Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Plzeň, 2013

Havel Kotál

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Simulátor základních algoritmů OS

Plzeň, 2013 Havel Kotál

zadání

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 14.6.2013

.....

Havel Kotál

Abstract

Simulator of basic OS algorithms

Within the framework of a thesis there were examined and described algorithms that are used in operating systems. On agreement with a supervisor there were chosen three algorithms for their implementation in an outcoming program.

The thesis solves what technologies and utilities are the best to use for a realization of the program. It deals with a selection of suitable visual and data representations, including a model that allows a simple expandability of any further algorithm simulations. In the thesis is also analyzed an implementation of the program that allows simple and illustrative simulation of particular algorithms.

For the purpose of simplifying the process of adding new algorithms there is written a programmer's guide. Then there is written a user's guide that describes numerous ways to use the program and that specifies the logic of implemented modules with algorithms.

Versions of the program functioning under Linux and Windows operating systems are a part of an attached CD that also includes both of the above mentioned guides.

Abstrakt

Simulátor základních algoritmů OS

V rámci bakalářské práce jsou prostudovány a popsány algoritmy používané v operačních systémech (OS). Z nich jsou po domluvě se zadavatelem vybrány tři k implementaci ve výsledné aplikaci.

Práce obsahuje rozvahu nad výběrem nejvhodnějších technologií a prostředků pro realizaci aplikace. Zabývá se výběrem vhodných vizuálních a datových reprezentací včetně modelu zaručujícího jednoduchou rozšiřitelnost o další simulace algoritmů. Rozebírá implementaci aplikace umožňující jednoduché a názorné simulace jednotlivých algoritmů.

Za účelem co největšího usnadnění přidávání nových algoritmů, je sepsán průvodce vytvořením nového modulu – programátorská příručka. Uživatelská příručka pak popisuje ovládání aplikace a popis logiky implementovaných modulů s algoritmy.

Funkční aplikace ve verzích pod systémy Linux i Windows je součástí přiloženého CD, které obsahuje i obě výše zmíněné příručky.

Obsah

1. Úvod

Cílem této práce je vytvořit aplikaci umožňující jednoduchou a názornou simulaci různých algoritmů využívaných v operačních systémech, zejména za účelem jejich snazšího pochopení při výuce. Aplikace musí být multiplatformní, nabízet uživateli interakci s běžící simulací a poskytnout mu měnit různé její aspekty. Zásadním požadavkem je pak možnost jednoduchého rozšiřování o další simulace algoritmů.

Součástí práce bude přehled různých algoritmů používaných v operačních systémech (OS), z nichž tři budou zvoleny k implementaci jako moduly výsledné aplikace.

Dále bude nutné se zabývat výběrem nejvhodnějších technologií a prostředků pro vlastní realizaci, a návrhem vhodných vizuálních a datových reprezentací včetně modelu zaručujícího jednoduchou rozšiřitelnost o další simulace algoritmů.

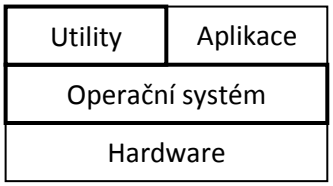
Po implementaci aplikace bude nutné se ještě zabývat formou distribuce pod všechny podporované operační systémy.

Nakonec bude sepsána podrobná uživatelská příručka popisující, jak ovládání aplikace, tak i vnitřní principy fungování implementovaných algoritmů. Dále pak programátorská příručka obsahující návod, jak do aplikace přidat nový modul s algoritmem. Obě příručky budou součástí přílohy této práce.

2. Přehled vybraných algoritmů používaných v OS

Hlavním účelem operačního systému (OS) je řešení komunikace mezi hardwarem a softwarem. OS reprezentuje jakýsi zobecněný abstraktní stroj, který poskytuje lehko použitelné rozhraní k těžko uchopitelné vrstvě hardwaru, a nabízí uživateli programovou základnu k vývoji a spouštění aplikací. Součástí OS pak často bývají takzvané utility, tedy programy usnadňující monitorování a správu systému. Vzájemný vztah hardware, OS a aplikací pak znázorňuje ***Obr. 1***. [1]

V rámci každého OS je zapotřebí řešit značné množství rozličných úloh, ať už se jedná o správu informací v operační paměti, formu uložení dat na různá média, nebo principy spouštění uživatelských aplikací. K řešení takovýchto úloh se v OS používají různé přístupy, a každý z nich přináší své výhody i nevýhody. Vybrané algoritmy využívané v OS jsou popsány níže.



Obr. 1 – Uspořádání počítače

2.1. Algoritmy plánování procesů

Problematika plánování procesů se zabývá tím, jakým způsobem zpracovávat úlohy (rozumějme procesy nebo vlákna) tak, aby jejich běh odpovídal zadaným kriteriím. Zmíněná kritéria jsou pak do značné míry závislá na samotném účelu daného OS. Například u dávkových systémů bude kladen důraz na výpočetní efektivitu, nebo na sdílení zdrojů; u systémů reálného času bude zásadním kritériem předvídatelnost chování a dokončení úlohy v předem stanoveném čase; u interaktivních systémů půjde především o plynulost a zdánlivou současnost běhu více úloh najednou, a tak dále.

Na pozadí každého algoritmu plánování procesů je proto vždy určitý primární záměr jeho využití, který předurčuje jisté jeho vlastnosti a dělá jej více či méně vhodným kandidátem pro řešení dané úlohy. Algoritmů reálně používaných v OS je pak celá řada, zde budou vyjmenovány jen ty nejpoužívanější nebo nejzajímavější.

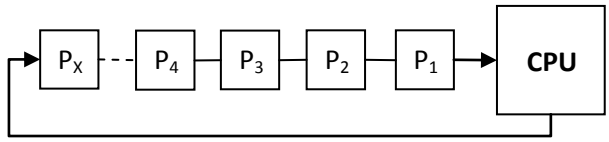
2.1.1. Cyklická obsluha (Round Robin)

Informace o algoritmu cyklické obsluhy byly čerpány ze zdrojů [1, 2, 3].

Algoritmus cyklické obsluhy, jak je již z názvu patrné, obsluhuje jednotlivé procesy v kruhu (viz ***Graf 1***), standardně za využití fronty typu FIFO (First-In, First-Out). Každému procesu pak přidělí určitý čas na CPU (tzv. časové kvantum), po které může běžet. Pokud proces běží ještě na konci časového kvanta, je provedena preempce, tedy přerušení právě vykonávané úlohy, a je naplánován a spuštěn další připravený proces. K témuž dojde i v případě, že proces skončil nebo se zablokoval ještě před spotřebováním celého kvanta.

Jedná se o jeden z nejstarších a nejpoužívanějších algoritmů (v různých variantách), který byl navržen pro systémy se sdílením času. Například ve verzi označované jako

virtuální cyklická obsluha se procesy po skončení blokace upřednostní před ostatními, a algoritmus tak lépe pracuje s I/O (vstupně/výstupně) vázanými úlohami.



Graf 1 – Cyklická obsluha typu FIFO

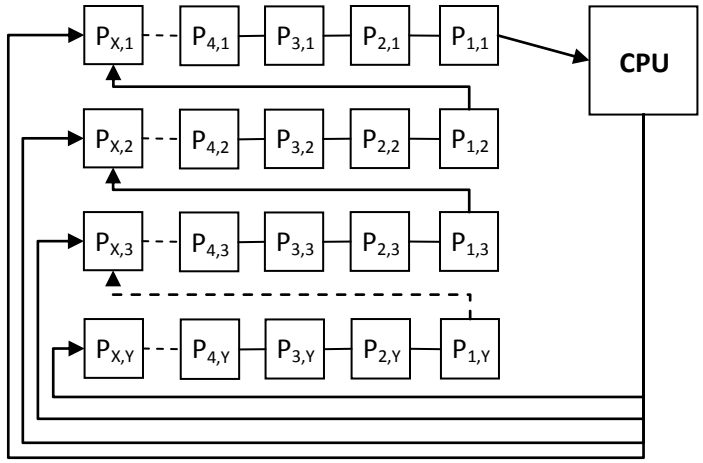
2.1.2. Víceúrovňová zpětná vazba (Multi-level Feedback)

Informace o algoritmu víceúrovňové zpětné vazby byly čerpány ze zdrojů [1, 4].

Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. Pro každou prioritu existuje jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty (viz ***Graf 2***).

Každý proces má přidělenou omezenou výpočetní dobu, kterou může strávit na dané prioritní úrovni, čímž se zaručí upřednostnění I/O vázaných procesů před výpočetně vázanými. Čas přidělený nižší prioritní úrovni je zpravidla dvojnásobný než čas přidělený prioritní úrovni o jedna vyšší.

Některé vybrané procesy mohou mít nastavené prioritní minimum, pod které jejich priorita v průběhu času již neklesne. Pro procesy po skončení blokace může být také použit takzvaný boost – tedy dočasné navýšení priority.



Graf 2 – Víceúrovňová fronta procesů

2.1.3. Víceúrovňová prioritní fronta (Multi-level Priority Queue)

Informace o algoritmu víceúrovňové prioritní fronty byly čerpány ze zdrojů [1, 5, 6].

Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. V praxi pak existuje pro každou prioritu jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně

prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty (viz výše ***Graf 2***).

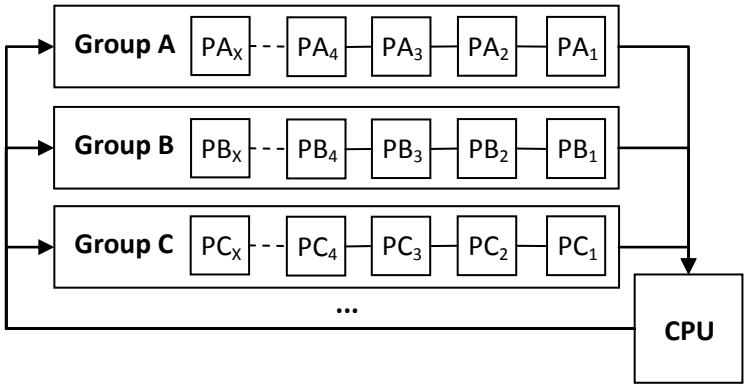
Každý proces disponuje statickou prioritou, která je mu nastavena při jeho vytvoření a zůstává zpravidla po celou dobu jeho existence neměnná, a dynamickou prioritou, která zaručuje, že i procesy s nižší statickou prioritou se občas dostanou na procesor (CPU) a nenastane takzvané jejich vyhladovění. Výsledná priorita procesu je pak určena součtem její statické a dynamické složky. Dynamická priorita se v pravidelných intervalech u všech procesů přepočítá. Základem výpočtu dynamické priority pak je poměrově strávená doba daného procesu na CPU. Takovýmto způsobem se upřednostní I/O vázané procesy před výpočetně vázanými.

V praxi tohoto algoritmu využívá například systém Windows (XP a novější), který pracuje s 32 prioritami. Tato konkrétní implementace umožňuje také vyhrazení určitému procesu právě jeden z přítomných procesorů, což způsobí útlum jeho aktivity, ale částečně uvolní frontu čekajících pro ostatní CPU.

2.1.4. Plánování spravedlivého sdílení (Fair-share scheduling)

Informace o algoritmu spravedlivého sdílení byly čerpány ze zdroje [1].

Algoritmus je založen na základech cyklického prioritního plánování, kde každý uživatel nebo uživatelská skupina dostává přiděleno přibližně stejné množství výpočetního času procesoru. V praxi se pak určitým uživatelským skupinám přidělují různé priority v rámci nichž dochází ke spravedlivému sdílení výpočetního času mezi jednotlivými uživateli. Při přidělování času uživateli není pak zohledňován počet jím spuštěných procesů, tedy dva uživatelé ze stejné skupiny mají vždy spravedlivě přidělenou část CPU bez ohledu na to, jaké programy mají spuštěné. Grafickou reprezentaci front procesů uživatelů příslušejících jednotlivým skupinám znázorňuje ***Graf 3***.



Graf 3 – Fronty procesů dle uživatelských skupin

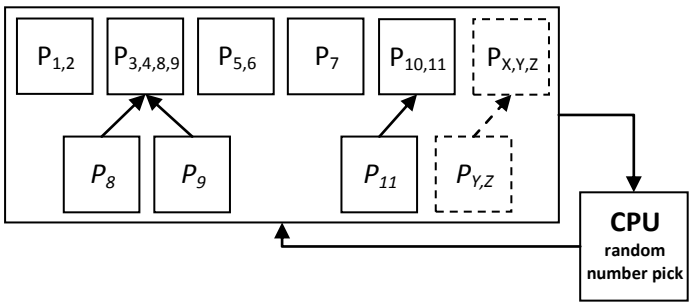
2.1.5. Plánování pomocí loterie (Lottery Scheduling)

Informace o algoritmu spravedlivého sdílení byly čerpány ze zdrojů [7, 8].

Algoritmus pracuje na principu rozdávání takzvaných losů. Každý procesor jich má k dispozici určité množství, a rozdá je mezi jednotlivé procesy s tím, že jeden proces jich může dostat i více čímž získá jakoby vyšší prioritu. Následně dojde k náhodnému losování a výherce obdrží jedno časové kvantum na daném procesoru.

V rámci algoritmu lze jednoduše optimalizovat například vzájemné čekání procesů tím, že čekající předá své losy procesu, na který čeká, čímž zvýší jeho šanci na přiřazení výpočetního kvanta. Možné je také procesu vyhradit procentuální část celého CPU tím, že je mu přiděleno dané procento ze všech losů procesoru.

Příslušnost losů k procesům, vzájemnou závislost procesů, a náhodný výběr losu přibližuje ***Graf 4***.



Graf 4 – Výběr procesu pomocí loterie

2.2. Algoritmy správy paměti

Problematika správy paměti se zabývá na jedné straně efektivitou uchovávání informace v operační paměti počítače, na straně druhé se snaží nabízet způsoby řešení při nedostatku paměti pro všechny běžící úlohy v systému.

Efektivitou uchovávání informace je pak myšlena rychlost nalezení volné oblasti paměti, rychlost nalezení požadované informace v uložených datech, a využitelnost velikosti paměti vzhledem k uloženým datům. Z logiky věci pak vyplývá, že rychlost zápisu/čtení půjde proti efektivnosti využití paměťového prostoru a naopak.

2.2.1. Správa paměti pomocí seznamů (Variable Sized Partitioning)

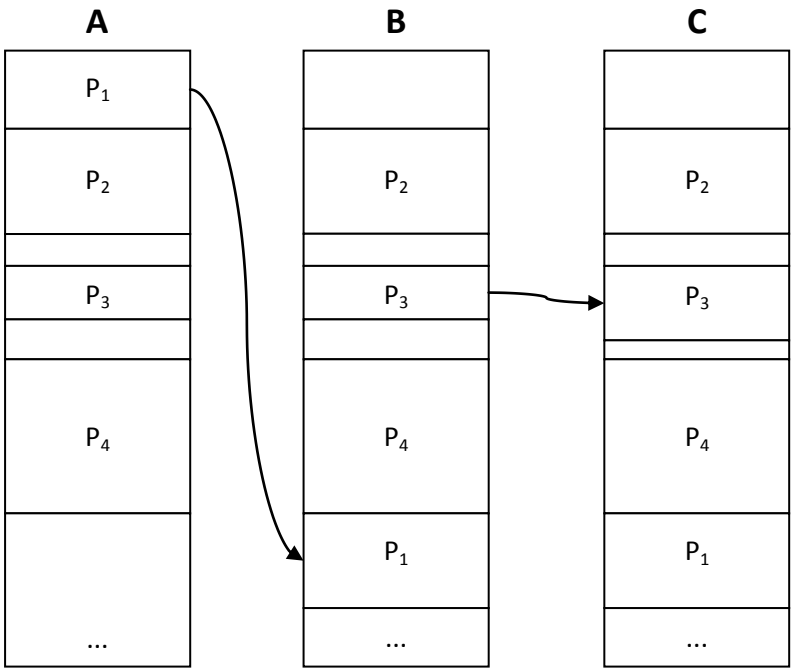
Informace o správě paměti pomocí seznamů byly čerpány ze zdroje [9].

Tato metoda umožňuje rozdělení paměti do různě velkých alokačních jednotek, jejichž velikost a umístění ve fyzické paměti se může během času měnit. Alokační jednotky jsou reprezentovány dvousměrně zřetězeným seznam, nebo dvěma oddělenými seznamy, ve kterých každá položka obsahuje informace o tom, zda se jedná o volnou nebo obsazenou oblast, dále pak počáteční adresu jednotky v paměti, a na konec její délku. Obsazování paměti zpravidla probíhá na základě hledání volného místa „First-Fit“, „Next-Fit“, nebo „Best-Fit“.

Při skončení procesu se oblast paměti přeznačí na uvolněnou a zkontroluje se obsazenost předchozí a následující jednotky; ty se případně sloučí. Algoritmus je také adaptivní vzhledem k růstu paměťové náročnosti jednotlivých procesů, v takovém případě je proces přesunut do jiné části paměti, nebo v případě volného místa za alokovanou oblastí procesu, je její velikost jenom rozšířena.

Správa paměti pomocí tohoto algoritmu je obecně efektivnější než jednoduché rozdělení paměti na fixně velké oblasti, při jejím používání ovšem časem dochází k postupnému přibývání malých volných oblastí, a je pak zapotřebí občas paměť defragmentovat.

Využití paměti procesy a přesuny při jejich růstu znázorňuje ***Graf 5***, kde je možno pozorovat tři různé stavy paměti A, B a C, mezi nimiž se napřed zvýší paměťové požadavky procesu P_1, a pak procesu P_3.



Graf 5 – Alokace paměti při růstu procesů za správy pomocí seznamů

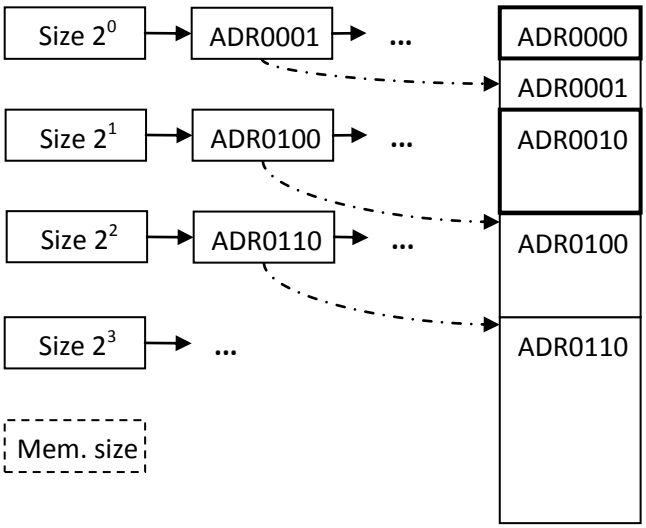
2.2.2. Správa paměti pomocí „Buddy System“

Informace o správě paměti pomocí „Buddy System“ byly čerpány ze zdrojů [2, 4, 10].

Algoritmus pracuje se seznamy volných bloků všech velikostí mocniny dvou, a to od jedné až do velikosti celé paměti. Při požadavku na vyhrazení paměti se vždy velikost zaokrouhlí nahoru na mocninu dvou a prohledají se seznamy. Pokud se nalezne jen větší úsek, rozdělí se na dva poloviční a test se opakuje až do získání oblasti o potřebné velikosti, která se vyhradí. Při uvolňování paměti se pak kontroluje i takzvaný soused (buddy), zda není také volný. Pokud ano, dojde ke sloučení a přeřazení do následujícího seznamu. Tento krok se pak provádí se všemi následujícími sousedy, dokud není nalezen obsazený buddy, nebo není volná celá paměť.

Oproti metodám jako „First-Fit“ nebo „Best-Fit“ zaručuje tento přístup výrazné urychlení hledání volných bloků. Jeho nevýhodou je ovšem plýtvání pamětí při zaokrouhlování potřebné velikosti na mocninu dvou.

Logiku algoritmu „Buddy System“ vizualizuje ***Graf 6***, na kterém je znázorněn příklad seznamu adres volných bloků ukazujících do paměti, v níž jsou i bloky obsazené, které jsou tučně orámovány.



Graf 6 – Algoritmus „Buddy System"

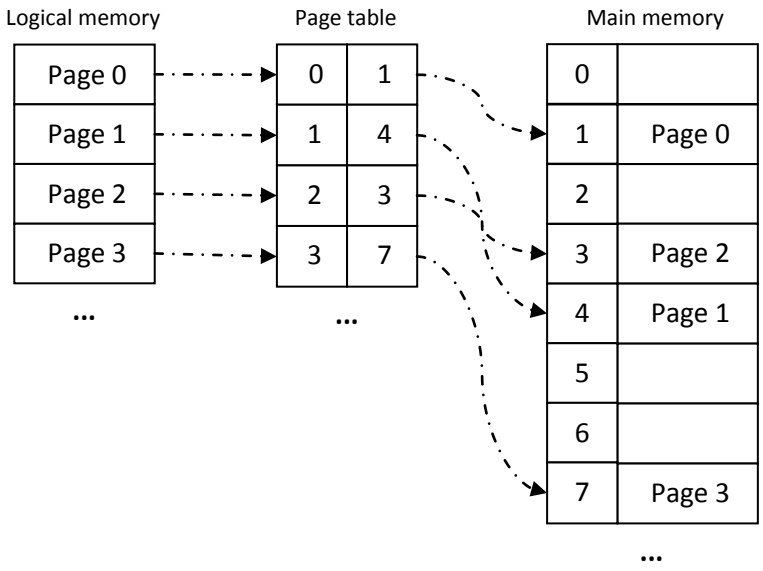
2.2.3. Stránkování paměti (Memory Paging)

Informace o stránkování paměti byly čerpány ze zdrojů [11, 12].

Při stránkování paměti může být proces uložen v různých místech fyzické paměti, která nemusí být souvislá. Tohoto je dosaženo za pomoci využití logické paměti rozdělené na tzv. stránky o fixní velikosti. Fyzická paměť je pak rozdělena do rámců o stejné velikosti jako jsou stránky. Při každém přístupu do paměti procesem, je pak logická adresa překládána na fyzickou, kde jsou uložena hledaná data. Vztah mezi logickou a fyzickou pamětí popisuje ***Graf 7***.

Implementačně se problém řeší udržováním tabulky stránek pro každý běžící proces. Dále se udržuje tabulka rámců fyzické paměti s informací, zda je daný rámec volný nebo obsazený. Při růstu procesu se pak v paměti alokuje nějaký volný rámec.

Této metody využívá většina dnešních operačních systémů, a to často v kombinaci s odkládáním rámců na disk. Tím je docíleno zvětšení celkově dostupné operační paměti systému, ovšem za cenu výrazně zpomaleného přístupu k odloženým rámcům. Dále pak stojí za zmínku, že metoda stránkování paměti vyžaduje podporu hardwaru, od něhož se pak odvíjí například velikost použitých rámců. Nespornou výhodou metody je její redukce fragmentace fyzické paměti.



Graf 7 – Vztah mezi logickou a fyzickou pamětí

2.3. Algoritmy správy souborů

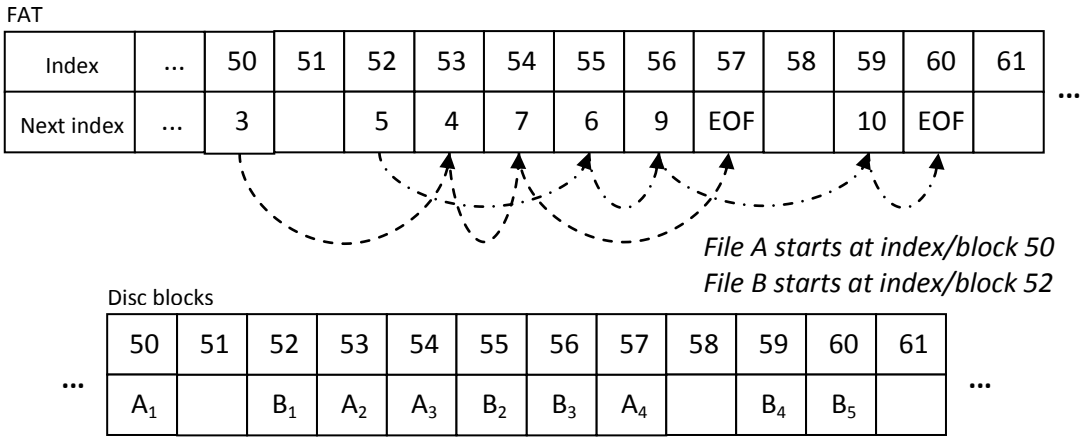
Problematika správy souborů se zabývá nalezením vhodné reprezentace informace pro uchování dat na různá média, s tím, že klade důraz na jejich odolnost vůči poškození, na rychlost jejich čtení a zápisu, nebo například na jejich ochranu proti neoprávněnému přístupu.

