyskytuje i z důvodu že její výhody nejsou v ostatních vizualizacích zapotřebí.

3.2.7 Průběh animace

O průběh animace se stará funkce draw, která je volána metodou Paper.js onFrame, jež je implementována v koncovém HTML dokumentu. Funkce onFrame obstarává animační cyklus.

```
AlgorithmOS.TEMPLATE.draw = function () {
    if (this.paperScope == null || !this._animate.isRunning() ||
        this.screenplays[this.screenplay] == null) {
        return;
    }

    switch (this._animate.next.action) {
        case this.AnimateAction.NEXT:
            this._animate.next = this.screenplays[this.scree-
nplay][this._animate.index];
            this._animate.index++;
            console.log(this._animate.next.action);
            break;
        case this.AnimateAction.WAIT:
            this._animate.waitTime -= 1;
            if (this._animate.waitTime <= 0) {
                this._goNext();
            }
            break;
        case this.AnimateAction.SET_WAIT:
            this._animate.setWait(this._animate.next.time);
            break;
        case this.AnimateAction.SET_TEXT:
            this.setText(this._animate.next.text);
            this._goNext();
            break;
        case this.AnimateAction.SHOW:
            this.objects[this._animate.next.key].visible = true;
            this._goNext();
            break;
        case this.AnimateAction.END:
            this._animate.state = AlgorithmOS.AnimateState.STOP;
            console.log("END");
            break;
    }
};
```

Pokud je animace spuštěna podle scénáře, dochází k načtení objektu ze scénáře do objektu _animate.next. Podle vlastnosti action se rozhoduje, jaká akce se má provést. Po provedení dané akce je vyvolána funkce _goNext, která vyvolá akci NEXT a díky tomu je načten další objekt ve scénáři. Tento průběh se opakuje, dokud není vyvolána akce END, která zastaví animaci.

Každá animace může obsahovat zcela jiné akce podle potřeb animace. Pokud animace není tvořena scénářem, dochází k přechodu mezi akcemi na základě algoritmu. Mezi vizualizace bez scénáře patří vizualizace bariéry nebo vizualizace dynamické alokace paměti. Tato vizualizace řídí celý svůj průběh zcela sama.

Další metoda je interaktivní, kdy na základě podnětu od uživatele dochází ke změně ve scéně. Tuto metodu využívá vizualizace obědvajících filozofů. Kdy po kliknutí na filozofa dojde ke změně textu a případně změny barvy vidličky. Zde dochází pouze k aktualizaci vykreslení.

Pro interaktivní animaci lze využít metodu onClick, kterou lze definovat pro většinu grafických objektů vytvořených pomocí Paper.js. Po kliknutí na grafický objekt dochází k předdefinované akci. Paper.js umožňuje i obecnější metodu zvanou hitTest, která provádí nad celým projektem test, zda v okolí zadaného bodu podle jeho vlastností se nachází grafický objekt.

Mezi užitečnou akci patří akce MOVE, která je využita například při vizualizaci Context Switch, kdy dochází k přesunu grafického objektu k jinému objektu. Celý kód pro přesun k bodu ve scéně nebo k objektu ve scéně lze získat na stránkách Paper.js. Jelikož se při realizaci vizualizací nevyužívá PaperScript, ale čistý JavaScript, bylo zapotřebí kód lehce modifikovat.

```
AlgorithmOS.ContextSwitch._move = function (objectKey, toKey) {
    var object = this.objects[objectKey];
    var from = object.bounds.topLeft;
    var destination = this.objects[toKey].bounds.topLeft;
    var vector = destination.subtract(from);
    var movePart = from.add(vector.divide(30));
    object.bounds.topLeft = movePart;
    if (vector.length < 5) {
        object.bounds.topLeft = destination;
        this._goNext();
    }
};
```

Jelikož rychlost animace nemusí vyhovovat všem uživatelům, lze danou animaci pozastavit a opět spustit pomocí mezerníku.

3.2.8 Vytvoření scénáře

Scénář je tvořen přesně setříděnými objekty. Tyto objekty jsou uchovány v poli a pomocí indexu je tento scénář procházen. Celý scénář je umístěn do objektu (asociativního pole), který uchovává všechny scénáře podle klíče. Klíč z pravidla reprezentuje název scénáře.

```
AlgorithmOS.TEMPLATE._createScreenplay = function () {
    var screenplay = [];
    var a = this.AnimateAction;
    screenplay.push({
        action: a.SET_TEXT,
        text: "initial state"
    });
    screenplay.push({action: a.SET_WAIT, time: 120});
    screenplay.push({
        action: a.SET_TEXT,
        text: "new state"
    });
    screenplay.push({action: a.END});
    this.screenplays['name'] = screenplay;
};
```

Mezi jednotlivé akce nebo množinu akcí je vhodné vkládat akci SET_WAIT. Aby uživatel stíhal registrovat změny v animaci případně číst text v informačním panelu.

