Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Grafické animace modelů pro podporu výuky operačních systémů

Bakalářská práce

Autor: Vojtěch Havlík

Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Josef Horálek, Ph.D.

Hradec Králové duben 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové 15.4.2021 Vojtěch Havlík

Poděkování:

Děkuji vedoucímu bakalářské práce Mgr. Josefu Horálkovi, Ph.D. za metodické vedení práce, za pomoc a rady při zpracování této práce.

Anotace

Bakalářská práce se zabývá problematikou operačních systémů. Jedná se o popis témat, u kterých byly části bakalářské práce graficky zpracovány pro podporu výuky operačních systémů. Práce je rozdělena do tří větších témat, která jsou podrobněji popsána. Jedná se o témata: adresní prostor a abstrakce paměti, virtuální paměť, procesy. Práce se zabývá těmito tématy více do hloubky a animace k těmto tématům slouží pro lepší porozumění, či dosažení lepší představivosti studenta při výuce operačních systémů. Animace jsou voleny převážně tam, kde jednoduchý statický obrázek nedokáže dostatečně popsat řešenou problematiku a student může mít problémy s představou konkrétního problému, který by měl být řešen. Popis, který je obsažen v bakalářské práci by měl studenta uvést do dané problematiky a pomocí animací mu poskytnout další studijní materiál, který bude vést k lepšímu porozumění.

Annotation

Title: Graphic animations of models to support the learning of operating systems

The bachelor thesis works with the issue of operating systems. This is a description of topics where some parts of the bachelor's thesis were graphically processed to support the learning of operating systems. The bachelor thesis is divided into three major topics. These topics are: address space and memory abstraction, virtual memory, processes. These topics are covered in more details and the animation serves for a better understanding or to achieve a better imagination of student in subject of operating systems. Animations are chosen mainly where a simple static image cannot sufficiently describe the problem and the student may have problems with the main idea of a specific problem that should be solved. Description contained in this work should introduce the student to the issue and use animations to provide additional study material that will lead to a better understanding.

Obsah

1 Úvod ... 10

2 Cíl práce ... 12

3 Metodika zpracování ... 13

4 Teoretická část ... 14

4.1 Adresní prostor a abstrakce paměti ... 14

4.1.1 Adresní prostor ... 14

4.1.2 Abstrakce paměti ... 15

4.1.3 Swapování (angl. Swapping) ... 20

4.1.4 Správa volné paměti ... 23

4.2 Virtuální paměť ... 32

4.2.1 Stránkování a MMU ... 33

4.2.2 Tabulka stránek ... 38

4.2.3 Zrychlení stránkování ... 40

4.3 Procesy ... 48

4.3.1 Blok řízení procesu ... 48

4.3.2 Model procesu ... 51

4.3.3 Vytvoření procesu ... 53

4.3.4 Ukončení procesu ... 55

4.3.5 Stavy procesu ... 56

4.3.6 Plánování procesů ... 58

4.3.7 Strategie plánování procesů ... 60

5 Praktická část ... 67

5.1 Animační program Synfig Studio ... 67

5.1.1 Vytvoření nového projektu ... 67

5.1.2 Nástroje programu ... 69

5.1.3 Vlastnosti objektů a časová osa ... 69

5.1.4 Uspořádání projektu do balíčku, vrstvení ... 71
5.1.5 Export projektu ... 72
5.2 Animace vytvořené programem Synfig Studio ... 74
5.2.1 Animace pro 1. téma – adresní prostor a abstrakce paměti ... 74
5.2.2 Animace pro 2. téma – virtuální paměť ... 81
5.2.3 Animace pro 3. téma – procesy ... 87
6 Shrnutí výsledků ... 90
7 Závěry a doporučení ... 91
8 Seznam použité literatury ... 92
9 Přílohy ... 93

Seznam obrázků

Obr. 1. Souvislý blok fyzické paměti 15

Obr. 2. Modely pro ilustraci paměti 15

Obr. 3. Problém relokace 17

Obr. 4. Ilustrace statické relokace 18

Obr. 5. Base a limit registry 19

Obr. 6. Dynamická relokace 20

Obr. 7. Pevné oddíly 21

Obr. 8. (a) Řazení do samostatných front; (b) Jediná fronta pro paměť 22

Obr. 9. Dynamický oddíl 22

Obr. 10. Ukázka správy paměti pomocí bitmapy 24

Obr. 11. Ukázka správy paměti pomocí spojeného seznamu 25

Obr. 12. Znázornění situací po ukončení nebo přesunutí procesu 26

Obr. 13. Ukázka algoritmu First fit 27

Obr. 14. Ukázka algoritmu Next fit 28

Obr. 15. Ukázka algoritmu Best fit 29

Obr. 16. Ukázka algoritmu Worst fit 30

Obr. 17. Ilustrace fungování jednotky MMU 34

Obr. 18. Relace mezi virtuálním adresním prostorem a fyzickým adresním prostorem 35

Obr. 19. Změna mapování při chybě stránky 37

Obr. 20. Ukázka mapování pomocí MMU 38

Obr. 21. Záznam tabulky stránek 39

Obr. 22. Ukázka TLB 42

Obr. 23. Víceúrovňová tabulka stránek 44

Obr. 24. Porovnání klasické tabulky stránek s převrácenou tabulkou stránek 46

Obr. 25. Vrstvy počítačového systému 49

Obr. 26. Ukázka bloku řízení procesu 50

Obr. 27. (a) - Multiprogramování čtyř programů; (b) - Konceptuální model čtyř nezávislých, sekvenčních procesů; (c) - Pouze jeden proces je aktivní v jednu chvíli 51

Obr. 28. Stavy procesu 56

Obr. 29. Nejnižší vrstva procesně strukturovaného operačního systému zpracovává přerušení a plánování (plánovač) 57

Obr. 30. Plánování a přechody stavu procesu 59

Obr. 31. Znázornění fronty 63

Obr. 32. Znázornění zásobníku 64

Obr. 33. (a) - Seznam spustitelných procesů; (b) - Seznam spustitelných procesů potom, co proces B vyčerpal své kvantum 65

Obr. 34. Karta vlastnosti, výchozí nastavení 68

Obr. 35. Prázdné okno nového projektu 68

Obr. 36. Nástroje programu 69

Obr. 37. Vlastnosti objektu 70

Obr. 38. Okno pro práci s časovou osou 70

Obr. 39. Aktivovaný animační stav 71

Obr. 40. Projekt strukturován do balíčků 71

Obr. 41. Nastavení renderování 72

Obr. 42. Renderování do formátu MP4 73

Obr. 43. Paměť – algoritmus first fit 77

Obr. 44. Paměť – algoritmus next fit 78

Obr. 45. Paměť – algoritmus best fit 79

Obr. 46. Paměť – algoritmus worst fit 81

Obr. 47. Mapování 83

Obr. 48. Mapování – chyba stránky 84

Obr. 49. Mapování – aktualizace mapování 85

Obr. 50. Tabulka stránek 86

Seznam tabulek

Tabulka 1. Charakteristiky procesu 49

Tabulka 2. Shrnutí typů plánování 58

Tabulka 3. Plánovací kritéria 61

1 Úvod

Bakalářská práce se skládá ze dvou částí. Z teoretické části, kde jsou popsány problematiky, kterými se bakalářská práce zabývá a praktickou částí, kde jsou popsány jednotlivé animace sloužící pro podporu výuky operačních systémů.

Teoretická část bakalářské práce se zabývá převážně třemi oblastmi z problematiky operačních systémů. Tato témata byla vybrána z hlediska oblastí, které se řeší převážně při výuce předmětu Operační systémy 1. K těmto tématům jsou zpracovány animace pro lepší porozumění a znázorní dané situace. Animace jsou voleny převážně tam, kde mohou pomoci s podporou výuky operačních systémů pro získání lepší představivosti studentů. Zmíněná témata v bakalářská práci jsou: adresní prostor a abstrakce paměti, virtuální paměť, procesy. První dvě témata by se dala shrnout jedním tématem, a to paměť počítače. Z hlediska lepší orientace je zde toto téma rozděleno a zpracováno zvlášť.

První téma se zabývá adresním prostorem a abstrakcí paměti. Úvod je věnován vysvětlováním pojmů, jejich smyslu a co si student může pod těmito pojmy představit. Pojem adresní prostor zde slouží jako úvod do této problematiky a větší část je věnována abstrakci paměti. Zde jsou vysvětleny otázky, s kterými se můžeme setkat při řešení této problematiky. Toto se týká převážně statické a dynamické relokace, která nám řeší problém abstrakce paměti. Dále je zde zmíněn mechanismus swapování, jeho problematika a následná práce s hlavní pamětí počítače. Na závěr je zde zmíněna správa volné paměti, kde jsou uvedeny i algoritmy pro přiřazování volné paměti procesům, resp. alokování procesů do hlavní paměti počítače.

Druhé téma volně navazuje na první téma a dále se zabývá řešením správy volné paměti počítače. Jedná se o virtuální paměť. V tomto tématu je převážná část věnována stránkování a fungování jednotky MMU. Toto téma se dále zabývá dvěma zásadními problémy, které mohou nastat v problematice virtuální paměti. Jedná se o rychlost mapování adres virtuálního adresního prostoru na fyzický adresní prostor a velikost virtuálního adresního prostoru, která je spojena s velikostí tabulky stránek.

Třetí téma zde vytváří novou problematiku, jedná se o téma procesy. V tomto tématu jsou popisovány základní principy procesů v operačním systému jako je jejich vytváření, či ukončení. Dále jsou zmiňovány stavy procesy, přes které se naváže na problematiku plánování procesů. Na závěr jsou zde zmíněné tři strategie plánování procesu, s kterými operační systém pracuje.

V praktické části bakalářské práce je popsán program Synfig Studio, ve kterém byly animace vytvářeny a je zde popsána základní práce s tímto programem. Dále tato část obsahuje doprovodný text k animacím, které byly vytvořeny pro konkrétní problematiky k tématům, která jsou rozebírána v teoretické části. Tento text slouží k popisu, co daná animace představuje.

2 Cíl práce

Smyslem této práce je dosáhnout co nejlepšího porozumění zmíněné problematiky, která je obsažena v této bakalářské práci. Práce by měla poskytnout dostatečný materiál s vysvětlením řešené problematiky a dostatečné informace k pochopení animací, které byly v průběhu bakalářské práce vytvořeny. Tento materiál by měl v plném rozsahu poskytnout další studijní materiál pro podporu výuky operačních systémů.

3 Metodika zpracování

Na základě konzultací s vedoucím bakalářské práce byly identifikovány oblasti, které mohou být pro studenty obtížně pochopitelné a je vhodné v těchto případech studentům poskytnout další studijní materiál. Doplňující materiál spočívá v textovém doprovodu a v samotných animací, které byly vytvořeny za tímto účelem.

Vybrané oblasti byly zpracovány a podrobně popsány na základě studia relevantních zdrojů. Výsledné texty byly z pohledu věcného i terminologického s vedoucím práce konzultovány.

Animace byly zpracovávány v programu Synfig Studio. Jedná se o bezplatný 2D animační software s otevřeným zdrojovým kódem. Výsledné animace jsou renderovány do formátu GIF, který lze přiložit ke studijnímu materiálu v kurzu předmětu na portálu Blackboard. Animace byly také vyrenderovány do formátu MP4 pro možnost zastavení, či přetočení animace, sloužící k lepší manipulaci pro vyučujícího ale i studenta.

4 Teoretická část

V teoretické části bakalářské práce jsou popsány detailněji témata, kterými se bakalářská práce zabývá. Zabývá se třemi hlavními tématy a těmi jsou: adresní prostor a abstrakce paměti, virtuální paměť, procesy. K získání potřebných informací o dané problematice byla použita uvedená literatura.

4.1 Adresní prostor a abstrakce paměti

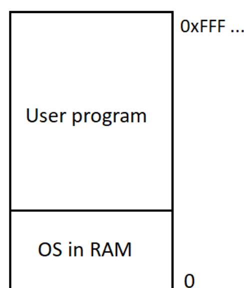
První téma bakalářské práce se věnuje především problematice abstrakce paměti. Je zde zmíněn adresní prostor, co si pod tím představit a jak je spojován s pojmem paměť. Následně je vysvětlena abstrakce paměti a její problémy, které navazují na případná řešení, která se zde vyskytují. Závěr tohoto tématu je věnován správě volné paměti, kde jsou představeny algoritmy pro správu volné paměti.

4.1.1 Adresní prostor

Hlavní paměť počítače, paměť RAM, je důležitým zdrojem, který je třeba pečlivě obsluhovat. V multiprogramovacím systému je hlavní paměť rozdělena na dvě části. Jedna část je určena pro operační systém a druhá část náleží aktuálně vykonávanému programu. Tato paměť by měla být rozdělena tak, aby dokázala vyhovět více procesům. Úkol dělení je prováděn dynamicky operačním systémem a je znám jako správa paměti.

Adresní prostor je označení pro souvislý rozsah fyzických či virtuálních adres. Je tvořen sadou jedinečných identifikátorů, pomocí kterých nemůže dojít k záměně jednotlivých adres. Pro lepší představu můžeme mít např. adresní prostor fyzických osob. Takovýto adresní prostor se může skládat z míst jako je ulice, obec, stát atd. Identifikátory adresy mohou být podobné, ovšem pokud se neshodují všechny uvedené hodnoty, poté ukazují vždy na jinou adresu v tomto adresním prostoru.

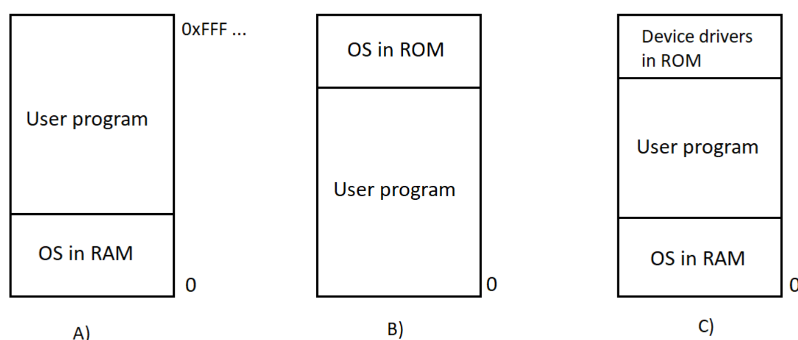
Jedná se o rozsah adres, které operační systém přiřadí uživateli nebo běžícímu procesu. Můžeme tedy říci, že se jedná o oblast souvislých adres, které jsou k dispozici pro provádění pokynů a ukládání dat. Rozsah adres začíná na nule a může se rozšiřovat na maximální adresu, která je určena architekturou operačního systému.



Obr. 1. Souvislý blok fyzické paměti
Zdroj: vlastní zpracování

4.1.2 Abstrakce paměti

Hardware a operační systém spravují paměť tak, aby se tvářila jako jedna souvislá entita. Paměť máme tedy reprezentovanou jako jeden souvislý blok fyzické paměti. Jedná se o sadu adres, které začínají od nuly do maxima, které je limitováno architekturou operačního systému. Pro ilustraci paměti máme znázorněné tři modely na Obr. 2.



Obr. 2. Modely pro ilustraci paměti
Zdroj: vlastní zpracování

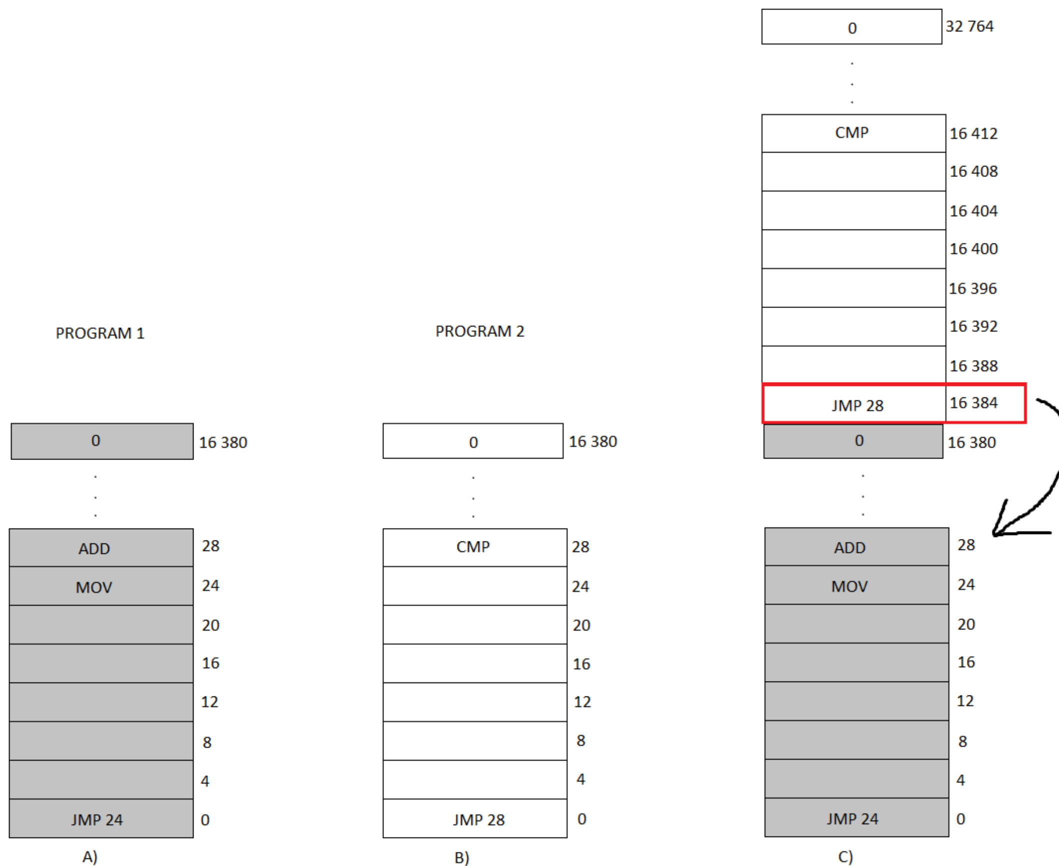
Operační systém se může nacházet v dolní části paměti RAM nebo v horní části paměti ROM. V horní části paměti ROM se také mohou nacházet ovladače zařízení. První model je používán zřídka. Druhý model byl použit na vestavěných (angl. Embedded) systémech. Třetí model začaly využívat jedny z prvních osobních počítačů, kde se část systému v paměti ROM nazývá BIOS. Fungování v tomto modelu paměti je takové, že po zadání příkazu uživatelem operační systém zkopíruje požadovaný program z disku do paměti a provede jej. Po dokončení programu operační systém zobrazí příkazový řádek a očekává nový příkaz. Když operační systém přijme nový příkaz, načte jej do paměti a přepíše předchozí.

Každý program tedy jednoduše vidí celou fyzickou paměť, kde každá adresa odpovídá buňce obsahující počet bitů. V paměti nelze mít dva programy v jednom časovém okamžiku, v tomto případě by nastal problém. Pokud je ve stejný čas vykonána instrukce jiným programem, potom hodnota prvního vykonaného programu bude přepsána. To jednoduše znamená, že kam zapíše jeden program, druhý program by místo přepsal a nic by nebylo funkční, což vede k chybě programu. To je hlavní důvod, který vede k abstrakci paměti. Abstrakce paměti spočívá v tom, že každý proces má svůj vlastní adresní prostor.

Ovšem i bez abstrakce paměti je možné spouštět více programů v jeden časový okamžik. Operační systém uloží celý obsah paměti do souboru na disku, poté přivede nový program a spustí jej. Dokud je v paměti pouze jeden program, v jeden časový okamžik, nedojde ke konfliktu. Jedná se o tzv. mechanismus swapování neboli výměny (angl. Swapping), který je popsán později v podkapitole 4.1.3.

Zpátky ke spuštění programů bez abstrakce paměti. S přidáním speciálního hardwaru lze spouštět více programů současně i bez použití zmíněného swapování. Máme paměť, která je rozdělena do 2kB bloků a každému tomuto bloku je přidělen 4bitový ochranný klíč, který je uchován ve speciálních registrech uvnitř CPU. Například paměť o velikosti 1 MB je rozdělena na 512 takových to bloků. Těmto blokům jsou přiděleny 4bitové klíče. Velikost tohoto registru by tedy činila 256 kB. Toto řešení mělo ovšem jednu nevýhodu. Nevýhoda spočívá v problému relokace.