2.3.1. Souborová alokační tabulka – File Allocation Table (FAT)

Informace o souborové alokační tabulce byly čerpány ze zdrojů [13, 14].

Souborový systém FAT je dnes podporován vesměs všemi operačními systémy a to především z důvodů kompatibility a přenositelnosti. Jedná se o velice jednoduchý model přístupu k souborům, který je založen na uchovávání si tabulky odkazů na jednotlivé bloky souborů na disku. Každý záznam tabulky vždy buď odkazuje na záznam následující (jakýkoliv jiný v celé tabulce FAT), nebo obsahuje speciální značku EOF (End Of File). Vizualizaci tohoto principu znázorňuje ***Graf 8***.

Velikost tabulky FAT je pak omezena velikostí binárního čísla použitého k adresování každého diskového bloku, s čímž souvisí omezení na maximální velikost použitého bloku, na maximální velikost jednoho souboru, nebo maximální velikost diskového oddílu. V současnosti se běžně využívají verze FAT16 a FAT32.



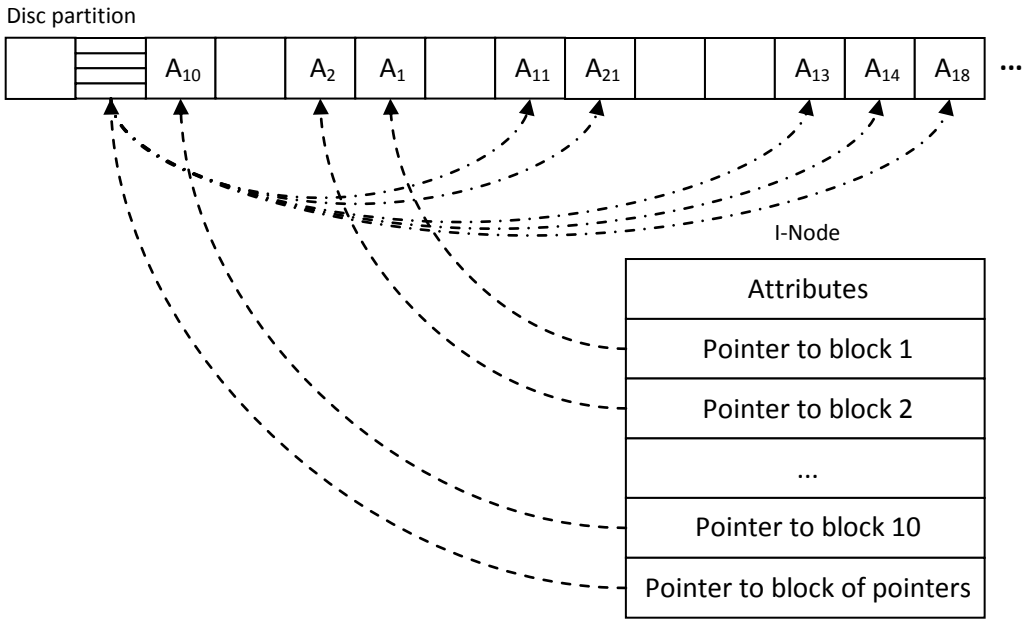
Graf 8 – Princip odkazování ve FAT

2.3.2. Index-Node

Informace o Index-Node byly čerpány ze zdroje [15].

Ke každému souboru existuje datová struktura I-Node, která obsahuje atributy souboru, odkazy na prvních N bloků souboru a jeden nebo více odkazů na diskové bloky obsahující další diskové adresy (viz nákres struktury I-Node v ***Graf 9***). Jednotlivé I-Node jsou pak zřetězeny do takzvaných seznamů I-List.

Výhodou této implementace je jednoduché načtení I-Node a případně bloku s dalšími adresami souboru do paměti, čímž se urychlí přístup k němu.



Graf 9 – Princip odkazování v I-Node

3. Aplikace – Simulátor algoritmů

V této kapitole je představena aplikace, která je součástí bakalářské práce. Jsou zde uvedeny jak požadavky na její funkcionalitu, tak rozbor výsledného návrhu a konečné implementace. Kapitola obsahuje také popis uživatelského rozhraní, nástin vytvoření nového modulu, a další.

3.1. Požadavky

Účelem aplikace je doplnění výuky algoritmů operačních systémů o názorné interaktivní ukázky tak, aby si studenti algoritmy lépe osvojili a pochopili jejich princip. Na tuto funkcionalitu je tedy kladen zvláštní důraz. Podporované algoritmy budou z oblasti správy procesů, paměti, souborů a dalších. Dále musí být aplikace uzpůsobena k jednoduchému rozšiřování do budoucna za pomoci modulů s novými algoritmy. V neposlední řadě je žádoucí, aby byla aplikace multiplatformní a to alespoň pro systémy typu Windows a Linux.

Aplikace by měla obsahovat grafickou reprezentaci běhu samotného algoritmu, dále pak ovládací prvky pro zpomalení a urychlení simulace, možnost uložení a opětovného načtení aktuálního stavu daného algoritmu a samozřejmě výběr z implementovaných algoritmů. Pro ovlivnění stavu jednotlivých prvků, rozumějme například procesů, by aplikace měla obsahovat příslušné ovládací prvky, tedy například pro změnu priority, či vynucení ukončení procesu, a podobně. Ke každému prvku by pak měla obsahovat možnost zobrazení podrobných informací.

3.2. Upřesnění zadání a výběr technologií a prostředků vývoje

Z požadavků na aplikaci vyvstává několik základních otázek:

- *Na jakých platformách musí být program s jistotou funkční?*
- *Jaký programovací jazyk a jaké technologie a nástroje pro vývoj využít?*
- *Jakým způsobem dostatečně jasně prezentovat jednotlivé děje simulace?*
- *Jak implementovat jednoduchou rozšiřovatelnost o nové simulace?*
- *Jaké algoritmy v rámci bakalářské práce zapracovat?*

Po dohodě se zadavatelem bylo stanoveno jako dostačující zprovoznění programu pod operačními systémy Linux (distribuce typu Debian) a Windows.

Pro zvolení programovacího jazyka a dalších nástrojů vývoje byl proveden malý výzkum:

- Jelikož jedním z požadavků na aplikaci je multiplatformnost, bylo nutné zvolit takový jazyk, který by tento požadavek respektoval. Kandidáty se tedy staly jazyky Java a Python. Oba měly svá pro i proti. Java v komfortu vývoje v prostředích jako NetBeans nebo Eclipse a velice dobré podpory GUI jako Swing. Python na druhé straně v rychlosti růstu kódu (člověk se nemusí ve většině případů zabývat řešením mnoha okrajových problémů a může se soustředit přímo na řešení svého úkolu) a celkově značně jednodušší prací oproti běžným jazykům.
- Následoval pokus o jednoduché zapracování grafického rozvržení budoucí aplikace v obou jazycích, z čehož vzešlo zjištění, že vzhledem k nárokům na vizuální

stránku simulací bude nutné využít OpenGL a tedy odpadají veškeré výhody využití Swingu. Jako podpůrná knihovna pro jednodušší realizaci GUI v OpenGL v Pythonu se pak zdála být nejvhodnější knihovna Pyglet, která umožňuje jednoduchou práci s událostmi, poskytuje prostředky pro efektivní seskupování grafických elementů, zavádí tzv. layout pro jednodušší práci s textem včetně jeho rolování, nebo umožňuje jednoduchou realizaci kurzoru pro textový vstup od uživatele.

- Nakonec tedy pro vývoj aplikace byla vybrána kombinace nástrojů Python2.7 [16] a knihovny PyOpenGL [17] a Pyglet [18], které společně poskytují vysokou universalitu a zároveň dostatečnou přenositelnost. Jako vývojové prostředí pak systém Linux, konkrétně distribuce KUbuntu 12.10 desktop 64bit, za pomoci textového editoru Vim.

Pro názornost dějů jednotlivých simulací a vazeb mezi různými stavy algoritmu byla vybrána grafová reprezentace, kde jednotlivé oblasti jsou zastoupeny poli čtverců svázanými šipkami. Elementy simulace (např. procesy) pak obrázkem, který se v rámci stavů simulace po grafu přesouvá, čímž vzniká pohyblivý model chování daného algoritmu. Navíc se jednotlivé události simulace zapisují v textové podobě tak, aby uživatel vždy měl přehled o jejich následnosti a mohl si vždy zkontrolovat jejich historii.

Pro jednoduché rozšiřování o nové simulace algoritmů se zdálo být jako nejvhodnější využití nativních modulů jazyka Python za pomoci podědění od abstraktní třídy algoritmu, která by určovala rozhraní pro programátora modulu a určitým způsobem jej naváděla.

Po prostudování algoritmů používaných v operačních systémech a po domluvě se zadavatelem byly jako dostatečně reprezentativní vzorek stanoveny algoritmy cyklická obsluha (round-robin), víceúrovňová zpětná vazba (multi-level feedback queue) a víceúrovňová prioritní fronta (multi-level priority queue), s tím, že další algoritmy bude jednoduché implementovat kýmkoliv jiným.

3.3. Ovládání aplikace z pohledu uživatele

Po spuštění programu se uživatel dostane do menu, kde si může zvolit spuštění nějakého algoritmu (tzv. modulu). Po výběru se pak rozšíří možnost ovládacích prvků o regulaci rychlosti simulace, jejího uložení a načtení, a dalších nastavení, souvisejících přímo s danou simulací. S tím se zobrazí také grafická reprezentace běhu vybraného algoritmu dle v algoritmu definovaných parametrů prostředí a nastavení. S grafickou reprezentací bude možné za pomoci pravého tlačítka myši po obrazovce pohybovat, nebo ji kolečkem myši přibližovat či oddalovat. Simulace se vždy spustí v zastaveném režimu. Každý prvek (např. proces) bude reprezentován ikonou a dalšími informacemi v podobě značek nebo čísel znázorňujících údaje jako prioritu, stav, apod. Na každý prvek bude možné kliknout myší, a tak jej vybrat k bližšímu prozkoumání – zobrazí se podrobné informace a statistiky k prvku a nastavení některých hodnot jeho vlastností.

Ovládání programu je rozděleno do několika sekcí: Hlavní menu, Panel textového logu, Nástrojový panel modulu a Oblast simulace.

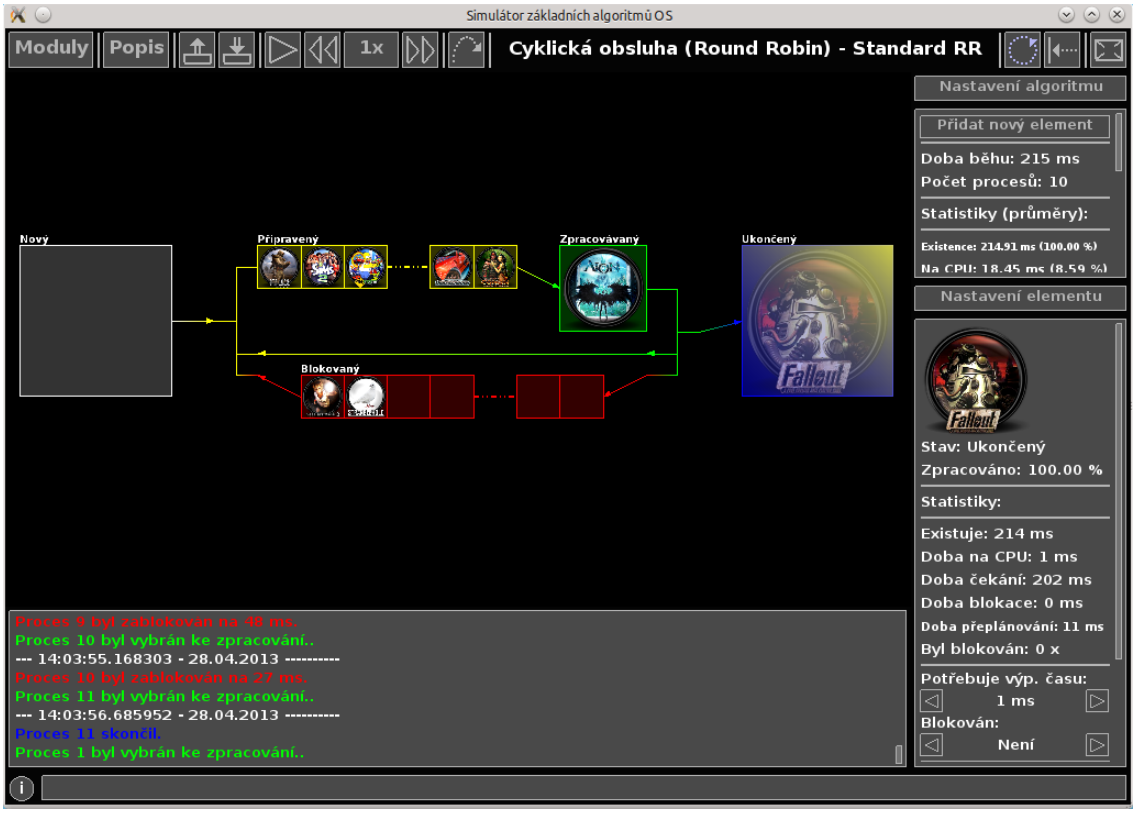
Hlavní menu se nachází v horní části okna a poskytuje možnosti jako načtení modulu s algoritmem, spuštění a zastavení simulace, regulaci její rychlosti, uložení a načtení aktuálního stavu simulace a podobně.

Panel textového logu se nachází v dolní části okna a obsahuje podrobný textový záznam všech prováděných akcí simulovaných prvků. Uživatel tak má vždy přehled o všem, co se v daném kroku událo. Záznamy logu je pak možné prohlížet i zpětně.

Nástrojový panel modulu se nachází v pravé části okna a je rozdělen do dvou částí. Horní část panelu je určena k zobrazení informací a nastavení k celému modulu. Může obsahovat například globální statistiky probíhající simulace, nebo nastavení počtu použitých procesorů nebo velikosti simulované paměti a pod. Dolní část panelu je pak učena k zobrazení informací a nastavení k vybranému prvku simulace. Zde pak mohou být například nastavení priority daného prvku, nebo možnost jeho vynuceného ukončení, atd.

Oblast simulace je pak zbytek scény, do kterého se vykresluje rozvržení aktuální simulace. Simulací je pak možné pohybovat za pomoci myši způsobem kliknu-držím-táhnu, nebo ji za pomoci kolečka oddalovat nebo přibližovat tak. Uživateli je tak umožněno sledovat simulaci buď jako celek, nebo si přiblížit jen její určitou oblast, která jej zajímá.

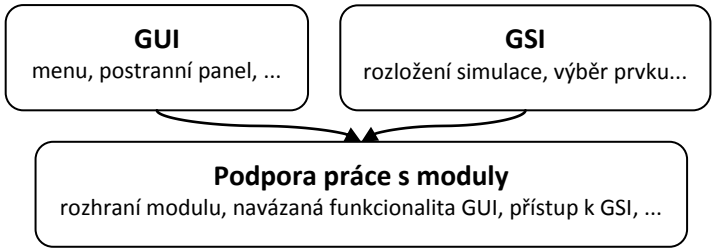
Výsledný náhled celé aplikace je pak možné vidět na ***Obr. 2*** znázorňujícím simulaci algoritmu cyklické obsluhy.



Obr. 2 – Náhled aplikace s běžícím algoritmem cyklické obsluhy

3.4. Implementační rozvržení aplikace

Implementaci aplikace můžeme rozdělit na tři základní logické celky, a to na grafické uživatelské rozhraní (GUI), grafické rozhraní simulace (GSI) a podporu práce s moduly. GUI a GSI jsou na sobě zcela nezávislé části, naopak podpora práce s moduly je přímo navázána na funkčnost obou částí předchozích (viz ***Graf 10***).



Graf 10 – Přehled jednotlivých částí aplikace a jejich závislost

3.4.1. Grafické uživatelské rozhraní (GUI)

GUI je hlavním nástrojem uživatele k ovládání aplikace. Obsahuje zejména lištu hlavního menu umístěnou v horní části okna, ze které jsou dostupné jak veškeré funkcionality samotné aplikace, tak i funkcionality společné pro všechny moduly. Dále dolní informační lištu, kde se uživatel dozví zda se právě prováděná akce zdařila, či nikoliv a případně z jakého důvodu. Po načtení modulu se rozšíří hlavní menu a přibude dolní panel textového logu simulace a pravý panel určený k zobrazení informací a specifických nastavení k právě načtenému modulu.

3.4.1.1. Základní přehled implementovaných prvků GUI

GUI je zcela založeno na knihovně Pyglet a je možné jej nalézt v souboru `lib/gControls.py`. Přítomné prvky lze rozdělit buď na aktivní a pasivní, na standardní a vložené, nebo dle použití na hlavní a pomocné. Aktivní prvky jsou ty, na které může uživatel kliknout a vyvolat nějakou akci, pasivní pak ty, na které žádná akce navázána není a mají jen informativní nebo estetický účel. Za vložené jsou považovány ty prvky, které je nutné vkládat do tzv. layoutu. Tyto prvky pak lze rolovat v rámci stanoveného výřezu a v aplikaci se využívají především pro zobrazení informací o stavu simulace nebo k nastavování jejích parametrů. Standardní prvky jsou pak všechny ostatní jako například obyčejné tlačítko v liště menu. Hlavní prvky jsou většinou seskupené prvkové celky jako lišta menu, která zaobaluje funkčnost pro celou sadu vnitřních pomocných prvků – tlačítek, textů, atd. Pomocnými prvky pak rozumíme ty prvky, které sice mají svůj vlastní účel a bylo by možné je samostatně použít, avšak jsou vždy použity jen v rámci nějakého prvku hlavního.

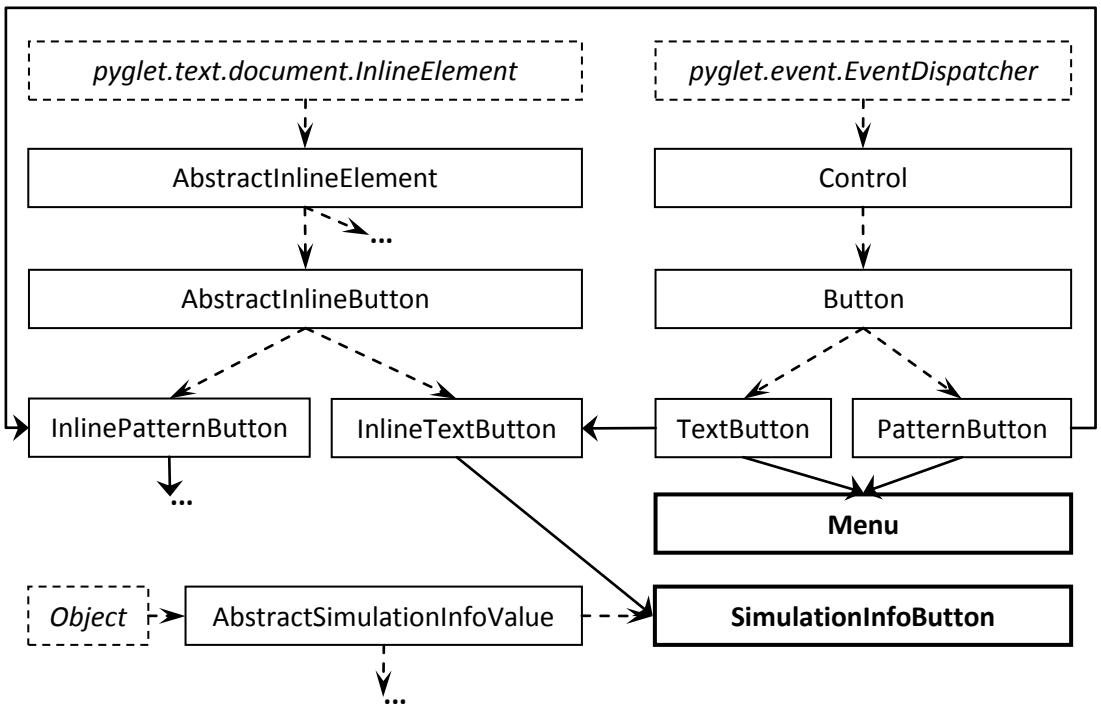
Aktivní prvky jsou děděny od `lib.gControls.Control`; vložené prvky od `lib.gControls.AbstractInlineElement`. Všechny prvky jsou tvořeny buď děděním od abstraktních tříd, nebo logicky či funkčně slučují prvky jiné. Jejich základní přehled můžete vidět v ***Tab. 1***. Typy prvků jsou: aktivní (A) nebo pasivní (P), standardní (S) nebo vložený (I) a hlavní (M) nebo pomocný (H)

Základní přehled prvků GUI (`lib.gControls`)		
Jméno objektu	**Typ**	**Popis**
`PatternButton`	ASH	Tlačítko vyplněné vzorem
`TextButton`	ASH	Textové tlačítko
`SpeedControls`	ASH	Skupina řešící nastavení rychlosti (dvě tlačítka, jeden text, dodatečná logika)

Základní přehled prvků GUI (`lib.gControls`)		
Jméno objektu	**Typ**	**Popis**
`Menu`	ASM	Hlavní menu (tlačítka, nastavení rychlosti a další objekty, dodatečná logika využívající `ListBox` a `Dialog`)
`Input`	ASH	Objekt pro zadávání textu uživatelem
`Dialog`	ASM	Objekt dialogové tabulky typu ANO/NE; může využívat `Input`
`TextBox`	ASH	Obecná oblast textu umožňující rolování
`LogBox`	ASM	Zobrazení logu; poděděné od `TextBox`
`InfoBox`	ASM	Informační lišta; využívá jednořádkový `LogBox`
`ListBox`	ASM	Výběr ze seznamu hodnot; poděděné od `TextBox`
`ModuleInfoBox`	ASM	Pravý informační panel; podporuje vložené objekty; využívá `TextBox`
`InlinePatternButton`	AIH	Vložené tlačítko vyplněné vzorem
`InlineTextButton`	AIH	Vložené textové tlačítko
`InlineText`	PIH	Vložený text; velikost písma se uzpůsobuje vyhrazené oblasti v závislosti na délce textu
`InlineSeparator`	PIH	Vložený horizontální oddělovač

Tab. 1 – Tabulka se základním přehledem prvků GUI

Provázanost tříd prvků GUI můžete vidět na ukázce ***Graf 11***, kde čárkované šipky značí dědičnost; plné značí použití v instanci třídy, na kterou ukazují; čárkované obdélníky značí vnější třídy (buď z knihoven nebo přímo Pythonu); plné slabé obdélníky pomocné třídy, a tučné plné obdélníky hlavní třídy.



Graf 11 – Ukázka provázanosti tříd prvků GUI

Program samotný umožňuje uživateli tři základní akce, a to načtení modulu, přepnutí mezi celoobrazovkovým režimem a režimem okna, a ukončení programu za pomoci klávesy „ESC“. Tlačítko pro načtení modulu a tlačítko přepnutí celoobrazovkového režimu (ve variantách „na celou obrazovku“ a „do okna“) je vyobrazeno na ***Obr. 3***.

Obr. 3 – Hlavní tlačítka

Kromě toho, je program vybaven „Informační lištou“ umístěnou v dolní části okna, ve které se zobrazují informace, jako, že se úspěšně podařilo načíst modul, nebo že v zadaném jméně uložení pozice se vyskytují nepovolené znaky, atp.; je tedy doporučeno lištu sledovat. Vizualizace lišty je vidět na ***Obr. 4***.


Obr. 4 – Informační lišta

Ovládání modulu se zpřístupní až po jeho načtení za pomoci tlačítka „Moduly“. Nové ovládací prvky zahrnují jak obecné akce s modulem jako uložení aktuálního stavu simulace apod., tak i akce zcela specifické pro daný modul.