4 NÁVRHY ROZŠÍŘENÍ

Během vypracování jednotlivých vizualizací se objevovaly nové možnosti vylepšení celkové práce. Jelikož většina přínosných nápadů přicházela v průběhu realizace práce, bylo by obtížné je přidat do již vytvořené práce především z časového hlediska.

4.1 Windows UWP

Operační systém Windows 10 umožňuje v rámci svých univerzálních aplikací vytvářet aplikace založené na technologii HTML5 a JavaScript. Z toho důvodu byla celá práce vypracována v čistém JavaScriptu s grafickou knihovou Paper.js bez využití PaperScipt.

Po nasazení výsledné aplikace do prostředí UWP došlo k zjištění, že některé vizualizace se buď nezobrazují, nebo je nutné upravit styly CSS. Nemělo by se jednat o nějak těžkou implementaci, ale z časové tísně se touto implementací nezabývalo.

Uživatelé by si mohli tak všechny vizualizace stáhnout pomocí Windows Store a pouštět si je tak bez nutnosti připojení k internetu.

4.2 Barevné mutace

Vzhledem k tomu, že práce bude sloužit především pro výuku, bylo by vhodné umožnit změnu barev ve scéně, jelikož při promítání na projektoru může dojít ke špatné viditelnosti. Barevná mutace byla implementována pro představu ve vizualizaci alokace dynamické paměti. Přidání barevné mutace by bylo poměrně složité, jelikož ve většině vizualizací jsou statické objekty.

4.3 Virtuální třída

Pomocí technologie Node.js by mohla být vytvořena virtuální třída pro studenty, ve které by učitel promítal vypracované vizualizace, popřípadě by mohl do nich kreslit vysvětlivky. V původním plánu bylo toto vylepšení bráno v potaz, ale jelikož Paper.js v době výběru technologie mělo kritickou chybu při vytváření canvas v npm distribuci, bylo od tohohle rozšíření odstoupeno.

V průběhu realizace práce ale došlo k možnému řešení, kdy by komunikaci mezi uživateli a učitelem molo být prováděno pomocí balíčku socket.io. Tím pádem není zapotřebí vytvářet canvas na straně serveru, ale každý uživatel vytvoří vlastní canvas, do kterého se budou vykreslovat objekty zaslané pomocí socket.io.

ZÁVĚR

Všechny vizualizace byly konzultovány s vedoucím práce tak, aby vyhovovaly výukovým účelům. Všechny požadované vizualizace byly zhotoveny v technologii, která je nezávislá na platformě. Výsledná práce splňuje zadání a požadavkům vedoucího práce.

Během vypracování práce byly získány velké praktické dovednosti v programovacím jazyce JavaScript. I přes získané zkušenosti docházelo k situacím, kdy bylo zapotřebí téměř celou vizualizaci přepracovat vhodněji.

Při postupu tvorby byly jednotlivé vizualizace rozděleny přibližně do deseti skupin. Každá skupina obsahovala minimálně tři vizualizace. Již při výběru tématu se počítalo s časovou náročností, a proto při plánování realizace byla vyhraněna téměř měsíční rezerva, která byla celá využita. Po vytvoření vizualizací docházelo ke kontrole v podobě opakovaného spuštění.

Výsledný produkt této práce bude sloužit k výukovým účelům v předmětu operačních systémů.

POUŽITÁ LITERATURA

[1] ŽÁRA, Ondřej. *JavaScript: programátorské techniky a webové technologie*. Brno: Computer Press, 2015. ISBN 978-80-251-4573-9.

[2] JavaScript Versions. *W3Schools Online Web Tutorials* [online]. W3Schools, c1999-2018 [cit. 2018-04-29]. Dostupné z: https://www.w3schools.com/js/js_versions.asp.

[3] Working with Paper.js: *What is PaperScript?*. Paper.js [online]. [cit. 2018-04-29]. Dostupné z: http://paperjs.org/tutorials/getting-started/working-with-paper-js/.

[4] DOMES, Martin. *333 tipů a triků pro CSS*. 2., aktualiz. vyd. Brno: Computer Press, 2011. ISBN 978-80-251-3366-8.

[5] HUDEC, Tomáš. *Správa paměti, metody alokace paměti, virtualizace paměti*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-03]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=575. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[6] HUDEC, Tomáš. *Plánování procesů*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-04]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=577. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[7] ANDREW S. TANENBAUM. *Modern operating systems*. 3rd ed., Pearson new international ed. Harlow: Pearson, 2014. ISBN 978-1-29202-577-3.