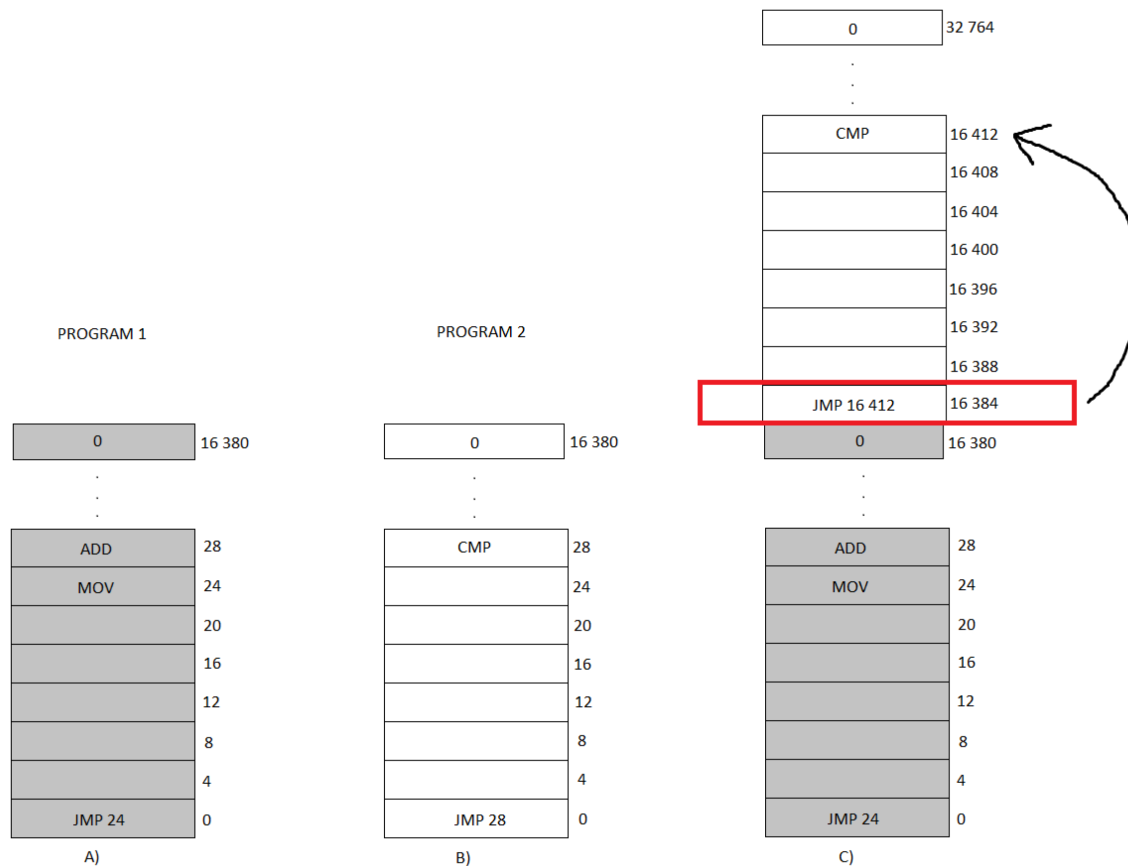
Pro příklad mějme dva programy, každý o velikosti 16 kB, jak je znázorněno na Obr. 3. První program (znázorněn šedým vyplněním) začíná instrukcí JMP 24, která obsahuje instrukci MOV. Druhý program začíná instrukcí JMP na adresu 28, kde máme instrukci CMP. Takto vypadá fungování těchto programů, pokud je máme oddělené. Když jsou tyto programy načteny do paměti, lze je spustit, protože mají různé paměťové klíče a žádný z nich nemůže poškodit druhý. Operační systém může rozhodnout, že spustí druhý program, který byl načten nad prvním programem na adrese 16 384. První provedenou instrukcí je tedy JMP 28, která skončí na instrukci v prvním programu ADD namísto instrukce, která je očekávána a tím je instrukce CMP. Zásadní problém je v tom, že oba programy odkazují na absolutní fyzickou adresu. Musíme dosáhnout toho, aby každý program odkazoval na privátní sadu místních adres. Řešením tohoto problému bylo zavedení statické a dynamické relokace.



Obr. 3. Problém relokace
Zdroj: vlastní zpracování

4.1.2.1 Statická relokace

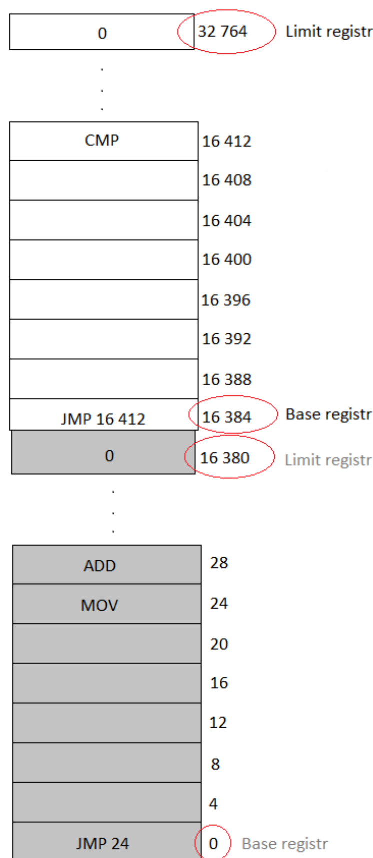
Statická relokace spočívá v přičítání konstanty ke každé adrese programu. Pokud by tedy byl program načten na adrese 16 384, byla by během načítání procesu přidána tato konstanta ke každé adrese programu. Zavaděč ovšem musí rozlišovat adresu a konstantu. Jedná se o řešení, které zpomaluje načítání a vyžaduje informaci navíc ve všech spustitelných programech, která označuje, která „slova“ obsahují přemístitelné adresy a která nikoli. Nesmí být například přemístěna instrukce MOV REGISTER 1, 28, která přesune obsah paměti na pozici 28 na pozici REGISTER 1. Proto je důležité, aby zavaděč rozlišoval adresu a konstantu. Příklad přičtení konstanty je uveden na Obr. 4.



Obr. 4. Ilustrace statické relokace
Zdroj: vlastní zpracování

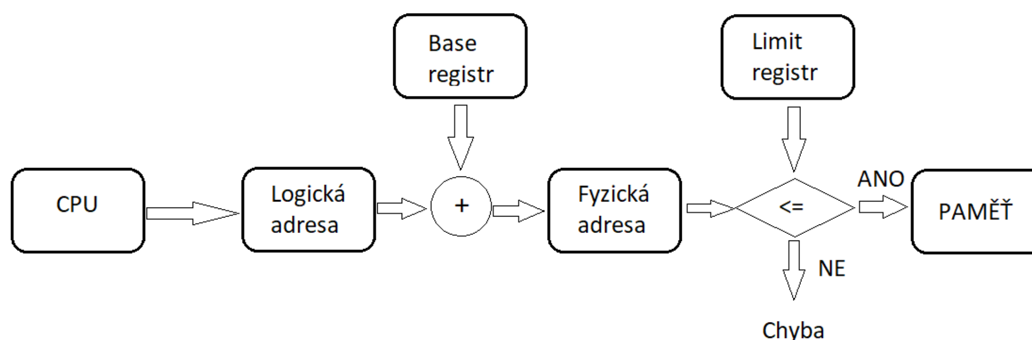
4.1.2.2 Dynamická relokace

Dynamická relokace funguje jednoduchým způsobem, kdy se mapuje adresní prostor každého procesu na jinou část fyzické paměti. CPU je v tomto případě vybaven dvěma speciálními hardwarovými registry. Programy se načítají do po sobě jdoucích paměťových míst. Tyto speciální registry se nazývají base registr a limit registr. Base registr nám označuje počáteční adresu programu ve fyzické paměti. Limitní registr značí délku programu. Registry jsou znázorněny na Obr. 5.



Obr. 5. Base a limit registry
Zdroj: vlastní zpracování

Při spuštění programu se base registr načte s fyzickou adresou, kde začíná. Limitní registr se načte s délkou programu. Pokaždé, když proces odkazuje na paměť za účelem vykonání instrukce, čtení či zápisu, CPU automaticky přidá base hodnotu na adresu generovanou procesem před odesláním adresy na paměťovou sběrnici. Současně kontroluje, zda je nabízená adresa větší nebo rovna než hodnota v limitním registru. V kladném případě je generována chyba a přístup je zamítnut. V opačném případě je adresa poslána na paměťovou sběrnici. Například na stejné situaci jako v předchozím příkladě bude hardware zacházet s instrukcí JMP 28 tak, že se jedná o JMP 16 412 a přemístíme se na instrukci CMP, jak je podle programu očekáváno. Nevýhoda base a limit registrů je znázorněna na Obr. 6, kdy je potřeba provádět sčítání a porovnávání při každém odkazu na paměť.



Obr. 6. Dynamická relokace
Zdroj: vlastní zpracování

Více podrobností k adresování bez abstrakce paměti, k dynamické a statické relokaci lze nalézt v knize Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 176, kap.: 3.1 No memory abstraction, str. 179, kap.: 3.2 A memory abstraction: Address spaces).

4.1.3 Swapování (angl. Swapping)

Jak již bylo zmíněno výše, pojem swapování označuje mechanismus pro výměnu procesů z hlavní paměti do sekundární paměti, nejčastěji na disk. Jedná se pouze o dočasné přesunutí, později můžeme procesy přesunout zpět do hlavní paměti. Výměnu provádí jádro operačního systému. Pohyb dat mezi diskem a operační pamětí má značné režijní náklady. V tomto případě je nejlepším postupem zvýšení fyzické paměti RAM, než aby systém umožňoval neustálé „žonglování" s daty mezi diskem a pamětí.

Například při samotném spouštění systému může být spuštěno mnoho procesů. Aplikace nainstalována na systému Windows často vydává příkazy tak, že při následném bootování systému bude spuštěn proces, který kromě kontroly aktualizace nedělá vůbec nic. Jiné procesy na pozadí mohou kontrolovat příchozí poštu, síťová připojení a další možné operace. Toto vše se děje ještě před spuštěním prvního uživatelského programu. V důsledku náročnosti uživatelských programů na paměť vyžaduje udržování všech procesů v paměti po celou dobu velké množství paměti a program nelze provést, pokud pro něj není dostatek volné paměti. Právě proto byl vyvinut mechanismus swapování. Dá se shrnout do třech následujících bodů:

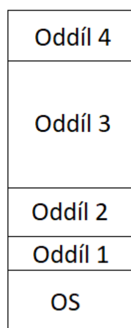
1. Operační systém uloží celou paměť na disk
2. Operační systém přivede program do paměti
3. Operační systém spustí program

Nečinné procesy jsou většinou uloženy na disku, takže nezabírají žádnou paměť, když nejsou aktuálně spuštěny. Další takovou strategií je virtuální paměť, která je vysvětlena v kapitole 4.2.

Pro mechanismus swapování jsou využívány dva způsoby implementace. Jedná se o Pevné oddíly (angl. Fixed partition) a Dynamické oddíly (angl. Dynamic partition).

4.1.3.1 Pevné oddíly

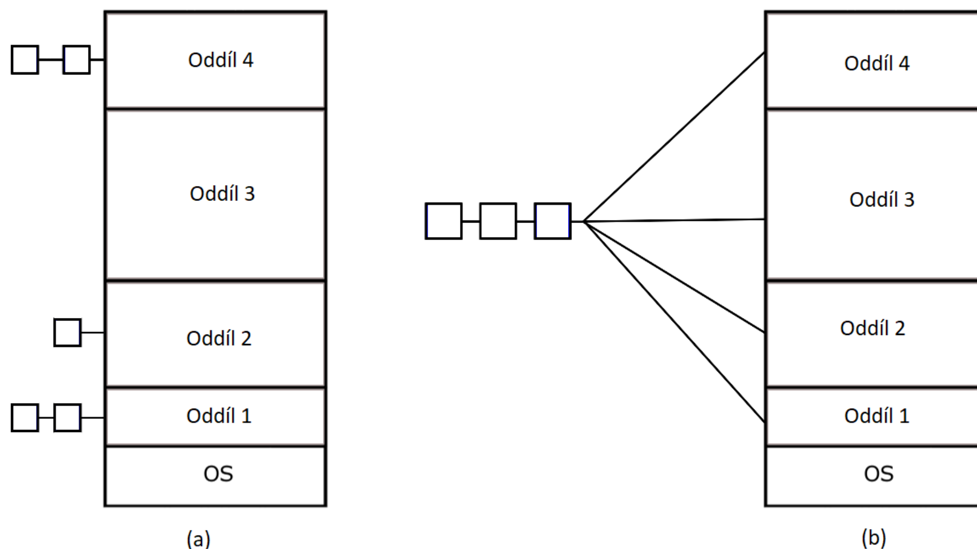
V případě pevných oddílů je paměť rozdělena do pevných velikostí jednotlivých oddílů, jak je znázorněno na Obr. 7. Velikost může být stejná nebo odlišná pro různé oddíly. Každý tento oddíl může přijmout právě jeden proces, tudíž jeden proces zabere právě jeden oddíl. Hranice těchto oddílů nejsou pohyblivé a zůstávají fixní. Kdykoli je potřeba, aby byl proces načten do paměti, je nalezen dost velký blok na to, aby mohl být proces alokován. Tento konkrétní oddíl bude poté přidělen procesu. Pokud není místo, kam bychom mohli proces alokovat, musí čekat a bude vložen do fronty. Řazení do fronty může probíhat dvěma způsoby. Jedná se o samotné fronty pro oddíly, nebo jednu jedinou frontu pro celou paměť.



Obr. 7. Pevné oddíly
Zdroj: vlastní zpracování

Nevýhoda řazení do samotných front pro jednotlivé oddíly se projeví, když je fronta pro velký oddíl prázdná, ale fronta pro malý oddíl je plná, jako je tomu u oddílů 1 a 3 na Obr. 8(a). Zde si malé procesy musí počkat, než se dostanou do paměti, i když je spousta volné paměti k dispozici.

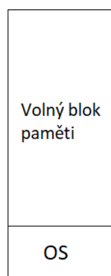
Alternativním uspořádáním je udržování pouze jediné fronty pro celou paměť, jak je tomu na Obr. 8(b). Kdykoli se oddíl uvolní, proces který se nachází ve frontě nejdéle a který nepřesahuje velikost tohoto oddílu, může být načten do prázdného oddílu a následně spuštěn.



Obr. 8. (a) Řazení do samostatných front; (b) Jediná fronta pro paměť
Zdroj: vlastní zpracování

4.1.3.2 Dynamické oddíly

V případě dynamických oddílů je paměť sdílena mezi operačním systémem a současně běžícími procesy. Zpočátku je paměť považována za jeden velký blok paměti, tzn. je brána jako jeden velký souvislý oddíl, jak je znázorněno na Obr. 9. Proces má potom alokováno přesně tolik místa, kolik potřebuje. Postup je následující. Jakýkoli proces je zpracován, je nalezeno dostatečně velké místo v paměti proto, aby mohl být proces alokován a následně je mu toto místo přiděleno. Se zbytkem volného prostoru se opět zachází jako s volným oddílem. Pokud v paměti není dostatečně velké místo pro alokování, musí proces čekat, dokud se místo neuvolní. Kdykoli je proces ukončen, je uvolněn obsazený prostor. Pokud tento prostor sousedí s dalším volným prostorem, sloučí se do jednoho většího volného prostoru a obsazené oddíly se posouvají do dolní části paměti. Tento princip se nazývá zhutnění paměti.



Obr. 9. Dynamický oddíl
Zdroj: vlastní zpracování

Pokud se očekává růst procesu při jeho běhu, je mu v paměti přidělena paměť navíc. Ovšem při swapování procesu zpět na disk, by měla být přesunuta pouze skutečná používaná velikost. Je zbytečné přesouvat paměť, která byla přidělena navíc. V případě přidělování paměti navíc mohou mít procesy dva rostoucí segmenty. Datový segment pro proměnné, které jsou dynamicky alokovány a uvolňovány, dále zásobník pro lokální proměnné a zpětné adresy. Pokud místo dojde, proces bude muset být přesunut do místa s dostatečně velkým prostorem, nebo přesunutý na disk, dokud jej nebude možné dát do dostatečně volného prostoru. Poslední možností je proces ukončit.

Více podrobností k mechanismu swapování a způsobu implementace lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 181, kap.: 3.2.2 Swapping); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 316, kap.: 7.2 Memory partitioning).

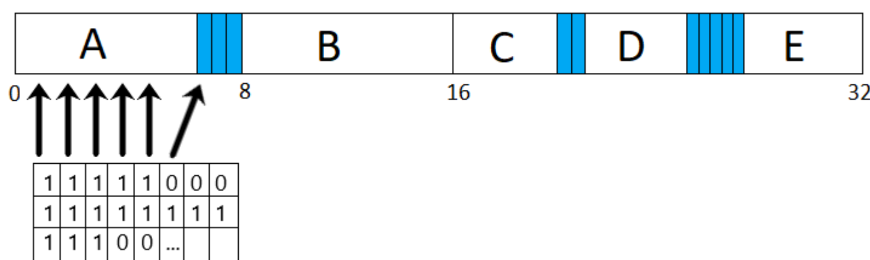
4.1.4 Správa volné paměti

Jak již zde bylo zmíněno procesy můžeme přiřazovat do paměti dynamicky, kdy se celá paměť jeví jako jeden souvislý blok paměti, nebo pomocí rozdělení paměti na pevné oddíly, které jsou neměnné, jedná se o tzv. pevné oddíly (angl. fixed partitions).

Pokud máme paměť rozdělenou na jeden souvislý blok paměti, znamená to, že přiřazujeme procesy do paměti dynamicky. Operační systém musí efektivním způsobem spravovat tuto paměť. Obecně jsou definovány dva základní postupy pro sledování využití stavu paměti. A to pomocí bitmapy nebo pomocí spojeného seznamu, resp. linked listu.

4.1.4.1 Bitmapa

Pokud se zaměříme na postup sledování využití paměti pomocí bitmapy, je celá paměť rozdělena na alokační jednotky, které standartně nabývají velikosti několik kilobajtů. Každá alokační jednotka obsahuje bity, které nabývají hodnot 1 nebo 0. Hodnota 1 nám značí místo, které je v paměti obsazené procesem. Hodnota 0 značí volné místo v paměti tzv. díru (angl. hole).



Obr. 10. Ukázka správy paměti pomocí bitmapy
Zdroj: vlastní zpracování

Pro ukázku využití bitmapy nám poslouží Obr. 10. Je zde znázorněna paměť, která obsahuje procesy A, B, C, D, E. Každý proces se tedy skládá z bitů, které nabývají hodnoty 1. Podle obrázku tedy proces A bude nabývat v bitmapě pěti jedniček, následuje volné místo (počet děr), které má velikost tři bity čili v bitmapě tvoří tři nuly. Tímto postupem by se vyplnila celá bitmapa popisující rozdělení procesů a děr v paměti, kdy každý řádek tabulky představuje záznam jedniček a nul pro jednotlivé procesy a díry.

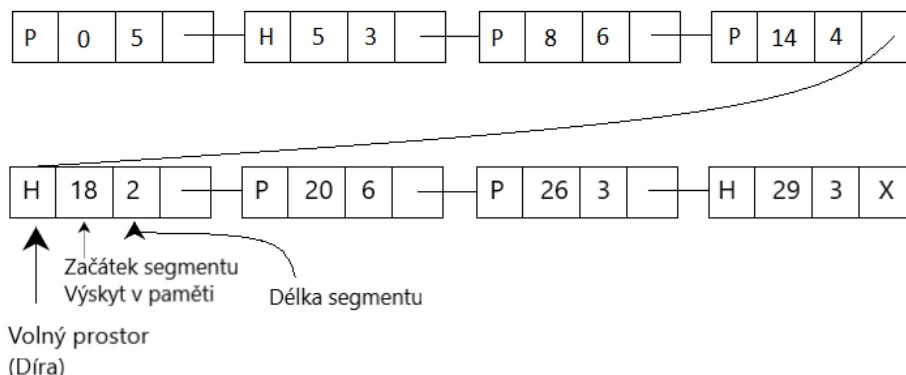
Velikost alokační jednotky je jeden z důležitých konstrukčních prvků celé bitmapy. Čím menší máme alokační jednotku, tím bude bitmapa větší. Naopak, když zvolíme alokační jednotku větší, bude bitmapa menší. Ovšem zde nastává riziko, kdy může být ztracena znatelná část paměti a to, pokud velikost procesu není přesným násobkem alokační jednotky, tudíž by nemohl být zapsán v této bitmapě.

Bitmapa nám poskytuje jednoduchý způsob sledování paměti. Celková velikost bitmapy je pak určena v závislosti na velikosti operační paměti a zvolené velikosti alokační jednotky.

4.1.4.2 Spojené seznamy

Další způsob pro sledování využití paměti je pomocí použití spojených seznamů, které zobrazují alokované a volné segmenty paměti. V tomto případě segment v seznamu obsahuje proces nebo je volným prostorem, resp. dírou, mezi procesy.

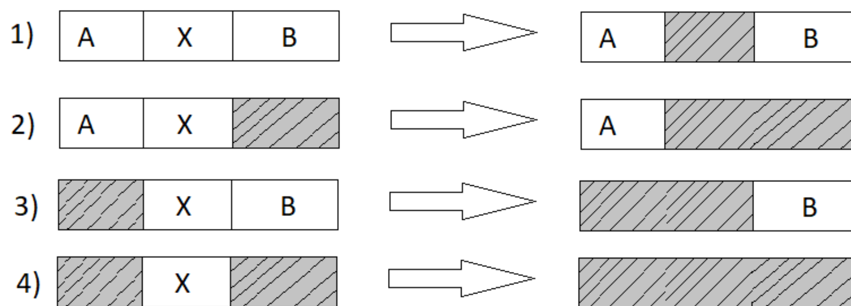
Seznam se skládá ze segmentů, které specifikují volný prostor (H) nebo proces (P). Segment dále obsahuje dva záznamy, kdy je zaznamenána adresa, na které segment začíná a druhý záznam obsahuje informaci o celkové délce segmentu v seznamu.



Obr. 11. Ukázka správy paměti pomocí spojeného seznamu
Zdroj: vlastní zpracování

Pro objasnění principu spojených seznamů je princip znázorněn na Obr. 11. Ze seznamu segmentů je vidět, že v paměti se nachází 5 procesů a 3 volné prostory, resp. díry. Každý segment je specifikován písmenem H nebo P, což značí, zda se jedná o proces nebo o volný prostor v paměti. Druhou položkou segmentu je číslo, které značí výskyt segmentu v paměti, jedná se o počátek segmentu v paměti. Poslední položka segmentu značí jeho délku.

Výhoda tohoto systému spočívá v tom, že pokud je proces ukončen nebo přesunut z operační paměti na disk, aktualizace seznamu je přímočará. Ukončený proces má pak většinou dva sousedy, pokud se ovšem nejedná o proces v horní nebo dolní části paměti. Sousedi mohou být buď procesy nebo volné prostory, což obecně vede ke čtyřem možným situacím. Po odebrání procesu je toto místo nahrazeno volným prostorem. Na Obr. 12. je takový proces značen písmenem X, tento proces bude odebrán. Další situace znázorňují spojení volných prostorů do jednoho většího. Po ukončení nebo přesunutí všech procesů se celá paměť uvolní a jeví se jako jeden souvislý blok volné paměti. V průběhu těchto situací se také odebírají segmenty ze seznamu.