OBECNÉ AKCE

Jsou umístěny v hlavním menu a obsahují:

- **Zobrazení popisu modulu**

  Zobrazovaný popis je text sestavený autorem modulu tak, aby přiblížil logiku simulovaného algoritmu a upozornil na jeho výhody a nevýhody.

- **Načtení dříve uložené pozice simulace**

  
  Načíst je možné jen pozice simulace právě načteného modulu. Výběr uložených pozic je řazen sestupně podle data a času vytvoření a obsahuje uživatelem zadané jméno a datum a čas vytvoření souboru.

- **Uložení aktuálního stavu simulace**

  
  Pro uložení aktuálního stavu simulace musí uživatel zadat její jméno. Znaky použitelné pro pojmenování jsou omezeny na malé a velké znaky anglické abecedy, čísla a znaky podtržíto, pomlčka a tečka.

- **Spuštění a zastavení běhu simulace**

  
  Simulace se spustí s aktuálně nastavenou rychlostí. Standardní rychlost simulace je 1 krok za sekundu. Po spuštění simulace běží, dokud není uživatelem opět pozastavena.

- **Zpomalení a zrychlení běhu simulace**

  Rychlost běhu simulace může být nastavena v rozmezí hodnot od 1 kroku za 5 sekund do 30 kroků za sekundu. Rychlost simulace je jen orientační a je vždy omezena výkonem počítače a náročností výpočtu kroku právě načteného modulu.

- **Ruční posun v simulaci o jeden krok**

  
  Umožňuje uživateli procházet si jednotlivé kroky simulace postupně dle svého uvážení, aniž by byl vázán na nějaké nastavené tempo simulace. Tato možnost je obzvláště výhodná při prezentacích, kdy uživatel jednotlivé kroky simulace komentuje.

- **Obnovení výchozího stavu simulace**

  Umožňuje jednoduché vrácení se do stavu posledního načtení uložené pozice. V případě, že žádná uložená pozice načtena nebyla, je programem stav simulace zcela vymazán.

- **Vymazání stavu simulace**

  Umožňuje jednoduše vymazat aktuální stav simulace a v podstatě se vrátit do momentu těsně po načtení modulu.

SPECIFICKÉ AKCE PRO DANÝ MODUL

Jsou umístěny v pravém ovládacím panelu, a jsou, jak je již z názvu patrné, pro každý modul jiné. Obecně můžeme tedy jen popsat určitou logiku jednotlivých prvků, které se v pravém ovládacím panelu mohou objevit:

- **Popis**

  Informační text, který se bude pravděpodobně během simulace aktualizovat, ale neumožňuje uživateli vyvolat žádnou akci.

- **Hodnota**

  Záznam, který má jak informativní účel, tak umožňuje uživateli měnit hodnotu daného parametru simulace nebo elementu v simulaci.

- **Tlačítko**

  Prvek, který slouží čistě k vyvolání určité akce, např. k přidání nového elementu do simulace.

Obr. 5 – Nastavení algoritmu

Obr. 6 – Nastavení prvku simulace

Dále bychom měli zmínit, že rozložení jednotlivých informací v pravém ovládacím panelu je rozděleno na informace a ovládací prvky k celé simulaci (viz ***Obr. 5***) a na informace a ovládací prvky k právě vybranému elementu simulace (viz ***Obr. 6***) – element je možné vybrat pomocí prokliku levého tlačítka myši nad jeho ikonou v simulaci.

Více informací o používání aplikace obsahuje příloha A – Uživatelská příručka.

3.4.2. Grafické rozhraní simulace (GSI)

GSI je primárně nástroj pro tvůrce modulu a snaží se mu co nejvíce zjednodušit návrh grafického rozvržení simulace. Jeho hlavním účelem je jednoduché vzájemné propojování vizuálních entit a zastřešení jejich jednotného ovládání pro uživatele. K docílení výše zmíněných požadavků se využívá „vkládání objektů do sebe“ s uchováním parametricky zadané vzájemné polohy, čímž vzniká jakýsi strom těchto

elementů. Těchto stromů pak může být pro danou simulaci vytvořeno více, v implementovaných modulech se však využívá vždy jen jeden.

3.4.2.1. Základní přehled implementovaných entit GSI

Jednotlivé entity je možné rozdělit na kontejnerové, které mohou uchovávat jednotlivé prvky simulace, jako například `EntityArray` nebo `EntitySet`, a na vazebné, které slouží k vzájemnému propojování entit ostatních. Propojení lze provést dvěma způsoby, a to buď za pomoci objektu `Direction` (předávají se mu právě dvě entity, které propojí v zadaném směru o zadané relativní vzdálenosti; vizuálně se může jednat o šipku, nebo jen úsečku), nebo za pomoci objektu `EntityGroup` (horizontálně nebo vertikálně seskupí libovolný počet entit, což lze využít také k vytvoření pomyslného cyklu). K těmto vazebným entitám měl původně patřit ještě objekt `Branche`, který by vytvořil grafické rozvětvení, nebo naopak sloučení; ovšem vzhledem k povaze simulovaných algoritmů nevyvstala potřeba tento objekt implementovat, a není tedy součástí programu. Základní přehled entit GSI si je možné prohlédnout v ***Tab. 2*** a příklad jejich použití v ***Př. 1***. Ten ukazuje způsob vytvoření fronty připravených procesů a instance procesoru, a jejich vzájemné grafické propojení.

Základní přehled entit GSI (`lib.gComponents`)	
Jméno objektu	**Popis**
`EmptyElement`	Prázdný element – slouží jako pomocný prvek bez grafické reprezentace; lze jej využít například k vytvoření lomené čáry (v kombinaci s `Direction`)
`Direction`	Propojí dvě entity v zadaném směru
`EntityGroup`	Skupina entit – vizuálně sloučí několik entit dohromady
`EntityArray`	Pole entit – slouží například k reprezentaci fronty; má pevně daný maximální počet prvků; umožňuje obsažené prvky jednoduše třídit, aktualizovat, atd.; obsahuje metody `push` a `pop`
`EntitySet`	Množina entit – slouží například k reprezentaci fronty; nemá omezený počet prvků; umožňuje obsažené prvky jednoduše třídit, aktualizovat, atd.; obsahuje metody `push` a `pop`
`PriorityGroup`	Skupina prioritních front - jedná se o rozšiřující entitu implementovanou za speciálním účelem; je součástí knihovny `lib.gAdvancedComponents`; umožňuje obsažené prvky jednoduše třídit, aktualizovat, atd.; obsahuje metody `push` a `pop`

Tab. 2 – Tabulka se základním přehledem entit GSI

Příklad použití grafických entit simulace v modulu

```
# Fronta čekajících procesů
# EntitySet(
#      <počet položek na začátku>, <počet položek na konci>,
#      <délka přerušení>, <barva RGBA>, <popisek>, <směr – horiz./vert.>,
#      <smysl plnění – zleva/zprava>, <porovnávací funkce>, <třída entit>)
self._simulationElements["ready"] = lGC.EntitySet(3, 2, 1.0,
      (1.0, 1.0, 0.0, 1.0), "Připravený", True, True,
      self._local_readyComapreFunc, lGC.Entity)

# Zpracovávané procesy (CPU)
# EntityArray(
#      <počet položek>, <barva RGBA>, <popisek>, <směr – horiz./vert.>,
#      <smysl plnění – zleva/zprava>, <porovnávací funkce>, <třída entit>,
#      <velikost položek>)
self._simulationElements["working"] = lGC.EntityArray(1,
      (0.0, 1.0, 0.0, 1.0), "Zpracovávaný", True, True,
      None, lGC.Entity, 2.0)

# Propojení dvou grafických elementů pomocí šipky zleva doprava
# Direction(
#      <element 1>, <element 2>, <směr>, <vzdálenost>, <zda šipka>)
top = lGC.Direction(self._simulationElements["ready"],
      self._simulationElements["working"], "right", 1.0, ">")
```

Př. 1 – Příklad použití grafických entit simulace v modulu

3.4.2.2. Ovládání simulace

Se simulací (grafickou reprezentací algoritmu) je možné provádět několik základních věcí:

- **Pohyb**

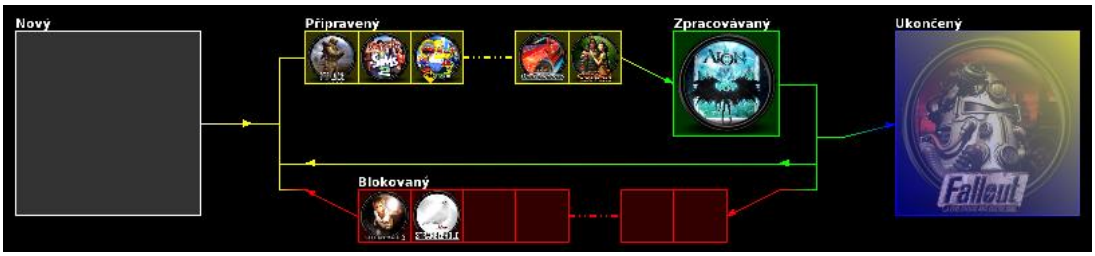
  Simulací je možné pohybovat za pomoci levého nebo pravého tlačítka myši tak, že jej stisknete a při jeho držení myší pohybujete. Stejného efektu lze dosáhnout za použití šipek na klávesnici.

- **Přiblížení/Oddálení**

  Simulaci je možné přibližovat a oddalovat za pomoci kolečka myši, nebo kláves „PageUp“ a „PageDown“.

- **Výběr prvku**

  Pokud jsou v simulaci přidány prvky, je možné nějaký z nich vybrat za pomoci prokliku levého tlačítka myši na jeho ikoně. Chceme-li vybraný prvek odznačit, proklikneme jej pravým tlačítkem myši, nebo označíme prvek jiný. Příklad vysvícení označeného prvku je možné vidět na **Obr. 7.**



Obr. 7 – Ukázka rozvržení simulace algoritmu s označeným prvkem

3.4.3. Podpora modulů

Aby byla aplikace jednoduše rozšiřitelná o simulace dalších algoritmů, umožňuje načítání takzvaných modulů s těmito algoritmy. Všechny moduly jsou klasické knihovny jazyka Python a jsou uloženy v podsložce `modules`. Každý takovýto modul musí obsahovat definici třídy poděděné od `lib.module.AbstractAlgorithm` a několik dalších drobností.

Abstraktní předek algoritmu slouží především jako rozhraní pro přístup k vybraným funkcím GUI a umožňuje jejich uzavřené propojení tak, aby programátor modulu nemohl poškodit funkčnost aplikace. Dále tato třída obsahuje sadu metod logicky rozdělující jednotlivé funkčnosti modulu, takže plní do jisté míry i úlohu jakési šablony.

Rozhraní nabízí zprostředkovaný přístup k panelu textového logu, k jednoduchému vytváření obsahu pravého informačního panelu algoritmu a vybraného elementu simulace, a samozřejmě ke grafickému rozvržení samotné simulace. Mezi jinými pak ještě vyžaduje udržování si své verze, která se zapisuje do souboru při uložení pozice simulace uživatelem. Na základě této informace může pak programátor ošetřovat případné problémy kompatibility mezi verzemi a podobně.

Třída je navržena tak, aby se programátor modulu nemusel zabývat vnitřními zákonitostmi aplikace, a přesto získal funkcionalitu jako například ovládání rychlosti běhu simulace, její uložení a načtení, zobrazení popisu algoritmu a podobně. Na druhou stranu, programátor si musí velice pečlivě ohlídat veškerou funkcionalitu na jím přidaných ovládacích prvcích měnící hodnoty algoritmu nebo vybraného elementu, na správnost výpočtu zobrazovaných statistik, na přesuny elementů v rámci GSI a na jejich časovou posloupnost; musí si také pečlivě rozmyslet, jaké všechny akce provádět v rámci jednoho kroku simulace, a tak dále. Z výše uvedeného je tedy patrné, že rozhraní se snaží programátorovi mnohé usnadnit a zároveň jej co nejméně omezovat.

3.4.3.1. Základní přehled struktury modulu

Standardní modul se skládá z povinných, volitelných a doplňkových částí, s tím, že doplňkovými částmi se rozumí kód modulu nesouvisející se strukturou programu – pomocné funkce, třídy a metody.

Povinnými částmi modulu jsou definované proměnné:

- `ALGORITHM_CLASS`

  Musí obsahovat jméno třídy poděděné od `AbstractAlgorithm`.

- `ALGORITHM_NAME`

  Musí obsahovat řetězec, který by měl jednoznačně identifikovat algoritmus.

- `ALGORITHM_DESCRIPTION`

  Musí obsahovat řetězec, který by měl popisovat funkcionalitu algoritmu, jeho výhody a nevýhody a obsahovat další relevantní informace k němu.

Z výše uvedeného vyplývá nutnost přítomnosti třídy poděděné od `AbstractAlgorithm`, která musí obsahovat nastavení proměnných a implementaci metod:

- Třídní proměnná `_VERSION` – musí obsahovat celé číslo (iniciálně 1)
- Třídní proměnná `_SUBVERSION` – musí obsahovat celé číslo (iniciálně 0)
- Třídní proměnná `_FILENAME_PREFIX` – musí obsahovat řetězec jednoznačný mezi všemi moduly, který může obsahovat jen znaky, které umožňuje systém použít v názvech souborů (je tedy doporučeno používat jen znaky anglické abecedy a podtržítka)
- Metoda `initAlgInfoData` – slouží k inicializaci dat, informací a nastavení určených k celému modulu, bez ohledu na vybraný element simulace.

  Pro nastavování hodnot vnitřních dat běhu algoritmu se musí využívat slovník `self._simulationData`, který může obsahovat jen standardní typy Pythonu (`int`, `float`, `str`, `unicode`, `tuple`, `list`, `dict`)

  Pro definici zobrazovaných informací a nastavení k celému modulu je nutné využít vnitřní proměnnou `self._simulationInfo`, a to konkrétně její metodu `addToAlgData`.

  Příklad implementace metody algoritmu `initAlgInfoData` obsahuje ***Př. 2***.

```
# Inicializace pomocných proměnných
self._simulationData["elmCount"] = 0

# Vytvoření oddělovače od standardního tlačítka "Nový element"
self._simulationInfo.addToAlgData( \
      lib.module.SimulationInfoSeparator())

# Vytvoření informačního textu
infoText = lib.module.SimulationInfoText( \
      lambda: "Počet procesů: %s" % \
      (self._simulationData["elmCount"], ))

self._simulationInfo.addToAlgData(infoText)
```

Př. 2 – Příklad implementace metody algoritmu „initAlgInfoData“

- Metoda `initSimulationElements` – slouží k inicializaci rozvržení simulace a uchování si později potřebných částí rozvržení v separátní struktuře

  Pro uchování si později potřebných částí rozvržení slouží slovník `self._simulationElements`, který může obsahovat jen instance tříd poděděných od třídy `Element`, které mají nadefinované určité metody. Pro zjednodušení se jedná o instance tříd `EntityArray`, `EntitySet` a `PriorityGroup`.

  Pro samotné vytvoření rozvržení simulace slouží proměnná `self._simulation`, do které se za pomoci metody `addElement` přidají jednotlivé části simulace. Pokud ovšem potřebujete jednoduché rozvržení, je doporučeno jednotlivé části pospojovat

za pomoci instancí tříd jako **Direction**, **EntityGroup** a **EmptyElement**, a vkládat do **self._simulation** jen jeden prvek.

Příklad implementace metody algoritmu **initSimulationElements** obsahuje ***Př. 3***.

```
# Fronta čekajících procesů
self._simulationElements["ready"] = lGC.EntitySet(3, 2, 1.0,
      (1.0, 1.0, 0.0, 1.0), "Připravený", True, True, None, lGC.Entity)

# ...

# Seskupení (vytvoření optického cyklu)
circle = lGC.EntityGroup([top, arrow, bottom],
      self._simulationElements["ready"].inColor,
      self._simulationElements["working"].outColor)

# ...

# Vložení grafu do simulace
self._simulation.addElement(graph, 0, 0, 0)

# Vycentrování simulace
self._simulation.position.setMiddle()
```

Př. 3 – Příklad implementace metody algoritmu „initSimulationElements“

- Metoda **fillElmInfoData** – slouží k inicializaci zobrazovaných informací a nastavení k právě vybranému prvku simulace (např. procesu). Obsah vybraného elementu je metodě předáván jako argument.

  Pro definici zobrazovaných informací a nastavení k vybranému prvku je nutné opět využít vnitřní proměnnou **self._simulationInfo**, ovšem tentokráte za pomoci její metody **addToElmData**.

  Příklad implementace metody algoritmu **fillElmInfoData** obsahuje ***Př. 4***.

```
fData = filling.getData()


# Stav
if "state" in fData:
      stateText = lib.module.SimulationInfoText( \
            lambda: "Stav: %s" % (fData["state"], ))


      self._simulationInfo.addToElmData(stateText)
#endif

# ...
```

```
# Zda blokován
blockedValue = lib.module.SimulationInfoValue(self._parent,
      "Blokován (kroků)",
      lambda: fData.get("blocked", "Není"), fData)

blockedValue.setValChangeFunc(self._local_blockedValDec,
      self._local_blockedValInc)

self._simulationInfo.addToElmData(blockedValue)
```

Př. 4 – Příklad implementace metody algoritmu „fillElmInfoData“

- Metoda **newElement** – slouží k přidání nového prvku do simulace (např. procesu).

  Příklad implementace metody algoritmu **newElement** obsahuje ***Př. 5***.

```
# Vyprázdnění pole pro nové elementy
filling = self._simulationElements["new"].pop()

# Případný přesun vyprázdněného elementu do fronty připravených
if filling:
      filling.getData()["state"] = "Připravený"

      self._simulationElements["ready"].push(filling)

      self._log.insertLine("Proces %s byl naplánován." % \
            (filling.getId(), ), self._COL_READY)
#endif

# Vytvoření nového elementu a jeho inicializace
filling = self.newFilling({"counter" : 0, "state" : "Nový"})

# Zařazení nového elementu do pole pro nové elementy
self._simulationElements["new"].push(filling)

# Záznam do logu simulace o přidání elementu
self._log.insertLine("Proces %s byl vytvořen." % \
      (filling.getId(), ), self._COL_NEW)

# Zvýšení hodnoty počtu aktivních elementů v simulaci
self._simulationData["elmCount"] += 1
```

Př. 5 – Příklad implementace metody algoritmu „newElement“

- Metoda **nextState** – slouží k výpočtu jednoho kroku simulace a aktualizaci jejího stavu. Jedná se o metodu, ve které se děje veškerá logika simulace daného algoritmu. Ve standardních modulech jsou použity dva odlišné přístupy simulací:

  1) Jednodušší přístup s částečně pevným krokem. Akce časově náležející mezi jednotlivé kroky se jen statisticky dopočítávají jako například skončení blokace procesu. Jedná se o jednoduše implementovatelnou variantu, ovšem nevhodnou pro složitější simulace. Příklady takovéto implementace je možné nalézt v modulech **example.py** a **roundRobin.py**.

2) Komplexnější způsob, kdy se dopředu spočte, co by se mělo stát (i se započtením náhodnosti), a pak se vyberou časově nejbližší akce, pro něž se aplikuje daný krok. Jedná se o implementačně i výpočetně náročnější přístup, ale jelikož je simulace vizuálně plynulejší a srozumitelnější, lze jej využít i pro celkem komplikované simulace. Příklady takovéto implementace je možné nalézt v modulech `multilevelFeedback.py` a `priorityQueue.py`.

Mezi volitelné části modulu patří implementace nepovinných metod hlavní třídy algoritmu:

- Metoda `getUserText` – slouží k rozšíření textu zobrazovanému v liště programu. Příklad použití naleznete v `roundRobin.py`.
- Metoda `genLogStatus` – slouží k zobrazení textu v logu simulace při načtení modulu nebo uložené pozice. Rozumné je vypsat hodnoty globálních nastavení simulace apod. Příklad použití naleznete v `roundRobin.py`.

3.4.3.2. Seznam užitečných funkcí a tříd při vytváření modulu

TŘÍDY PRVKŮ PRAVÉHO INFORMAČNÍHO PANELU

Jedná se o třídy poděděné od `lib.module.AbstractSimulationInfoValue` a jsou výhradně určeny jako prvky pravého informačního panelu. Jejich instance je možné do informačního panelu přidávat za pomoci příkazů:

- `self._simulationInfo.addToAlgData(`*<instance prvku>*`)`
- `self._simulationInfo.addToElmData(`*<instance prvku>*`)`

Prvky se vždy vkládají postupně ze shora dolů, a pokud se nevejdou již do výřezu, je možné se k nim prorolovat kolečkem myši.

- `SimulationInfoSeparator` *(lib.module)*

  Vloží do informačního panelu horizontální oddělení o zadané tloušťce čáry.

- `SimulationInfoImage` *(lib.module)*

  Vloží do informačního panelu obrázek (texturu). Velikost obrázku se přepočte dle zadané maximální výšky.

- `SimulationInfoText` *(lib.module)*

  Vloží do informačního panelu text generovaný zadanou funkcí. Velikost textu se bude upravovat podle jeho aktuální délky tak, aby se vešel do šířky informačního boxu a dané výšky. Pokud by ovšem velikost textu měla být příliš malá, zvolí se stanovená minimální čitelná velikost, a tedy text může přetéci vyhrazené místo.

- `SimulationInfoValue` *(lib.module)*

  Vloží do informačního panelu dvouřádkový prvek složený ze zadaného popisku a hodnoty vrácené zadanou funkcí. Zadávaná `data` slouží k předání funkcím měnící hodnotu, které lze nastavit za pomoci třídní metody `setValChangeFunc (valDecFunc, valIncFunc)`. Pokud jsou tyto funkce zadány, objeví se pak kolem hodnoty šipky, kterými ji může uživatel měnit. Argument `parent` musí obsahovat ukazatel na instanci okna – v případě vytváření modulu použijte `self._parent`, kde je tento ukazatel uložen.

- `SimulationInfoButton` *(lib.module)*

  Vloží do informačního panelu textové tlačítko o zadané výšce. Popisek tlačítka se určí z výsledku zadané funkce `nameFunc` a při jeho stisku se provede funkce `actionFunc`, které se předají `data` podobně jako v `SimulationInfoValue`. Do argumentu `parent` taktéž nastavte `self._parent`.

Tyto třídy by měly při sestavování informací a nastavení do pravého informačního panelu stačit. Pokud by tomu tak nebylo, a byl zapotřebí nějaký nový prvek, je vždy možné si jej doprogramovat, je ovšem nutné jej podědit od třídy `AbstractSimulationInfoValue` nebo jejích potomků.

TŘÍDY ELEMENTŮ SIMULACE

Jedná se o třídy poděděné od `Element` a slouží ke grafickému rozvržení simulace. Některé jako např. `EntityArray` slouží k přímému držení prvků simulace (procesů apod.), jiné jako např. `Direction` slouží zcela jen jako doplňkové a definují pouze prostorové vztahy mezi ostatními elementy.

- `EntityArray` *(lib.gComponents)*

  Vloží do simulace kontejnerové pole pro uchovávání prvků simulace. Zásadní vlastností pole je, že má pevný maximální počet vložených prvků. Instanci třídy je možné předat údaje jako délku, barvu, jméno, orientaci, porovnávací funkci, aktualizační funkci, atd. Objekt podporuje vložení a zatřídění prvku, nebo jeho odebrání z konce dle třídění. Dále podporuje třídění, aktualizaci a vymazání všech vložených prvků.

- `EntitySet` *(lib.gComponents)*

  Vloží do simulace kontejnerové pole bez omezení počtu prvků – množinu. Instanci třídy je možné předat údaje jako délku, barvu, jméno, orientaci, porovnávací funkci, aktualizační funkci, atd. Objekt podporuje vložení a zatřídění prvku, nebo jeho odebrání z konce dle třídění. Dále podporuje třídění, aktualizaci a vymazání všech vložených prvků.

- `PriorityGroup` *(lib.gAdvancedComponents)*

  Vloží do simulace specializovaný kontejner reprezentující prioritní frontu (vnitřně se jedná o skupinu množin propojených šipkami). Prioritní fronty fungují dle očekávání jen s prvky `AvancedEntity`; standardní prvky se považují za prvky s nejnižší prioritou. Instanci třídy je možné předat údaje jako délku, počet prioritních front, seznam pojmenování jednotlivých front, vstupní a výstupní barvu, orientaci, porovnávací funkci, aktualizační funkci, atd. Objekt podporuje vložení a zatřídění prvku, nebo jeho odebrání z konce dle třídění. Dále podporuje třídění, aktualizaci a vymazání všech vložených prvků.