[8] HUDEC, Tomáš. *Plánování procesů*. Operační systémy [online]. Pardubice [cit. 2018-05-04]. Dostupné z: http://asuei01.upceucebny.cz/usr/hudec/vyuka/os/materialy/pub2015/OS-04.pdf. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[9] HUDEC, Tomáš. *Konkurence procesů, řízení přístupu do kritické sekce pomocí SW a HW metod*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-04]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=578. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[10] HUDEC, Tomáš. *Hardware a operační systémy*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-04]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=507. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[11] HUDEC, Tomáš. *Komunikace procesů (IPC)*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-04]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=579. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[12] HUDEC, Tomáš. *Prostředky programovacích jazyků pro IPC*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-04]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=1909. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[13] HUDEC, Tomáš. *Správa paměti* [online]. Pardubice, 2015 [cit. 2018-05-05]. Dostupné z: http://asuei01.upceucebny.cz/usr/hudec/vyuka/os/materialy/pub2015/OS-07.pdf. Dostupné pouze z vnitřní sítě Univerzity Pardubice.

[14] HUDEC, Tomáš. *Soubory a souborové systémy*. E-learningový portál FEI - LEARN [online]. Pardubice, 2013 [cit. 2018-05-05]. Dostupné z: https://fei-learn.upceucebny.cz/mod/page/view.php?id=572.

[15] DAVISON, Andrew. Programování dokonalých her v Javě: [programování her a grafiky v Javě]. Brno: Computer Press, 2006. ISBN 80-7226-944-5.

PŘÍLOHY

PŘÍLOHA A – HTML ŠABLONA PRO VIZUALIZACI

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Tempagte</title>
    <link rel="stylesheet" href="../view/css/style.css">

    <script src="../view/js/paper-full.min.js"></script>
    <script src="../view/js/algorithm/AlgorithmOS.js"></script>
    <!--<script src="../view/js/algorithm/... .js"></script>-->

    <script>

        window.onload = function () {
            paper.install(window);
//            var obj = AlgorithmOS....;
//            obj.paperScope = new paper.PaperScope();
//            obj.paperScope.setup('example-canvas');
//            obj.build();
            //        memory._destroy();

            paper.view.onResize = function (event) {
//                obj.update();
            };

            paper.view.onFrame = function (event) {
//                obj.draw();
            };

            paper.view.onKeyUp = function (event) {
//                if (event.key === 'space') {
//                    obj.playOrPause();
//                }
            };

            window.addEventListener('keydown', function (e) {
                if (e.keyCode == 32 && e.target == document.body) {
                    e.preventDefault();
                }
            });
        }
    </script>
</head>
<body>
<div class="wrapper-block bg-main">
    <div class="wrapper-center algorithm-info">
        <h1>Nadpis h1</h1>
        <article>
            <h2>Nadpis h2</h2>
            <p>
                Popis tematu
            </p>
        </article>
        <article>
            <h3>Blisi specifikace h3</h3>
            <p>
                text
                <span>zdurazneni</span>
            </p>
```

```
        </article>
    </div>
</div>
<div class="wrapper-tools" id="tools">

    <div class="wrapper-center">
        <div class="tool tool-color">

            <input type="color" value="#ffffff"
                   onchange="AlgorithmOS.canvasBackground('example-canvas',
this.value)"/>
        </div>
    </div>
</div>
<div class="wrapper-block" id="exampleBlock">
    <canvas class="example-canvas" id="example-canvas" resize>
        Browser not supported HTML5 Canvas
    </canvas>
</div>
</body>
</html>
```

PŘÍLOHA B – JAVASCRIPT ŠABLONA PRO VIZUALIZACI

```
var AlgorithmOS = AlgorithmOS || {};

AlgorithmOS.TEMPLATE = {
    paperScope: null,
    sceneGroup: null,
    _infoText: null,
    screenplays: [],
    screenplay: 'screenplayName',

    // _texts:  AlgorithmOS.Texts.eng.PlanningAlgorithms.TEMPLATE,
    objects:[],

    AnimateAction: {
        NEXT: 'NEXT',
        WAIT: 'WAIT',
        SET_WAIT: 'SET_WAIT',
        SET_TEXT: 'SET_TEXT',
        SHOW: 'SHOW',
        END: 'END'
    },
    _animate: {
        state: 'STOP',
        waitTime: 120,
        speed: 2,
        next: {action: 'NEXT'},
        index: 0,

        setWait: function (waitTime) {
            this.next = {action: 'WAIT'};
            this.waitTime = waitTime;
        },
        reset: function () {
            this.state = 'STOP';
            this.index = 0;
            this.next = {action: 'NEXT'}
        },
        isRunning: function () {
            return this.state == AlgorithmOS.AnimateState.PLAY;
        }
    }

};


AlgorithmOS.TEMPLATE.build = function () {
    if (this.paperScope == null) {
        return;
    }

    if (this.sceneGroup == null) {
        this.paperScope.activate();
        this.sceneGroup = new paper.Group();

        var border = new paper.Path.Rectangle({
            from: [0, 0],
            to: [700, 500],
            strokeColor: "red"
        });
```