Obr. 12. Znázornění situací po ukončení nebo přesunutí procesu
Zdroj: vlastní zpracování

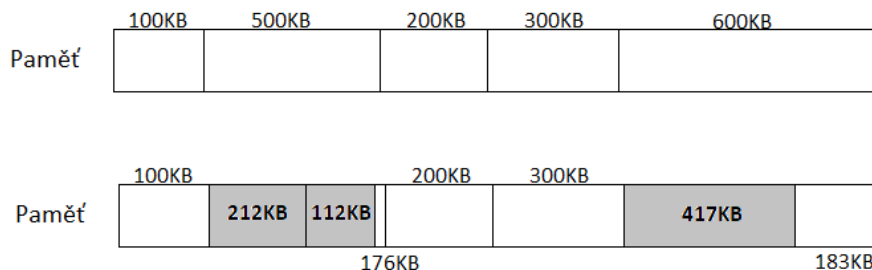
Pokud jsou procesy a volné prostory udržovány takto v seznamu a dokážeme je řadit podle velikosti, lze k přidělené paměti pro nově vytvořený proces nebo existující proces, který je přesunut z disku do operační paměti použít několik algoritmů. Jedná se o algoritmy:

1. First fit
2. Next fit
3. Best fit
4. Worst fit

4.1.4.3 Algoritmy přiřazování volné paměti

First fit

Nejjednodušší z uvedených algoritmů je algoritmus First fit. V tomto případě je paměť skenována od začátku paměti podél seznamu segmentů, dokud správce paměti nenarazí na dostatečně velký volný prostor, aby mohl být proces alokován. Tento volný prostor je následně rozdělen na dva díly. Jeden díl je alokovaný proces v paměti a druhý díl je zbytek volného prostoru, jedná se o nevyužitou paměť, kterou proces neobsadil.



Obr. 13. Ukázka algoritmu First fit
Zdroj: vlastní zpracování

Princip algoritmu First fit je znázorněn na Obr. 13. Úkolem je do paměti přiřadit procesy o velikosti 212 kB, 417 kB, 112 kB a 426 kB **v tomto pořadí**.

Hledání volného prostoru začíná od začátku paměti. První nalezený volný prostor je velikosti 100 kB, který není dostatečně velký pro alokování prvního procesu o velikosti 212 kB. Další volný prostor o velikosti 500 kB je dostatečně velký na alokování tohoto procesu. Volný prostor nyní obsahuje proces a zbytek tohoto prostoru se dále jeví jako volný prostor. Tudíž máme 500kB volný prostor rozdělen na dvě části. První část je alokovaný proces o velikosti 212 kB a druhá část o velikosti 288 kB je nevyužitá paměť a jedná se tedy o volný prostor.

Pro alokování druhého procesu opět algoritmus začíná od začátku paměti a skenuje celou paměť, dokud nenajde dostatečně velký prostor. Nalezený prostor je o velikosti 600 kB, kam je alokován proces o velikosti 417 kB. Prostor je opět rozdělen na dvě části. První část je alokovaný proces a druhá nevyužitá paměť.

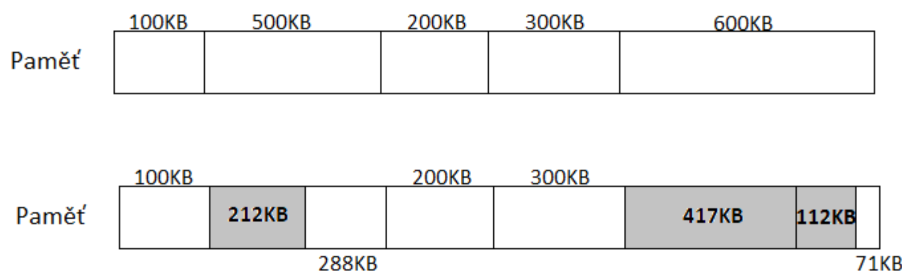
Tímto postupem jsou alokovány skoro všechny procesy v pořadí, v jakém přišly. Na obrázku vidíme, že proces o velikosti 426 kB nemá v paměti dostatečně velký prostor pro alokování. Musí proto čekat, dokud se neuvolní dostatečný prostor na to, aby mohl být alokován do paměti.

First fit je nejrychlejší ze zmíněných algoritmů, protože se snaží prohledávat paměť co nejméně. Jakmile najde dostatečný prostor, alokuje proces a prohledává paměť opět od začátku a tímto způsobem pokračuje pro všechny procesy. Nevýhodou v tomto algoritmu je celková

ztráta paměti, protože i pro malý proces může být vybrán velký volný prostor, který by byl dostatečný pro alokování větších procesů. Při alokování menšího procesu do velkého volného prostoru se volný prostor zmenší a poté nemusí být v paměti místo pro alokování většího procesu.

Next fit

Algoritmus Next fit funguje na podobném principu jako zmíněný algoritmus First fit. Hledá první dostatečně velký volný prostor, aby mohl alokovat proces. Prostor je poté opět rozdělen na dvě části. První část je alokovaný proces a druhá část zbytek volného prostoru, nevyužitá paměť. Do teď se jedná o stejný princip jako u algoritmu First fit. Rozdíl oproti algoritmu First fit spočívá v tom, že u algoritmu Next fit nezačíná prohledávání paměti vždy od začátku, ale od místa, kde při předchozím hledání skončil.



Obr. 14. Ukázka algoritmu Next fit
Zdroj: vlastní zpracování

Princip algoritmu Next fit je znázorněn na Obr. 14. Do paměti se mají přiřadit procesy o velikosti 212 kB, 417 kB, 112 kB a 426 kB **v tomto pořadí**.

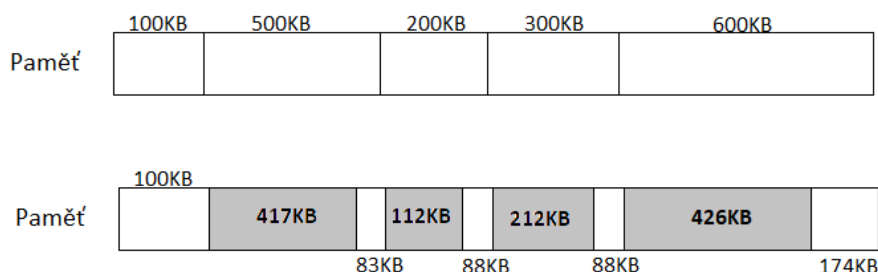
První hledání začíná od začátku paměti a funguje stejně jako u algoritmu First fit. Dokud nenarazí na první dostatečně velký volný prostor, tak bude algoritmus pokračovat v hledání. V našem případě chceme alokovat proces o velikosti 212 kB. První takový volný prostor je o velikosti 500 kB. Do tohoto prostoru je alokován proces a zbytek je opět volný prostor, paměť, která je nevyužita. V dalším kroku nastává hlavní rozdíl mezi algoritmem First fit a algoritmem Next fit. Algoritmus Next fit začíná následně prohledávat paměť od místa kde skončil, zatímco algoritmus First fit by se opět vracel na začátek paměti.

Tímto postupem jsou alokovány skoro všechny procesy v pořadí, v jakém přišly. Na Obr. 14 je vidět, že proces o velikosti 426 kB nemá v paměti dostatečně velký prostor pro alokování. Musí proto čekat, dokud se neuvolní dostatečný prostor na to, aby mohl být proces alokován.

Stejně jako u algoritmu First fit, tak i v tomto případě se jedná o rychlý algoritmus, který má menší čas prohledávání paměti, protože nezačíná při každém volání od začátku, ale od místa, kde naposledy skončil. Z tohoto důvodu musí správce paměti sledovat poslední přidělený volný prostor, který byl zvolen ke zpracování. Podle simulací dosahuje algoritmus Next fit horšího výkonu než algoritmus First fit.

Best fit

Nejvhodnější a velmi často používaný je algoritmus Best fit. Algoritmus Best fit při vyhledávání místa pro alokování procesu v paměti prochází celou paměť od začátku do konce a vybírá nejmenší možný volný prostor, který je pro alokaci procesu adekvátní. Místo použití prvního adekvátního volného prostoru, jak je to u dvou předchozích algoritmů, se algoritmus Best fit snaží vybrat co nejmenší volný prostor a větší volné prostory ponechat pro využití jinými, náročnějšími, procesy.



Obr. 15. Ukázka algoritmu Best fit
Zdroj: vlastní zpracování

Princip algoritmu Best fit je znázorněn na Obr. 15. Do paměti se mají přiřadit procesy o velikosti 212 kB, 417 kB, 112 kB a 426 kB **v tomto pořadí.**

Jako první proces má být alokován proces o velikosti 212 kB. Algoritmus Best fit je zavolán a prohledá celou paměť od začátku do konce. Jako adekvátní volný prostor pro proces

o velikosti 212 kB našel volný prostor o velikosti 300 kB. Volný prostor je opět rozdělen na dvě části. Jedna část je alokovaný proces a druhá část je zbytek volného prostoru, který proces nevyužije. Pro další alokování procesu opět algoritmus Best fit prohledává paměť od začátku do konce. Pro proces o velikosti 417 kB vybírá jako adekvátní volný prostor s nejmenší možnou velikostí volný prostor velikosti 600 kB. Volný prostor je opět rozdělen na dvě části. Při dalším volání algoritmu opět prohledává algoritmus celou paměť od začátku do konce a opět hledá nejmenší adekvátní volný prostor.

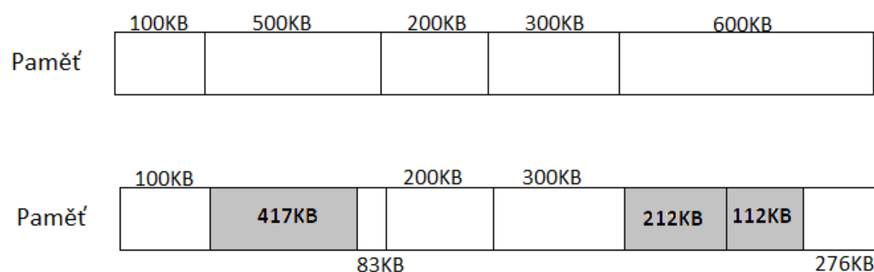
Takto jsou postupně alokovány všechny procesy v pořadí, v jakém přišly. Oproti předchozím dvěma algoritmům jsou zde alokovány všechny procesy do paměti.

Algoritmus Best fit je značně pomalejší oproti předchozím dvěma, protože při každém volání prohledává celou paměť od začátku do konce a vybírá nejmenší adekvátní volný prostor pro alokaci procesu. Další nevýhoda tohoto algoritmu spočívá v tom, že z podstaty výběru minimálního vhodného místa pro jednotlivé procesy vzniká velké množství malých, dále již nevyužitelných, volných částí paměti (fragmentace).

Worst fit

Algoritmus Worst fit je hodně podobný s algoritmem Best fit. Při každém volání i tento algoritmus prohledává celou paměť od začátku do konce. Tentokrát ovšem není účelem najít co nejmenší adekvátní volný prostor, ale největší volný prostor, kam je následně proces alokován. Stejně jako u všech algoritmů přiřazování procesů do paměti je volný prostor rozdělen do dvou částí. Jedna část je tvořená procesem a druhá část je nevyužitý volný prostor.

Máme procesy o velikosti 212KB, 417KB, 112KB, 426KB v tomto pořadí.



Proces o velikosti 426KB nemá dostatečně velký volný prostor pro alokaci do paměti

Obr. 16. Ukázka algoritmu Worst fit
Zdroj: vlastní zpracování

Princip algoritmu Worst fit je znázorněn na Obr. 16. Do paměti se mají přiřadit procesy o velikosti 212 kB, 417 kB, 112 kB a 426 kB **v tomto pořadí**.

Jako první chceme alokovat proces o velikosti 212 kB. Algoritmus Worst fit je zavolán a prohledá celou paměť od začátku do konce. Jako největší volný prostor najde volný prostor o velikosti 600 kB. Volný prostor je opět rozdělen na dvě části. Jedna část je alokovaný proces a druhá část je zbytek volného prostoru, který proces nevyužije. Pro další alokování procesu opět algoritmus Worst fit prohledává paměť od začátku do konce. Nyní je vybrán opět největší volný prostor, tentokrát o velikost 500 kB. Volný prostor je opět rozdělen na dvě části. Při dalším volání algoritmu opět prohledává algoritmus celou paměť od začátku do konce a hledá největší volný prostor.

Tímto postupem je alokována většina procesů v pořadí, v jakém přišly. Na Obr. 16 vidíme, že proces o velikosti 426 kB nemá v paměti dostatečně velký prostor pro alokování. Musí proto čekat, dokud se neuvolní dostatečný prostor na to, aby mohl být alokován do paměti.

I u tohoto algoritmu je čas hledání poměrně vysoký kvůli prohledávání celé paměti od začátku do konce při každém zavolání. Nevýhoda spočívá v tom, že do velkých volných prostorů jsou přiřazovány procesy o malé velikosti a pro procesy větších velikostí se nedostává dostatečného místa v paměti.

Více podrobností ke správě volné paměti a konkrétních algoritmů lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 184, kap.: 3.2.3 Managing free memory); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 321, kap.: 7.2 Memory partitioning).

4.2 Virtuální paměť

V tomto tématu je především zmiňováno stránkování a způsob mapování stránek virtuálního adresního prostoru na rámce stránek fyzického adresního prostoru. Vysvětlení pojmů stránka a rámec stránky je potřeba pro účely správného pochopení, kdy je musíme rozlišovat a pochopit jejich význam. Dále se toto téma zabývá fungováním jednotky procesoru MMU, které je v souvislosti s virtuální pamětí velmi důležité. Toto téma volně navazuje na předchozí téma a dále se zabývá principem fungování operační paměti a její správy.

Virtuální paměť (též virtualizace paměti) je způsob správy operační paměti počítače, který umožňuje předložit běžícímu procesu adresní prostor paměti, který je uspořádán jinak, nebo je větší než fyzická operační paměť RAM. Z tohoto důvodu procesor rozlišuje mezi virtuálními adresami (adresy s kterými pracují strojové instrukce, resp. běžící proces) a fyzickými adresami (adresy, které odkazují na konkrétní adresové buňky hlavní paměti počítače).

V předchozí kapitole byly zmíněny base a limit registry, které mohou být použity k abstrakci adresního prostoru. Ovšem je zde problém, který by měl být také řešen, jedná se o správu tzv. bloatware.

Bloatware je termín, který označuje tendenci novějších počítačových programů po své instalaci zanechat větší stopu, než je nutné. Tím můžeme chápat mnoho zbytečných funkcí, které nejsou běžnými uživateli využívány. Nebo se také jedná o využití více systémových prostředků, než je nutné, přestože uživatelům nabízejí velmi malý až skoro žádný užitek. Termín bloatware taky můžeme chápat jako software, který se nachází na již zakoupeném počítači a obvykle se jedná o časově omezené zkušební verze předinstalovaných programů.

Bloatware je jednou z příčin, který způsobuje nevyváženost mezi růstem operační paměti a růstem využití paměti samotným programem. Toto se stává novým trendem multimédií, které kladou ještě větší nároky na paměť. V důsledku tohoto problému vznikají situace, kdy je potřeba spouštět programy, které jsou příliš velké na to, aby se vešly do paměti nebo existují systémy, které vyžadují více programů současně, z nichž se každý zvlášť vejde do operační paměti, ale společně tuto paměť přesahují. Swapování zde není atraktivní řešení kvůli přenosové rychlosti a delšímu času zpracování.

Jedno z řešení přišlo v 60. letech. Jednalo se o rozdělení programů na malé kousky, tzv. překryvy (angl. Overlays). Pokud byl program spuštěn, tak se do paměti načetl správce překrytí

a okamžitě načetl a spustil překrytí 0. Když bylo překrytí 0 dokončeno, bylo správci překrytí řečeno, aby načetl překrytí 1 a to buď nad překrytím 0 přímo v paměti, pokud tam bylo dostatečné místo pro načtení, nebo přes překrytí 0, pokud v paměti místo nebylo. Některé systémy překrytí byly vysoce komplexní. V tomto ohledu byly vrstvy překrytí uchovávány na disku a swapovány z, resp. do, paměti. Práci s přepínáním obsluhoval operační systém. Zde se jednalo o klasický swapovací systém. Rozdělení programu na překryvy muselo být uděláno ručně programátorem. Jednalo se o způsob, který nebyl jednoduchý a byl také časově náročný. Netrvalo dlouho a byl vymyšlen způsob, jak předat celou práci samotnému počítači.

Metoda byla vyvinuta v roce 1961 Johnem Fortheringhamem. Metoda se stala dodnes známou jako virtuální paměť. Základní myšlenkou virtuální paměti je, že každý program má svůj vlastní adresní prostor, který je rozdělen na bloky zvané stránky. Tyto stránky jsou mapovány na fyzickou paměť.

Pokud program odkazuje na část svého adresního prostoru, která je ve fyzické paměti, hardware provede potřebné mapování za běhu programu. Ovšem pokud program odkazuje na část adresního prostoru, který není ve fyzické paměti, operační systém je upozorněn, aby získal chybějící část a znovu provedl instrukci, u které selhal. V následujícím textu bude popsáno, co je tím myšleno a jak je virtuální paměť implementována.

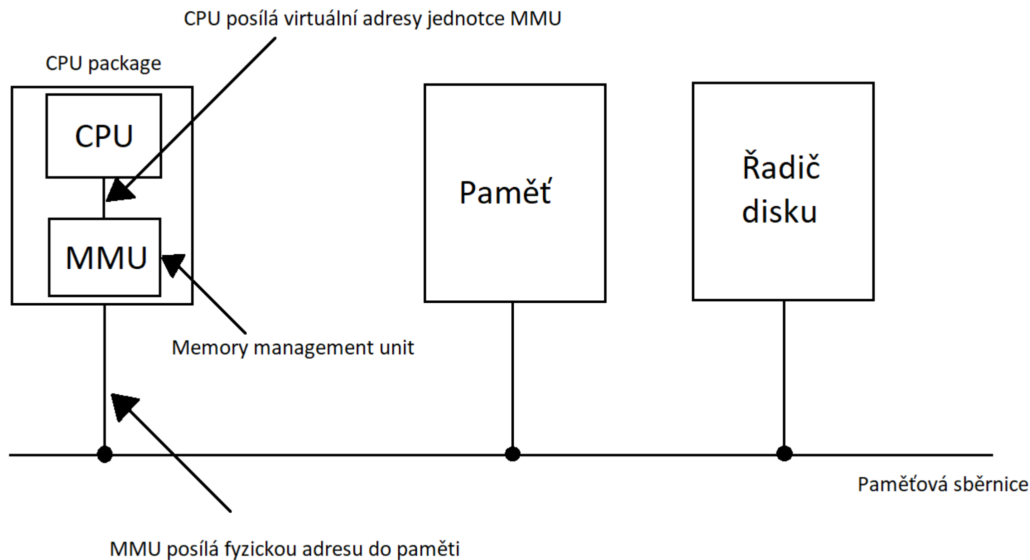
4.2.1 Stránkování a MMU

Zaměřme se tedy na to, co virtuální paměť dělá. Virtuální paměť můžeme chápat jako vytvoření nového adresního prostoru, což je abstrakce fyzické paměti. Virtuální paměť lze implementovat rozdělením virtuálního adresního prostoru na stránky a mapováním každé z nich na rámec stránky fyzické paměti. Jedná se tedy především o abstrakci vytvořenou operačním systémem. Mapujeme tedy stránky (virtuální adresní prostor) na rámce stránek (fyzický adresní prostor).

Systém virtuální paměti používá techniku zvanou stránkování, kde programy odkazují na sadu adres paměti. Vysvětlení je uvedeno na následujícím příkladě.

Když program zavede instrukci MOV REG, 1000 zkopíruje obsah adresy 1000 na pozici REG. Adresy lze generovat například pomocí indexování, základních registrů nebo segmentových registrů. Tyto programem generované adresy nazýváme virtuální adresy a tvoří společně virtuální adresní prostor. Počítače bez virtuální paměti ukládají virtuální adresy přímo na paměťovou sběrnici a způsobují, že se bude číst nebo zapisovat slovo fyzické adresy na

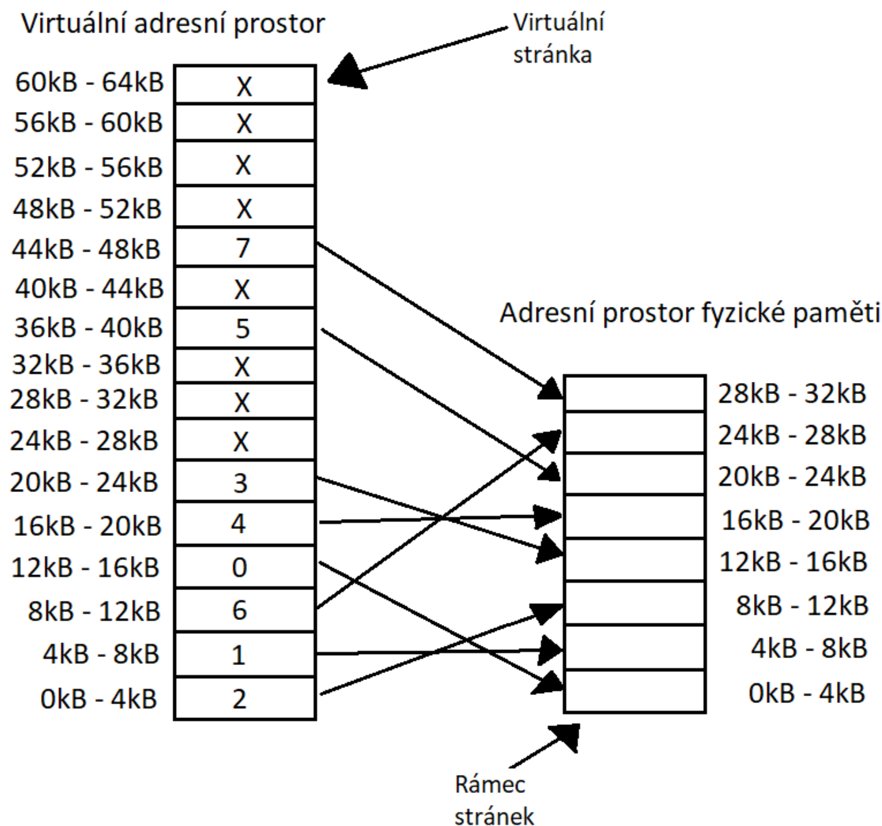
stejné adrese. Pokud ovšem počítač využívá virtuální paměť, virtuální adresy nepřejdou přímo na paměťovou sběrnici. Místo toho putují do jednotky procesoru, která se nazývá MMU. MMU mapuje virtuální adresy na fyzické adresy, jak je znázorněno na Obr. 17.