- `EntityGroup` *(lib.gComponents)*

  Propojí několik elementů do skupiny. Instanci třídy je možné předat údaje jako pole elementů, vstupní a výstupní barvu, orientaci, atd. Objekt podporuje přepočet rozložení obsažených entit, jejich aktualizaci, apod.

- `Direction` *(lib.gComponents)*

  Propojí právě dva elementy a vizuálně mezi nimi definuje vztah směru. Instanci třídy je možné předat údaje jako první element, druhý element, směr, délku, zda se má použít šipka, atd.

Za pomoci těchto tříd by měl být každý schopen sestavit rozvržení simulace dle svých představ. V případě nutnosti je možné si doprogramovat svou vlastní třídu, je ovšem silně doporučováno ji založit na třídách již existujících a poděděním doplnit jen nutnou funkcionalitu.

ROZŠÍŘENÁ ENTITA

Rozšířená entita (`AdvancedEntity` a výplň `AdvancedEntityFilling`) navíc od entity standardní umožňuje jednoduše pracovat s prioritou a se statusem, které také při svém zobrazení vizualizuje. Obě třídy (`Priority` i `Status`) podporují získání své instance z prvku za pomoci třídní metody `get`. Použití je pak zcela jednoduché, a to například příkazem: `prio = lGAC.Priority.get(filling)`, kde `lGAC` je alias pro `lib.gAdvancedComponents` a `filling` je instance třídy `AdvancedEntityFilling`. Pokud by se této metodě dostala jako argument instance neregistrované třídy, jako např. `EntityFilling`, metoda vrátí `None`.

3.5. Adresářová struktura aplikace

HLAVNÍ SLOŽKA `ALG-SIMULATOR`:

Složka obsahuje soubory uvedené v ***Tab. 3***.

Název souboru	Systém	Funkce
`alg-simulator.bat`	Windows	Spouštěč programu ve Windows
`alg-simulator.sh`	Linux	Spouštěč programu v Linuxu (jen po instalaci)
`configuration.cfg`	Všechny	Konfigurace programu
`doxygen.cfg`	Všechny	Konfigurace generátoru automatické nápovědy. (`doxygen doxygen.cfg`)
`install.bat`	Windows	Batch soubor k vytvoření instalace ve Windows
`Makefile`	Linux	Makefile soubor k vytvoření balíčku v Linuxu (za pomoci příkazu `debuild`)
`run.py`	Všechny	Hlavní program (`python run.py configuration.cfg`)
`tasklist.txt`	Všechny	Soubor s úkoly k zapracování

Tab. 3 – Tabulka přehledu souborů z hlavní složky aplikace s jejich popisy

PODSLOŽKA `DATA`:

Složka obsahuje ikony pro samotný program a archív `icons.zip` s ikonami sloužícími jako reprezentace prvků simulace (např. procesů). V případě záměny těchto ikon za jiné, je možné stejným způsobem vytvořený archív umístit do této cesty a nadefinovat jej v konfiguraci programu `configuration.cfg`.

PODSLOŽKA **DEBIAN**:

Tato složka obsahuje skripty a nastavení určené k vytvoření balíčků pro distribuce Linuxu založené na Debianu (např. Ubuntu). [19, 20, 21]

PODSLOŽKA **DOC**:

Do této složky generuje Doxygen automatickou nápovědu ke zdrojovým kódům. Při standardním nastavení bude tato složka obsahovat složku **html**, ve které bude mezi jinými vytvořena hlavní stránka automaticky generované nápovědy **index.html**.

PODSLOŽKA **DOCS**:

Tato složka obsahuje materiály týkající se problematiky algoritmů operačních systémů a manuály k programu.

PODSLOŽKA **INSTALL**:

V této složce a jejích podsložkách jsou uloženy prerekvizity pro instalaci pod Windows. Součástí je také batch soubor pro instalaci prerekvizit.

PODSLOŽKA **LIB**:

Tato složka obsahuje veškeré knihovny programu. Pokud by se program rozšiřoval o nějakou knihovnu, měla by být přidána sem.

PODSLOŽKA **LOGS**:

Do této složky se standardně ukládají logy o běhu programu. V případě, že program nebo nějaký modul selže, v této složce naleznete log s bližšími informacemi k tomuto selhání.

PODSLOŽKA **MODULES**:

Z této složky se načítají veškeré moduly dostupné v programu. V případě vytváření nového modulu, toto je místo, kde musí být založen.

PODSLOŽKA **PATCHES**:

Tato složka obsahuje skript a soubory nutné k opravě chyby 471 knihovny Pyglet verze 1.1.4. Toto je zapotřebí při vytváření balíčku pod Linux, de-facto při jeho instalaci. [22, 23]

PODSLOŽKA **SAVES**:

Do této složky se ukládají pozice simulací, které je možné následně do programu opět načíst. Uložené pozice se nezahrnují do instalačních balíčků programu, po instalaci je tedy program bez těchto pozic.

PODSLOŽKA **`WIN`**:

Do této složky nic neukládejte, při spuštění **`install.bat`** se vždy přemazává a kopíruje se do ní aktuální instance programu pro přípravu instalace pod Windows.

4. Implementované moduly

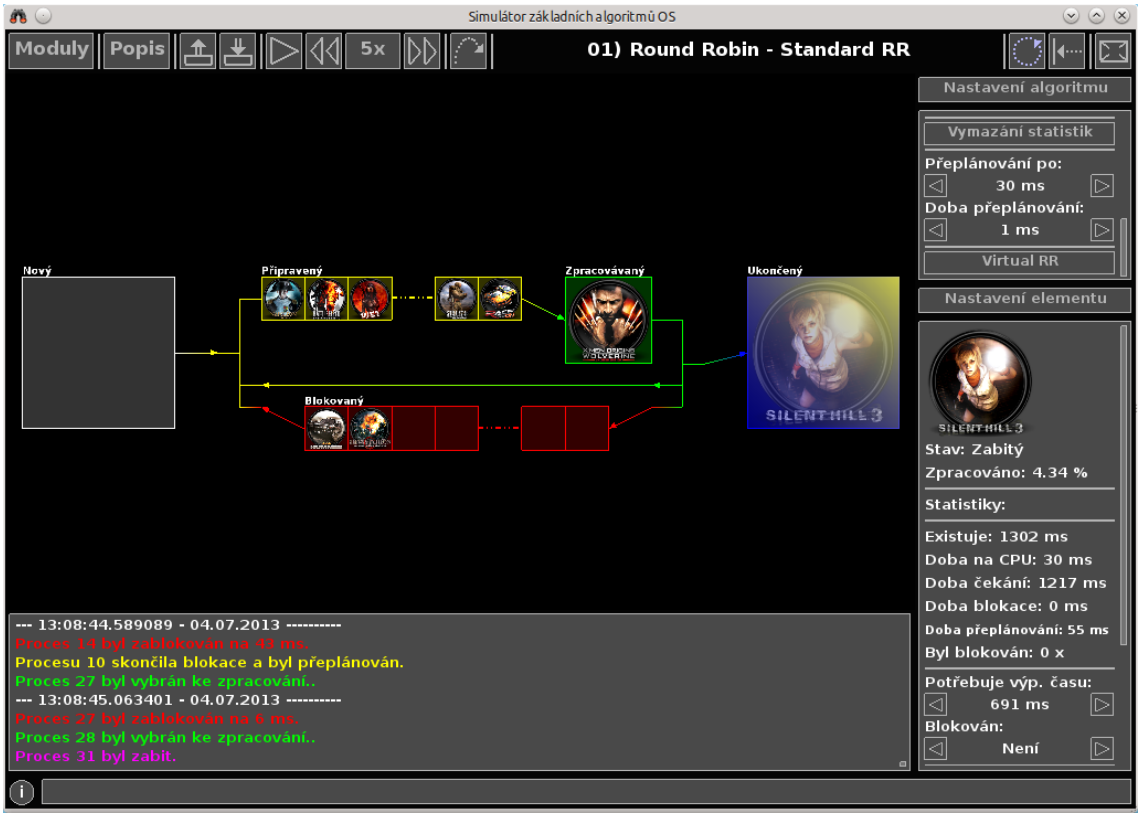
Tato kapitola se zabývá třemi implementovanými algoritmy. U každého je pak uveden jeho stručný popis, rozebrány veškeré možnosti jeho nastavení a vysvětleny veškeré zobrazované informace tak, aby uživatel měl kompletní přehled o funkcionalitě a všech možnostech daného modulu. Nakonec je u každého modulu popsána jeho nejzákladnější implementační logika.

4.1. Cyklická obsluha (Round Robin)

Jedná se o jeden z nejstarších a nejpoužívanějších algoritmů (v různých variantách) a byl navržen obzvláště pro systémy se sdílením času. Algoritmus přiřadí každému procesu časový interval (tzv. časové kvantum), po které může běžet. Pokud proces běží ještě na konci kvanta, je provedena preempce, tedy přerušení právě vykonávané úlohy, a je naplánován a spuštěn další připravený proces. Pokud proces skončil nebo se zablokoval ještě před spotřebováním časového kvanta, je také naplánován a spuštěn další připravený proces. Ve verzi algoritmu označované jako virtuální cyklická obsluha se procesy po skončení blokace upřednostní před ostatními.

Modul s algoritmem cyklické obsluhy je umístěn v podsložce **`modules`** a je pojmenován **`roundRobin.py`**.

Vizualizaci implementovaného algoritmu je možné vidět v příkladu stavu simulace na ***Obr. 8***.



Obr. 8 – Příklad stavu simulace algoritmu cyklické obsluhy

4.1.1. Nastavení modulu

V horní části pravého panelu se nalézá:

- Tlačítko **Přidat nový element** – přidá proces s náhodně určenými parametry do simulace
- Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
- Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
- Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
  - **Existence** – průměrná existence procesu
  - **Na CPU** – průměrně strávený čas procesu na procesoru
  - **Čekání** – průměrná hodnota čekání procesu na procesor
  - **Blokace** – průměrná délka času procesu stráveného blokací
  - **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
  - **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)
  - **Poč. blokací** – průměrný počet blokací procesu
- Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
- Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
- Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)
- Tlačítko **Virtual RR/Standard RR** – přepíná mezi variantami algoritmu (Virtual RR upřednostňuje ve frontě připravených procesy přicházející z blokace, Standard RR nikoliv)

4.1.2. Nastavení elementu

V dolní části pravého panelu se při výběru procesu nalézá:

- **Ikona procesu** – napomáhá rozpoznání označeného procesu
- Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
- Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)

- Informace **Statistiky** – nadpis níže uvedených statistik
  - **Existuje** – čas existence procesu
  - **Doba na CPU** – čas procesu strávený na procesoru
  - **Doba čekání** – čas procesu strávený čekáním na procesor
  - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, uspáním, apod.)
  - **Doba přeplánování** – čas procesu strávený přeplánováváním
  - **Byl blokován** – počet blokací procesu
- Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
- Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)
- Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
- Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Tlačítko **Zabít proces** – umožní uživateli vynutit ukončení procesu, ten se pak okamžitě přesune do sekce ukončených procesů

4.1.3. Logika simulace

V této simulaci odpovídá jeden krok maximálně době jednoho kvanta a minimálně době do blokace nebo ukončení běžícího procesu. V každém kroku se zkontrolují a přepočtou procesy ve všech frontách (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený). V rámci jednoho kroku se mohou v simulaci stát taktéž události, které nemusí být na první pohled zřejmé; jsou ovšem vždy zaznamenány v logu simulace – je tedy dobrým zvykem log během simulace sledovat. Jednou z takovýchto událostí může být například to, že při ukončení blokace procesu a prázdné frontě připravených procesů, se vizuálně doteď blokovaný proces přesune rovnou na procesor; z logu je ovšem patrné, že tento dojem je zavádějící, a že proces byl napřed řádně naplánován do fronty připravených procesů, a následně z ní byl vybrán ke zpracování procesorem.

4.2. Víceúrovňová zpětná vazba (Multi-level feedback queue)

Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. Pro každou prioritu existuje jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty.

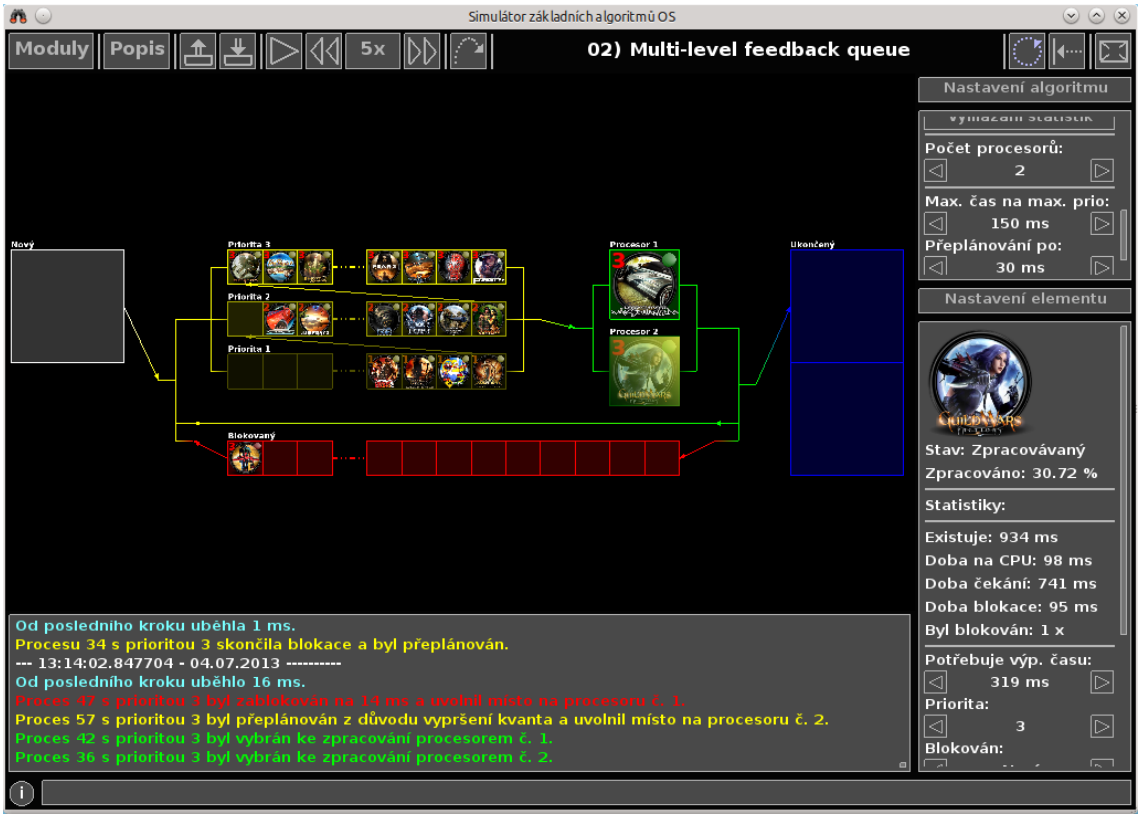
Každý proces má přidělenou omezenou výpočetní dobu, kterou může strávit na dané prioritní úrovni, čímž se zaručí upřednostnění I/O vázaných procesů před

výpočetně vázanými. Čas přidělený nižší prioritní úrovni je zpravidla dvojnásobný než čas přidělený prioritní úrovni o jedna vyšší.

Tento modul pro názornost pracuje s třemi prioritními frontami (v praxi jich bývá více) a tedy prioritou procesů v rozmezí 1 až 3, kde priorita 3 je brána jako nejvyšší.

Modul s algoritmem víceúrovňové zpětné vazby je umístěn v podsložce `modules` a je pojmenován `multilevelFeedback.py`.

Vizualizaci implementovaného algoritmu je možné vidět v příkladu stavu simulace na ***Obr. 9***.



Obr. 9 – Příklad stavu simulace algoritmu víceúrovňové zpětné vazby

4.2.1. Nastavení modulu

V horní části pravého panelu se nalézá:

- Tlačítko **Přidat nový element** – přidá do simulace proces s náhodně určenými parametry
- Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
- Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
- Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
  - **Existence** – průměrná existence procesu
  - **Na CPU** – průměrně strávený čas procesu na procesoru
  - **Čekání** – průměrná hodnota čekání procesu na procesor

- **Blokace** – průměrná délka času procesu stráveného blokací
- **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
- **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)
- **Poč. blokací** – průměrný počet blokací procesu

- Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
- Parametr **Počet procesorů** – určuje počet procesorů použitých v simulaci
- Parametr **Max. čas na max. prio.** – určuje množství výpočetního času v milisekundách, které může maximálně proces na nejvyšší prioritní úrovni strávit; poté se jeho priorita poníží. Hodnota parametru slouží jako základ pro výpočet množství výpočetního času pro ostatní prioritní úrovně.
- Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
- Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)

4.2.2. Nastavení elementu

V dolní části pravého panelu se při výběru procesu nalézá:

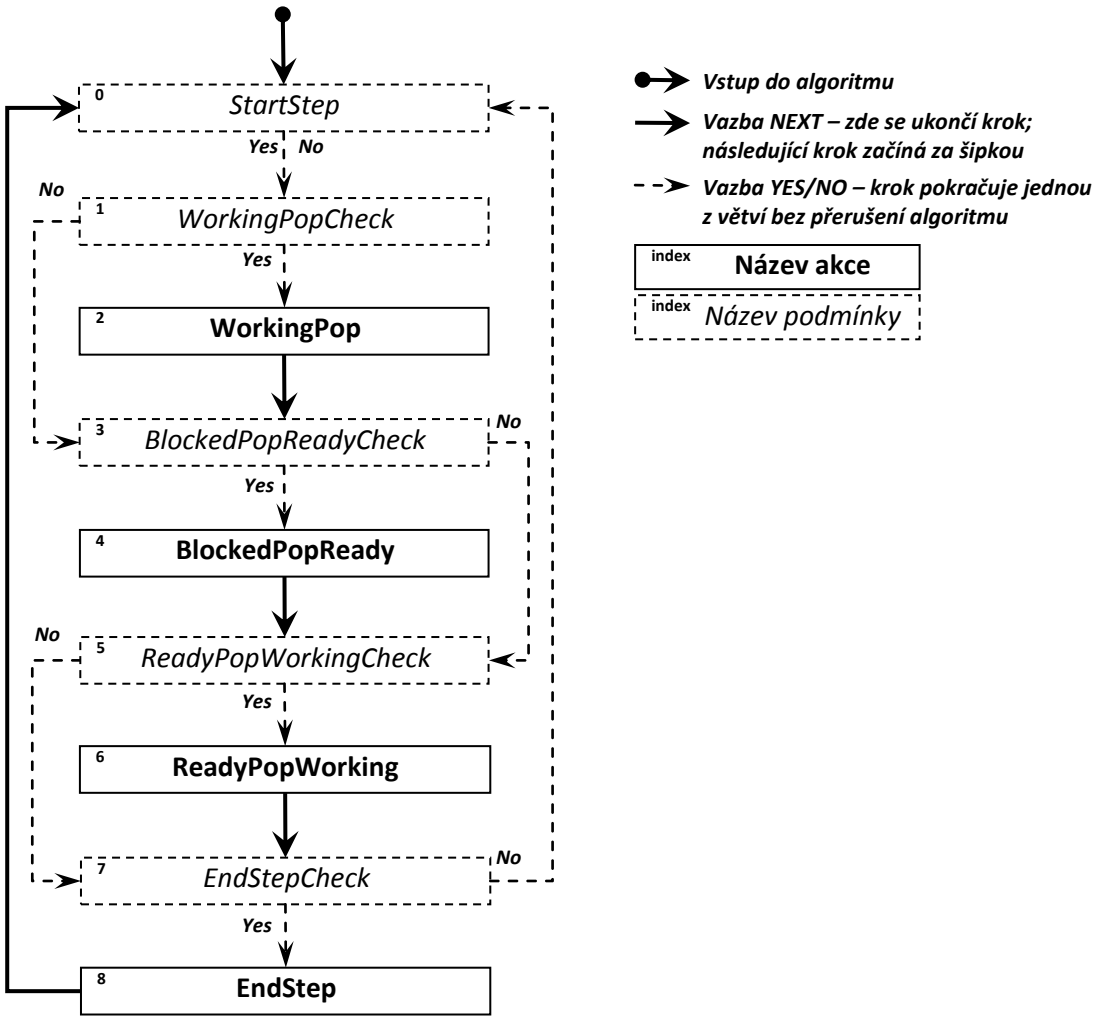
- **Ikona procesu** – napomáhá rozpoznání označeného procesu
- Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
- Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)
- Informace **Statistiky** – nadpis níže uvedených statistik
  - **Existuje** – čas existence procesu
  - **Doba na CPU** – čas procesu strávený na procesoru
  - **Doba čekání** – čas procesu strávený čekáním na procesor
  - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, atp.)
  - **Byl blokován** – počet blokací procesu
- Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
- Parametr **Priorita** – určuje hodnotu priority procesu
- Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)

- Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
- Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Tlačítko **Zabít proces** – umožní uživateli vynutit ukončení procesu, ten se pak okamžitě přesune do sekce ukončených procesů

4.2.3. Logika simulace

V této simulaci odpovídá jeden krok maximálně době jednoho výpočetního kvanta a minimálně době jedné milisekundy. Aktuální čas kroku je vždy určen jako čas do nejbližší akce – změny stavu jakéhokoliv procesu. Posloupnost jednotlivých kroků znázorňuje ***Graf 12*** a popis jejich funkcí pak ***Tab. 4***.

Pokud srovnáme kroky tohoto algoritmu například s implementací Cyklické obsluhy popsanou výše, zjistíme, že jeden krok v Cyklické obsluze odpovídá celému cyklu kroků zde, a že tento modul simuluje daný algoritmus mnohem podrobněji a názorněji.



Graf 12 – Posloupnost jednotlivých kroků při simulaci algoritmu víceúrovňové zpětné vazby

Název	Typ	Popis
StartStep	Podmínka	Počátek cyklu – zaloguje oddělovač a nastaví pomocné proměnné.
WorkingPopCheck	Podmínka	Zjistí, zda se má nějaký blokovaný proces přeplánovat, zpracovávaný proces ukončit, přeplánovat, nebo zablokovat. Vrací seznam nejbližších akcí nebo None.
WorkingPop	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.
BlockedPopReadyCheck	Podmínka	Aktualizuje statistiky blokovaných procesů a zjistí, zda nějakému procesu skončila blokace.
BlockedPopReady	Akce	Přesune všechny procesy, kterým skončila blokace do fronty připravených.
ReadyPopWorkingCheck	Podmínka	Aktualizuje statistiky připravených procesů a zjistí, zda je nějaký z procesorů prázdný a vrátí jejich seznam nebo None.
ReadyPopWorking	Akce	Pro každý prázdný procesor z přijatého seznamu vybere nejprioritnější čekající proces a přiřadí mu jej.
EndStepCheck	Podmínka	Zjistí, zda se v daném cyklu něco událo; pokud ne, přejde na „EndStep“, aby nedošlo k nekonečné smyčce.
EndStep	Akce	Slouží jako zarážka – prázdná metoda.