```
        this._infoText = new paper.PointText({
            content: '',
            justification: 'center',
            fillColor: AlgorithmOS.defaultStyle.mainFont.fillColor,
            fontSize: AlgorithmOS.defaultStyle.mainFont.fontSize,
            fontFamily: AlgorithmOS.defaultStyle.mainFont.fontFamily,
            fontWeight: AlgorithmOS.defaultStyle.mainFont.fontWeight
        });

        this.sceneGroup.addChildren([border, this._infoText]);
        this._infoText.position.x = this.sceneGroup.bounds.center.x;
        //TODO:SCREEN PLAY this._createScreenplay();
    }

    this._reset(true);
    this.update();
};

AlgorithmOS.TEMPLATE._reset = function (animation) {
    if (animation) {
        this._animate.reset();
    }
    //TODO add reseet
};

//-------------- ANIMATION TOOLS ---------------
AlgorithmOS.TEMPLATE.play = function () {
    this._animate.state = AlgorithmOS.AnimateState.PLAY;
};

AlgorithmOS.TEMPLATE.stop = function () {
    this.build();
};

AlgorithmOS.TEMPLATE.setScenario = function (screenPlayName) {
    this.screenplay = screenPlayName;
    this.build();
};

AlgorithmOS.TEMPLATE.playOrPause = function () {
    switch (this._animate.state){
        case AlgorithmOS.AnimateState.PAUSE:this.play();
            break;
        case AlgorithmOS.AnimateState.PLAY:  this._animate.state = Algo-
rithmOS.AnimateState.PAUSE;
            break;
        case AlgorithmOS.AnimateState.STOP: this.play();
            break;
    }
};


AlgorithmOS.TEMPLATE.setText = function (text) {
    if (this._infoText != null) {
        this._infoText.content = text;
        this._infoText.position.x = this.sceneGroup.bounds.center.x;
    }
};


AlgorithmOS.TEMPLATE.update = function () {
```

```
    if (this.sceneGroup == null && this.paperScope != null) {
        return;
    }
    this.paperScope.activate();

    var offsetX = paper.view.size.width * 0.065;
    var offsetY = paper.view.size.height * 0.025;

    var height = paper.view.size.height - 2 * offsetY;
    var width = paper.view.size.width - 2 * offsetX;
    var scale;

    if (paper.view.size.width < paper.view.size.height) {
        // scale by width
        scale = width / this.sceneGroup.bounds.width;
    } else {
        // scale by height
        scale = height / this.sceneGroup.bounds.height;
    }
    this.sceneGroup.scale(scale);

    if(this.sceneGroup.bounds.width > width){
        offsetX = paper.view.size.width * 0.065;
        width = paper.view.size.width - 2 * offsetX;
        scale = width / this.sceneGroup.bounds.width;
        this.sceneGroup.scale(scale);
    }

    this._infoText.position.x = this.sceneGroup.bounds.center.x;

    this.sceneGroup.position = paper.view.bounds.center;
};


AlgorithmOS.TEMPLATE._goNext = function () {
    this._animate.next = {action: this.AnimateAction.NEXT};
};

AlgorithmOS.TEMPLATE.draw = function () {
    if (this.paperScope == null || !this._animate.isRunning() ||
        this.screenplays[this.screenplay] == null) {
        return;
    }

    switch (this._animate.next.action) {
        case this.AnimateAction.NEXT:
            this._animate.next = this.screenplays[this.scree-
nplay][this._animate.index];
            this._animate.index++;
            console.log(this._animate.next.action);
            break;
        case this.AnimateAction.WAIT:
            this._animate.waitTime -= 1;
            if (this._animate.waitTime <= 0) {
                this._goNext();
            }
            break;
        case this.AnimateAction.SET_WAIT:
            this._animate.setWait(this._animate.next.time);
            break;
        case this.AnimateAction.SET_TEXT:
```

```
            this.setText(this._animate.next.text);
            this._goNext();
            break;
        case this.AnimateAction.SHOW:
            this.objects[this._animate.next.key].visible = true;
            this._goNext();
            break;
        case this.AnimateAction.END:
            this._animate.state = AlgorithmOS.AnimateState.STOP;
            console.log("END");
            break;
    }
};

AlgorithmOS.TEMPLATE._createScreenplay = function () {
    var screenplay = [];
    var a = this.AnimateAction;
    //TDOO: create screen
    // screenplay.push({
    //     action: a.SET_TEXT,
    //     text: "initial state"
    // });
    // screenplay.push({action: a.SET_WAIT, time: 120});
    // screenplay.push({action: a.END});

    this.screenplays['name'] = screenplay;
};
```