Obr. 17. Ilustrace fungování jednotky MMU
Zdroj: vlastní zpracování

MMU (angl. Memory Management Unit) je součást mikroprocesoru, která umožňuje přístup do virtuální paměti. Dříve se jednalo o samotnou hardwarovou část mimo procesor. MMU zajišťuje především překlad virtuální adresy na fyzickou adresu, dále ochranu paměti a v závislosti na architektuře také zajišťuje přepínání mezi paměťovými buňkami.

Příklad samotného mapování virtuálních adres je uveden na Obr. 18. V tomto příkladě je uveden počítač, který generuje 16bitové adresy od 0 kB do 64 kB. Tady se jedná se o virtuální adresy. Nicméně tento počítač má menší fyzickou paměť o velikosti 32 kB. Virtuální adresní prostor je rozdělen na jednotky pevné velikosti. Tyto jednotky ve virtuálním adresním prostoru nazýváme stránky. Odpovídající položky stránkám ve fyzické paměti se nazývají rámce stránek. Stránky a rámce stránek obecně mívají stejnou velikost. V tomto případě se jedná o 4 kB, ale v reálných systémech můžou být využity velikosti stránek od 512 bajtů do 64 kB. S 64kB virtuálním adresním prostorem a 32kB fyzickou pamětí dostaneme 16 virtuálních stránek a 8 rámců stránek. Následně bude představeno fungování jednotky MMU a mapování stránek na rámce stránek.



Obr. 18. Relace mezi virtuálním adresním prostorem a fyzickým adresním prostorem
Zdroj: vlastní zpracování

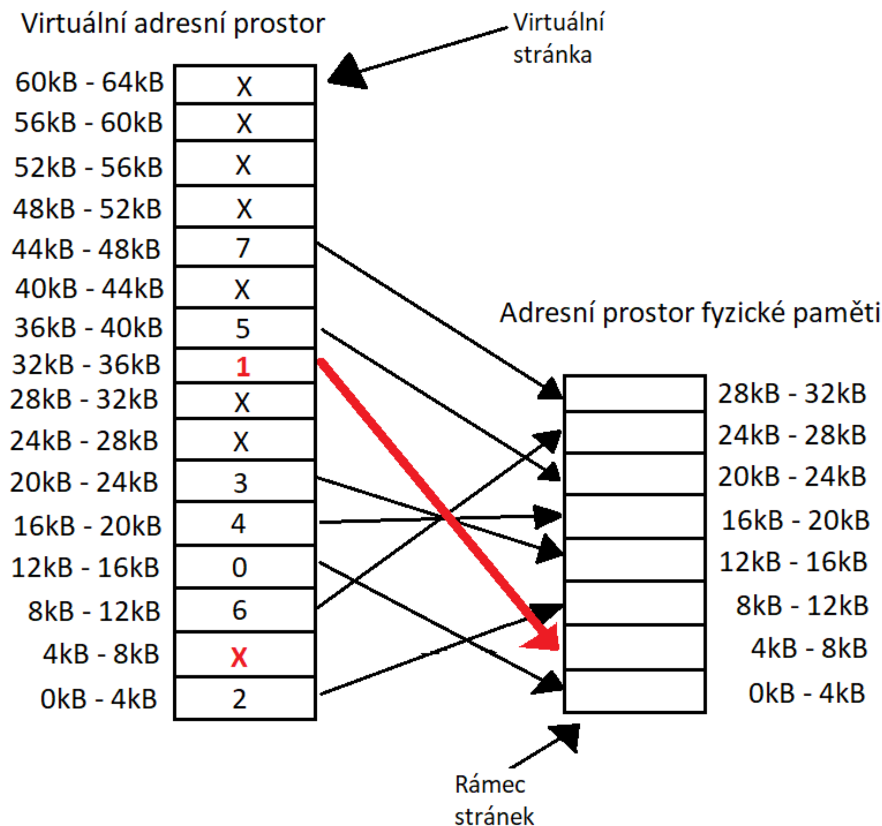
Popis Obr. 18. je následující. Rozsah 0 kB – 4 kB znázorňuje virtuální nebo fyzický rozsah adres stránky, resp. rámce stránek. Jedná se tedy o adresy od 0 do 4095 a jejich násobek. Z obrázku je jasné, že každá stránka či rámec stránky má stejnou velikost.

Nyní přistupme k fungování jednotky MMU. Pokud se program pokouší získat přístup k adrese 0, například pomocí instrukce MOV REG, 0, je virtuální adresa 0 poslána do MMU. MMU vidí, že tato virtuální adresa spadá do 0. stránky ve virtuálním adresním prostoru, což odpovídá adresám 0–4095. Podle mapování má referenci na 2. rámec stránky ve fyzickém adresním prostoru (pozor, všechny položky číslujeme od nuly). Můžeme tedy říct, že stránky o rozsahu virtuálních adres 0–4095 (jedná se o virtuální adresní prostor) jsou mapovány na rámec stránky o rozsahu fyzických adres 8192-12287 (zde se jedná o fyzický adresní prostor). Tudíž adresa 0 je transformována na adresu 8192, která je následně poslána na paměťovou sběrnici. MMU účinně mapuje všechny virtuální adresy 0 až 4095 na fyzické adresy v paměti v rozsahu 8192 do 12287. Obdobně to bude například s instrukcí MOV REG, 8192. Pomocí MMU je instrukce transformována na MOV REG, 24576, jelikož virtuální adresa 8192 (jedná se o 2. stránku) má referenci na 6. rámec stránky ve fyzickém adresním prostoru.

Poslední příklad je na adrese 20500. Hodnota 20500 je hodnota 5. stránky, jelikož máme daný počátek stránky, tj. 20480 + 20 bajtů od počátku. Tato stránka je mapována na rozsah fyzických stránek od 12 288 do 16 384, jedná se o 3. rámec stránky. Výsledná adresa ve fyzické paměti, která bude poslána v případě instrukce na paměťovou sběrnici, bude počáteční hodnota rámce stránky s přičtením počtu bajtů od počátku, tj. 12 288 + 20 = 12 308.

Tímto způsobem dokážeme namapovat 16 virtuálních stránek na kterýkoli z osmi rámců stránek vhodným mapováním pomocí MMU. Sám o sobě ale tento způsob neřeší problém v tom, že máme větší virtuální prostor než fyzický adresní prostor. Podle Obr. 18 máme osm fyzických rámců stránek, tudíž je mapováno osm stránek z virtuální paměti do fyzické paměti. Ostatní stránky jsou na obrázku vyplněné křížkem a nejsou mapovány. V hardwaru nám k tomuto účelu slouží přítomný/nepřítomný bit (angl. Present/absent bit), který sleduje, zda se stránka fyzicky nachází v paměti, či nikoliv. Tady nastává otázka, co se stane, pokud program odkazuje na stránku, která není mapována na fyzický rozsah adres. Pro získání odpovědi použijeme následující příklad.

Mějme instrukci MOV REG, 32780. Podle Obr. 18 se jedná o 8. virtuální stránku. Počáteční adresa stránky je 32 768 + 12 bajtů od počátku. MMU zjistí, že stránka není namapována (označeno v obrázku křížkem) a způsobí, že se CPU tzv. „zachytí“ v operačním systému. Toto zachycení, můžeme označit i jako past (angl. trap), se nazývá chyba stránky (angl. page fault). Operační systém v tomto případě vybere málo používaný rámec stránky a zapíše jeho obsah na disk. Poté načte právě odkazovanou stránku do nyní uvolněného rámce stránky, změní mapování a restartuje instrukci, u které byla způsobena chyba stránky. Pro příklad berme v úvahu, že se operační systém rozhodl vyřadit 1. rámec stránky. Operační systém nyní provede potřebné změny v mapování MMU. Nejprve označí virtuální stránku s referencí na 1. rámec stránky jako nezmapovanou, aby zachytil budoucí přístupy k virtuálním adresám mezi adresami 4096-8192. Pak nahradí „křížek“, resp. změní present/absent bit v záznamu 8. virtuální stránky číslem 1. Při opětovném provedení zachycené instrukce je mapována virtuální adresa 32 780 na fyzickou adresu 4108. Na Obr. 19 je znázorněná změna mapování MMU oproti předchozímu příkladu uvedeném na Obr. 18.

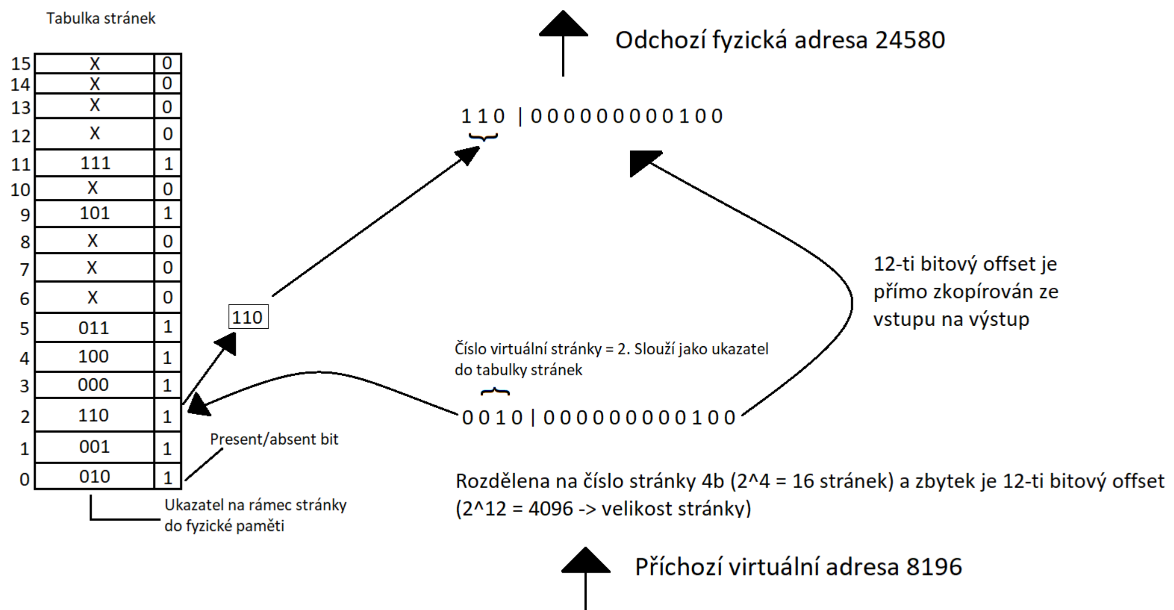


Obr. 19. Změna mapování při chybě stránky
Zdroj: vlastní zpracování

Toto byly příklady k samotnému mapování a relaci mezi stránkami a rámci stránek. Představení odděleného virtuálního adresního prostoru a fyzického adresního prostoru. Následující příklad znázorní samotné fungování MMU, jak rozhodne, na který rozsah adres bude mapovat stránky. Následně bude popsáno vnitřní fungování jednotky MMU.

Na Obr. 20 vidíme příklad mapování adresy pomocí MMU. Na vstup přichází virtuální adresa 8196, která je zapsána binárně. Příchozí 16bitová virtuální adresa je rozdělena na 4bitové číslo stránky a 12bitový offset. Čtyři bity slouží pro získání čísla stránky. Čtyři bity proto, že máme celkem 16 stránek, tj. $2^4 = 16$. 12 zbývajících bitů je pro offset. Offset značí, že můžeme adresovat 4096 adres, jedná se o velikost stránky ($2^{12} = 4096$). Virtuální adresu 8196 máme napsanou binárně a následně binární číslo rozdělíme na číslo stránky a offset. Číslo stránky je využito jako index do tabulky stránek, čímž získáme odpovídající číslo rámce stránky, které odpovídá virtuální stránce. V tomto příkladě získáme binární číslo 0010, v desítkové soustavě se jedná o hodnotu 2. Takže se nyní zaměříme na druhou stránku v tabulce stránek (číslujeme od nuly). Pokud je zde přítomný/nepřítomný bit roven 0, je způsobena chyba operačního systému, jedná se o chybu stránky. Pokud je bit nastaven na jedničku, číslo rámce stránky

nalezené v tabulce stránek se zkopíruje na první 3 bity výstupní adresy společně s 12bitovým offsetem, který nebyl změněn. Vzniká 15bitová fyzická adresa, která se pošle na paměťovou sběrnici jako získaná adresa fyzické paměti.



Obr. 20. Ukázka mapování pomocí MMU
Zdroj: vlastní zpracování

Více podrobností ke stránkování a fungování MMU lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 188, kap.: 3.3 Virtual memory, dále str. 189, kap.: 3.3.1 Paging); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 326, kap.: 7.3 Paging).

4.2.2 Tabulka stránek

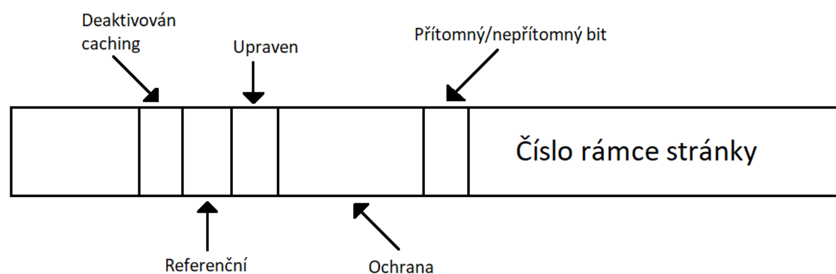
V jednoduché implementaci lze mapování virtuálních adres na fyzické adresy shrnout následovně. Virtuální adresa je rozdělena na číslo stránky a offset, offset zůstává neměnný. Například jak zde bylo zmíněno, 16bitovou adresu s velikostí stránky 4 kB rozdělíme na 4 bity pro určení jedné z 16-ti virtuálních stránek a 12 bitů by určovalo offset, který odpovídá velikosti jednotlivé stránky, resp. rámce stránek.

Číslo virtuální stránky je použito jako index do tabulky stránek, kde je nalezeno číslo rámce stránek, pokud existuje. Číslo tohoto rámce je připojeno (binárně) k neměnnému offsetu, čímž nahradíme číslo virtuální stránky. V tomto případě se jedná o právě zmíněné 4 bity. Toto spojení nám vytvoří fyzickou adresu, která je poslána na paměťovou sběrnici.

Účelem tabulky stránek je indexování virtuálních stránek na rámce stránek. Z matematického hlediska je tabulka stránek funkcí, jejíž argumentem je číslo virtuální stránky a výsledkem je číslo fyzického rámce. Pomocí výsledku této funkce lze číslo virtuální stránky nahradit číslem rámce, čímž se vytvoří adresa fyzické paměti.

4.2.2.1 Struktura položky tabulky stránek

Nyní se zaměříme na strukturu záznamu tabulky stránek. Přesné rozvržení záznamu tabulky stránek je závislé na daném stroji, ale druh obsažených informací je u každého stroje stejný. Na Obr. 21 je uveden příklad záznamu tabulky stránek.



Obr. 21. Záznam tabulky stránek
Zdroj: vlastní zpracování

Nejdůležitější částí je číslo rámce stránky, jelikož cílem mapování stránky je právě vydat tuto hodnotu k offsetu. Následuje přítomný/nepřítomný bit (angl. present/absent bit). Tento bit zde již byl zmíněn. Jedná se o určení platnosti položky. Pokud nabývá hodnoty 1, lze stránku použít. V opačném případě virtuální stránka není aktuálně mapována na žádný rámec stránek. Pokud je bit nastavený na 0 způsobí chybu stránky.

Ochranné bity (angl. Protection bit) určují, jaké druhy přístupu jsou povoleny. V nejjednodušší formě toto pole obsahuje právě jeden bit. Hodnota 0 určuje čtení/zápis a hodnota 1 je pouze pro čtení. Více sofistikované uspořádání mají tři bity. Každý jeden bit umožňuje čtení, zápis nebo vykonání stránky (angl. read, write, execute).

Modifikovaný bit (angl. Modified bit) říká, zda byla stránka upravena nebo ne. To znamená, že se můžeme pokusit o zápis na stránku. Pokud je stránka upravena, pak kdykoli bychom ji měli nahradit nějakou jinou stránkou, upravené informace by měly být uloženy na pevném disku. Tento bit je nastaven hardwarem na jedničku pro zápis na stránku, která se používá, aby se zabránilo zápisu při výměně. Někdy se tento upravený bit nazývá také jako „Dirty bit".

Referenční bit (angl. Referenced bit) se nastaví vždy, když se odkazuje na stránku, ať už pro čtení nebo zápis. Hodnota tohoto bitu pomáhá operačnímu systému vybrat stránku, která by měla být vyřazena v případě, že dojde k chybě stránky. Stránky, které se nepoužívají jsou lepšími kandidáty než stránky, které jsou využívány často.

Poslední bit umožňuje deaktivaci ukládání stránky do mezipaměti. Tato funkce je důležitá pro stránky, které se mapují na registry zařízení. Pokud operační systém vyčkává ve smyčce a čeká, až nějaké vstupní/výstupní zařízení zareaguje na zadaný příkaz, je nezbytné, aby hardware stále načítal slovo ze zařízení a nepoužíval starou kopii v mezipaměti. S tímto bitem lze ukládání do mezipaměti vypnout. Počítače, které mají oddělený I/O prostor a nepoužívají mapování I/O paměti tento bit nepotřebují.

Tabulka stránek obsahuje pouze ty informace, které hardware potřebuje k překladu virtuální adresy na fyzickou adresu. Informace, které operační systém potřebuje ke zpracování chyb stránky jsou uloženy v softwarových tabulkách uvnitř operačního systému. Hardware toto znát nepotřebuje.

4.2.3 Zrychlení stránkování

Nyní se detailněji podíváme na možné problémy, které je potřeba řešit v případě virtuální paměti. V jakémkoli stránkovacím systému musíme čelit dvěma zásadním poznatkům:

1. Mapování z virtuální adresy na fyzickou adresu musí být rychlé
2. Čím větší je virtuální adresní prostor, tím větší je tabulka stránek

První bod je důsledkem toho, že mapování virtuální adresy na fyzickou musí být provedeno při každém odkazování na paměť. Všechny instrukce pocházejí z paměti a mnoho instrukcí odkazuje na operandy v paměti. Je tedy nutné udělat jeden nebo více odkazů na tabulku stránek za instrukci. Musí se zabránit tomu, aby se pomalá rychlost mapování nestala velkým problémem.

Druhý bod vyplývá ze skutečnosti, že většina počítačů využívá alespoň 64bitové virtuální adresy. Řekněme například, že na 32bitovém adresním prostoru budeme mít stránky o velikosti 4 kB. Těchto stránek bude milion. Z toho vyplývá, že tabulka stránek bude obsahovat právě milion záznamů. Každý proces potřebuje mít vlastní tabulku stránek, jelikož každý proces má svůj vlastní adresní prostor. Pokud si toto představíme pro 64bitový adresní prostor, dostaneme se do obrovských čísel.

Nejjednodušší design je mít tabulku jedné stránky sestávající z řady rychlých hardwarových registrů s jedním záznamem pro virtuální stránku v mezipaměti, která je indexována podle čísla virtuální stránky, jak již zde bylo představeno. Když je proces spuštěn, operační systém načte registry s tabulkou stránek procesu převzatou z kopie uložené v hlavní paměti. Během vykonávání procesu nejsou pro tabulku stránek potřeba žádné další odkazy na paměť. Tato metoda je přímá a nevyžaduje žádné další odkazy na paměť během mapování. Nevýhodou je její nákladnost, pokud je tabulka stránek větší.

Jiný design ukazuje, že může být celá tabulka stránek přímo v paměti. Všechny hardwarové potřeby pak tvoří jediný registr, který odkazuje na začátek tabulky stránek. Tento design umožňuje změnu virtuální mapy na fyzickou mapu při přepnutí kontextu opětovným načtením jednoho registru. Je tu nevýhoda v tom, že vyžaduje jeden nebo více odkazů na paměť ke čtení položek tabulky stránek během provádění každé instrukce, což je velmi pomalé.

4.2.3.1 TLB

Výchozím bodem většiny optimalizačních technik spočívá v tom, že celá tabulka stránek se nachází v paměti. Potenciálně má toto schéma dopad na celkový výkon. Řešení je založeno na pozorování, kdy většina programů má tendenci vytvářet velké množství odkazů na malý počet stránek. Proto je ve výsledku jen malý zlomek položek tabulky stránek těžce čitelný, to znamená, že se skoro vůbec nepoužívají.

V tomto problému nám pomáhá hardwarové zařízení pro mapování virtuálních adres na fyzické adresy bez nutnosti procházet tabulku stránek. Jedná se o TLB. TLB (angl. Translation lookaside buffer) je speciální hardwarová cache v MMU, která má za úkol řešení hlavního problému, a to je především zrychlení stránkování.

Na Obr. 22 je uveden příklad TLB. Obvykle se nachází uvnitř MMU a skládá se z malého počtu záznamů, v tomto případě se jedná o osm záznamů. Každý záznam obsahuje informace o jedné stránce. První informace nám značí bit, který označuje, zda je záznam platný, resp. je používán. Následuje číslo virtuální stránky, dále bit, který je nastaven, pokud je stránka upravena. Ochranný bit, který zahrnuje oprávnění pro čtení, zápis a spuštění. Nakonec číslo fyzického rámce, na který je stránka mapována.

Valid	Č. virt. stránky	Upravený	Ochranný	Č. rámce stránky
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Obr. 22. Ukázka TLB
Zdroj: vlastní zpracování

Fungování TLB je následující. Virtuální stránka je dána MMU k překladu, hardware zkontroluje, zda je číslo virtuální stránky v TLB a porovná je paralelně se všemi položkami. Pokud je nalezena shoda a přístup není porušen ochranným bitem, je rámec stránky převzat z TLB, aniž by se muselo pracovat a odkazovat na tabulku stránek. V případě že se instrukce snaží zapsat na stránku, kde je ochranný bit pouze pro čtení, je vygenerována chyba a přístup zamítnut.