Tab. 4 – Tabulka s popisem jednotlivých kroků při simulaci algoritmu víceúrovňové zpětné vazby

4.3. Víceúrovňová prioritní fronta (Multi-level priority queue)

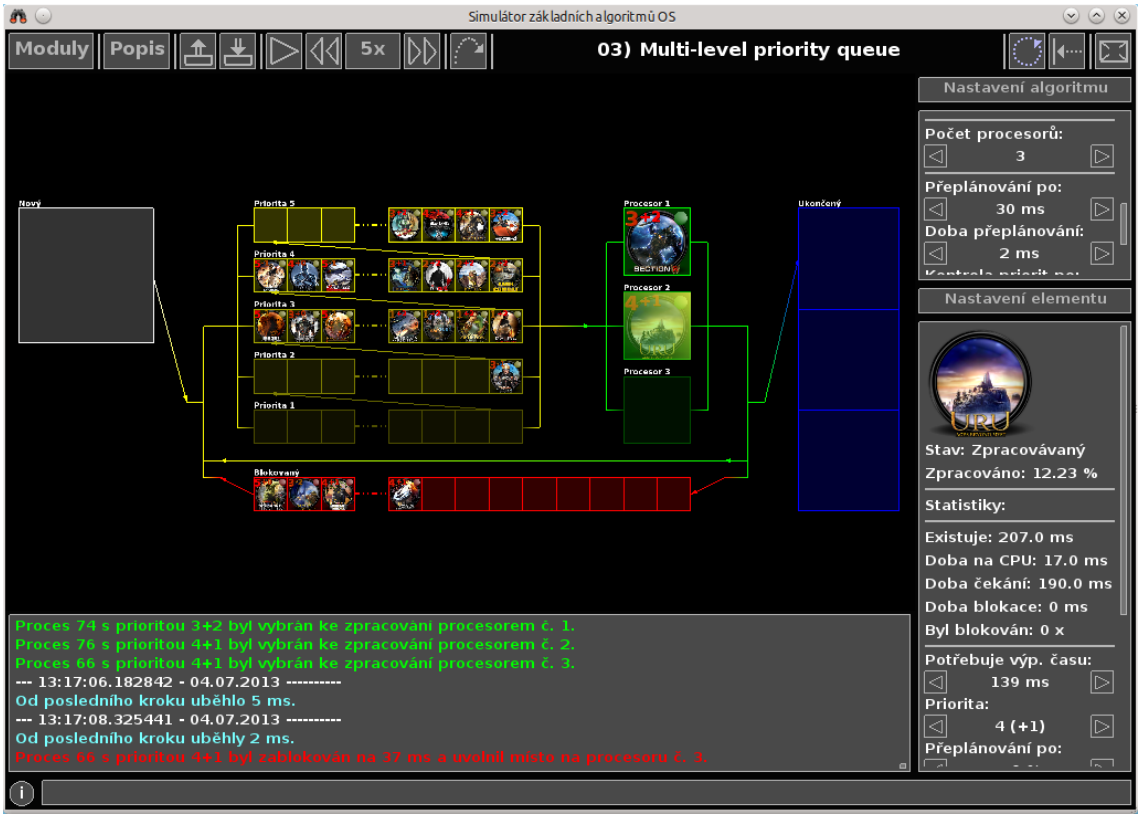
Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. V praxi pak existuje pro každou prioritu jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty.

Každý proces disponuje statickou prioritou, která je mu nastavena při jeho vytvoření a zůstává po celou dobu jeho existence neměnná, a dynamickou prioritou, která zaručuje, že i procesy s nižší statickou prioritou se občas dostanou na procesor (CPU) a nenastalo takzvané jejich vyhladovění. Tato dynamická priorita se čas od času pro každý proces přepočítá, a to na základě jím strávené doby na CPU, čímž se také upřednostní I/O vázané procesy.

Tento modul pro názornost pracuje s pěti prioritními frontami (v praxi jich bývá 30 až 50) a tedy statickou prioritou procesů v rozmezí 1 až 5, kde priorita 5 je brána jako nejvyšší. Dynamická priorita se může pohybovat v rozmezí -2 až +2, a výsledná priorita procesu se pak vždy skládá jako součet priority statické a dynamické.

Modul s algoritmem víceúrovňové prioritní fronty je umístěn v podsložce **`modules`** a je pojmenován **`priorityQueue.py`**.

Vizualizaci implementovaného algoritmu je možné vidět v příkladu stavu simulace na ***Obr. 10***.



Obr. 10 – Příklad stavu simulace algoritmu víceúrovňové prioritní fronty

4.3.1. Nastavení modulu

V horní části pravého panelu se nalézá:

- Tlačítko **Přidat nový element** – přidá proces s náhodně určenými parametry do simulace
- Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
- Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
- Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
  - **Existence** – průměrná existence procesu
  - **Na CPU** – průměrně strávený čas procesu na procesoru
  - **Čekání** – průměrná hodnota čekání procesu na procesor
  - **Blokace** – průměrná délka času procesu stráveného blokací
  - **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
  - **Přep. prio** – průměrná hodnota času procesu strávená během přepočtu priorit
  - **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)

- **Poč. blokací** – průměrný počet blokací procesu

- Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
- Parametr **Počet procesorů** – určuje počet procesorů použitých v simulaci
- Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
- Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)
- Parametr **Kontrola priorit po** – určuje dobu, po které se provádí kontrola priorit u běžících procesů (a případná následná záměna za proces s vyšší prioritou)
- Parametr **Přep. priorit každou** – určuje, kolikátou každou kontrolu se provede přepočet priorit všech živých procesů
- Parametr **Doba přepočtu priorit** – určuje dobu strávenou jedním přepočtem priorit

4.3.2. Nastavení elementu

V dolní části pravého panelu se při výběru procesu nalézá:

- **Ikona procesu** – napomáhá rozpoznání označeného procesu
- Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
- Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)
- Informace **Statistiky** – nadpis níže uvedených statistik
  - **Existuje** – čas existence procesu
  - **Doba na CPU** – čas procesu strávený na procesoru
  - **Doba čekání** – čas procesu strávený čekáním na procesor
  - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, uspáním, apod.)
  - **Byl blokován** – počet blokací procesu
- Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
- Parametr **Priorita** – určuje hodnotu statické priority procesu (v praxi je změna statické priority procesu umožněna administrátorovi systému většinou jen v rozmezí -3 až +3) [6]
- Parametr **Přeplánování po** – určuje procento změny standardního výpočetního kvanta pro daný proces od -90% po +100% (v praxi je možnost úpravy výpočetního kvanta procesu omezena na jeho zařazení do jedné ze dvou skupin a to „programu“

(kratší kvantum; standardní možnost pro uživatelské procesy) nebo „démonu“ (delší kvantum; většinou využívají systémové procesy))

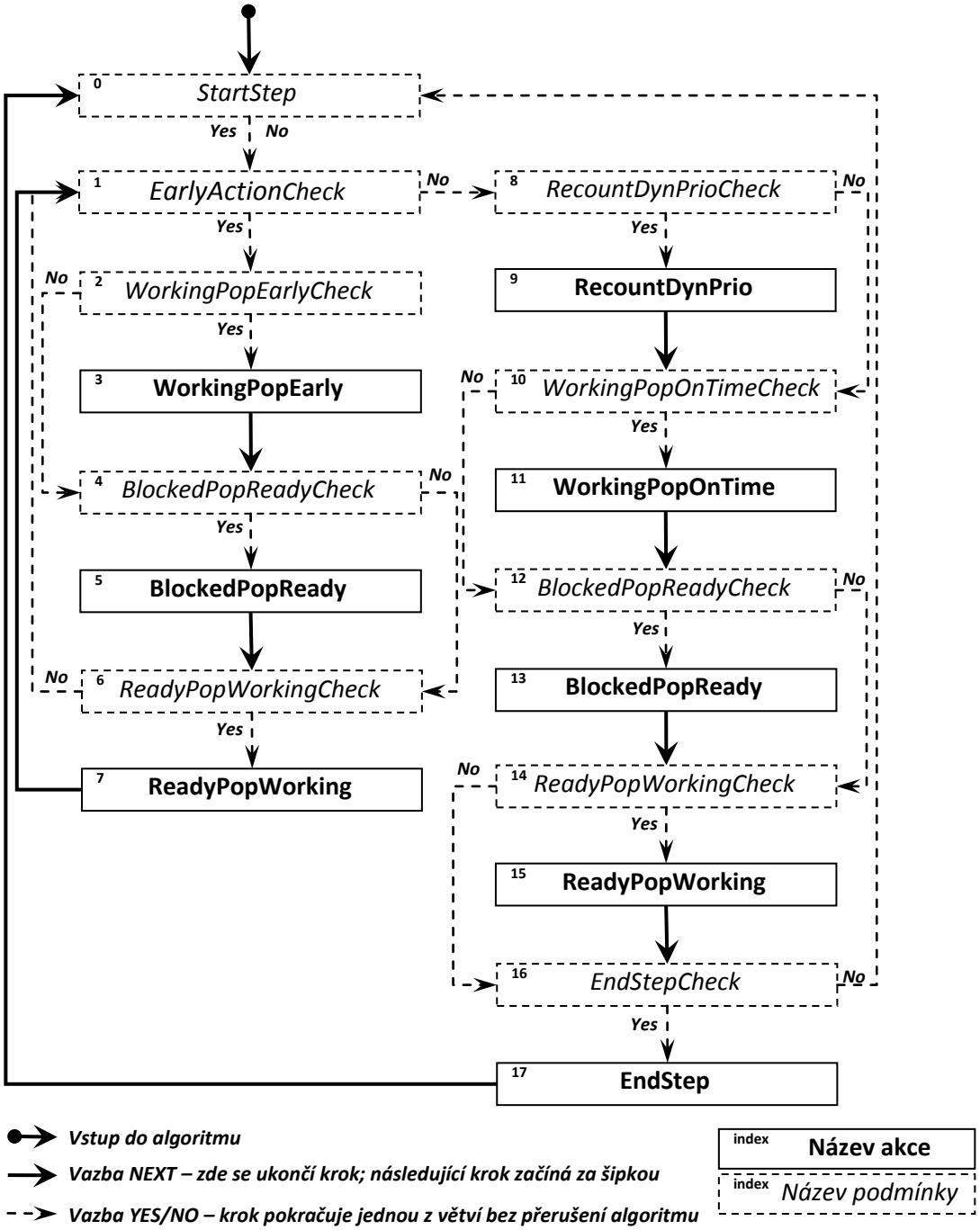
- Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)
- Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
- Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
- Tlačítko **Zabít proces** – umožní uživateli vynutit ukončení procesu, ten se pak okamžitě přesune do sekce ukončených procesů

4.3.3. Logika simulace

V této simulaci odpovídá jeden krok maximálně době jedné kontroly priorit (ta se provádí standardně několikrát za kvantum) a minimálně době jedné milisekundy. Aktuální čas kroku je vždy určen jako čas do nejbližší akce. Akcí pak je změna stavu jakéhokoliv procesu, nebo kontrola priorit.

Posloupnost jednotlivých kroků je rozdělena do dvou základních logických částí a to do obsluhy předčasných (rozumějme dřívějších než je plánovaná kontrola) akcí a obsluhy akcí při plánované kontrole. Hlavním rozdílem je, že při plánované kontrole se navíc kontroluje dostatečnost priorit právě běžících procesů – v této části tedy může proces být nucen ukončit svůj výpočet na základě nižší priority, než má nějaký z připravených procesů – a pokud se jedná o X-tou kontrolu (lze nastavit; standardně každou 10. kontrolu), provádí se v této části přepočet priorit všech živých procesů. Posloupnost jednotlivých kroků znázorňuje ***Graf 13*** a popis jejich funkcí pak ***Tab. 5***.

Pokud srovnáme kroky tohoto algoritmu například s implementací Cyklické obsluhy popsanou výše, zjistíme, že jeden krok v Cyklické obsluze odpovídá celému cyklu kroků zde, a že tento modul simuluje daný algoritmus mnohem podrobněji a názorněji. Tento přístup byl zvolen, jelikož se scéna v případě prvního přístupu krokování stávala vzhledem ke komplexnosti prováděných akcí nepřehlednou.



Graf 13 – Posloupnost jednotlivých kroků při simulaci algoritmu víceúrovňové prioritní fronty

Název	Typ	Popis
StartStep	Podmínka	Počátek cyklu – zaloguje oddělovač a nastaví pomocné proměnné.
EarlyActionCheck	Podmínka	Zjistí, zda se má nějaký blokovaný proces přeplánovat, zpracovávaný proces ukončit, přeplánovat na základě vypršení kvanta nebo zablokovat dříve, než uběhla perioda kontroly (standardně 5 ms).
WorkingPopEarlyCheck	Podmínka	Na základě přijatých dat zjistí, zda se má nějaký zpracovávaný proces ukončit, přeplánovat nebo zablokovat. Předává přijatá data nebo None.
WorkingPopEarly	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.
WorkingPopOnTimeCheck	Podmínka	Zjistí, zda se má nějaký zpracovávaný proces ukončit, přeplánovat na základě vypršení kvanta nebo nedostatečné priority nebo zablokovat při skončení periody kontroly (standardně 5 ms).
WorkingPopOnTime	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.
BlockedPopReadyCheck	Podmínka	Aktualizuje statistiky blokovaných procesů a zjistí, zda nějakému procesu skončila blokace.
BlockedPopReady	Akce	Přesune všechny procesy, kterým skončila blokace do fronty připravených.
ReadyPopWorkingCheck	Podmínka	Aktualizuje statistiky připravených procesů a zjistí, zda je nějaký z procesorů prázdný a vrátí jejich seznam nebo None.
ReadyPopWorking	Akce	Pro každý prázdný procesor z přijatého seznamu vybere nejprioritnější čekající proces a přiřadí mu jej.
RecountDynPrioCheck	Podmínka	Zjistí, zda již nastal čas pro přepočet dynamických priorit procesů.
RecountDynPrio	Akce	Přepočte dynamické priority na základě statistických hodnot procesu tak, aby se všechny procesy občas dostaly na procesor.
EndStepCheck	Podmínka	Zjistí, zda se v daném cyklu něco událo; pokud ne, přejde na „EndStep“, aby nedošlo k nekonečné smyčce.
EndStep	Akce	Slouží jako zarážka – prázdná metoda.

Tab. 5 – Tabulka s popisem jednotlivých kroků při simulaci algoritmu víceúrovňové prioritní fronty

5. Závěr

Závěrečná kapitola souhrnně popisuje obsah bakalářské práce a zabývá se hodnocením aplikace, která je její součástí. Mezi jinými jsou pak v rámci závěru navrhována její možná vylepšení.

5.1. Shrnutí obsahu práce

V rámci práce byly prostudovány algoritmy používané v operačních systémech, a po domluvě se zadavatelem z nich vybrány tři, jež byly následně implementovány. Jedná se o algoritmy cyklická obsluha (round-robin), víceúrovňová zpětná vazba (multi-level feedback queue) a víceúrovňová prioritní fronta (multi-level priority queue).

Práce obsahuje rozvahu nad výběrem nejvhodnějších technologií a prostředků pro vlastní realizaci aplikace. Zabývá se výběrem vhodných vizuálních a datových reprezentací včetně modelu zaručujícího jednoduchou rozšiřitelnost o další simulace algoritmů.

Dále je popsána realizace aplikace, její rozdělení na samostatné implementační celky, rozebrány některé použité datové struktury a uvedeny příklady provázání jednotlivých tříd a podobně. Všechny implementované moduly jsou v práci podrobně rozebrány, jsou u nich vysvětlena všechna specifická nastavení simulace, a je u nich nastíněna implementační logika přepočtu mezi jednotlivými stavy simulace.

V rámci vývoje byla také zapracována podpora balíčkování pod systémy Linux (balíčkování typu Debian) a vytvořeny skripty pro jednoduché vytvoření samorozbalovacího archivu pod systémy Windows. Běh aplikace byl pak testován pod systémy Ubuntu 12.10 desktop 64bit, Windows XP 32bit a Windows 7 64bit.

Jako samostatná příloha byla sepsána podrobná uživatelská příručka popisující jak ovládání aplikace, tak i vnitřní principy fungování implementovaných algoritmů. Dále je pak součástí příloh také takzvaná programátorská příručka obsahující návod, jak do aplikace přidat nový modul s algoritmem.

5.2. Popis a zhodnocení výsledné aplikace

Výsledkem bakalářské práce je aplikace, která je dle požadavků spustitelná jak pod systémem Linux, tak pod systémem Windows. Vzhledem k přenositelnosti použitých technologií a knihoven by ji mělo být možné zprovoznit i pod systémem Mac OS, což ovšem nebylo testováno.

Aplikace podporuje modulární načítání jednotlivých algoritmů k simulaci, s tím, že při práci se simulací algoritmu je uživateli umožněna nejen regulace její rychlosti přehrávání, ale i změna jejích parametrů a parametrů simulovaných prvků. Simulace pak samozřejmě na takovéto změny reaguje a uživatel tak má možnost důkladně prozkoumat i méně obvyklé či zcela extrémní stavy, do kterých se algoritmus může dostat. Všechny implementované algoritmy také zaznamenávají veškeré své akce v textové podobě a uživatel tak má naprostý přehled jak o změnách probíhajících, tak o těch již proběhlých.

Aby program mohl zastávat funkci pomůcky při výuce, obsahuje možnosti uložení a opětovného načtení stavu simulovaného algoritmu, dále možnost přechodu od

aktuálního do následujícího stavu na vyžádání uživatele, návrat ke stavu načtení uložené pozice, nebo návrat do výchozího stavu algoritmu. Všechny tyto vlastnosti by měly co nejvíce usnadnit prezentaci chování simulovaného algoritmu.

Program tak plní dostatečně svůj účel, tak jak je, mohl by ovšem být vylepšen například animovaným přesunem prvků z jedné pozice do druhé (nyní vždy jen prvek zmizí a objeví se jinde), nebo možností uložení právě probíhající simulace jako videa, případně pro lepší uživatelský zážitek by mohl být ozvučen. Za zmínku také stojí vytvoření plnohodnotného instalátoru programu pod Windows (nyní je distribuován jen jako samorozbalovací archív, který vyžaduje ruční instalaci prerekvizit).

Přehled zkratek

CPU central processing unit neboli procesor

FIFO first-in, first-out / “kdo dřív přijde, ten dřív mele”

GSI graphic simulation interface / grafické rozhraní simulace

GUI graphic user interface / grafické uživatelské rozhraní

I/O input/output / vstupně-výstupní

OS operating system / operační systém

Seznamy

Grafy

Obrázky

Příklady

Tabulky

Literatura

[1] **Haldar, Sibsankar a Arvind, Alex A.** Operating Systems. New Delhi : Dorling Kindersley (India) Pvt. Ltd., 2010, stránky 1-3,111-119,125-127. ISBN: 978-81-317-3022-5

[2] **Stuart, Brian L.** Principles of Operating Systems: Design & Applications. USA : Course Technology, a division of Thomson Learning, Inc., 2009, stránky 90-92, 211-212. ISBN: 1-4188-3769-5

[3] **Debasis, Samanta.** Classic Data Structures. 2nd. New Delhi : PHI Learning Private Limited, 2009, pp. 185-187. ISBN: 978-81-203-3731-2

[4] **Krishnamoorthy, R. a Kumaravel, G. Indirani.** Data Structures Using C. New Delhi : Tata McGraw-Hill Publishing Company Limited, 2008, stránky 153-154, 251. ISBN-13: 978-0-07-066919-2, ISBN-10: 0-07-066919-8

[5] **Hughes, Cameron a Hughes, Tracey.** Parallel and Distributed Programming Using C++. USA, Canada : Pearson Education, Inc., 2004, stránky 46-47. ISBN: 0-13-101376-9

[6] **Malina, Patrik.** Procesy a jejich „běh“ ve Windows. [Online] Gopas. - Březen 2013. - *http://www.wug.cz/zaznamy/32-Procesy-a-jejich-beh-ve-Windows*.

[7] **Dhamdhere, D. M.** Operating Systems: A Concept-based Approach. 2nd. New Delhi : Tata McGraw-Hill Publishing Company Limited, 2006, p. 167. ISBN: 0-07-061194-7

[8] **Kwong, Yuen Chung.** Annual Review of Scalable Computing. Singapore : Singapore University Press, World Scientific Publishing Co., Pte., Ltd., 2001, Sv. 3, stránky 71-73. ISBN: 981-02-4579-3

[9] **Remesh, Sri. V.** Principles of Operating Systems. New Delhi : University Science Press, 2010, stránky 90-95. ISBN: 978-93-80386-17-1

[10] **Rangan, C. Pandu, Raman, V. a Ramanujam, R.** Foundations of Software Technology and Theoretical Computer Science: 19th conference, Chennai, India, December 1999. Springer : autor neznámý, 1999, stránky 84-86. ISBN: 3-540-66836-5

[11] **Sharma, Vivek, Varshney, Manish a Sharma, Shantanu.** Design and Implementation of Operating System. New Delhi : Laxami Publications Pvt. Ltd., 2010, stránky 294-304. ISBN: 978-93-80386-41-6

[12] **Dhotre, I. A.** Operating Systems. India : Technical Publications Pune, 2009, 7, pp. 19-21. ISBN: 978-81-8431-644-5

[13] **Sharma, D. P.** Foundation of Operating Systems. New Delhi : EXCEL BOOKS, 2008, str. 117. ISBN: 978-81-7446-626-6

[14] **Wright, Byron a Plesniarski, Leon.** MCTS Guide to Microsoft Windows 7: Exam #70-680. USA : Course Technology, Cangage Learning, 2011, str. 189. ISBN-13: 978-1-1113-0977-0, ISBN-10: 1-1113-0977-9

[15] **Deoppner, Thomas W.** Operating Systems in Depth: Design and programming. USA : MPS Limited, A Macmillan Company, Haminton printing Company, 2010, pp. 219-223. ISBN: 978-0-471-68723-8

[16] Dokumentace Python2.7. [Online]. - Březen 2013. - *http://docs.python.org/release/2.7.5/*.

[17] Dokumentace OpenGL. [Online]. - Duben 2013. - *http://www.opengl.org/sdk/docs/man/*.

[18] Dokumentace Pyglet. [Online]. - Duben 2013. - *http://www.pyglet.org/doc/programming_guide/*.

[19] Balíčkování Python aplikace pro Debian. [Online]. - Květen 2013. - *http://savetheions.com/2010/01/20/packaging-python-applicationsmodules-for-debian/*.

[20] Příručka udržovatele balíčku pro Debian. [Online]. - Květen 2013. - *http://www.debian.org/doc/manuals/maint-guide/*.

[21] Úvod do balíčkování pod Debianem. [Online]. - Květen 2013. - *http://wiki.debian.org/IntroDebianPackaging*.

[22] Popis chyby 471 knihovny Pyglet. *Google.* [Online]. - Duben 2013. - *http://code.google.com/p/pyglet/issues/detail?id=471*.

[23] Popis řešení chyby 471 knihovny Pyglet. *Google.* [Online]. - Duben 2013. - *http://code.google.com/p/pyglet/source/detail?r=64e3a450c83bd2245f047bb96fdacd79208d8b6a*.

Přílohy

Příloha A – Uživatelská příručka

Zabývá se instalací aplikace pod operační systémy Windows a Linux (Ubuntu). Dále popisuje její ovládání a používání, a nakonec popisuje a rozebírá moduly jednotlivých implementovaných algoritmů.

Součástí příručky je také licenční ujednání, vymezující podmínky, za kterých může být aplikace beztrestně používána a modifikována.

Příloha B – Programátorská příručka

Zabývá se instalací zdrojů aplikace a potřebných prerekvizit k jejímu vývoji. Popisuje základní strukturu projektu a především se zabývá tématem vytvoření nového modulu. Nakonec popisuje způsob aktualizace verze aplikace a vytvoření nové instalace.

Součástí příručky je také licenční ujednání, vymezující podmínky, za kterých může být aplikace beztrestně používána a modifikována.

Simulátor algoritmů OS – uživatelská příručka

k verzi 0.1.5 (5. 5. 2013)

Obsah

1. Účel programu

Simulátor algoritmů OS slouží jako názorná ukázka chování jednotlivých algoritmů s možností měnit různé aspekty simulace a pozorovat výsledky jejich změn. Je primárně určen jako výukový doplněk.

2. Licenční ujednání

3. Instalace

a. Linux (Ubuntu)

Podporovaným systémem je Ubuntu 12.10 a novější. Ostatní distribuce nebyly testovány a funkčnost, či bezproblémová instalace pod nimi není zaručena.

Instalace pod Linux vyžaduje rootovská práva a provede se následovně:

1) Spuštění terminálu a zadání příkazu **`sudo apt-get update`** pro obnovení informací o dostupných balících.
2) Spuštění příkazu **`sudo dpkg -i alg-simulator_0.1.5_all.deb`** (verze balíku se může lišit) pro přidání programu k instalaci a zjištění potřebných závislostí.
3) Spuštění **`sudo apt-get -f install`** pro opravu předchozího selhání instalace na základě nepřítomnosti prerekvizit instalace. Potřebné balíky prerekvizit se samy doinstalují.

Pokud instalace proběhla v pořádku, program bude možné spustit z příkazové řádky za pomoci příkazu **`alg-simulator`**.

Je třeba si uvědomit, že obzvláště první spuštění programu může chvíli trvat.