Pokud virtuální stránka není nalezena v TLB, MMU detekuje chybu (angl. miss) a provede klasické vyhledávání pomocí tabulky stránek. Následně je prohozena jedna položka v TLB a je nahrazena právě vyhledanou položkou pomocí tabulky stránek. V případě příštího přístupu k této stránce bude již proveden přístup v TLB. Odstraněná položka z TLB je zkopírována do tabulky stránek.

Softwarová správa TLB

Do této doby bylo předpokládáno, že každý stroj s virtuální pamětí rozpoznává tabulku stránek pomocí hardwaru. V tomto designu se správa a zpracování chyb TLB provádí výhradně pomocí hardwaru MMU. Chyby, které jsou mířeny na operační systém se vyskytují za předpokladu, že se stránka nenachází v paměti. Pokud dojde k chybě TLB, MMU vygeneruje pouze chybu a přesune problém na operační systém. Operační systém poté musí najít správnou stránku, odebrat položku z TLB, zadat novou položku do TLB a restartovat instrukci, u které selhala.

Softwarová správa TLB se ukáže jako efektivní, pokud je v TLB velký počet záznamů. Hlavní zisk je zde v podobě jednoduššího MMU, který uvolní značné množství prostoru na čipu CPU pro mezipaměť a další funkce, které mohou zlepšit celkový výkon. Aby operační systém

snížil počet ztrát TLB, může zjistit, které stránky se budou používat a předem pro ně načíst položky v TLB.

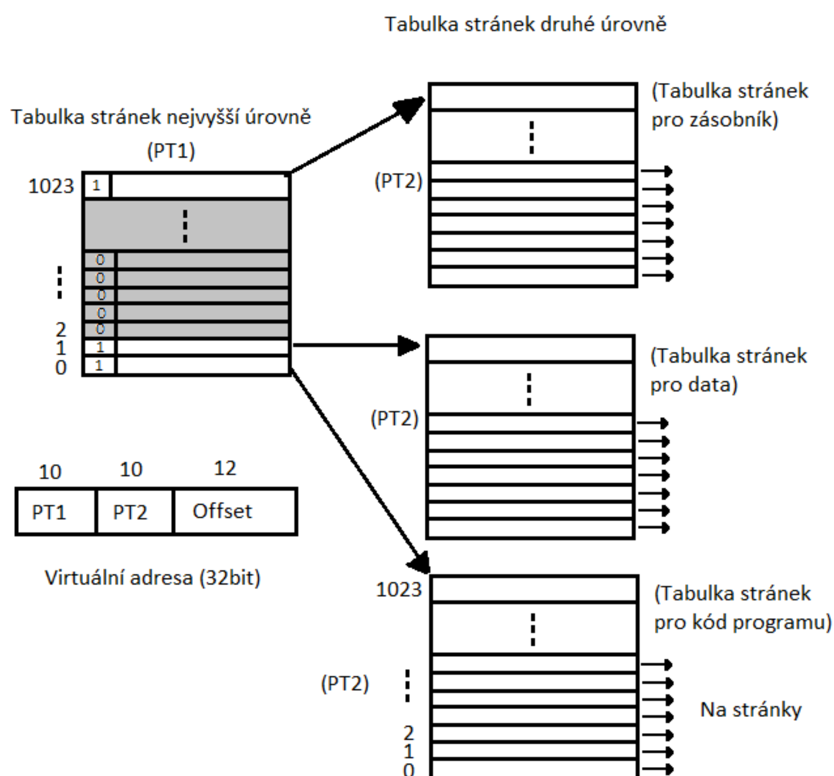
Při použití softwarové správy TLB může dojít ke dvěma druhům chyb (resp. ztrát, angl. miss). K měkké ztrátě (angl. Soft miss) dojde, pokud odkazovaná stránka není v TLB, ale nachází se v paměti v tabulce stránek. Zde je za potřebí vyřešit aktualizaci položky v TLB. K tvrdé ztrátě (angl. Hard miss) dojde, když odkazovaná stránka není v TLB ani v tabulce stránek v paměti. Zde je zapotřebí přístup na disk, kde získáme požadovanou stránku. Je logické, že řešení tvrdé ztráty je několikrát pomalejší než řešení měkké ztráty.

4.2.3.2 Tabulky stránek pro velké paměti

Druhý problém spočívá ve velikosti tabulky stránek ve velkém virtuálním adresním prostoru. K tomuto řešení jsou použity dva přístupy do tabulek stránek.

Víceúrovňová tabulka stránek

Jako první přístup je použití tzv. víceúrovňové tabulky stránek. Příklad je uveden na Obr. 23. Na obrázku je znázorněna 32bitová virtuální adresa, která je rozdělena na 10bitové pole PT1, PT2 a 12bitový offset. Víceúrovňové tabulky zabraňují tomu, aby byly všechny tabulky stránek stále uchovávány v paměti. Zejména by neměly být uchovávány ty, které nejsou potřeba. Jde tedy hlavně o to, že proces používá pouze podmnožinu adres svého virtuálního adresního prostoru, tudíž by stačilo mít v paměti pouze ty položky z tabulky stránek, které bude operační systém potřebovat k překladu.



Obr. 23. Víceúrovňová tabulka stránek
Zdroj: vlastní zpracování

Pro příklad tedy mějme 32bitový virtuální adresní prostor s 4kB stránkami. Předpokládejme, že proces bude skutečně používat pouze 12 MB, a to dolní 4 MB paměti pro kód programu, následující 4 MB pro data a horní 4 MB pro zásobník. Proces má virtuální adresní prostor velký 1 MB (2^{20}), což odpovídá jednomu milionu položek v tabulce stránek. Stačí mít pouze čtyři tabulky stránek, každou mající jeden tisíc položek (2^{10}). Každý z těchto 1024 záznamů reprezentuje 4 M (miliony), protože celý virtuální adresní prostor je velký 4 GB (2^{32}) a tento prostor je rozdělen na kusy o velikosti 4096 B. Proto 4 M (1024 x 4096). Máme následující tabulky:

1. Tabulka nejvyšší úrovně (angl. Top level page table)
2. Tabulka stránek pro kód programu (angl. Program code page table)
3. Tabulka stránek pro data (angl. Data page table)
4. Tabulka stránek pro zásobník (angl. Stack page table)

Záznam, který je umístěn v tabulce stránek nejvyšší úrovně poskytuje adresu, či číslo rámce stránky tabulky stránek druhé úrovně. V tomto příkladě 0. záznam tabulky stránek

nejvyšší úrovně odkazuje na tabulku stránek pro kód programu, 1. záznam odkazuje na tabulku stránek pro data a 1023. záznam odkazuje na tabulku stránek pro zásobník. Ostatní (stínované) položky nejsou využívány. PT2 se nyní využívá jako index do vybrané tabulky stránek druhé úrovně k vyhledání čísla rámce stránky pro získání samostatné stránky.

Mějme 32bitovou virtuální adresu 0x00403004 (4 206 596), zapsanou binárně a rozdělenou na PT1, PT2 a offset.

- Binární adresa = 00000000010000000011000000000100
- PT1 (10 b) = 0000000001
- PT2 (10 b) = 0000000011
- Offset (12 b) = 000000000100

MMU nejprve použije PT1 k indexování do tabulky stránek nejvyšší úrovně pro získání záznamu z položky jedna. PT1 odpovídá binárně záznamu 1. Záznam 1 odpovídá adresám od 4 M do 8 M. Poté pomocí PT2 indexuje právě nalezené tabulky stránek druhé úrovně, kde najde 3. záznam. PT2 odpovídá binárně třetímu záznamu. Tato položka v rámci svého 4M bloku odpovídá absolutním adresám od 4 206 592 do 4 210 687. Dále obsahuje číslo rámce stránky, který obsahuje stránku obsahující virtuální adresu 4 206 596. Pokud není stránka v paměti bude její přítomný/nepřítomný bit roven hodnotě nula, což nám způsobí chybu stránky. Pokud se stránka v paměti nachází, je převzato číslo rámce stránky z tabulky stránek druhé úrovně a je společně spojeno s offsetem k vytvoření fyzické adresy. Tato adresa je poté poslána na paměťovou sběrnici a poslána do paměti.

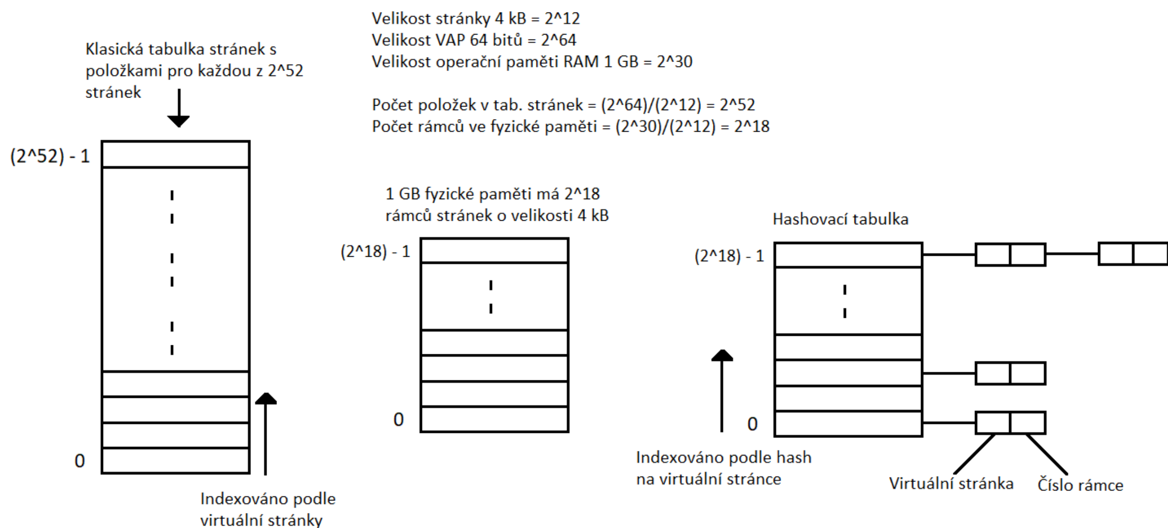
Převrácená tabulka stránek

Víceúrovňová tabulka stránek funguje poměrně dobře u 32bitových adresních prostorů. Běžnější je ovšem 64bitový adresní prostor. Pokud máme 64bitový adresní prostor se 4kB stránkami, potřebujeme v tomto případě tabulku stránek s 2^{52} záznamů. Zde je potřeba pro stránkování virtuálního adresového prostoru jiné řešení. Jedná se o převrácenou tabulku stránek.

V tomto návrhu je v paměti jeden záznam na rámec stránky oproti předchozímu, kde byl jeden záznam na stránku virtuálního adresového prostoru. Například máme 64bitovou virtuální adresu, stránky o velikosti 4 kB a operačním paměť RAM o velikosti 1 GB. Převrácená tabulka bude vyžadovat 262 144 záznamů. Záznam zde sleduje, který proces, či virtuální stránka, je umístěna v rámci stránek.

Ačkoliv mají převrácené tabulky stránek výhodu v šetření obrovského množství prostoru, pokud je virtuální adresní prostor větší než fyzická paměť, mají i velkou nevýhodu. Překlad z virtuální adresy na fyzickou adresu je mnohem složitější. Když proces X odkazuje na virtuální stránku P, tak hardware již nemůže najít fyzickou stránku pomocí indexu stránky P do tabulky stránek, jako to bylo uváděno doteď. Změna je taková, že záznam hledá v celé převrácené tabulce stránek. Kromě toho musí být hledání provedeno u každého odkazu na paměť, nejen při chybě stránek, která nastane, pokud není stránka obsažená v tabulce stránek.

Z tohoto problému se dostaneme pomocí použití TLB. Pokud TLB pojme všechny často používané stránky, může dojít k překladu adres stejně rychle jako u běžných tabulek stránek. Při chybě TLB, kdy není nalezena požadovaná stránka musí být prohledána celá převrácená tabulka stránek. Jedním způsobem, jak toto hledání provést je pomocí hashovací tabulky. Všechny virtuální stránky aktuálně v paměti, které mají stejnou hodnotu hash jsou zřetězeny dohromady, jak je to uvedeno na Obr. 24. Pokud má hashovací tabulka tolik slotů, kolik je fyzických stránek, bude průměrný řetězec dlouhý pouze jeden záznam, což výrazně zvýší mapování. Jakmile je nalezeno číslo rámce stránky, je do TLB zadán nový (virtuální, fyzický) pár.



Obr. 24. Porovnání klasické tabulky stránek s převrácenou tabulkou stránek
Zdroj: vlastní zpracování

Převrácená tabulka stránek je běžná na 64bitových strojích, protože čím je větší velikost stránky, tím bude větší počet záznamů tabulky stránek. Pokud budeme mít 4MB (2^{22}) stránky a 64bitové virtuální adresy, bude potřeba 2^{42} záznamů tabulky stránek.

Více podrobností k problémům stránkování lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 194, kap.: 3.3.3 Speeding Up Paging); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 353, kap.: 8.1 Hardware and control structures).

4.3 Procesy

V posledním tématu bakalářské práce jsou zmiňovány procesy. Jedná se o představení konceptu proces v operačním systému, jeho vytvoření, ukončení a jakých stavů může proces nabývat a jak je s těmito stavy zacházeno. Závěr je věnován plánování procesů a plánovacím strategiím, resp. plánovací politice operačního systému.

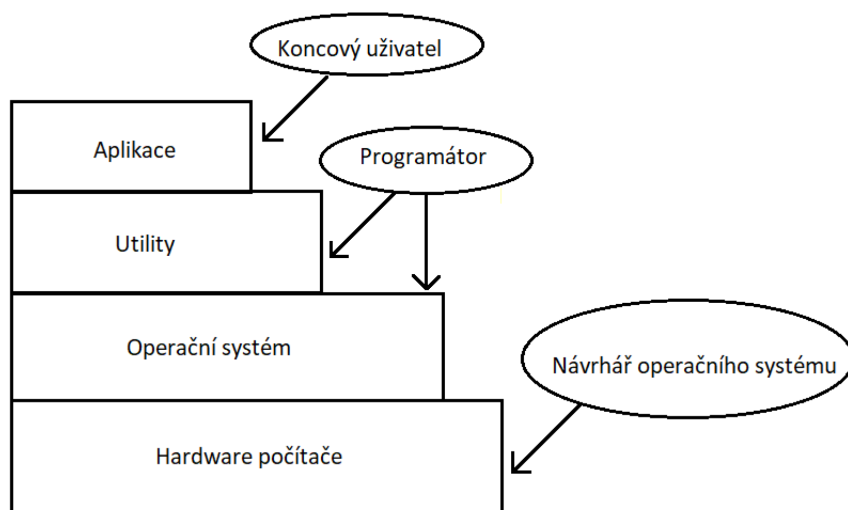
Jedním z hlavních konceptů v každém operačním systému je proces. Zjednodušeně lze říct, že se jedná o abstrakci běžícího programu. Všechno ostatní závisí na tomto konceptu a je důležité, aby návrhář operačního systému, důkladně porozuměl tomu, co je proces.

Procesy jsou jednou z nejstarších a nejdůležitějších abstrakcí, které operační systém poskytuje. Podporují schopnost souběžného provozu, i když je k dispozici pouze jeden procesor. Z jednoho CPU udělají více virtuálních CPU. Bez této procesní abstrakce by moderní výpočetní technika nemohla vůbec existovat.

4.3.1 Blok řízení procesu

Před definováním pojmu proces a širšímu vysvětlení k tomuto tématu je užitečné shrnout některé koncepty:

1. Počítač se skládá z hardwaru jako je procesor, hlavní paměť, I/O moduly, diskové jednotky atd.
2. Aplikace jsou vyvíjeny k provádění určitých úkolů. Obvykle jsou schopni přijímat vstup, provádějí určité zpracování a poté generují výstup.
3. Je velice neefektivní psát aplikace pouze pro jednu platformu.
4. Operační systém je vyvinutý tak, aby poskytoval pohodlné, bezpečné funkce a konstantní rozhraní pro použití uživatelem. Operační systém je vrstva softwaru mezi aplikacemi a hardwarem počítače, která podporuje aplikace a nástroje. Tyto vrstvy jsou uvedeny na Obr. 25.
5. Můžeme si představit, že operační systém poskytuje jednotné, abstraktní znázornění zdrojů, ke kterým je možné požádat i získat přístup. Jakmile operační systém vytvoří tyto abstrakce pro použití v aplikacích, musí také spravovat jejich použití.



Obr. 25. Vrstvy počítačového systému
Zdroj: vlastní zpracování

Proces si můžeme představit jako entitu, která se skládá ze dvou základních prvků. Jedná se o kód programu a soubor dat souvisejících s tímto kódem. Předpokládejme, že procesor začne spouštět tento programový kód, a na tuto entitu odkazujeme jako na proces. V kterémkoli daném časovém okamžiku, kdy program probíhá, lze tento proces jedinečně charakterizovat počtem prvků, popsané následujícími charakteristikami:

Tabulka 1. Charakteristiky procesu

Identifikátor	Jedinečný identifikátor přidělený k tomuto procesu, aby se odlišil od ostatních procesů.
Stav	Pokud se např. proces aktuálně vykonává je ve spuštěném stavu.
Priorita	Určuje úroveň priority k jiným procesům.
Čítač programu	Jedná se o adresu další instrukce v programu, která má být provedena.
Ukazatelé paměti	Zahrnuje ukazatele na programový kód a data spojená s tímto procesem, dále všechny paměťové bloky sdílené s jinými procesy.
Kontextová data	Jedná se o data, která jsou přítomna v registrech procesoru během vykonávání procesu.
Informace o stavu I/O	Zahrnuje nevyřízené I/O požadavky, dále jednotky přiřazené k tomuto procesu.

Účetní informace	Může zahrnovat množství použitého času procesoru, časové limity atd.

Zdroj: Vlastní zpracování

Identifikátor
Stav
Priorita
Čítač programu
Ukazatelé paměti
Kontextová data
Informace o stavu I/O
Účetní informace
⋮

Obr. 26. Ukázka bloku řízení procesu
Zdroj: vlastní zpracování

Informace z tohoto seznamu jsou obvykle uloženy v datové struktuře, která se nazývá blok řízení procesu (angl. Process control block), Obr. 26. Důležitým bodem bloku procesu je, že obsahuje dostatečné informace, aby bylo možné probíhající proces přerušit a později pokračovat v provádění procesu. Blok řízení procesů je klíčovým nástrojem, který umožňuje operačnímu systému podporovat více procesů.

Tím se dostáváme k myšlence, že všechny moderní počítače často dělají několik úkolů naráz. Lidé jsou na toto zvyklí například při práci s osobním počítačem a tuto skutečnost si nemusí plně uvědomovat. Zvažme nyní uživatelský počítač. Při spuštění počítače je spuštěno mnoho procesů, které jsou často uživatelem neznámé. Například může být spuštěn proces pro kontrolu aktualizace, či pro kontrolu příchozí pošty. Kromě toho mohou být spuštěny explicitní uživatelské procesy, například tisknutí souboru, či přesouvání souboru na disk. A toto vše probíhá, zatímco uživatel poslouchá hudbu a surfuje na internetu. Celá tato hromadná aktivita musí být nějakým způsobem řízena. V tomto případě se mluví o systému, který podporuje více procesů, resp. multiprogramovací systém.

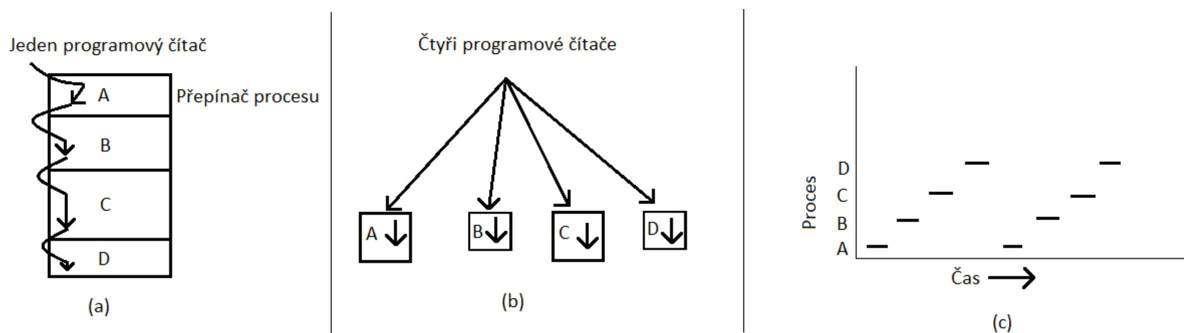
V jakémkoli multiprogramovacím systému probíhá rychlé přepínání procesů. V kterémkoli okamžiku běžící proces na jednoprocesorovém počítači nabývá stavu spuštěný,

resp. běžící a k výměně přidělení CPU dojde pomocí rychlého přepnutí. Ovšem tímto rychlým přepínáním mezi procesy, můžeme vidět, že celý systém pracuje paralelně. Vývojáři operačních systému vyvinuli koncepční model (sekvenční model), který usnadňuje řešení paralelismu.

Když je proces přerušen, aktuální hodnoty čítače programu a registry procesoru (v seznamu se jedná o kontextová data) se uloží do příslušných polí řídícího bloku procesu a stav procesu se změní na jinou hodnotu. Operační systém může nyní zavést a spustit nový proces. Čítač programu a kontextová data pro tento proces jsou načtena do registrů procesoru, tento proces se nyní začne provádět v operačním systému. Můžeme tedy říct, že proces se skládá z programového kódu a souvisejících dat řídícího bloku procesu.

4.3.2 Model procesu

Jedná se o model, kde je veškerý spustitelný software v počítači organizován do několika sekvenčních procesů. Jak zde bylo zmíněno proces je pouze instancí provádějícího programu, včetně aktuálních hodnot částí programu, registrů a proměnných. Každý proces má koncepčně svůj vlastní virtuální procesor. Ve skutečnosti zde probíhá rychlé přepínání za pomoci procesoru. Pro snazší pochopení je ovšem lepší přemýšlet o kolekci procesů běžících paralelně než sledovat, jak CPU přepíná, resp. skáče, z programu na program.



Obr. 27. (a) - Multiprogramování čtyř programů; (b) - Konceptuální model čtyř nezávislých, sekvenčních procesů; (c) - Pouze jeden proces je aktivní v jednu chvíli
Zdroj: vlastní zpracování

Na Obr. 27(a) je vidět multiprogramování čtyř programů v paměti. Na Obr. 27(b) vidíme čtyři procesy, každý s vlastním tokem řízení, tzn. s vlastním čítačem programu, a každý běží nezávisle na ostatních. Samozřejmě existuje pouze jeden fyzický čítač programu, takže když běží nějaký proces, jeho logický programový čítač se načte do skutečného programového čítače. Po jeho dokončení se fyzický programový čítač uloží do procesoru uloženého čítače logických programů v paměti. Na Obr. 27(c) vidíme, že při dostatečně dlouhém časovém

intervalu všechny procesy dosáhly nějakého pokroku. V jednom časovém okamžiku běží pouze jeden proces.

V této části budeme předpokládat, že existuje pouze jedno CPU, i když existují vícejádrové procesory. Prozatím je jednodušší přemýšlet o jednom CPU. S rychlým přepínáním CPU mezi procesy nebude rychlost, jakou proces provádí svůj výpočet, jednotná. Procesy tedy nesmí být naprogramovány s integrovanými předpoklady o načasování. Pokud má nějaký proces kritické požadavky v reálném čase, znamená to, že konkrétní události musí nastat během stanoveného počtu milisekund. Zde je třeba přijmout opatření, aby se zajistilo, že k tomu skutečně dojde. Za normálních okolností však většina procesů není ovlivněna základním multiprogramováním CPU nebo relativní rychlostí různých procesů.

Pro další pochopení je třeba rozlišovat proces a program. Rozdíl mezi procesem a programem není zase tak velký, ale je zcela zásadní. Pro příklad berme v úvahu analogii s lidským prostředím. Máme otce, který chce upéct narozeninový dort pro svoji dceru. Má recept na dort a zásobenou kuchyň se všemi potřebami (jedná se o vstupy). V této analogii je recept program, jedná se o algoritmus vyjádřený k nějaké notaci, který vede k získání výstupu, v tomto případě se jedná o dort. Otec je procesor a vstupní data jsou přísady k dortu. Proces spočívá v tom, že otec přečte recept, načte ingredience a upeče dort.

Nyní do naší analogie přijde syn se zraněním. Otec zaznamená, kde skončil v receptu (stav aktuálního procesu je uložen) a začne ošetřovat syna, jedná se o proces s vyšší prioritou, na pomoc si vezme knihu první pomoci. Zde je myšleno přepínání mezi procesy. Jedná se o přepínaní procesů, kde každý proces má jiný program (jedná se o recept a knihu první pomoci). Když je o syna postaráno, vrátí se otec zpátky k dortu a pokračuje tam, kde skončil.

Hlavní myšlenkou je, že proces je činnost nějakého druhu. Má program, vstup, výstup a stav. Jeden procesor může být sdílen mezi několika procesy, přičemž je použit plánovací algoritmus k určení, kdy se má zastavit na jednom procesu a obsloužit jiný. Pokud jeden program běží dvakrát, počítá se jako dva procesy. Například je možné spustit dva soubory textového procesoru současně, nebo vytisknout dva soubory v jeden okamžik, pokud jsou k dispozici dvě tiskárny. Skutečnost, že dva běžící procesy spouští stejný program není důležitá, stále se jedná o odlišné procesy.

4.3.3 Vytvoření procesu

Operační systém potřebuje nějaký způsob, jakým bude vytvářet procesy. V jednoduchých systémech nebo v systémech určených pro provozování pouze jedné aplikace (např. mikrovlnná trouba) je možné, aby byly k dispozici všechny procesy, které budou vždy potřeba, když se systém spustí. V systémech pro všeobecné účely je nutný určitý způsob vytváření a ukončování procesů podle potřeby během běhu operačního systému. Existují čtyři události, které vytvářejí procesy:

1. Inicializace systému
2. Provedení systémového volání procesu, které je vytvořeno spuštěným procesem
3. Žádost uživatele o vytvoření nového procesu
4. Zahájení dávkové úlohy (angl. Batch job)

Jak již zde bylo zmíněno před spuštěním operačního systému se obvykle vytvoří několik procesů. Některé z nich jsou procesy, které interagují s uživateli a vykonávají za ně určité úlohy. Jiné procesy jsou procesy na pozadí, které nejsou spojeny s konkrétními uživateli, ale místo toho mají konkrétní funkci. Například se jedná o kontrolu příchozí pošty. Takto navržený proces může být většinu dne neaktivní, ale příchozí email může „oživit“ tento proces a převést ho do aktivního stavu. Procesy, které zůstávají takto na pozadí za účelem zpracovávání aktivit, jako je emailový klient, webová stránka, notifikace, tisk atd., se nazývají démoni. Velké systémy obvykle obsahují desítky takovýchto procesů. V operačním systému Windows je možné tyto procesy zobrazit pomocí správce úloh.

Kromě procesů, které jsou vytvořeny při spuštění, vytváříme také nové procesy. Spuštěný proces vydá systémové volání, aby vytvořil jeden či více nových procesů, které mu umožní dosáhnout svého cíle. Například pokud se v síti načítá velké množství dat pro následné zpracování, může být vhodné vytvořit jeden proces pro načtení dat a dát je do sdíleného bufferu, zatímco druhý proces odstraní datové položky a zpracuje je. Na multiprocesoru je toto řešeno pomocí běhu každého procesu na jiném CPU, což může zrychlit celkovou práci.

V interaktivních systémech mohou uživatelé spustit libovolný program. Tato akce spustí nový proces a v něm se spustí vybraný program. Většině procesů, které jsou spuštěny uživatelem je přiřazeno okno, které je následně zobrazeno. Uživatelé tedy mohou mít otevřeno více oken najednou a pro každé okno je spuštěn proces. Uživatel může dále s těmito okny interagovat například pomocí myši a komunikovat tak s procesem. Například může proces ukončit zavřením jeho okna.

Poslední situace, ve které může vzniknout proces se týká dávkových systémů. Zde mohou uživatelé odesílat dávkové úlohy do systému. Když se operační systém rozhodne, že má dostatek prostředků ke spuštění jiné úlohy, vytvoří nový proces a spustí další úlohu, kterou získá ze vstupní fronty dávkového systému.

Technicky je ve všech těchto případech nový proces vytvořen tak, že stávající proces provede systémové volání k vytvoření procesu. Tímto procesem může být spuštěný uživatelský proces, systémový proces vyvolaný klinutím na klávesnici či kliknutím na myš a dávkový systém. Proces tedy provede spuštění systémového volání a dojde k vytvoření nového procesu. Toto systémové volání říká operačnímu systému, aby vytvořil nový proces a označuje, jaký program v něm spustit.

V systému UNIX, existuje pouze jedno systémové volání k vytvoření nového procesu, systémové volání fork. Toto volání vytvoří přesný klon procesu volání. Po vykonání volání máme dva procesy, nadřazený (angl. Parent process) a podřazený (angl. Child process), stejný obraz paměti, stejné řetězce prostředí a stejné otevřené soubory. Podřazený proces potom vykonává příkaz execve, nebo podobné systémové volání, aby změnil svůj obraz paměti a spustil nový program. Například pokud uživatel zadá do shellu příkaz sort, shell se rozdělí na dva procesy a podřízený proces vykoná příkaz sort. Důvodem tohoto, řekněme, dvoustupňového procesu je umožnit podřízenému procesu manipulovat se svými deskriptory souborů po provedení volání fork, ale před vykonáním příkazu execve, aby bylo dosaženo přesměrování standartního vstupu, výstupu a chyb.

Naopak ve Windows systémech máme systémové volání funkce CreateProcess, které zpracovává jak vytvoření procesu, tak i načítání správného programu do nového procesu. Toto volání má 10 parametrů, které zahrnují program, který má být spuštěn, parametry příkazového řádku, atributy zabezpečení, informace o prioritě a specifikaci okna, které má být pro proces vytvořeno. Navíc v systému máme přibližně 100 dalších funkcí pro správu a synchronizaci procesů.

Jak v systému UNIX, tak v systému Windows mají po vytvoření procesu nadřazený a podřazený proces vlastní odlišné adresní prostory. Pokud některý z procesů změní slovo v adresním prostoru, změna nebude pro druhý proces viditelná. V systému UNIX je počáteční adresní prostor podřazeného kopií rodičovského adresního prostoru, ale stále se jedná o dva odlišné adresní prostory. V systémech Windows se adresní prostor nadřazeného a podřazeného procesu liší již od začátku, od vytvoření nového procesu.

4.3.4 Ukončení procesu

Po vytvoření procesu se tento proces spustí a vykonává svoji práci. Nic však netrvá věčně a ani procesy nejsou nekonečné. Dříve nebo později bude nový proces ukončen. Obvykle se jedná o jednu z následujících podmínek:

1. Normální výstup (dobrovolné ukončení)
2. Chyba výstupu (dobrovolné ukončení)
3. Závažná chyba (nedobrovolné ukončení)
4. Ukončen, resp. zabit, jiným procesem (nedobrovolné ukončení)

Většina procesů se ukončí, pokud mají splněno, resp. dokončili svoji úlohu. Když kompilátor zkompiluje program, který mu byl přidělen, a provede systémové volání a sdělí operačnímu systému, že jeho aktivita je dokončena. Toto volání v systému UNIX je nazýváno exit, v systému Windows se jedná o volání ExitProcess. Programy, které jsou zaměřené na ovládání uživatelem, jako jsou textové procesory, internetový prohlížeč a podobné programy mají možnost, kterou může uživatel sdělit procesoru, aby odstranil dočasné soubory, které jsou aktuálně otevřeny a poté proces ukončil.

Druhý důvod ukončení je chyba způsobená procesem, často kvůli chybě programu. Mezi příklady patří provádění neplatné instrukce, například odkazování na neexistující místo v paměti. V některých systémech (např. UNIX) může proces říct operačnímu systému, že si přeje zpracovat určité chyby sám. V takovém případě je proces přerušen, místo ukončení, když dojde k chybě.

Třetí důvod ukončení je, že proces zjistí závažnou chybu. Například uživatel zadá příkaz cc file.c, aby zkompiloval soubor file.c, jenže žádný takový soubor neexistuje. Kompilátor jednoduše proces ukončí. Procesy orientované na interakci uživatelem se obecně neukončí, pokud dostanou špatné vstupní parametry. Místo toho zobrazí uživateli dialogové okno a požádají ho o přiřazení správného souboru.

Posledním důvodem ukončení je to, že proces provede systémové volání a řekne operačnímu systému, aby ukončil (resp. zabil) jiný proces. V systému UNIX je toto volání nazýváno kill. V systémech Windows se jedná o volání TerminateProcess. V obou případech musí mít „zabiják“ potřebné povolení k tomu, aby mohl proces ukončit.

V některých systémech, když je proces ukončen, ať už dobrovolně či nedobrovolně, jsou okamžitě ukončeny také ostatní procesy, které vytvořil.

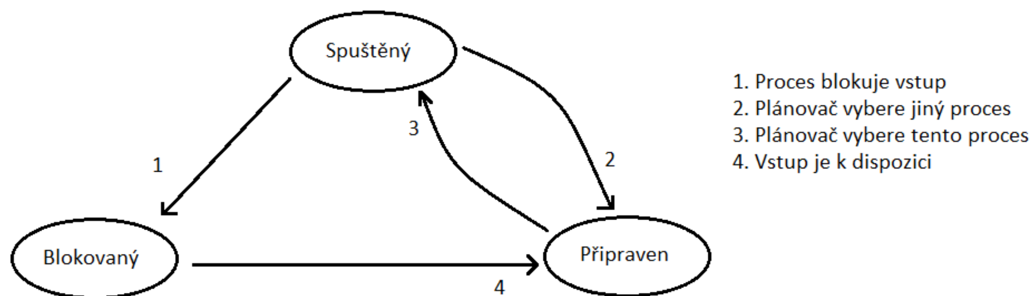
4.3.5 Stavy procesu

Každý proces je nezávislou entitou, která má vlastní čítač programu a vnitřní stav. Přesto je nutné, aby i přes svou nezávislost dokázaly interagovat s ostatními procesy. Jeden proces může generovat výstup a druhý proces může tento výstup potřebovat jako vstup.

Když je proces blokován, dochází k situaci, kdy proces nemůže pokračovat. Typická situace blokování nastává právě při čekání na vstup, který stále není k dispozici. Je také možné zastavit proces, který je koncepčně připravený a schopný provozu, protože se operační systém rozhodl přidělit CPU jinému procesu. V druhém případě se jedná o technickou stránku systému, máme zde nedostatek CPU, aby každý proces dostal svůj vlastní privátní procesor. Na Obr. 28 je znázorněn stavový diagram ukazující tři stavy, ve kterých se může proces nacházet:

1. Spuštěný (angl. Running) – v daném okamžiku proces využívá CPU
2. Připraven (angl. Ready) – jedná se o spustitelný proces, který je nyní zastaven, aby mohl běžet jiný proces
3. Blokovaný (angl. Blocked) – nelze spustit, dokud nenastane externí událost

První dva stavy jsou si podobné. V obou případech je možné proces spustit, pouze v druhém případě pro něj není dočasně k dispozici žádný procesor. Třetí stav se liší od prvních dvou v tom, že proces nelze spustit vůbec, i když CPU nemá nic jiného na práci.



Obr. 28. Stavy procesu
Zdroj: vlastní zpracování

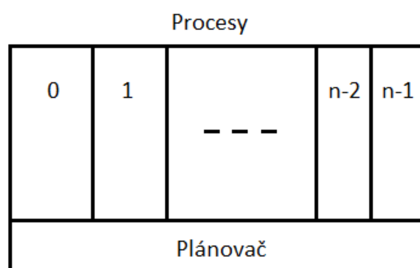
Jak je znázorněno na Obr. 28 mezi těmito třemi stavy jsou možné čtyři přechody. K přechodu dojde, když operační systém zjistí, že proces nemůže v danou chvíli pokračovat. V některých systémech může proces provést systémové volání, například systémové volání pause, aby se dostal do blokovaného stavu. V jiných systémech, když proces čte z kanálu nebo speciálního souboru (např. z terminálu) a není k dispozici žádný vstup, je proces automaticky zablokován.

Přechody 2 a 3 jsou způsobeny plánovačem procesů, který je součástí operačního systému, aniž by o nich proces věděl. Přechod 2 nastane, když plánovač rozhodne, že běžící proces běžel již dostatečně dlouhou dobu a je čas, aby bylo CPU přiděleno jinému procesu. K přechodu 3 dochází, když všechny procesy mají spravedlivý podíl času a CPU tak může spustit první proces. Důležitou částí je zde plánování, resp. rozhodování, který proces by měl kdy a jak dlouho běžet.

Přechod 4 nastane, když dojde k externí události, na kterou proces čekal. Pokud v daném okamžiku není spuštěn jiný proces, spustí se přechod 3 a proces je následně přidělen CPU. V opačném případě bude muset čekat v připraveném stavu, dokud nebude CPU k dispozici.

Pomocí modelu procesu je snazší přemýšlet o tom, co se děje uvnitř systému. Některé procesy spouští programy, které provádějí příkazy zadané uživatelem. Další procesy jsou součástí systému a zpracovávají úkoly, jako jsou např. požadavky na souborové služby nebo správa podrobností o spuštění disku. Dále jsou zde uživatelské procesy, diskové procesy nebo terminálové procesy atd. Tyto procesy mohou být blokovány nebo čekají na vykonání vhodné události.

Tento pohled vede k Obr. 29. Zde je nejnižší úroveň operačního systému plánovač, na kterém je celá řada procesů. Veškeré zpracování přerušení a podrobnosti o skutečně spouštěných a zastavených procesech jsou skryty v tom, co se zde nazývá plánovač. Plánování procesů je v popsáno v podkapitole 4.3.6.



Obr. 29. Nejnižší vrstva procesně strukturovaného operačního systému zpracovává přerušení a plánování (plánovač)
Zdroj: vlastní zpracování

Více podrobností ke stavům procesů lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 90, kap.: 2.1.5 Process States); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 111, kap.: 3.2 Process states).

4.3.6 Plánování procesů

Plánování procesů (angl. scheduling) je úkol jádra operačního systému, ve kterém je spuštěno více procesů najednou. Týká se víceúlohových systémů, které podporují multitasking. Plánování procesů řeší výběr, kterému následujícímu procesu bude přidělen procesor a proces bude tak aktivní, tzn. bude se nacházet ve spuštěném stavu. Běžné operační systémy vyžadují, aby byla v přidělování procesoru jednotlivým procesům zachována jistá míra spravedlnosti.

Cílem plánování procesu je tedy přiřadit procesy, které mají být zpracovány procesorem nebo procesory v průběhu času, a to takovým způsobem, který splňuje systémové cíle, jako je doba odezvy, propustnost a efektivita procesoru. V mnoha systémech je plánovací aktivita rozdělena následovně:

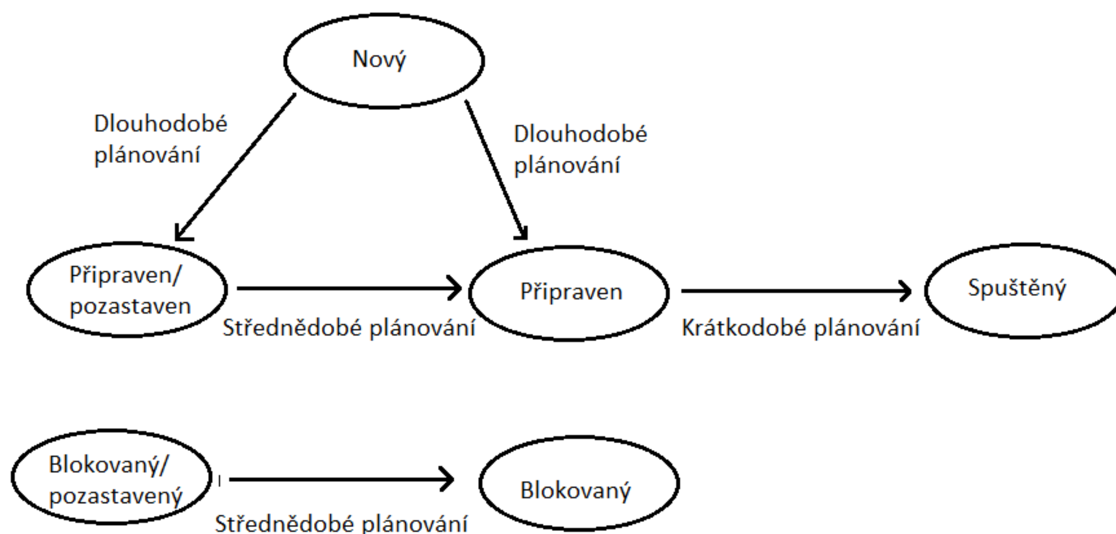
1. Dlouhodobé plánování
2. Střednědobé plánování
3. Krátkodobé plánování

Tabulka 2. Shrnutí typů plánování

Dlouhodobé plánování	Rozhodnutí o přidání do skupiny procesů, které mají být provedeny.
Střednědobé plánování	Rozhodnutí o přidání počtu procesů, které jsou částečně nebo úplně v hlavní paměti.
Krátkodobé plánování	Rozhodnutí o tom, který dostupný proces bude zpracován procesorem.

Zdroj: vlastní zpracování

Při vytváření nového procesu se provádí dlouhodobé plánování. Zde nastává rozhodnutí, zda přidat nový proces do sady procesů, které jsou aktuálně aktivní. Střednědobé plánování je součástí funkce výměny (angl. swapping). Jedná se o rozhodnutí, zda přidat proces k těm, které jsou alespoň částečně v hlavní paměti, a proto nejsou k dispozici pro provedení. Krátkodobé plánování je finální rozhodnutí, kterému následujícímu procesu bude přiděleno CPU.



Obr. 30. Plánování a přechody stavu procesu
Zdroj: vlastní zpracování

Plánování ovlivňuje výkon systému, protože určuje, které procesy budou čekat a které budou aktivní. Plánování je v zásadě otázkou správy front tak, aby se minimalizovalo zpoždění čekání ve frontě a optimalizoval se výkon ve frontě.

4.3.6.1 Dlouhodobé plánování

Dlouhodobý plánovač určuje, které programy jsou přijímány do systému ke zpracování. Jakmile je pracovní nebo uživatelský program přijat, stává se procesem a je přidán do fronty pro krátkodobý plánovač. V některých systémech začíná nově vytvořený proces v podmínce výměny. V takovém případě je proces přidán do fronty pro střednědobý plánovač.

V dávkovém systému (angl. Batch system) jsou nově odeslané úlohy směrovány na disk a jsou drženy v tzv. dávkové frontě. Pokud je to možné, dlouhodobý plánovač vytváří procesy z fronty. Zde jsou zahrnuta dvě rozhodnutí. Nejprve musí plánovač rozhodnout, kdy může operační systém převzít jeden nebo více dalších procesů. Zadruhé, plánovač musí rozhodnout, kterou úlohu nebo úlohy přijme a udělá z nich procesy.

Rozhodnutí o tom, kdy vytvořit nový proces je obvykle řízeno požadovaným stupněm multiprogramování. Čím více procesů se vytvoří, tím menší je procento času, které může využít každý proces. Dlouhodobý plánovač tak může omezit stupeň multiprogramování, aby poskytl dostatečnou službu sadě procesů. Při každém ukončení úlohy se plánovač může rozhodnout přidat jednu nebo více nových úloh.

Rozhodnutí o tom, jakou úlohu přijmout, může být na jednoduchém principu „kdo dřív přijde, ten dřív bere“, nebo to může být nástroj pro správu výkonu systému. Použitá kritéria mohou zahrnovat prioritu, očekávanou dobu provedení a požadavky I/O. Například pokud jsou informace k dispozici, může se plánovač pokusit zachovat kombinaci procesů vázaných na procesor a I/O. Rozhodnutí může být učiněno v závislosti na tom, které I/O prostředky mají být požadovány ve snaze vyvážit celkové využití I/O modulů.