Funkčnost programu byla testována na KUbuntu 12.10 Desktop (64bit).

b. Windows®

Podporovanými systémy jsou Windows XP a Windows 7.

Instalace pod Windows® sestává ze dvou až tří kroků a vyžaduje práva super-uživatele nebo administrátora:

1) Spuštění **`alg-simulator_0.1.5.exe`** (verze programu se může lišit) rozbalí do aktuální složky, kde je soubor umístěn, složku **`alg-simulator`** obsahující program a instalátor prerekvizit.
2) Spuštění **`install-prereq.bat`** pak rozpozná architekturu systému (32/64bit) a spustí instalaci prerekvizit. Pro bezproblémové použití je doporučeno prerekvizity instalovat do standardních cest.
3) V případě, že prerekvizity nebyly nainstalovány do standardních cest, je zapotřebí ještě upravit spouštěcí soubor **`alg-simulator.bat`** tak, aby cesta k interpretu Pythonu odpovídala vámi zvolené cestě instalace, tedy je třeba nahradit **`c:\Python27\pythonw.exe`** za například **`c:\MojeSlozka\Python27\pythonw.exe`**. Mějte přitom na paměti, že názvy složek obsahujících mezery je třeba uzavírat do uvozovek.

Samotný program se pak spustí souborem **`alg-simulator.bat`**, který je umístěn v hlavní složce **`alg-simulator`**. Pro pohodlnější spouštění si na něj můžete vytvořit zástupce na pracovní plochu pomocí stisku pravého tlačítka myši na jeho ikoně a výběru možnosti **`Odeslat->Plocha (vytvořit zástupce)`**.

Je třeba si také uvědomit, že obzvláště první spuštění programu může chvíli trvat.

Funkčnost programu byla testována na Windows XP (32bit) a Windows 7 (64bit).

c. Jiné systémy

Jiné než výše uvedené systémy nejsou standardně podporovány, je ovšem nutné říci, že pokud daný systém podporuje OpenGL, Python2.7 (a novější z dvojkové řady), PyOpenGL, Pyglet a NumPy, je velká šance, že s trochou úsilí bude možné program na

tomto systému provozovat. Součástí programu je totiž i zdrojový archív `alg-simulator_0.1.5.tar.gz` (verze archívu se může lišit), který by se k tomuto účelu dal využít.

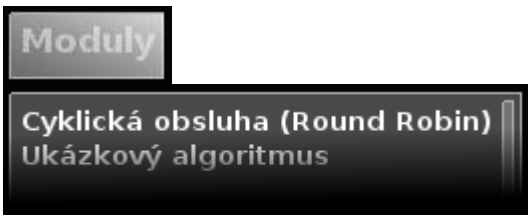
4. Ovládání

Program se ovládá výhradně za pomoci myši, a to levého a pravého tlačítka a kolečka. Ovládání je rozděleno do tří sekcí a to ovládacího menu (lišta nahoře; po načtení modulu se rozšíří), pravého ovládacího panelu (přibude po načtení modulu) a oblasti simulace (černá oblast).

a. První krůčky

Jelikož hlavním účelem tohoto programu je zobrazovat simulace, ukážeme si jak takovou nějakou simulaci spustit.

Začneme tím, že si zobrazíme seznam dostupných modulů, což provedeme stisknutím levého tlačítka myši na tlačítku **Moduly**. Po této akci se nám zobrazí něco podobného jako na obrázku vpravo.



Pro výběr modulu proklikneme levým tlačítkem myši například název **Cyklická obsluha (Round Robin)**.

Pokud vše proběhlo v pořádku, mělo by se zobrazit rozvržení simulace, rozšířit se horní menu o nová tlačítka, přibýt pravý ovládací panel modulu a logovací box dole nad informační lištou.

Abychom mohli v simulaci něco sledovat, musíme do ní přidat nějaké prvky, což učiníme stiskem tlačítka **Přidat nový element** v pravém ovládacím panelu. Můžeme jej stisknout několikráte, a přidat tak do simulace více nových elementů (v tomto případě procesů nebo vláken procesů).

Samotnou simulaci spustíme stisknutím tlačítka **Spuštění** z hlavního menu. Nyní by se měly prvky simulace začít hýbat a do logovacího boxu vypisovat informace o tom, co se během simulace vlastně děje. Stejným tlačítkem (ovšem se symbolem **Zastavení**), pak můžeme simulaci pozastavit.

Nakonec si povíme jak program ukončit. Ukončení programu je velice jednoduchá záležitost, buď stiskneme klávesu **Esc**, nebo program zavřeme standardně křížkem v liště okna.

Pro získání více informací o práci s moduly navštivte sekci Ovládání modulu.

Pro vysvětlení významů jednotlivých tlačítek navštivte Seznam všech ovládacích prvků.

Pro podrobnější informace o standardních modulech navštivte sekci Dostupné moduly.

b. Ovládání programu

Program samotný umožňuje uživateli tři základní akce, a to načtení modulu, přepnutí mezi celoobrazovkovým režimem a režimem okna, a ukončení programu za pomoci klávesy **ESC**.

Kromě toho, je program vybaven **Informační lištou** umístěnou v dolní části okna, ve které se zobrazují informace, jako, že se úspěšně podařilo načíst modul, nebo že v zadaném jméně uložení pozice se vyskytují nepovolené znaky, atp.; je tedy doporučeno lištu sledovat.

c. Ovládání modulu

Ovládání modulu se zpřístupní až po jeho načtení za pomoci tlačítka **Moduly**. Nové ovládací prvky zahrnují jak obecné akce s modulem jako uložení aktuálního stavu simulace apod., tak i akce zcela specifické pro daný modul.

Obecné akce jsou umístěny v hlavním menu a obsahují:

- **Zobrazení popisu modulu** – Uživatel má možnost si nechat zobrazit popis k právě načtenému modulu. *Zobrazovaný popis je text sestavený autorem modulu tak, aby přiblížil logiku simulovaného algoritmu a upozornil na jeho výhody a nevýhody.*

- **Načtení dříve uložené pozice simulace** – Uživateli je umožněno načíst dříve uložený stav simulace právě načteného modulu. *Výběr uložených pozic je řazen sestupně podle data a času vytvoření a obsahuje uživatelem zadané jméno a datum a čas vytvoření souboru.*

- **Uložení aktuálního stavu simulace** – Uživateli je umožněno uložit aktuální stav simulace pod jím zvoleným jménem. *Znaky použitelné pro pojmenování jsou omezeny na malé a velké znaky anglické abecedy, čísla a znaky podtržíto, pomlčka a tečka.*

- **Spuštění a zastavení běhu simulace** – Uživatel má možnost spustit a zastavit běh simulace za aktuálně nastavené rychlosti. *Standardní rychlost simulace je 1 krok za sekundu. Po spuštění simulace běží, dokud není uživatelem opět pozastavena.*

- **Zpomalení a zrychlení běhu simulace** – Uživatel má možnost měnit rychlost běhu simulace a to v rozmezí hodnot rychlosti od 1 kroku za 5 sekund do 30 kroků za sekundu. *Rychlost simulace je jen orientační a je vždy omezena výkonem počítače a náročností výpočtu kroku právě načteného modulu.*

- **Ruční posun v simulaci o jeden krok** – Uživatel má možnost si jednotlivé kroky simulace procházet postupně dle svého uvážení, aniž by byl vázán na nějaké nastavené tempo simulace. *Tato možnost je obzvláště výhodná při prezentacích, kdy uživatel jednotlivé kroky simulace komentuje.*

- **Obnovení výchozího stavu simulace** – Uživatel má možnost se jednoduše vrátit do stavu posledního načtení uložené pozice. *V případě, že žádná uložená pozice načtena nebyla, je programem simulace zcela vymazána.*

- **Vymazání stavu simulace –** Uživatel má možnost jednoduše vymazat aktuální stav simulace a v podstatě se vrátit do momentu těsně po načtení modulu.

Specifické akce pro daný modul jsou umístěny v pravém ovládacím panelu, a jsou, jak je již z názvu patrné, pro každý modul jiné. Obecně můžeme tedy jen popsat určitou logiku jednotlivých prvků, které se v pravém ovládacím panelu mohou objevit:

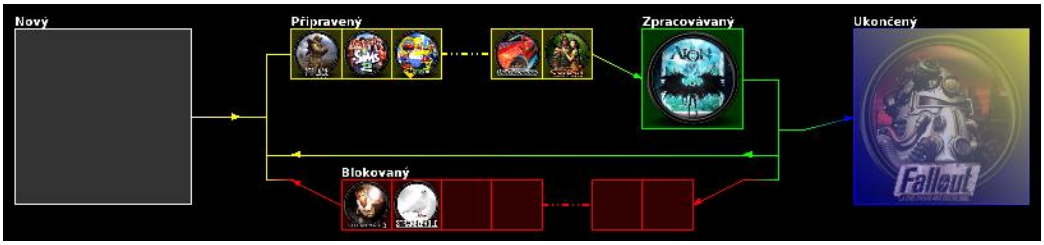
- **Popis** – Jde o informační text, který se bude pravděpodobně během simulace aktualizovat, ale neumožňuje uživateli vyvolat žádnou akci.
- **Hodnota** – Jde o záznam, který má jak informativní účel, tak umožňuje uživateli měnit hodnotu daného parametru simulace nebo elementu v simulaci.
- **Tlačítko** – Jde o prvek, který slouží čistě k vyvolání určité akce, např. k přidání nového elementu do simulace.

Dále bychom měli zmínit, že rozložení jednotlivých informací v pravém ovládacím panelu je rozděleno na informace a ovládací prvky k celé simulaci a na informace a ovládací prvky k právě vybranému elementu simulace (element je možné vybrat pomocí prokliku levého tlačítka myši nad jeho ikonou v simulaci).

d. Ovládání simulace

Se simulací (grafickou reprezentací algoritmu) je možné provádět několik základních věcí:

- **Pohyb** – Simulací je možné pohybovat za pomoci levého nebo pravého tlačítka myši tak, že jej stisknete a při jeho držení pohybujete myší. Stejného efektu lze dosáhnout za použití šipek na klávesnici.
- **Přiblížení/Oddálení** – Simulaci je možné přibližovat a oddalovat za pomoci kolečka myši, nebo kláves **PageUp** a **PageDown**.
- **Výběr elementu** – Pokud jsou v simulaci přidány elementy, je možné nějaký z nich vybrat za pomoci prokliku levého tlačítka myši na jeho ikoně. Pokud chceme vybraný element odznačit, proklikneme jej pravým tlačítkem myši, nebo označíme jiný.



e. Seznam všech ovládacích prvků

Prvek	Popis
Moduly	Tlačítko **Moduly** slouží k výběru modulu simulace. Po stisknutí se otevře seznam dostupných modulů; po výběru jednoho z nich se daný modul načte.
Popis	Tlačítko **Popis** slouží k zobrazení informací k vybranému modulu, sepsanými autorem modulu.

	Tlačítko **Načtení** slouží k načtení dříve uložených stavů simulace daného modulu (zobrazují se jen pozice příslušné k právě načtenému modulu).
	Tlačítko **Uložení** slouží k uložení aktuálního stavu simulace. Po stisku se zobrazí dialogové okno pro zadání jména, pod kterým se pozice bude zobrazovat.
	Tlačítko **Spuštění/Zastavení** slouží ke spuštění a pozastavení automatického běhu simulace dle aktuálně nastavené rychlosti.
	Tlačítka **Zpomalení** a **Zrychlení** slouží ke změně rychlosti automatického běhu simulace. Hodnoty rychlosti se pohybují od 1 kroku za 5 sekund do 30 kroků za sekundu. Rychlost je jen orientační a je omezena výkonem počítače a náročností výpočtu kroku načteného modulu.
	Tlačítko **Další krok** slouží k ručnímu přechodu simulace do následujícího kroku.
	Tlačítko **Obnovení** slouží k obnovení výchozího stavu simulace. Výchozí stav je buď nové načtení modulu, nebo načtení naposledy načtené uložené pozice.
	Tlačítko **Vymazání** slouží k návratu do stavu načtení daného modulu.
	Tlačítko **Celá obrazovka/Okno** slouží k přepínání mezi celoobrazovkovým režimem a režimem okna.
Klávesa **Esc**	Stisknutím klávesy **Esc** je možné program ukončit.

5. Dostupné moduly

a. Cyklická obsluha (Round Robin)

Krátké představení algoritmu

Jedná se o jeden z nejstarších a nejpoužívanějších algoritmů (v různých variantách). Každému procesu je přiřazen časový interval (časové kvantum), po které může běžet. Pokud proces běží ještě na konci kvanta, je provedena preempce a je naplánován a spuštěn další připravený proces. Pokud proces skončil nebo se zablokoval ještě před spotřebováním časového kvanta, je také naplánován a spuštěn další připravený proces. Ve verzi algoritmu označované jako virtuální cyklická obsluha se procesy po dokončení I/O upřednostní před ostatními.

Popis nastavení modulu

V horní části pravého panelu se nalézá:

1. Tlačítko **Přidat nový element** – přidá proces s náhodně určenými parametry do simulace
2. Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
3. Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
4. Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
   - **Existence** – průměrná existence procesu
   - **Na CPU** – průměrně strávený čas procesu na procesoru
   - **Čekání** – průměrná hodnota čekání procesu na procesor
   - **Blokace** – průměrná délka času procesu stráveného blokací

- **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
- **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)
- **Poč. blokací** – průměrný počet blokací procesu

5. Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
6. Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
7. Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)
8. Tlačítko **Virtual RR/Standard RR** – přepíná mezi variantami algoritmu (Virtual RR upřednostňuje ve frontě připravených procesy přicházející z blokace, Standard RR nikoliv)

Popis nastavení elementu

V dolní části pravého panelu se při výběru procesu nalézá:

1. **Ikona procesu** – napomáhá rozpoznání označeného procesu
2. Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
3. Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)
4. Informace **Statistiky** – nadpis níže uvedených statistik
   - **Existuje** – čas existence procesu
   - **Doba na CPU** – čas procesu strávený na procesoru
   - **Doba čekání** – čas procesu strávený čekáním na procesor
   - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, uspáním, apod.)
   - **Doba přeplánování** – čas procesu strávený přeplánováváním
   - **Byl blokován** – počet blokací procesu
5. Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
6. Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)
7. Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
8. Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
9. Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
10. Tlačítko **Zabít proces** – umožní uživateli zabít proces, ten se okamžitě přesune do sekce ukončených procesů

Poznámka k logice simulace

V této simulaci odpovídá jeden krok maximálně době jednoho kvanta a minimálně době do blokace nebo ukončení běžícího procesu. Během této doby se mohou v simulaci udát další věci, které nemusí být na první pohled zřejmé; jsou ovšem vždy zaznamenány v logu simulace – je tedy dobrým zvykem log během simulace sledovat. Jednou z takovýchto věcí může být například to, že při ukončení blokace procesu a

prázdné frontě připravených procesů, se vizuálně doteď blokovaný proces přesune rovnou na procesor; z logu je ovšem patrné, že tento dojem je zavádějící, a že proces byl napřed řádně naplánován do fronty připravených procesů, a následně z ní byl vybrán ke zpracování procesorem.

b. Víceúrovňová zpětná vazba (Multi-level feedback queue)

Krátké představení algoritmu

Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. Pro každou prioritu existuje jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty. Každý proces má přidělenou omezenou výpočetní dobu, kterou může strávit na dané prioritní úrovni – tím se zaručí upřednostnění I/O vázaných procesů před výpočetně vázanými. Čas přidělený nižší prioritní úrovni je vždy dvojnásobný než čas přidělený prioritní úrovni o jedna vyšší.

Tento modul pro názornost pracuje s třemi prioritními frontami (v praxi jich bývá více) a tedy prioritou procesů v rozmezí 1 až 3, kde priorita 3 je brána jako nejvyšší.

Popis nastavení modulu

V horní části pravého panelu se nalézá:

1. Tlačítko **Přidat nový element** – přidá proces s náhodně určenými parametry do simulace
2. Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
3. Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
4. Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
   - **Existence** – průměrná existence procesu
   - **Na CPU** – průměrně strávený čas procesu na procesoru
   - **Čekání** – průměrná hodnota čekání procesu na procesor
   - **Blokace** – průměrná délka času procesu stráveného blokací
   - **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
   - **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)
   - **Poč. blokací** – průměrný počet blokací procesu
5. Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
6. Parametr **Počet procesorů** – určuje počet procesorů použitých v simulaci
7. Parametr **Max. čas na max. prio.** – určuje množství výpočetního času v milisekundách, které může maximálně proces na nejvyšší prioritní úrovni strávit; poté se jeho priorita poníží. Hodnota parametru slouží jako základ pro výpočet množství výpočetního času pro ostatní prioritní úrovně.
8. Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
9. Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)

Popis nastavení elementu

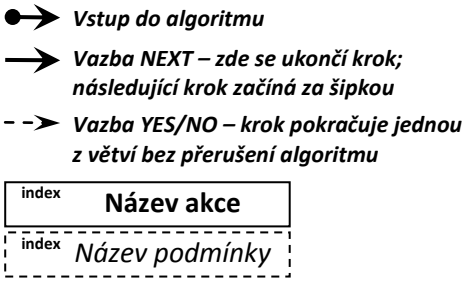
V dolní části pravého panelu se při výběru procesu nalézá:

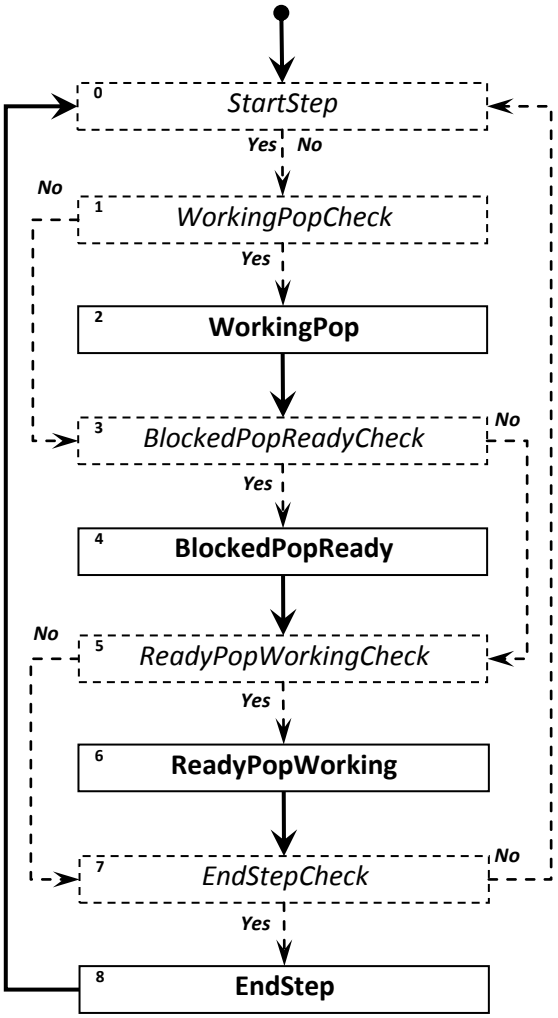
1. **Ikona procesu** – napomáhá rozpoznání označeného procesu
2. Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
3. Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)
4. Informace **Statistiky** – nadpis níže uvedených statistik
   - **Existuje** – čas existence procesu
   - **Doba na CPU** – čas procesu strávený na procesoru
   - **Doba čekání** – čas procesu strávený čekáním na procesor
   - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, atp.)
   - **Byl blokován** – počet blokací procesu
5. Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
6. Parametr **Priorita** – určuje hodnotu priority procesu
7. Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)
8. Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
9. Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
10. Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
11. Tlačítko **Zabít proces** – umožní uživateli zabít proces, ten se okamžitě přesune do sekce ukončených procesů

Poznámka k logice simulace

V této simulaci odpovídá jeden krok maximálně době jednoho výpočetního kvanta a minimálně době jedné milisekundy. Aktuální čas kroku je vždy určen jako čas do nejbližší akce – změny stavu jakéhokoliv procesu.

Posloupnost jednotlivých kroků znázorňuje následující graf.





Název	***Typ***	***Popis***
StartStep	Podmínka	Počátek cyklu – zaloguje oddělovač a nastaví pomocné proměnné.
WorkingPopCheck	Podmínka	Zjistí, zda se má nějaký blokovaný proces přeplánovat, zpracovávaný proces ukončit, přeplánovat, nebo zablokovat. Vrací seznam nejbližších akcí nebo None.
WorkingPop	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.
BlockedPopReadyCheck	Podmínka	Aktualizuje statistiky blokovaných procesů a zjistí, zda nějakému procesu skončila blokace.
BlockedPopReady	Akce	Přesune všechny procesy, kterým skončila blokace do fronty připravených.
ReadyPopWorkingCheck	Podmínka	Aktualizuje statistiky připravených procesů a zjistí, zda je nějaký z procesorů prázdný a vrátí jejich seznam nebo None.
ReadyPopWorking	Akce	Pro každý prázdný procesor z přijatého seznamu vybere nejprioritnější čekající proces a přiřadí mu jej.
EndStepCheck	Podmínka	Zjistí, zda se v daném cyklu něco událo; pokud ne, přejde na „EndStep“, aby nedošlo k nekonečné smyčce.
EndStep	Akce	Slouží jako zarážka – prázdná metoda.

Pokud srovnáme kroky tohoto algoritmu například s Cyklickou obsluhou popsanou výše, zjistíme, že jeden krok v Cyklické obsluze odpovídá celému cyklu kroků zde, a že tento modul simuluje daný algoritmus mnohem podrobněji a názorněji.

c. Víceúrovňová prioritní fronta (Multi-level priority queue)

Krátké představení algoritmu

Algoritmus pracuje s prioritami procesů, na základě kterých určuje pořadí výběru procesů z fronty čekajících. V praxi pak existuje pro každou prioritu jedna prioritní fronta, v rámci níž jsou procesy řazeny podle principu FIFO. Algoritmus pak sestupně prochází jednotlivé fronty, dokud nenajde nějakou obsahující čekající proces, nebo bez úspěchu neprojde všechny fronty. Aby nenastalo vyhladovění procesů s nízkou prioritou, každému procesu je čas od času vypočítávána takzvaná dynamická priorita, která má za úkol občas „pustit na řadu“ i „méně šťastné“ procesy.

Tento modul pro názornost pracuje s pěti prioritními frontami (v praxi jich bývá 30 až 50) a tedy statickou prioritou procesů v rozmezí 1 až 5, kde priorita 5 je brána jako nejvyšší. Dynamická priorita se může pohybovat v rozmezí -2 až +2, a výsledná priorita procesu se pak vždy skládá jako součet priority statické a dynamické.