4.3.6.2 Střednědobé plánování

Střednědobé plánování používají systémy s virtuální pamětí. Jde o výběr, kde blokovaný nebo připravený proces bude odsunut z vnitřní paměti na pevný disk, není-li k dispozici dostatek vnitřní paměti. Jedná se o mechanismus výměny, resp. swapování. Důvodem pro odložení procesu může být absence aktivity procesu, nízká priorita, časté výpadky stránek, odblokovaný proces, který již nečeká na systémové prostředky nebo alokace příliš velké části paměti, když je potřeba paměť pro jiné procesy. Rozhodnutí o výměně je obvykle založeno na aktuální potřebě procesu.

4.3.6.3 Krátkodobé plánování

Co se týká frekvence provádění, dlouhodobý plánovač se vykonává relativně často a činí rozhodnutí, zda přijmout jeden z dostupných procesů. Střednědobý plánovač se spouští ještě častěji, aby se rozhodlo o jeho přesunutí (swapování). Krátkodobý plánovač, známý též jako dispečer, se spouští nejčastěji a dělá rozhodnutí o tom, který následující proces se má provést, resp. kterému z připravených procesů bude přidělen procesor.

Tento plánovač je také vyvolán vždy, když dojde k události, která může vést k blokování aktuálního procesu nebo která může poskytnout příležitost předcházet aktuálně běžícímu procesu ve prospěch jiného. Mezi příklady takovýchto událostí patří:

- Přerušení I/O
- Volání operačního systému
- Signály (např. semafory)

4.3.7 Strategie plánování procesů

Hlavním cílem krátkodobého plánování je alokování času procesoru tak, aby optimalizoval jeden nebo více aspektů chování operačního systému. Obecně je pro toto stanovena sada kritérií, podle nichž lze vyhodnotit různé plánovací strategie.

Běžně používaná kritéria můžeme rozdělit do dvou kategorií. Nejprve můžeme rozlišovat mezi uživatelsky orientovanými a systémově orientovanými kritérii. Kritéria zaměřená na uživatele se týkají chování systému vnímaného jednotlivým uživatelem nebo procesem. Příkladem je doba odezvy v interaktivním systému. Doba odezvy je uplynulý čas mezi odesláním požadavku, dokud se odpověď nezačne zobrazovat jako výstup. V případě doby odezvy může být definována hodnota 2 sekundy. Pak by cílem plánovacího mechanismu mělo být maximalizovat počet uživatelů, kteří zaznamenají průměrnou dobu odezvy 2 sekundy a méně.

Další kritéria jsou orientována na systém. Tyto kritéria se zaměřují na efektivnost procesoru. Příkladem je průchodnost. Jedná se o rychlost, ve které jsou procesy dokončeny. Průchodnost přispívá užitečnou mírou k výkonu systému. Zaměřuje se tedy více na výkon systému než na službu poskytnutou uživateli. Systémově orientovaná kritéria mají menší význam pro systémy jednoho uživatele. Na jednouživatelském systému není důležité dosáhnout vysokého použití procesoru nebo vysoké propustnosti, pokud je přijatelná reakce systému k aplikacím uživatele.

Další kategorií je rozdělení kritérií, která souvisejí s výkonem a které nejsou přímo spojována s výkonem. Kritéria, která souvisejí přímo s výkonem jsou kvantitativní a obecně je lze snadno měřit. Mezi příklady tohoto kritéria může patřit již zmíněná doba odezvy, či propustnost. Kritéria, která přímo nesouvisí s výkonem jsou kvalitativní povahy, nebo se nepoužívají k měření či analýze. Příkladem takového kritéria může být předvídatelnost.

Tabulka 3. shrnuje plánovací kritéria. Tato kritéria jsou na sobě vzájemně závislá a je nemožné je optimalizovat současně. Například poskytnutí doby odezvy může vyžadovat plánovací algoritmus, který často přepíná mezi procesy. To zvyšuje režii systému, ale snižuje propustnost. Návrh těchto kritérií, jinak řečeno politiky plánování, zahrnuje kompromisy mezi požadavky.

Tabulka 3. Plánovací kritéria

Uživatelsky orientované, související s výkonem	
Doba obratu (angl. Turnaround time)	Jedná se o časový interval mezi předložením procesu a jeho dokončením. Zahrnuje skutečný čas spuštění plus čas strávený čekáním na zdroje.

Doba odezvy (angl. Response time)	Jedná se o dobu od odeslání požadavku do zahájení přijímání odpovědi. Plánovací politika by se měla snažit dosáhnout nízké doby odezvy.
Deadlines	Pokud lze určit lhůtu ukončení procesu, měla by plánovací politika podřídit jiné cíle maximalizaci procenta splněných termínů.
Uživatelsky orientované, ostatní	
Předvídatelnost (angl. Predictability)	Daná úloha by měla běžet přibližně stejnou dobu bez ohledu na zatížení systému. Široká variace času odezvy nebo doby obratu může signalizovat výkyvy v celkové zátěži systému.
Systémově orientované, související s výkonem	
Propustnost (angl. Throughput)	Plánovací politika by se měla pokusit maximalizovat počet procesů dokončených za jednotku času. Jedná se o míru toho, kolik práce se provádí.
Využití procesoru (angl. Processor utilization)	Jedná se o procento času, kdy je procesor zaneprázdněn.
Systémově orientované, ostatní	
Spravedlnost (angl. Fairness)	Při absenci pokynů od uživatele by se s procesy mělo zacházet stejně a žádný proces by neměl trpět „hladem“.
Prosazování priorit (angl. Enforcing priorities)	Pokud jsou procesům přiřazeny priority, plánovací politika by měla upřednostňovat procesy s vyšší prioritou.
Vyvažování zdrojů (angl. Balancing resources)	Plánovací politika by měla udržovat zdroje systému zaneprázdněné. Toto kritérium zahrnuje střednědobé a dlouhodobé plánování.

Zdroj: vlastní zpracování

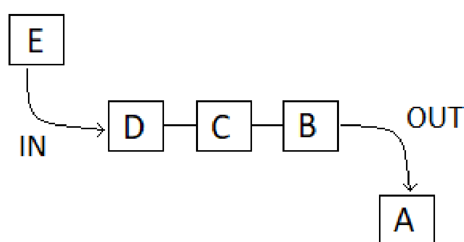
V případě, že máme pouze jeden procesor, jak je zde zmiňováno, je potřeba zvolit, který proces bude spuštěn jako další. Toto nám obstarává již zmíněný plánovač a algoritmus, který plánovač využívá. Tento algoritmus je nazýván plánovací algoritmus. Algoritmy plánování můžeme rozdělit do dvou skupin: nepreemptivní plánovací algoritmus a preemptivní plánovací algoritmus.

Nepreemptivní plánovací algoritmus vybere proces ke spuštění a nechá jej běžet, dokud není tento proces zablokován, nebo dobrovolně neuvolní CPU tím, že dokončí svoji úlohu. To znamená, že operační systém nemůže násilně odebrat procesu přidělené systémové prostředky a nedrží tedy absolutní kontrolu nad počítačem. Důvodem může být absence hardwarových možností procesoru nebo nedostatečné využití schopností pokročilejšího procesoru operačním systémem.

Preemptivní plánovací algoritmus vybere proces a nechá ho běžet maximálně stanovený čas. Pokud proces stále běží i po vyčerpání svého času (kvantum procesu), je pozastaven a plánovač vybere jiný proces. Zde operační systém drží kontrolu nad počítačem a jeho prostředky, které procesům přiděluje. V tomto případě je operační systém schopen procesu odebrat CPU.

4.3.7.1 First-in-first-out (FIFO)

Nejjednodušší plánovací strategií, resp. plánovací politikou, je „první dovnitř, první ven“ (angl. FIFO). Jedná se o striktní schéma, které znázorňuje řazení procesů do fronty. Jakmile je nějaký proces připraven, připojí se do fronty. Když se aktuálně spuštěný proces přestane vykonávat, je pro další spuštění vybrán proces, který byl ve frontě nejdéle.



Obr. 31. Znázornění fronty
Zdroj: vlastní zpracování

Výhoda tohoto přístupu spočívá v jeho snadné srozumitelnosti. S tímto algoritmem sledujeme propojený seznam všech připravených procesů. Výběr procesu vyžaduje pouze

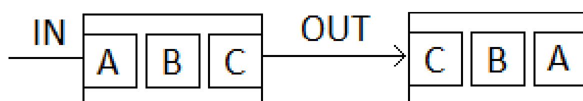
odebrání procesu ze začátku fronty. Přidání nové úlohy nebo odblokovaného procesu vyžaduje připojení na konec fronty.

Nevýhoda FIFO spočívá v tom, že má tendenci upřednostňovat procesy vázané na procesor před procesy vázané na I/O moduly. Berme v úvahu, že existuje kolekce procesů, z nichž jenom jeden proces využívá procesor (tzn. je vázaný na procesor) a řada z nich upřednostňuje I/O (tzn. jsou vázané na I/O). Když běží proces vázaný na procesor, musí všechny ostatní procesy vázané na I/O čekat. V tomto okamžiku může být většina nebo dokonce všechna I/O zařízení nečinná, i když pro ně existuje potenciální práce. Pokud je zde proces vázaný na procesor blokovaný, procesor se stane nečinným. FIFO může mít za následek neefektivní využití procesoru a I/O zařízení.

FIFO není sama o sobě atraktivní plánovací politikou pro jednoprocesorový systém. Často se však kombinuje s dalšími schématy, aby se zajistil efektivní plánovač. Plánovač tedy může udržovat několik front, jednu pro každou prioritní úroveň a odesílat každou frontu klasickým FIFO způsobem.

4.3.7.2 Last-in-first-in (LIFO)

Podobným přístupem je řazení do zásobníku, angl. LIFO. Jedná se o datovou strukturu, kde je manipulace s procesy taková, že jsou řazeny do zásobníku a poslední vložený proces je převzatý CPU jako první. Nově příchozí procesy zde nejsou řazeny na konec, ale na začátek fronty připravených procesů. Jako následující proces je odebrán proces, který strávil v zásobníku nejkratší dobu.

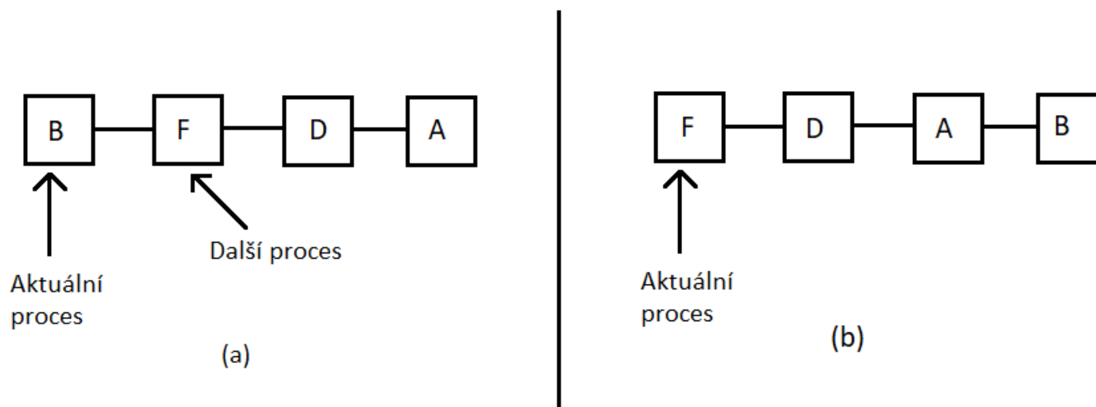


Obr. 32. Znázornění zásobníku
Zdroj: vlastní zpracování

Hlavní nevýhodou řazení do zásobníku je fakt, že brzké procesy nemusí dostat přidělené CPU. Použijeme zde analogii z lidského prostředí, konkrétně z restaurace. Pokud budeme mít kuchaře, který bude zpracovávat objednávky, tzn. vařit tak, jak zákazníci přicházejí, tak v případě prvně příchozích hostů do zásobníku se vůbec nemusí dostat k objednávce. Každá příprava jídla trvá jinou dobu a pokud budou stále přicházet další hosté, tak dřív příchozím se vyřízení objednávky vzdaluje víc a víc.

4.3.7.3 Round Robin

Jedním z nejférovějších a velice používaným algoritmem je algoritmus round robin. Každému procesu je zde přidělen časový interval, který nazýváme jeho kvantem a během tohoto kvanta procesu je povolen jeho běh. Pokud proces stále běží po vyčerpání svého kvanta, je CPU předáno jinému procesu. Pokud se proces zablokuje nebo se dokončí před ukončením svého kvanta, přepnutí CPU se také provede, aniž by se čekalo na vyčerpání kvanta procesu. Plánovač zde musí udržet jediný seznam spustitelných procesů, jak je znázorněno na Obr. 33(a). Když proces spotřebuje své kvantum, umístí se na konec seznamu, jak je znázorněno na Obr. 33(b).



Obr. 33. (a) - Seznam spustitelných procesů; (b) - Seznam spustitelných procesů potom, co proces B vyčerpal své kvantum
Zdroj: vlastní zpracování

Problémem algoritmu round robin je délka kvant. Přechod z jednoho procesu na druhý vyžaduje určitou dobu pro provedení administrace, tzn. ukládání a načítání registrů, paměťových map, opětovné načítání mezipaměti atd. Předpokládejme, že tento přepínač procesu, či přepínač kontextu, jak se někdy nazývá, bude trvat 1 milisekundu. Kvantum procesu bude nastaveno na 4 milisekundy. S těmito parametry bude muset CPU po provedení 4 milisekund „promarnit“ 1 milisekundu kvůli přepínání procesů. Z toho vyplývá, že 20 % času CPU bude využito na přepínání procesu. Což je v tomto případě celkem hodně.

Aby se zde zlepšila efektivita CPU, mohlo by se nastavit kvantum procesu na 100 milisekund. Nyní je promarněný čas CPU 1 %. Pokud budeme mít 50 požadavků, které dorazí na server se značně se měnícími požadavky na CPU, potom se na seznam spustitelných procesů zařadí těchto 50 procesů. Pokud je CPU nečinný, tak se první proces spustí okamžitě, druhý se může spustit až o 100 milisekund později atd. Poslední proces možná bude muset čekat 5 sekund, než přijde na řadu, ovšem za předpokladu, že všechny procesy využijí celé své kvantum. Většina uživatelů již bude 5sekundovou odezvu od serveru vnímat jako pomalou.

Závěr lze formulovat následovně: nastavení příliš krátkého kvanta způsobí mnoho procesních přepínačů a sníží se tím účinnost CPU. Naopak dlouhé nastavení může způsobit špatnou či pomalou reakci na krátké interaktivní požadavky. Uvádí se, že kompromisem kvanta procesu je něco mezi 20-50 milisekundami.

Více podrobností k plánování procesu a plánovacím algoritmům lze nalézt v knihách Modern Operating Systems, Andrew S. Tanenbaum (3. vyd., 2008, str. 145, kap.: 2.4 Scheduling); Operating systems: Internals and design, W. Stallings (6. vyd., 2009, str. 406, kap.: 9.1 Types of processor scheduling, dále str. 410, kap.: 9.2 Scheduling algorithms).

5 Praktická část

5.1 Animační program Synfig Studio

Pro tvorbu animací byl zvolen program Synfig Studio. Animační program Synfig Studio je bezplatný vektorový 2D animační software, který je publikován jako open-source software pod GNU General Public License. Program je zdarma ke stažení na oficiálních stránkách programu (https://www.synfig.org/). Synfig Studio podporuje operační systémy Windows, OS X, Linux. Na stránkách https://wiki.synfig.org/Category:Manual, či na stránkách https://synfig.readthedocs.io/en/latest/index.html lze nalézt dokumentaci pro jednotlivé nástroje, které program nabízí.

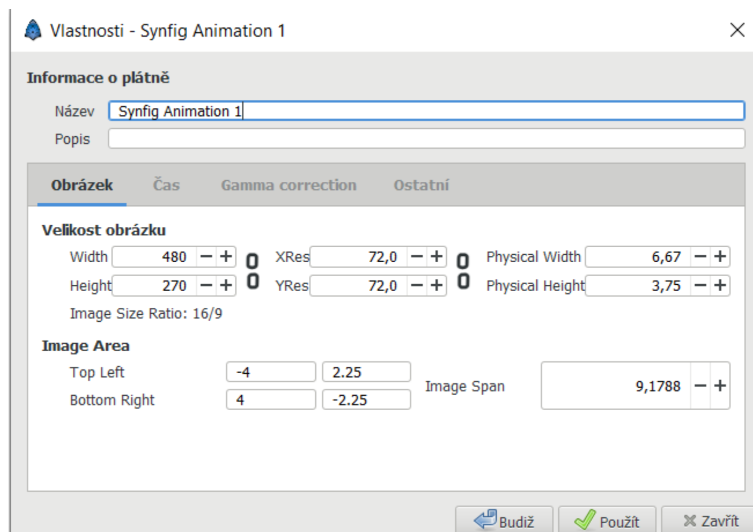
Program Synfig Studio byl vybrán z důvodů, že obsahuje dostatečné nástroje k tvorbě animací pro podporu výuky operačních systémů, je ke stažení zdarma a není problém nalézt k tomuto programu instruktážní videa a tutoriály sloužící pro lepší porozumění programu. Mezi hlavní požadavky, které byly brány v úvahu při volbě animačního programu patřilo nastavování parametrů pro práci s plátnem a časovou osou. Posledním požadavkem byl výstup vytvořeného projektu. Pro nejlepší a nejjednodušší variantu byl vybrán export do souboru GIF, který může být vložen ke studijnímu textu na Blackboard ke kterému mají studenti UHK přístup a nemusí si stahovat dodatečný software pro zobrazení animace a vše bude umístěno na jednom místě. Alternativou je formát MP4, kde si navíc studenti mohou animaci v libovolném čase pozastavit. Z důvodu ukončení podpory Flash od Adobe nebyl brán v potaz žádný animační program využívající Flash, jako například Macromedia Flash, který byl využíván na předmětu Multimediální systémy 1.

Všechny animace byly tvořeny ve větším měřítku obrazu (1280 x 720) pro účely zobrazení animací v dostatečném rozlišení bez nutnosti obraz přibližovat, kde by mohlo docházet ke kompresi videa a ztrátě kvality. Co se týká časové osy bylo změněno výchozí nastavení z přehrávání 24 snímků za sekundu na 10 snímků za sekundu, což je zcela adekvátní vzhledem k vytvořeným animacím.

5.1.1 Vytvoření nového projektu

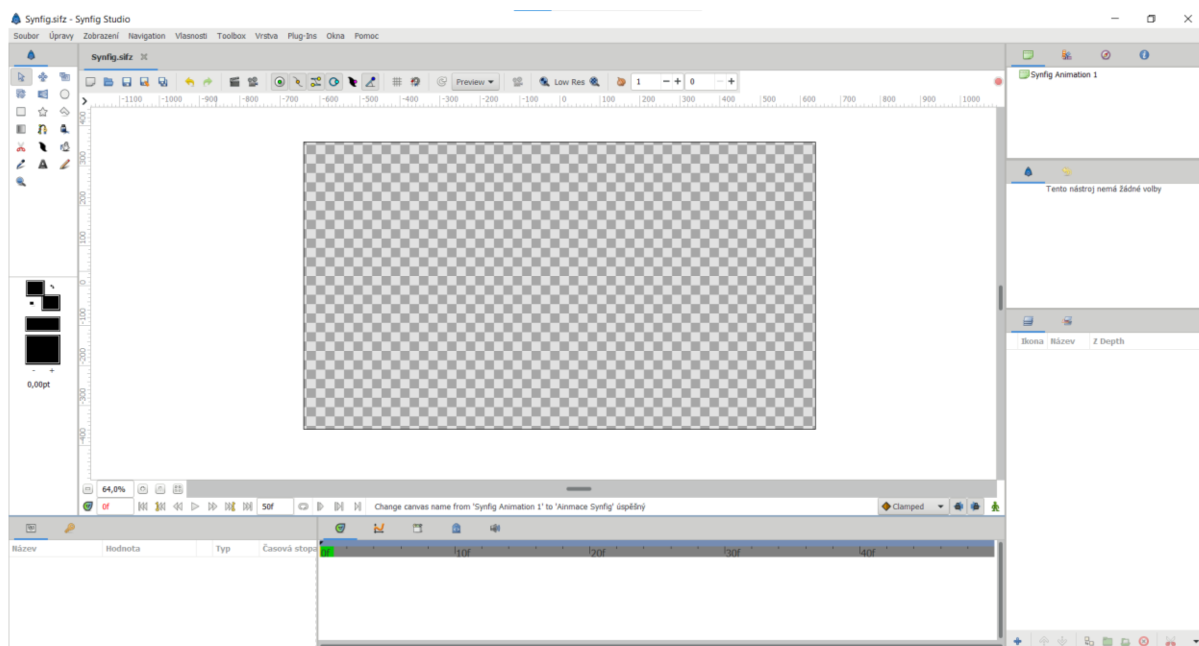
Při vytváření nového projektu si může uživatel určit vlastnosti projektu, jako je například plátno, resp. okno projektu a časovou osu. Na kartu Vlastnosti se uživatel dostane pomocí klávesové zkratky F8 nebo přes horní menu aplikace: Vlastnosti → Properties. Na kartě Vlastnosti je možno změnit název projektu, v záložce Obrázek je možno si nastavit výšku

a šířku okna projektu. V záložce Čas je možno nastavit délku časové osy a jakou frekvencí se bude animace přehrávat, ve výchozím nastavení je nastaveno 24 snímků za sekundu.



Obr. 34. Karta vlastnosti, výchozí nastavení
Zdroj: vlastní zpracování

Po nastavení námi zvolených vlastností můžeme nyní vstoupit do prostoru animace, kde můžeme vkládat jednotlivé objekty, či využívat nástrojů programu a pracovat s časovou osou.