Popis nastavení modulu

V horní části pravého panelu se nalézá:

2. Tlačítko **Přidat nový element** – přidá proces s náhodně určenými parametry do simulace
3. Informace **Doba běhu** – zobrazuje počet milisekund, které uběhly od počátku simulace (nejedná se o reálný čas, ale o imaginární čas v simulaci)
4. Informace **Počet procesů** – zobrazuje aktuální počet živých procesů (při ukončení procesu je hodnota ponížena)
5. Informace **Statistiky (průměry)** – nadpis níže uvedených statistik
   - **Existence** – průměrná existence procesu
   - **Na CPU** – průměrně strávený čas procesu na procesoru
   - **Čekání** – průměrná hodnota čekání procesu na procesor
   - **Blokace** – průměrná délka času procesu stráveného blokací
   - **Přeplánování** – průměrná hodnota času procesu strávená přeplánováním
   - **Přep. prio** – průměrná hodnota času procesu strávená během přepočtu priorit
   - **Náročnost** – průměrná časová náročnost procesu (jaký čas potřebuje strávit na procesoru, aby se dokončil)
   - **Poč. blokací** – průměrný počet blokací procesu
6. Tlačítko **Vymazání statistik** – vynuluje globální statistiky (hodnota **Náročnost** se přepočte na aktuální stav)
7. Parametr **Počet procesorů** – určuje počet procesorů použitých v simulaci
8. Parametr **Přeplánování po** – určuje časový úsek (kvantum), který maximálně může proces strávit na procesoru
9. Parametr **Doba přeplánování** – určuje dobu strávenou jedním přeplánováním (přeplánování se provádí vždy po uběhnutí kvanta, anebo při skončení nebo zablokování procesu)
10. Parametr **Kontrola priorit po** – určuje dobu, po které se provádí kontrola priorit u běžících procesů (a případná následná záměna za proces s vyšší prioritou)
11. Parametr **Přep. priorit každou** – určuje, kolikátou každou kontrolu se provede přepočet priorit všech živých procesů
12. Parametr **Doba přepočtu priorit** – určuje dobu strávenou jedním přepočtem priorit

Popis nastavení elementu

V dolní části pravého panelu se při výběru procesu nalézá:

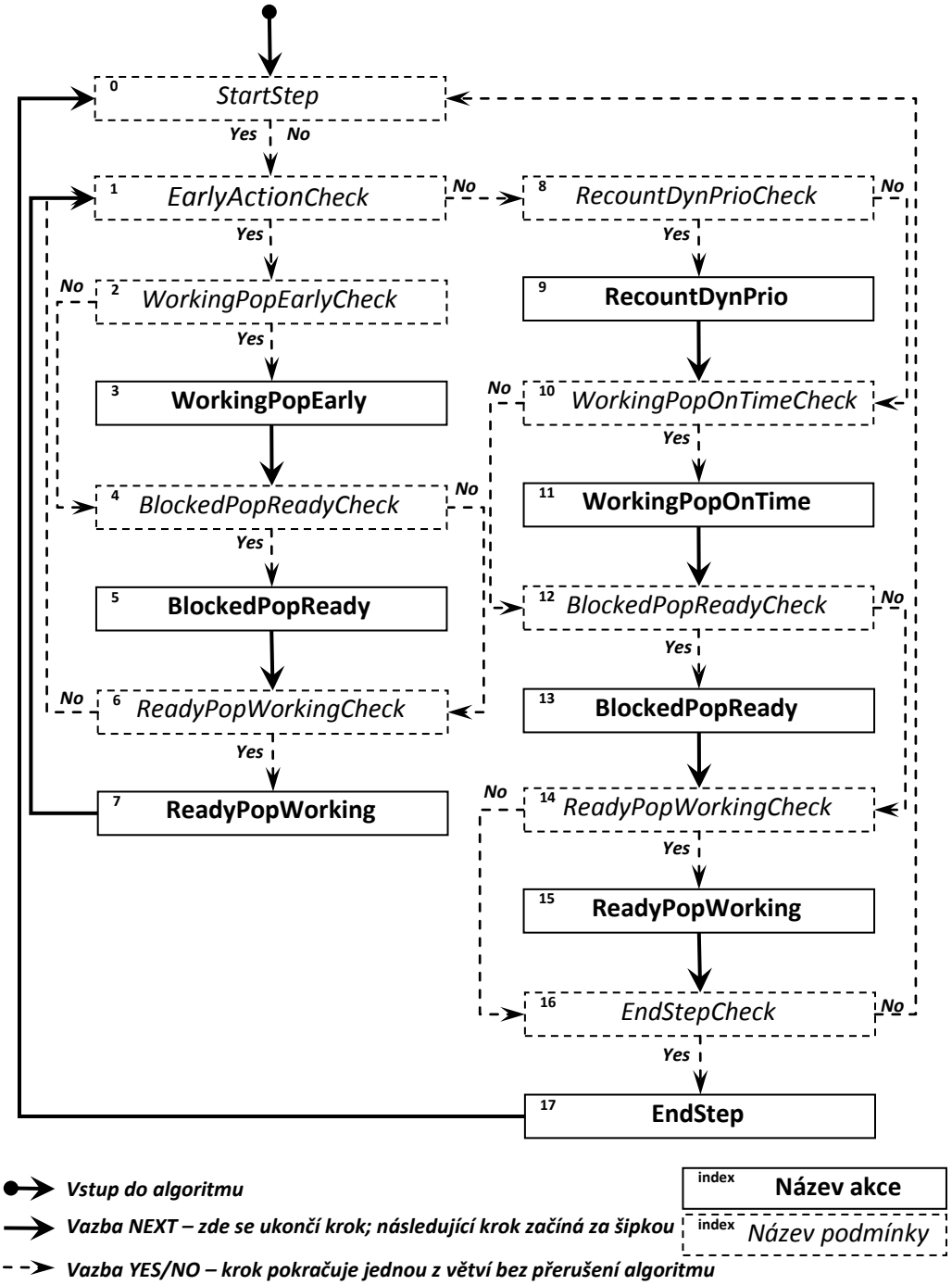
1. **Ikona procesu** – napomáhá rozpoznání označeného procesu
2. Informace **Stav** – zobrazuje slovní popis stavu, ve kterém se proces nachází (Nový, Připravený, Zpracovávaný, Blokovaný, Ukončený)
3. Informace **Zpracováno** – zobrazuje procento zpracování úkolu procesu, kde nový proces má hodnotu zpracování na 0% a ukončený na 100% (je třeba si uvědomit, že v reálu nelze obecně říci, kolik výpočetního času proces pro své dokončení bude potřebovat)
4. Informace **Statistiky** – nadpis níže uvedených statistik
   - **Existuje** – čas existence procesu
   - **Doba na CPU** – čas procesu strávený na procesoru
   - **Doba čekání** – čas procesu strávený čekáním na procesor
   - **Doba blokace** – čas procesu strávený blokací (čekáním na I/O operaci, uspáním, apod.)
   - **Byl blokován** – počet blokací procesu
5. Parametr **Potřebuje výp. času** – určuje celkovou dobu, kterou proces potřebuje strávit na procesoru, než se ukončí
6. Parametr **Priorita** – určuje hodnotu statické priority procesu (v praxi je změna statické priority procesu umožněna administrátorovi systému většinou jen v rozmezí -3 až +3)
7. Parametr **Přeplánování po** – určuje procento změny standardního výpočetního kvanta pro daný proces od -90% po +100% (v praxi je možnost úpravy výpočetního kvanta procesu omezena na jeho zařazení do jedné ze dvou skupin a to „programu“ (kratší kvantum; standardní možnost pro uživatelské procesy) nebo „démonu“ (delší kvantum; většinou využívají systémové procesy))
8. Parametr **Blokován** – určuje zda, a na jak dlouho je ještě proces blokován (hodnotu parametru lze měnit jen, když je proces blokován)
9. Parametr **Šance na blokaci** – určuje procentuální šanci na zablokování procesu během jednoho výpočetního kvanta
10. Parametr **Min. doba blokace** – určuje minimum náhodně určené doby blokace, v případě, že se proces má zablokovat
11. Parametr **Max. doba blokace** – určuje maximum náhodně určené doby blokace, v případě, že se proces má zablokovat
12. Tlačítko **Zabít proces** – umožní uživateli zabít proces, ten se okamžitě přesune do sekce ukončených procesů

Poznámka k logice simulace

V této simulaci odpovídá jeden krok maximálně době jedné kontroly priorit (ta se provádí standardně několikrát za kvantum) a minimálně době jedné milisekundy. Aktuální čas kroku je vždy určen jako čas do nejbližší akce. Akcí pak je změna stavu jakéhokoliv procesu, nebo kontrola priorit.

Posloupnost jednotlivých kroků je rozdělena do dvou základních logických částí a to do obsluhy předčasných (rozumějme dřívějších než je plánovaná kontrola) akcí a obsluhy akcí při plánované kontrole. Hlavním rozdílem je, že při plánované kontrole se navíc kontroluje dostatečnost priorit právě běžících procesů – v této části tedy může proces být nucen ukončit svůj výpočet na základě nižší priority, než má nějaký z připravených procesů – a pokud se jedná o X-tou kontrolu (lze nastavit; standardně každou 10. kontrolu), provádí se v této části přepočet priorit všech živých procesů.

Posloupnost jednotlivých kroků znázorňuje následující graf.



Název	*Typ*	*Popis*
StartStep	Podmínka	Počátek cyklu – zaloguje oddělovač a nastaví pomocné proměnné.
EarlyActionCheck	Podmínka	Zjistí, zda se má nějaký blokovaný proces přeplánovat, zpracovávaný proces ukončit, přeplánovat na základě vypršení kvanta nebo zablokovat dříve, než uběhla perioda kontroly (standardně 5 ms).
WorkingPopEarlyCheck	Podmínka	Na základě přijatých dat zjistí, zda se má nějaký zpracovávaný proces ukončit, přeplánovat nebo zablokovat. Předává přijatá data nebo None.
WorkingPopEarly	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.

Název	*Typ*	*Popis*
WorkingPopOnTimeCheck	Podmínka	Zjistí, zda se má nějaký zpracovávaný proces ukončit, přeplánovat na základě vypršení kvanta nebo nedostatečné priority nebo zablokovat při skončení periody kontroly (standardně 5 ms).
WorkingPopOnTime	Akce	Provede ukončení, přeplánování nebo zablokování dle přijatého seznamu nejbližších akcí. Aktualizuje příslušné statistiky.
BlockedPopReadyCheck	Podmínka	Aktualizuje statistiky blokovaných procesů a zjistí, zda nějakému procesu skončila blokace.
BlockedPopReady	Akce	Přesune všechny procesy, kterým skončila blokace do fronty připravených.
ReadyPopWorkingCheck	Podmínka	Aktualizuje statistiky připravených procesů a zjistí, zda je nějaký z procesorů prázdný a vrátí jejich seznam nebo None.
ReadyPopWorking	Akce	Pro každý prázdný procesor z přijatého seznamu vybere nejprioritnější čekající proces a přiřadí mu jej.
RecountDynPrioCheck	Podmínka	Zjistí, zda již nastal čas pro přepočet dynamických priorit procesů.
RecountDynPrio	Akce	Přepočte dynamické priority na základě statistických hodnot procesu tak, aby se všechny procesy občas dostaly na procesor.
EndStepCheck	Podmínka	Zjistí, zda se v daném cyklu něco událo; pokud ne, přejde na „EndStep“, aby nedošlo k nekonečné smyčce.
EndStep	Akce	Slouží jako zarážka – prázdná metoda.

Pokud srovnáme kroky tohoto algoritmu například s Cyklickou obsluhou popsanou výše, zjistíme, že jeden krok v Cyklické obsluze odpovídá celému cyklu kroků zde, a že tento modul simuluje daný algoritmus mnohem podrobněji a názorněji. Tento přístup byl zvolen, jelikož se scéna v případě prvního přístupu krokování stávala vzhledem ke komplexnosti prováděných akcí nepřehlednou.

d. ~ Prázdný algoritmus ~

Tento modul nesimuluje žádný algoritmus a slouží jako šablona pro vytváření nových modulů.

e. ~ Ukázkový algoritmus ~

Tento modul slouží jako ukázkový pro snadnější pochopení principů vytváření vlastních modulů. Algoritmus v něm simulovaný je zcela smyšlený a nemá žádnou výukovou hodnotu.

6. FAQ (Často kladené otázky)

a. Program nelze spustit

Zkontrolujte, zda máte správně nainstalované všechny podmiňující programy jako Python2.7, apod. V případě instalace pod Windows® je nutné buď dodržet standardní cesty jednotlivých instalací, nebo si upravit spouštěcí soubor `alg-simulator.bat` tak, aby obsahoval správnou cestu k interpretu Pythonu.

b. Program nemůže najít můj modul

Váš modul se musí nalézat v podsložce `modules`, pokud tomu tak je, pak došlo při jeho načítání k chybě. Podrobnější informace o problému pak naleznete v logu – podsložka `logs`.

Simulátor algoritmů OS – programátorská příručka

k verzi 0.1.5 (5. 5. 2013)

Obsah

1. Představení programu

Simulátor algoritmů OS slouží jako názorná ukázka chování jednotlivých algoritmů s možností měnit různé aspekty simulace a pozorovat výsledky jejich změn. Je primárně určen jako výukový doplněk.

Program byl navržen tak, aby do něj bylo možné relativně jednoduše přidávat další moduly (simulace). Jakým způsobem by se takový modul do programu přidělal, se pokusí přiblížit tato příručka.

2. Licenční ujednání

3. Instalace zdrojů

Program byl vyvíjen pod KUbuntu 12.10 a je proto pro vývoj v tomto prostředí nejlépe připraven. Ovšem, jelikož je program multiplatformní, vývoj i pod jiným systémem jako jsou Windows®, případně Mac OS, by neměl být velkým problémem.

Zdroj programu je obsažen v archívu **alg-simulator_0.1.5.tar.gz** a pod Linuxem jej lze rozbalit příkazem **tar -zxvf alg-simulator_0.1.5.tar.gz**, ve Windows® použijte k jeho rozbalení například program WinRAR.

Ubuntu

Závislosti potřebné k vývoji a vytváření balíčků pod Ubuntu naleznete v souboru **alg-simulator/debian/control** v částech **Build-Depends** a **Depends** a nainstalujete je jednoduše příkazy:

- **sudo apt-get update**
- **sudo apt-get install bash sed tar gzip libjs-jquery make devscripts dpkg-dev debhelper lintian lintian4python doxygen doxypy graphviz python=2.7.3 python-pyglet=1.1.4 python-opengl python-numpy**

Pokud byste měli problém s instalací přímo specifikované verze Pythonu, zvolte jakoukoliv novější, avšak menší než 3.0. Python 3.0 a vyšší není programem bez nezbytných úprav podporován.

Po nainstalování knihovny Pyglet verze 1.1.4 je nutné spustit patch, který opraví chybu 471 této distribuce (**alg-simulator/patches/patch.sh**). Podrobnější informace k chybě 471 naleznete na *http://code.google.com/p/pyglet/issues/detail?id=471*. Bez této opravy nebude program fungovat. Pokud jste nainstalovali novější verzi knihovny, je třeba zjistit, zda opravu již obsahuje, a dle toho případně upravit soubor **/usr/share/pyshared/pyglet/text/runlist.py**.

Windows®

Pod Windows bude nutné si nainstalovat prerekvizity za pomoci **install-prereq.bat** ze složky **alg-simulator\install\win**. Pro možnost přegenerování automatické nápovědy z kódu, budou zapotřebí ještě programy Doxygen, DoxyPy a GraphViz pro Windows®, které nejsou součástí archívu.

4. Struktura programu

a. Soubory a složky

Hlavní složka (**alg-simulator**):

Název souboru	Systém	Funkce
alg-simulator.bat	Windows	Spouštěč programu ve Windows
alg-simulator.sh	Linux	Spouštěč programu v Linuxu (jen po instalaci)
configuration.cfg	Všechny	Konfigurace programu
doxygen.cfg	Všechny	Konfigurace generátoru automatické nápovědy. (**doxygen doxygen.cfg**)
install.bat	Windows	Batch soubor k vytvoření instalace ve Windows
Makefile	Linux	Makefile soubor k vytvoření balíčku v Linuxu (za pomoci příkazu **debuild**)
run.py	Všechny	Hlavní program (**python run.py configuration.cfg**)
tasklist.txt	Všechny	Soubor s úkoly k zapracování

Podsložka **data**:

Složka obsahuje ikony pro samotný program a archív **icons.zip** s ikonami sloužícími jako reprezentace elementů (procesů) v simulacích. V případě záměny těchto ikon za jiné, je možné stejným způsobem vytvořený archív umístit do této cesty a nadefinovat jej v konfiguraci programu (**configuration.cfg**).

Podsložka **debian**:

Tato složka obsahuje skripty a nastavení určené k vytvoření balíčků pro distribuce Linuxu založené na Debianu (např. Ubuntu). Podrobnější informace můžete získat na adresách:

- *http://savetheions.com/2010/01/20/packaging-python-applicationsmodules-for-debian/*
- *http://www.debian.org/doc/manuals/maint-guide/*
- *http://wiki.debian.org/IntroDebianPackaging*

Podsložka **doc**:

Do této složky generuje Doxygen automatickou nápovědu ke zdrojovým kódům.

Podsložka **docs**:

Tato složka obsahuje materiály týkající se problematiky algoritmů operačních systémů a manuály k programu.

Podsložka **install**:

V této složce a jejích podsložkách jsou uloženy prerekvizity pro instalaci pod Windows®. Součástí je také batch soubor pro instalaci prerekvizit.

Podsložka **lib**:

Tato složka obsahuje veškeré knihovny programu. Pokud by se program rozšiřoval o nějakou knihovnu, měla by být přidána sem.

Podsložka **logs**:

Do této složky se standardně ukládají logy o běhu programu. V případě, že program nebo nějaký modul selže, v této složce naleznete log s bližšími informacemi k tomuto selhání.

Podsložka **modules**:

Z této složky se načítají veškeré moduly dostupné v programu. V případě vytváření nového modulu, toto je místo, kde musí být založen.

Podsložka **patches**:

Tato složka obsahuje skript a soubory nutné k opravě chyby 471 knihovny Pyglet verze 1.1.4. Toto je zapotřebí při vytváření balíčku pod Linux, de-facto při jeho instalaci. Bližší informace k chybě a opravě je možné nalézt na:

- *http://code.google.com/p/pyglet/issues/detail?id=471*
- *http://code.google.com/p/pyglet/source/detail?r=64e3a450c83bd2245f047bb96fdacd79208d8b6a*

Podsložka **saves**:

Do této složky se ukládají pozice simulací, které je možné následně do programu opět načíst. Uložené pozice se nezahrnují do instalačních balíčků programu, po instalaci je tedy program bez těchto pozic.

Podsložka `win`:

Do této složky nic neukládejte, při spuštění `install.bat` se vždy přemazává a kopíruje se do ní aktuální instance programu pro přípravu instalace pod Windows®. V této složce je pak k nalezení výsledná instalace pro Windows®.

b. Rozvržení logiky programu

Program se primárně skládá z části řešící GUI, části řešící rozvržení simulace, části řešící spolupráci modulů s programem, pomocných knihoven a hlavního programu.

GUI část řeší zejména knihovna `lib.gControls`, ze které si hlavní program načítá veškeré ovládací prvky pro vytvoření hlavního menu a logovacího a informačního panelu. Ovládací prvky jsou založeny na knihovně Pyglet (viz *http://www.pyglet.org/*).

Část rozvržení simulace je řešena zejména knihovnami `lib.simulation` a `lib.gComponents`. Objekty s rozšiřujícími vlastnostmi a další nástroje pro použití priorit obsahuje knihovna `lib.gAdvancedComponents`. Poslední dvě zmíněné knihovny pak využívají pomocnou knihovnu `lib.gTools`. Knihovna `lib.simulation` řeší potřebná nastavení OpenGL a možnosti pohybu, přiblížení / oddálení simulace a výběru (prokliku) elementu. Naopak knihovna `lib.gComponents` obsahuje objekty prvků rozvržení (jako jsou `EntityArray`, `EntitySet`, apod.) a veškerou logiku pro jejich propojování.

Částí řešící spolupráci modulů s programem je knihovna `lib.module`, která definuje předka všech algoritmů jednotlivých modulů, tzv. `AbstractAlgorithm`. Dále knihovna obsahuje objekty pro podporu vložených ovládacích prvků, které se využívají v pravém informačním panelu pro vytváření struktury zobrazovaných informací a nastavení k simulovanému algoritmu nebo vybranému elementu simulace.

Mezi pomocné knihovny pak patří `lib.config` zajišťující načítání konfigurace a `lib.util` obsahující různé pomocné funkce.

5. Vytvoření nového modulu

a. Jak začít

Před vytvářením každého modulu byste si měli uvědomit, co od nového modulu budete očekávat (jak by se měl algoritmus chovat, jaké parametry u něj budete chtít sledovat a nastavovat, jaké statistiky sbírat, atd.), a zda již podobná funkčnost v nějakém z již implementovaných modulů není obsažena.

V případě, že velká část vámi požadované funkcionality je již obsažena v nějakém existujícím modulu, jednoduše jej zkopírujte a přejmenujte. Uvnitř souboru pak musíte upravit několik věcí, aby program rozpoznal, že se jedná o nový modul, a to:

- Hodnoty `_VERSION` a `_SUBVERSION` nastavit na 1 a 0.
- Hodnotu `_FILENAME_PREFIX` nastavit na nějakou zatím nepoužitou (nejlépe, aby také respektovala pojmenování nového modulu). Pokud tak neučiníte, budou kolidovat názvy uložených pozic vašeho a původního modulu, a následně bude docházet k chybám při pokusu o načtení nesprávné uložené pozice.
- Hodnotu `ALGORITHM_NAME` přejmenovat na název nového modulu. Pokud byste tak neučinili, uživatel by nebyl schopen rozpoznat, který z modulů načítá.
- Upravit popis algoritmu modulu v proměnné `ALGORITHM_DESCRIPTION` tak, aby odpovídal vámi implementovanému algoritmu.
- Dobrým zvykem je pak přejmenovat hlavní třídu na název odpovídající novému algoritmu. Tento název je pak třeba aktualizovat i v proměnné `ALGORITHM_CLASS`.

Pokud plánujete vytvářet modul zcela odlišný od již implementovaných, vytvořte si kopii modulu **empty.py** pod vámi zvoleným názvem, a postupujte dle v ní uvedených pokynů v komentářích TODO.

b. Popis struktury modulu

Standardní modul se skládá z povinných, volitelných a doplňkových částí, s tím, že doplňkovými částmi je namysli kód modulu nesouvisející se strukturou programu – pomocné funkce, třídy a metody.

Povinnými částmi modulu jsou definované proměnné:

- **ALGORITHM_CLASS** – musí obsahovat jméno třídy poděděné od **AbstractAlgorithm**
- **ALGORITHM_NAME** – musí obsahovat řetězec; ten by měl jednoznačně identifikovat algoritmus
- **ALGORITHM_DESCRIPTION** – musí obsahovat řetězec; ten by měl popisovat funkcionalitu algoritmu, jeho výhody a nevýhody a obsahovat další relevantní informace k algoritmu

Z výše uvedeného vyplývá nutnost přítomnosti třídy poděděné od **AbstractAlgorithm**, která musí obsahovat nastavení proměnných a implementaci metod:

- Třídní proměnná **_VERSION** – musí obsahovat celé číslo (iniciálně 1)
- Třídní proměnná **_SUBVERSION** – musí obsahovat celé číslo (iniciálně 0)
- Třídní proměnná **_FILENAME_PREFIX** – musí obsahovat řetězec jednoznačný mezi všemi moduly, který může obsahovat jen znaky, které umožňuje systém použít v názvech souborů (je tedy doporučeno používat jen znaky anglické abecedy a podtržítka)
- Metoda **initAlgInfoData** – slouží k inicializaci dat, informací a nastavení určených k celému modulu, bez ohledu na vybraný element simulace.
  - Pro nastavování hodnot vnitřních dat běhu algoritmu se musí využívat slovník **self._simulationData**, který může obsahovat jen standardní typy Pythonu (**int**, **float**, **str**, **unicode**, **tuple**, **list**, **dict**)
  - Pro definici zobrazovaných informací a nastavení k celému modulu je nutné využít vnitřní proměnnou **self._simulationInfo**, a to konkrétně její metodu **addToAlgData**.
  - Například:

```
# Inicializace pomocných proměnných
self._simulationData["elmCount"] = 0

# Vytvoření oddělovače od standardního tlačítka "Nový element"
self._simulationInfo.addToAlgData( \
      lib.module.SimulationInfoSeparator())

# Vytvoření informačního textu
infoText = lib.module.SimulationInfoText( \
      lambda: "Počet procesů: %s" % \
      (self._simulationData["elmCount"], ))

self._simulationInfo.addToAlgData(infoText)
```

- Metoda **initSimulationElements** – slouží k inicializaci rozvržení simulace a uchování si později potřebných částí rozvržení v separátní struktuře
  - Pro uchování si později potřebných částí rozvržení slouží slovník **self._simulationElements**, který může obsahovat jen instance tříd poděděných od třídy **Element**, které mají nadefinované určité metody. Pro zjednodušení se jedná o instance tříd **EntityArray**, **EntitySet** a **PriorityGroup**.

- Pro samotné vytvoření rozvržení simulace slouží proměnná **self._simulation**, do které se za pomoci metody **addElement** přidají jednotlivé části simulace. Pokud ovšem potřebujete jednoduché rozvržení, je doporučeno jednotlivé části pospojovat za pomoci instancí tříd jako **Direction**, **Branche**, **EntityGroup** a **EmptyElement**, a vkládat do **self._simulation** jen jeden prvek.
- Například:

```
# Fronta čekajících procesů
self._simulationElements["ready"] = lGC.EntitySet(3, 2, 1.0,
      [c / 255.0 for c in self._COL_READY],
      "Připravený", True, True, None, lGC.Entity)

# ...

# Seskupení (vytvoření optického cyklu)
circle = lGC.EntityGroup([top, arrow, bottom],
      self._simulationElements["ready"].inColor,
      self._simulationElements["working"].outColor)

# ...

# Vložení grafu do simulace
self._simulation.addElement(graph, 0, 0, 0)

# Vycentrování simulace
self._simulation.position.setMiddle()
```

- Metoda **fillElmInfoData** – slouží k inicializaci zobrazovaných informací a nastavení k právě vybranému elementu (procesu) simulace. Obsah vybraného elementu je metodě předáván jako argument.
- Pro definici zobrazovaných informací a nastavení k vybranému elementu nutné opět využít vnitřní proměnnou **self._simulationInfo**, ovšem tentokráte za pomoci její metody **addToElmData**.
- Například:

```
fData = filling.getData()

# Stav
if "state" in fData:
      stateText = lib.module.SimulationInfoText( \
            lambda: "Stav: %s" % (fData["state"], ))

      self._simulationInfo.addToElmData(stateText)
#endif

# ...

# Zda blokován
blockedValue = lib.module.SimulationInfoValue(self._parent,
      "Blokován (kroků)", lambda: fData.get("blocked", "Není"),
fData)

blockedValue.setValChangeFunc(self._local_blockedValDec,
      self._local_blockedValInc)

self._simulationInfo.addToElmData(blockedValue)
```

- Metoda **newElement** – slouží k přidání nového elementu (procesu) do simulace.
- Například:

```
# Vyprázdnění pole pro nové elementy
filling = self._simulationElements["new"].pop()
```

```
# Případný přesun vyprázdněného elementu do fronty připravených
if filling:
      filling.getData()["state"] = "Připravený"

      self._simulationElements["ready"].push(filling)

      self._log.insertLine("Proces %s byl naplánován." % \
            (filling.getId(), ), self._COL_READY)
#endif

# Vytvoření nového elementu a jeho inicializace
filling = self.newFilling({"counter" : 0, "state" : "Nový"})

# Zařazení nového elementu do pole pro nové elementy
self._simulationElements["new"].push(filling)

# Záznam do logu simulace o přidání elementu
self._log.insertLine("Proces %s byl vytvořen." % \
      (filling.getId(), ), self._COL_NEW)

# Zvýšení hodnoty počtu aktivních elementů v simulaci
self._simulationData["elmCount"] += 1
```

- Metoda **nextState** – slouží k výpočtu jednoho kroku simulace a aktualizaci jejího stavu. Jedná se o metodu, ve které se děje veškerá logika simulace daného algoritmu. Ve standardních modulech jsou použity dva odlišné přístupy simulací:
  - **example.py** a **roundRobin.py** – jednodušší přístup s částečně pevným krokem; akce časově náležející mezi jednotlivé kroky se jen statisticky dopočítávají – např. skončení blokace procesu; jednoduše implementovatelné; nevhodné pro složitější simulace
  - **multilevelFeedback.py** a **priorityQueue.py** – komplexnější způsob, kdy se dopředu spočte co by se mělo stát (i se započtením náhodnosti) a pak se vyberou časově nejbližší akce a pro ty se aplikuje daný krok; jedná se o implementačně i výpočetně náročnější přístup; simulace je vizuálně plynulejší a srozumitelnější; lze využít i pro celkem komplikované simulace

Mezi volitelné části modulu patří implementace nepovinných metod hlavní třídy algoritmu:

- Metoda **getUserText** – slouží k rozšíření textu zobrazovanému v liště programu. Příklad použití naleznete v **roundRobin.py**.
- Metoda **genLogStatus** – slouží k zobrazení textu v logu simulace při načtení modulu nebo uložené pozice. Rozumné je vypsat hodnoty globálních nastavení simulace apod. Příklad použití naleznete v **roundRobin.py**.

Ostatní funkce a třídy v modulech slouží jen jako pomocné pro daný modul.

c. Seznam užitečných funkcí a tříd

Třídy prvků pravého informačního panelu

Jedná se o třídy poděděné od **lib.module.AbstractSimulationInfoValue** a jsou výhradně určeny jako prvky pravého informačního panelu. Jejich instance je možné do informačního panelu přidávat za pomoci příkazů:

- **self._simulationInfo.addToAlgData(*<instance prvku>*)**
- **self._simulationInfo.addToElmData(*<instance prvku>*)**

Prvky se vždy vkládají postupně ze shora dolů; pokud se nevejdou již do výřezu, je možné se k nim prorolovat kolečkem myši.

- **SimulationInfoSeparator** *(lib.module)*

  `height = 6`

  Vloží do informačního panelu horizontální oddělení o zadané tloušťce čáry.

- **SimulationInfoImage** *(lib.module)*

  `texture, height = 100`

  Vloží do informačního panelu obrázek (texturu). Velikost obrázku se přepočte dle zadané maximální výšky.

- **SimulationInfoText** *(lib.module)*

  `getTextFunc, height = 20`

  Vloží do informačního panelu text generovaný zadanou funkcí. Velikost textu se bude upravovat podle jeho aktuální délky tak, aby se vešel do šířky informačního boxu a dané výšky. Pokud by ovšem velikost textu měla být příliš malá, zvolí se stanovená minimální čitelná velikost, a tedy text může přetéci vyhrazené místo.

- **SimulationInfoValue** *(lib.module)*

  `parent, label, getValFunc, data = None`

  Vloží do informačního panelu dvouřádkový prvek složený ze zadaného popisku a hodnoty vrácené zadanou funkcí. Zadávaná **data** slouží k předání funkcím měnící hodnotu, které lze nastavit za pomoci třídní metody **setValChangeFunc(valDecFunc, valIncFunc)**. Pokud jsou tyto funkce zadány, objeví se pak kolem hodnoty šipky, kterými ji může uživatel měnit. Argument **parent** musí obsahovat ukazatel na instanci okna – v případě vytváření modulu použijte **self._parent**, kde je tento ukazatel uložen.

- **SimulationInfoButton** *(lib.module)*

  `parent, nameFunc, actionFunc, data = None, height = 20`

  Vloží do informačního panelu textové tlačítko o zadané výšce. Popisek tlačítka se určí z výsledku zadané funkce **nameFunc** a při jeho stisku se provede funkce **actionFunc**, které se předají **data** podobně jako v **SimulationInfoValue**. Do argumentu **parent** taktéž nastavte **self._parent**.

S těmito třídami byste si měli při sestavování informací a nastavení do pravého informačního panelu vystačit. Pokud i přesto zjistíte, že byste potřebovali nějaký nový prvek, je vždy možné si jej doprogramovat, je ovšem nutné jej podědit od třídy **AbstractSimulationInfoValue** nebo jejích potomků.

Třídy elementů simulace

Jedná se o třídy poděděné od **Element** a slouží ke grafickému rozvržení simulace. Některé jako např. **EntityArray** slouží k přímému držení prvků simulace (procesů apod.), jiné jako např. **Direction** slouží zcela jen jako doplňkové a definují pouze prostorové vztahy mezi ostatními elementy.

- **EntityArray** *(lib.gComponents)*

  Vloží do simulace kontejnerové pole pro uchovávání prvků simulace. Zásadní vlastností pole je, že má pevný maximální počet vložených prvků. Všechny argumenty kromě prvního jsou nepovinné a v případě jejich neuvedení se použijí standardní hodnoty.

  Argumenty inicializátoru jsou postupně:

  - **length** – délka pole uvedená v maximálním počtu prvků, které může obsahovat (povinný argument; celočíselná hodnota)
  - **color** – barva orámování pole uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně bílá barva bez průhlednosti)
  - **name** – pojmenování pole (standardně prázdné)

- `horizontal` – zda se má pole vykreslit v horizontálním nebo vertikálním směru (standardně `True`)
- `forwardSense` – příznak smyslu řazení položek v poli (standardně `True`)
- `compareFunc` – porovnávací funkce pro řazení položek v poli (standardně `None` značící řazení FIFO)
- `entityClass` – třída reprezentující prvky simulace (standardně `Entity`, může ovšem obsahovat například `lib.gAdvancedComponents.AdvancedEntity`).
- `entitySize` – velikost jedné buňky pole (standardně 1.0; rozumnými hodnotami jsou násobky 0.5)
- `isReadyFunc` – funkce umožňující regulovat výběr prvku z pole za pomoci metody `pop`. Pokud je tato funkce definována, před každým výběrem se jí metoda `pop` zeptá, zda je prvek opravdu připraven k výběru, v případě zamítnutí vrací metoda `pop` hodnotu `None`, jakoby pole bylo prázdné (využití např. při výběru prvků z fronty blokovaných, kdy žádnému z prvků ještě neskončila blokace). (standardně `None` – žádná kontrola)
- `updateFunc` – funkce, která se zavolá na všechny prvky v poli při zavolání metody `update` (využití např. při aktualizaci uběhlého času pro frontu blokovaných prvků). (standardně `None` – žádné aktualizace)

Nejpotřebnější metody třídy jsou:

- `fillingsList` – property vracející pole dvojic hodnot obsažených prvků; dvojce obsahuje čas naplnění a prvek (`filling`)
- `update` – pokud je nastavena `updateFunc`, provede ji nad všemi prvky pole
- `clear` – vymaže obsah pole
- `sort` – přetřídí prvky pole dle nastavené porovnávací funkce
- `push` – vloží prvek (`filling`) do pole; pokud je pole již plné, vrací `False`
- `getNextPop` – vrátí jeden nebo více prvků, který by se měly vybrat při příštích zavolání metody `pop`, nebo `None`; jaký maximální počet prvků vrátí se určí argumentem `count`
- `pop` – vybere z pole prvek (`filling`) a vrátí jej, nebo `None`
- `getData` – vrátí prvek obsažený v poli dle zadaného ID prvku, nebo `None`
- `removeFilling` – odebere z pole prvek (`filling`), definovaný jeho adresou v paměti

Popis všech metod najdete v automaticky generované nápovědě v podsložce `doc/html`.

- `EntitySet` *(`lib.gComponents`)*

  Vloží do simulace kontejnerové pole bez omezení počtu prvků - množinu. Všechny argumenty kromě prvních dvou jsou nepovinné a v případě jejich neuvedení se použijí standardní hodnoty.

  Argumenty inicializátoru jsou postupně:

  - `startLength` – počet zobrazovaných prvků ze začátku (povinný argument; celočíselná hodnota)
  - `endLength` – počet zobrazovaných prvků od konce (povinný argument; celočíselná hodnota)
  - `interrupt` – délka přerušení mezi začátkem a koncem (standardně 1.0)
  - `color` – barva orámování množiny uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně bílá barva bez průhlednosti)
  - `name` – pojmenování množiny (standardně prázdné)
  - `horizontal` – zda se má množina vykreslit v horizontálním nebo vertikálním směru (standardně `True`)

- `forwardSense` – příznak smyslu řazení položek v poli (standardně `True`)
- `compareFunc` – porovnávací funkce pro řazení položek v poli (standardně `None` značící řazení FIFO)
- `entityClass` – třída reprezentující prvky simulace (standardně `Entity`, může ovšem obsahovat například `lib.gAdvancedComponents.AdvancedEntity`).
- `entitySize` – velikost jedné buňky množiny (standardně 1.0; rozumnými hodnotami jsou násobky 0.5)
- `isReadyFunc` – funkce umožňující regulovat výběr prvku z množiny za pomoci metody `pop`. Pokud je tato funkce definována, před každým výběrem se jí metoda `pop` zeptá, zda je prvek opravdu připraven k výběru, v případě zamítnutí vrací metoda `pop` hodnotu `None`, jakoby množina byla prázdná (využití např. při výběru prvků z fronty blokovaných, kdy žádnému z prvků ještě neskončila blokace). (standardně `None` – žádná kontrola)
- `updateFunc` – funkce, která se zavolá na všechny prvky v poli při zavolání metody `update` (využití např. při aktualizaci uběhlého času pro frontu blokovaných prvků). (standardně `None` – žádné aktualizace)

Nejpotřebnější metody třídy jsou:

- `fillingsList` – property vracející pole dvojic hodnot obsažených prvků; dvojce obsahuje čas naplnění a prvek (`filling`)
- `update` – pokud je nastavena `updateFunc`, provede ji nad všemi prvky množiny
- `clear` – vymaže obsah množiny
- `sort` – přetřídí prvky množiny dle nastavené porovnávací funkce
- `push` – vloží prvek (`filling`) do množiny
- `getNextPop` – vrátí jeden nebo více prvků, který by se měly vybrat při příštích zavolání metody `pop`, nebo `None`; jaký maximální počet prvků vrátí se určí argumentem `count`
- `pop` – vybere z množiny prvek (`filling`) a vrátí jej, nebo `None`
- `getData` – vrátí prvek množiny dle zadaného ID prvku, nebo `None`
- `removeFilling` – odebere z množiny prvek (`filling`), definovaný jeho adresou v paměti

Popis všech metod najdete v automaticky generované nápovědě v podsložce `doc/html`.

- `PriorityGroup` *(`lib.gAdvancedComponents`)*

  Vloží do simulace specializovaný kontejner reprezentující prioritní frontu (vnitřně se jedná o skupinu množin propojených šipkami). Všechny argumenty kromě prvního jsou nepovinné a v případě jejich neuvedení se použijí standardní hodnoty.

  Prioritní fronty fungují dle očekávání jen s prvky `AvancedEntity`; standardní prvky se považují za prvky s nejnižší prioritou.

  Argumenty inicializátoru jsou postupně:

  - `length` – délka jednotlivých polí uvedená v maximálním počtu prvků, které může obsahovat (povinný argument; celočíselná hodnota)
  - `queueCount` – počet prioritních front; měl by odpovídat počtu priorit (standardně `5`)
  - `firstColor` – barva první fronty uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně bílá barva bez průhlednosti)
  - `lastColor` – barva poslední fronty uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně `None` – stejná jako `firstColor`)

- `inColor` – vstupní barva propojení front uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně bílá barva bez průhlednosti)
- `outColor` – výstupní barva propojení front uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně `None` – stejná jako `inColor`)
- `horizontal` – zda se má skupina propojit v horizontálním nebo vertikálním směru; ovlivňuje i směr front (standardně `True`)
- `forwardSense` – příznak smyslu řazení prvků ve frontách (standardně `True`)
- `compareFunc` – porovnávací funkce pro řazení položek v polích (standardně `None` značící řazení podle priorit a v rámci stejné priority FIFO)
- `useSet` – příznak použití množin nebo polí pro reprezentaci prioritních front (standardně `True`)
- `entitySize` – velikost jedné buňky front (standardně 1.0; rozumnými hodnotami jsou násobky 0.5)
- `labelsArray` – pole pojmenování prioritních front; počet pojmenování musí odpovídat `queueCount` (standardně `None`)
- `isReadyFunc` – funkce umožňující regulovat výběr prvku za pomoci metody `pop`. Pokud je tato funkce definována, před každým výběrem se jí metoda `pop` zeptá, zda je prvek opravdu připraven k výběru, v případě zamítnutí vrací metoda `pop` hodnotu `None`. (standardně `None` – žádná kontrola)
- `updateFunc` – funkce, která se zavolá na všechny prvky při zavolání metody `update`. (standardně `None` – žádné aktualizace

Nejpotřebnější metody třídy jsou:

- `fillingsList` – property vracející pole dvojic hodnot obsažených prvků; dvojce obsahuje čas naplnění a prvek (`filling`)
- `update` – pokud je nastavena `updateFunc`, provede ji nad všemi prvky
- `clear` – vymaže obsah prioritních front
- `sort` – přetřídí prvky dle nastavené porovnávací funkce
- `push` – zatřídí prvek (`filling`) do prioritních front
- `getNextPop` – vrátí jeden nebo více prvků, který by se měly vybrat při příštích zavolání metody `pop`, nebo `None`; jaký maximální počet prvků vrátí se určí argumentem `count`
- `pop` – vybere z prioritních front prvek (`filling`) a vrátí jej, nebo `None`
- `getData` – vrátí prvek prioritních front dle zadaného ID prvku, nebo `None`
- `removeFilling` – odebere prvek (`filling`) definovaný jeho adresou v paměti z prioritních front

Popis všech metod najdete v automaticky generované nápovědě v podsložce `doc/html`.

- `EntityGroup` *(lib.gComponents)*

  Propojí několik elementů do skupiny. Všechny argumenty kromě prvního jsou nepovinné a v případě jejich neuvedení se použijí standardní hodnoty.

  Argumenty inicializátoru jsou postupně:

  - `elementsArray` – pole instancí elementů, které se mají seskupit (povinný argument)
  - `inColor` – vstupní barva propojení elementů skupiny uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně bílá barva bez průhlednosti)
  - `outColor` – výstupní barva propojení elementů skupiny uvedená v RGBA s reálnými hodnotami od 0.0 do 1.0 (standardně `None` – stejná jako `inColor`)
  - `horizontal` – zda se má skupina propojit v horizontálním nebo vertikálním směru (standardně `True`)

- `forwardSense` – příznak smyslu řazení elementů pro vykreslení (standardně `True`)

Nejpotřebnější metody třídy jsou:

- `elementsArray` – property vracející seznam seskupených elementů
- `recountSize` – přepočte vzájemné pozice a propojení seskupených elementů
- `fillingsList` – property vracející pole dvojic hodnot prvků obsažených v seskupených elementech; dvojce obsahuje čas naplnění a prvek (`filling`)
- `update` – zavolá metodu `update` na všech seskupených elementech
- `clear` – zavolá metodu `clear` na všech seskupených elementech
- `getData` – vrátí prvek obsažený v některém ze seskupených elementů dle zadaného ID prvku, nebo `None`
- `removeFilling` – odebere z množiny prvek (`filling`), definovaný jeho adresou v paměti

Popis všech metod najdete v automaticky generované nápovědě v podsložce `doc/html`.

- `Direction` *(lib.gComponents)*

  Propojí právě dva elementy a vizuálně mezi nimi definuje vztah směru. Všechny argumenty kromě prvních dvou jsou nepovinné a v případě jejich neuvedení se použijí standardní hodnoty.

  Argumenty inicializátoru jsou postupně:

  - `elementFrom` – instance prvního elementu (povinný argument)
  - `elementTo` – instance druhého elementu (povinný argument)
  - `direction` – směr napojení ve vztahu k prvnímu elementu (standardně "vMid|right"; povolené hodnoty jsou "left", "right", "top", "bottom", "vMid", "hMid", a je možné vždy kombinovat horizontální a vertikální pozicování za pomoci svislítka)
  - `length` – určuje nejmenší horizontální / vertikální vzdálenost prvků; nejedná se o délku čáry (standardně 1.0; rozumné je používat násobky 0.5)
  - `arrows` – určuje zda se bude na spojnici elementů vykreslovat šipka a v jakém směru (standardně ">"; rozpoznávané hodnoty jsou ">" a "<" s tím, že je možné je uvést v řetězci obě, pak je šipka obousměrná; pro čáru bez šipky stačí uvést prázdný řetězec)

  Nejpotřebnější metody třídy jsou:

  - `getData` – vrátí prvek obsažený v některém z podřízených elementů dle zadaného ID prvku, nebo `None`

  Popis všech metod najdete v automaticky generované nápovědě v podsložce `doc/html`.

Za pomoci těchto tříd by měl být každý schopen sestavit rozvržení simulace dle svých představ. V případě nutnosti je možné si doprogramovat svou vlastní třídu, je ovšem silně doporučováno ji založit na třídách již existujících a poděděním si ji doplnit jen nutnou funkcionalitu.

Rozšířená entita

Rozšířená entita (`AdvancedEntity` a výplň `AdvancedEntityFilling`) navíc od entity standardní umožňuje jednoduše pracovat s prioritou a se statusem, které také při svém zobrazení vizualizuje. Obě třídy (`Priority` i `Status`) podporují získání své instance z prvku za pomoci třídní metody `get`. Použití je pak zcela jednoduché, a to

například příkazem: `prio = lGAC.Priority.get(filling)`, kde `lGAC` je alias pro `lib.gAdvancedComponents` a `filling` je instance třídy `AdvancedEntityFilling`. Pokud by se této metodě dostala jako argument instance neregistrované třídy, jako např. `EntityFilling`, metoda vrátí `None`.

K oběma třídám naleznete podrobnější popis v automaticky generované nápovědě v podsložce `doc/html`.

6. Vytvoření instalace

a. Kontrola a nezbytné úpravy před vytvořením instalace

Pokud máte možnost mít nainstalovaných více instancí systému (například ve virtuálních strojích), doporučuje se mít jednu instanci na vývoj, jednu čistou jen s nutnými programy na vytváření balíčků a třetí čistou na test instalace nově vytvořeného balíku.

Před samotným vytvořením nové verze instalace je dobré překontrolovat několik věcí:

- Pokud používáme nějaký verzovací systém jako `CVS`, `SVN`, `Git` apod., aktualizujeme si, na balící instanci systému, na aktuální verzi stabilní verze programu (`update`).
- Vyzkoušíme si program spustit a naklikat v něm náhodně alespoň několik scénářů, abychom nebalili nefunkční verzi. Při úpravách programu nebo jeho knihoven je nutností zkontrolovat správnou funkčnost každého modulu – nechceme, aby naše úpravy znefunkčnily staré moduly.
- Zkontrolovat `changelog` (v Linuxu za pomoci příkazu `dch`, nebo přidat nový záznam za pomoci `dch -i` v hlavní složce programu; ve Windows® editujte soubor například PSPadem, který podporuje utf-8, a pro vytvoření nového záznamu zkopírujte záznam předchozí a upravte jej – soubor vyžaduje přesnou strukturu). V tomto souboru by měly být zaznamenány veškeré změny provedené v kódu tak, aby kdokoli jiný mohl jednoduše zjistit, kdy a kde se co měnilo. Z tohoto souboru se také bere verze pro nově vytvářený Linuxový balík, je tedy nezbytné jej udržovat aktuální.
- Aktualizovat verzi programu v `install.bat` nastavením správné hodnoty proměnné `pVersion` (4. řádek skriptu).
- Dobrým zvykem je pak ještě přegenerovat automatickou nápovědu k aktuálním zdrojovým kódům za pomoci příkazu `doxygen doxygen.cfg` z hlavní složky programu a v případě chyb nebo upozornění, je před vytvořením nového balíku vyřešit.
- Pokud jsme při kontrole výše zmíněného provedli nějaké změny, je třeba je promítnout (`commit`) i do verzovacího systému a opět si vytáhnout (`update`) aktuální verzi.

Pokud jsme zkontrolovali a případně opravili vše výše zmíněné, můžeme přejít k samotnému vytvoření balíku.

b. Balíčkování v Linuxu (Ubuntu)

Pokud máme správnou verzi v souboru `changelog` (viz Kontrola a nezbytné úpravy před vytvořením instalace), můžeme přejít k vytvoření nového balíku, a to zadáním příkazu `debuild` z hlavní složky programu. Pokud vše proběhlo v pořádku, mělo by se v nadřazené složce hlavní složky programu objevit pět nových souborů a to:

- `alg-simulator_<verze>.dsc` – Soubor s popisem archivu zdrojových souborů
- `alg-simulator_<verze>.tar.gz` – Archív zdrojových souborů
- `alg-simulator_<verze>_all.deb` – Instalační balíček programu pod Debian

- `alg-simulator_<verze>_amd64.build` – Soubor se záznamem průběhu balíčkování
- `alg-simulator_<verze>_amd64.changes` – Soubor s popisem instalačního balíčku a výčtem změn mezi verzemi

Instalační balíček pak přesuneme na instanci systému určenou k testu, kde balíček nainstalujeme (viz uživatelská příručka) a ještě jednou vyzkoušíme funkčnost programu a jeho modulů. Pokud je vše v pořádku, můžeme balíček distribuovat.

c. Vytvoření instalace ve Windows®

Pokud máme nastavenou správnou verzi v souboru `install.bat`, (viz Kontrola a nezbytné úpravy před vytvořením instalace), můžeme jej spustit a nechat vytvořit instalační archív. Pokud vše proběhlo v pořádku, měl by se v podsložce `win` objevit archív `alg-simulator_0.1.5.exe` (s aktuální verzí v názvu).

Tento archív pak přesuneme na instanci systému určenou k testu, kde balíček nainstalujeme (viz uživatelská příručka) a ještě jednou vyzkoušíme funkčnost programu a jeho modulů. Pokud je vše v pořádku, můžeme archív distribuovat.