Obr. 35. Prázdné okno nového projektu
Zdroj: vlastní zpracování

5.1.2 Nástroje programu

V levé části okna nového projektu se nachází všechny nástroje, kterými program Synfig Studio disponuje. Nejčastěji pro účely bakalářské práce byly využívány nástroje pro vytváření tvarů (nástroj obdélník), nástroj spline tool, který byl využíván pro kreslení čar. Nástroj pro kreslení byl využíván pro vytvoření nepravidelného obrazce, nejčastěji pro vytvoření nějakého ukazatele, například šipky, která se neskládá z přímých čar. Tudíž nebylo adekvátní využívat nástroj spline tool. Dále nástroj pro text, který byl použit pro vytvoření textového doprovodu pro lepší pochopení animace.

Obr. 36. Nástroje programu
Zdroj: vlastní zpracování

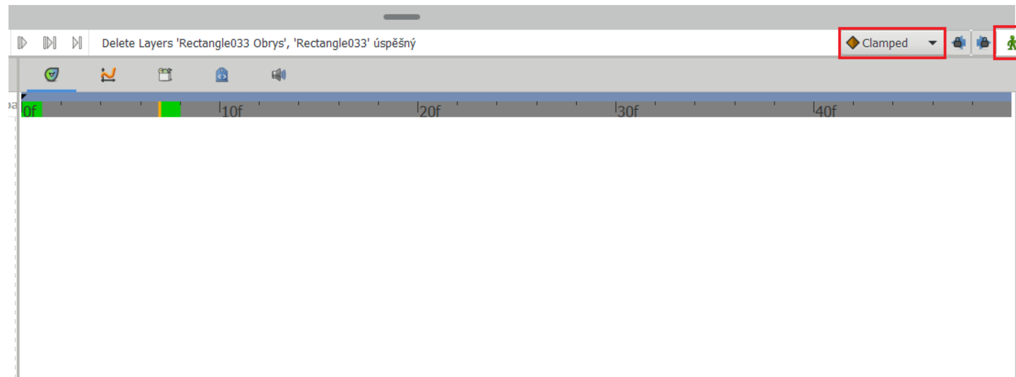
5.1.3 Vlastnosti objektů a časová osa

V dolní části okna projektu se nachází část, která je rozdělena na dvě okna. Levé okno slouží k nastavení vlastností vybraného objektu. Ukázka je uvedena na Obr. 37. V projektu je aktuálně vybrán nakreslený obdélník s výchozími hodnotami, které můžeme pod touto kartou upravovat. Pro účely bakalářské práce byly nejčastěji měněny hodnoty: průhlednost (resp. viditelnost), možnost změnit barvu objektu a jeho rozměry.

Název	Hodnota	Typ	Časová stopa
Z Depth	0,000000	real	
Průhlednost	1,000000	real	
Blend Method	Composite	integer	
Barva		color	
Bod 1	-258,050045px,172,3097	vector	
Bod 2	210,814424px,-95,34238	vector	
Expand amount	0,000000px	real	
Převrátit		bool	
Feather X	0,000000px	real	
Feather Y	0,000000px	real	
Zkosení	0,000000px	real	
Keep Bevel Circular	✓	bool	

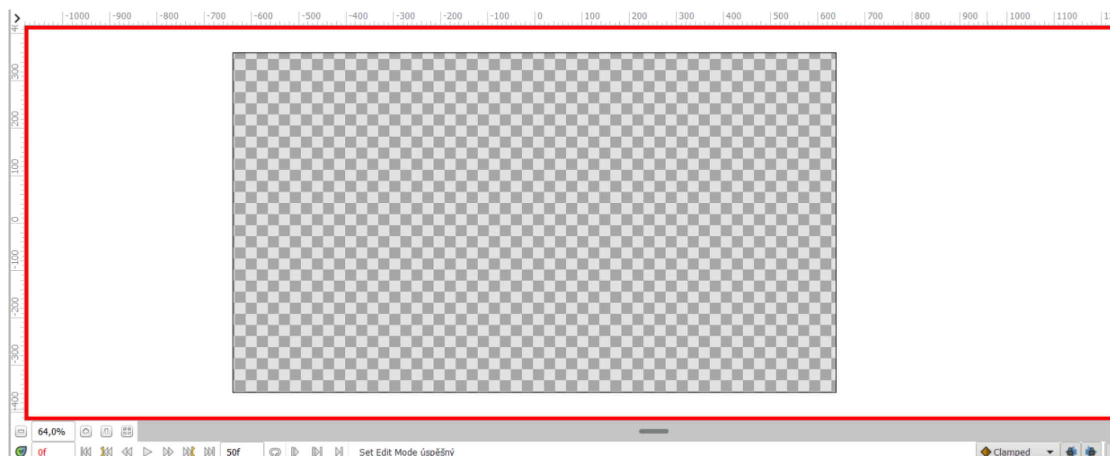
Obr. 37. Vlastnosti objektu
Zdroj: vlastní zpracování

Pravé okno slouží pro práci s objektem na časové ose. Do časové osy můžeme vkládat klíčové snímky na konkrétní snímky projektu. Klíčový snímek nám určí, do jakého okamžiku se má objekt transformovat, změnit pozici, či změnit barvu. V klíčovém snímku můžeme určit nový stav animace, tudíž od tohoto klíčového snímku bude animace, námi zvolené transformace či jiné zvolené operace nad objektem, vypadat takto i v dalším průběhu a objekty se v animaci nebudou vracet do předchozího stavu. Přechod animace jsem pro účely bakalářské práce volil dvěma způsoby.



Obr. 38. Okno pro práci s časovou osou
Zdroj: vlastní zpracování

Kliknutím na zeleného panáčka (pravý horní roh), vyvoláme v projektu animační stav. Označuje se tím, že okno projektu je červeně ohraničeno. Nyní můžeme vytvářet animaci na časové ose. Vedle zeleného panáčka je na výběr, jak bude animace probíhat. Pro účely bakalářské práce jsem volil mezi dvěma stavy. Stav Clamped (výchozí) a stav Constant. Stav Clamped způsobí plynulou animaci z bodu A do bodu B. Stav Constant způsobí jednotné zobrazení (náhlý přechod) změny objektu bez plynulého prolnutí. Kliknutím nyní na červeného panáčka ukončíme animační stav projektu.



Obr. 39. Aktivovaný animační stav
Zdroj: vlastní zpracování

5.1.4 Uspořádání projektu do balíčku, vrstvení

Na pravé straně okna programu se nachází část, která zobrazuje vytvořené objekty, které můžeme skládat do složek, resp. množin objektů či přesouvat a určovat překrývání objektů podle jejich souřadnice z depth, která udává, který objekt je blíže k pozorovateli. Hodnota 0 určuje, že se objekt nachází úplně v popředí. Čím vyšší hodnota z depth, tím bude objekt vzdálenější.

Ikona	Název	Z Depth
	ZobrazeniVirtualniPameti	0,000000
	Steps	0,000000
	UvodniText	1,000000
	uvodniText	0,000000
	MainPicture	2,000000
	MemoryBus	0,000000
	Memory	1,000000
	CPU Package	2,000000
	NewDrawing024 Obrys	0,000000
	CPU Package	1,000000
	CPU Package	0,000000
	Rectangle028 Obrys	1,000000
	MMU	2,000000
	MMU	0,000000
	Rectangle029 Obrys	1,000000
	CPU	3,000000
	CPU	0,000000
	Rectangle029 Obrys	1,000000

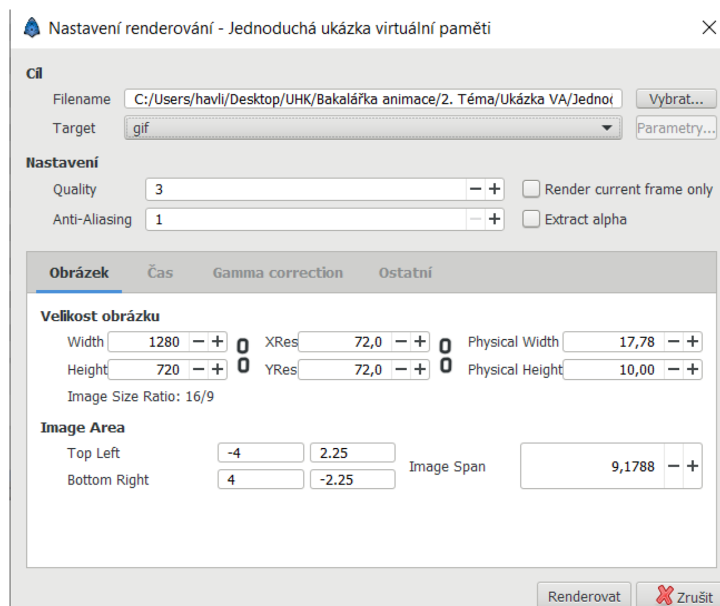
Obr. 40. Projekt strukturován do balíčků
Zdroj: vlastní zpracování

Na Obr. 40 je uveden příklad zobrazení objektů do balíčků. Většina objektů během vytváření animace byla strukturována takto do balíčků, aby se žádnému z objektů nedostával rozmazaný efekt značící velké posunutí objektu od pozorovatele, převážně kvůli textovému doprovodu. Pomocí hodnoty z depth můžeme také způsobit překrývání objektů.

5.1.5 Export projektu

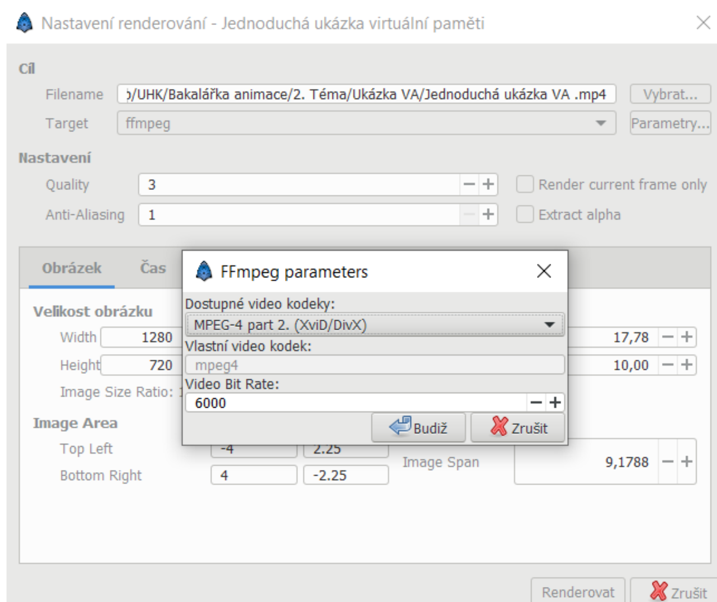
Hotový projekt můžeme vygenerovat do formátu GIF, či MP4. Pro vygenerování projektu ve zvoleném formátu slouží klávesová zkratka F9, nebo přes horní menu programu, kde zvolíme nabídku Soubor a následně možnost Renderovat. Dostaneme se do nastavení renderování. Pod volbou Target můžeme zvolit formát v jakém bude projekt vyrendrován. Pod položkou Nastavení můžeme upravit kvalitu a anti-aliasing animace, případně změnit vlastnosti, které jsme nastavovali při vytváření nového projektu. Ve výchozím nastavení jsou uvedeny hodnoty 3 a 1, jak je uvedeno na Obr. 41. Pro účely bakalářské nebyly tyto hodnoty měněny, jelikož jsou v rámci kvality verendrovaného projektu dostačující.

Pro renderování projektu v položce Target zvolíme formát GIF a kliknutím na tlačítko Renderovat spustíme renderování projektu do tohoto formátu.



Obr. 41. Nastavení renderování
Zdroj: vlastní zpracování

Pro vytvoření animace ve formátu MP4 je potřeba změnit položku Target na ffmpeg. Touto možností se nám umožní změnit možnosti v položce Parametry, které se nachází vedle položky Target, kde nastavíme kodek na mpeg4. Ukázka nastavení je uvedena na Obr. 42. Ve výchozím nastavení se renderuje soubor s koncovkou AVI, pro změnu na koncovku MP4 stačí přepsat koncovku v položce Filename na filename.mp4.



Obr. 42. Renderování do formátu MP4
Zdroj: vlastní zpracování

5.2 Animace vytvořené programem Synfig Studio

V této části jsou popsány jednotlivé animace, které byly vytvořeny pro účely podpory výuky operačních systémů. Ke každé animaci je vytvořen doprovodný text, který obsahuje informace o tom, co daná animace zachycuje. Více podrobností ke konkrétní problematice lze nalézt v teoretické části bakalářské práce či v literatuře, která byla při tvorbě bakalářské práce využita.

5.2.1 Animace pro 1. téma – adresní prostor a abstrakce paměti

Animace z prvního tématu zachycují většinu základních problémů, které jsou zmíněny v teoretické části bakalářské práce. Následující texty obsahují výčet animací, které byly vytvořeny za účelem lepšího porozumění této problematiky.

5.2.1.1 Abstrakce paměti, problém relokace

I bez použití mechanismu swapování lze spouštět více programů v paměti naráz, aniž by docházelo k přepsaní místa kódu jiným programem, či k jiným neshodám, které vedou k chybě prováděného programu. Jedná se o tzv. abstrakci paměti. Abstrakce paměti spočívá v tom, že každý proces má svůj vlastní adresní prostor.

Abstrakce funguje následovně: celá paměť je rozdělena na bloky a každému tomuto bloku je přidělen ochranný klíč. Programy se poté mohou nacházet současně v paměti, aniž by došlo ke kolizi, jelikož mají odlišné klíče.

Mějme v paměti dva programy. Každý o velikosti 16 kB. Dejme tomu, že první program začíná instrukcí JMP 24 na instrukci MOV. Druhý program začíná instrukcí JMP 28 na instrukci CMP. Oba programy odkazují na své instrukce a v případě vložení jednoho z tohoto programů do paměti se vše provede, jak má a nikde nenastane problém. Zásadní problém zde ovšem nastává, pokud budou oba programy společně v paměti. Mají odlišné klíče, takže se nebudou přepisovat a mohou být do paměti vloženy. Oba programy jsou v paměti a operační systém spustí druhý program, který byl načten nad prvním programem v paměti. Druhý program je tedy spuštěn a je vyvolána instrukce JMP 28. Pro správnou funkčnost zde očekáváme, že instrukce nás dovede k instrukci CMP. Instrukce JMP 28 ovšem odkazuje na absolutní fyzické adresy paměti, což vede k nesprávnému chování programu. Toto nám způsobí, že JMP 28 odkazuje na adresu 28 celé paměti, a ne na adresu 28 druhého 16kB programu. Musíme zde dosáhnout toho, že programy budou odkazovat na privátní sadu místních adres. Tímto řešením je statická a dynamická relokace.

5.2.1.2 Statická relokace

Statická relokace se nabízí jako jedno z řešení problému relokace při abstrakci paměti. Statická relokace spočívá v přičítání konstanty ke každé adrese programu. Budeme brát v úvahu stejnou situaci jako v předchozím případě. Pokud budeme mít tedy instrukci JMP 28, která byla načtena na adresa 16 384, stává se hodnota 16 384 konstantou a přičítá se k programu během načítání procesu. Získaná adresa tedy ve finále bude 16 412 a bude zavolána instrukce JMP 16 412, což vede k správnému chování programu a k vyřešení problému, který byl uveden výše. Instrukce CMP se nyní v paměti nachází na adrese 16 412.

Zavaděč ovšem musí rozlišovat mezi adresou programu a konstantou. Například nesmí být přemístěna instrukce MOV REGISTER 1, 28, které přesune obsah paměti na pozici REGISTER 1. Obsah paměti by mohl být následně přesunut na jinou pozici, což by způsobilo kritické chyby fungování programu. U statické relokace se jedná o řešení, které zpomaluje načítání procesu a vyžaduje informaci navíc ve všech spustitelných programech, která označuje, jaká „slova" obsahují přemístitelné adresy a která nikoli. Proto je zcela zásadní, aby byla rozlišována konstanta a adresa.

5.2.1.3 Dynamická relokace

U dynamické relokace rozlišujeme dva speciální hardwarové registry. Tyto registry se nazývají base a limit registr. Base registr nám značí počáteční adresu programu ve fyzické paměti. Limit registr značí délku samotného programu. Pokud nám program začíná v paměti na pozici 0, jeho base registr je roven nule a limit registr je poslední možná adresa programu. Pomocí součtu těchto registrů nám vyjde obsah paměti, který máme k dispozici pro zpracování programem.

Při každém spuštění programu se načte base registr s fyzickou adresou, kde program v paměti začíná. Limit registr se načítá s délkou programu. Dynamickou relokaci můžeme zachytit dvěma body, které se opakují při každém vstupu do paměti:

1. Hardware přidá base registr k logické adrese a získá tím úplnou fyzickou adresu
2. Hardware porovná adresu s limitním registrem

V bodě 2 nám vznikají dvě podmínky, které mohou nastat:

a. Podmínka selže, tzn. že fyzická adresa je větší než limitní registr, to znamená, že přesahuje délku svého program. V tomto případě je vrácena chyba adresy a fyzická adresa je ignorována.

b. V kladném případě je adresa poslána na paměťovou sběrnici.

Nevýhoda dynamické relokace je právě v provádění těchto kroků při každém přístupu, resp. odkazu do paměti.

5.2.1.4 Swapování

Swapování označuje jednoduchý mechanismus pro výměnu procesů z hlavní paměti do sekundární paměti, nejčastěji na disk. Jedná se pouze o dočasné přesunutí. Tato výměna je prováděna jádrem operačního systému. Swapování je velmi důležité v případě, kdy máme plnou paměť a nelze spustit další program, jelikož pro něj není dostatek volného prostoru v paměti. Tento mechanismus se dá shrnout třemi body:

1. Operační systém uloží celou paměť na disk
2. Operační systém přivede nový program do paměti
3. Operační systém spustí nový program

Na disk se ve většině případů odkládají nečinné procesy, které nejsou aktuálně spuštěny, proto není zapotřebí je uchovávat v paměti.

5.2.1.5 Pevné oddíly (angl. Fixed partitions)

V případě pevných oddílů je paměť rozdělena do pevných velikostí jednotlivých oddílů. Tyto oddíly jsou fixní, tudíž jsou neměnné. Každý tento oddíl je schopný pojmout právě jeden proces, který nepřesahuje velikost samotného oddílu. V případě, že je paměť plně obsazená, řadí se procesy do dvou front:

1. Samotná fronta pro jednotlivé oddíly
2. Jediná fronta pro celou paměť

V prvním případě se tvoří fronty pro každý oddíl paměti. Nevýhoda v tomto řazení spočívá v případě, kdy je fronta pro velký oddíl (velký kus paměti) zcela volná a fronta pro menší oddíl je plná, jak je znázorněno animací. Problém je v tom, že procesy, které zabírají malou část paměti nejsou alokovány do paměti i přesto, že je dostatek volné paměti k dispozici.

Alternativním způsobem řazení do fronty je vytvořit jednu frontu pro celou paměť. Kdykoli proces vykoná svoji práci a uvolní oddíl v paměti, je tento oddíl alokován procesem, který je na řadě ve frontě. Nevýhoda v tomto řazení spočívá v tom, že se ve frontě může objevit proces, který potřebuje alokovat větší množství paměti. Dokud ovšem nebude uvolněn oddíl

s dostatečně volným prostorem, nebude moci tento proces opustit frontu, zatímco ostatní procesy ve frontě by mohly být alokovány do jiných částí paměti.

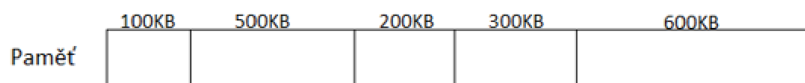
5.2.1.6 Dynamické oddíly (angl. Dynamic partitions)

V tomto případě se paměť oproti pevným oddílů jeví jako jeden velký blok paměti. Proces má v tomto případě alokováno přesně tolik místa, kolik potřebuje. Tzn. vezme si z dynamického bloku takovou část paměti, která je vyžadována pro alokování a následné spuštění procesu, jak je znázorněno animací.

Tento prostor paměti se alokuje a bude procesu přidělen. Zbytek volné paměti se dále jeví jako blok volné paměti. Kdykoli je proces ukončen, je tento blok paměti uvolněn. Pokud tento volný prostor sousedí s dalším volným prostorem, jsou tyto prostory spojeny do jednoho většího a obsazené oddíly se posouvají v paměti směrem dolů (animace Memory compaction – zhutnění paměti). Během dynamicky přiřazovaných oddílů máme paměť rozdělenou na dvě části, a to blok obsazený procesy a volný blok paměti. Tomuto přístupu se říká zhutnění paměti.

5.2.1.7 First fit

Tato animace zobrazuje chování algoritmu pro přiřazování volné paměti. Jedná se o algoritmus, který skenuje celou paměť od jejího počátku při každém pokusu o alokování procesu do paměti. Na vstup přijdou procesy v daném pořadí, které je neměnné a správce paměti skenuje paměť do té doby, dokud nenarazí na dostatečně velký volný prostor, aby zde mohl být proces alokován. Tento prostor je následně rozdělen na alokovaný prostor a volný prostor, jedná se o prostor, který nebyl procesem alokován.



Obr. 43. Paměť – algoritmus first fit
Zdroj: vlastní zpracování

Mějme takovouto paměť a procesy o velikosti 212 kB, 417 kB, 112 kB, 426 kB, které se budou alokovat **v tomto pořadí**. Dále budou popsány všechny kroky algoritmu first fit.

Jako první proces alokujeme proces o velikosti 212 kB. Správce paměti začíná skenovat paměť od jejího počátku. 100 kB není dostačující volný prostor pro tento proces, tudíž správce paměti pokračuje. Narazí na velikost bloku 500 kB, což je pro proces dostačující. Máme tedy

alokovaný proces v 500kB bloku. Blok se rozdělí na 212kB alokovaný proces a **288 kB** je zbytek volného prostoru.

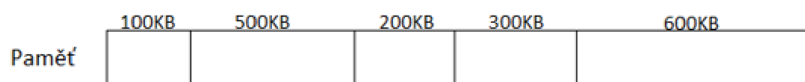
Chceme alokovat druhý proces o velikosti 417 kB. Správce paměti opět skenuje paměť od jejího počátku. Blok 100 kB není dostačující pro alokování. To samé platí pro bloky 288 kB, 200 kB, 300 kB. Blok 600 kB je dostatečně velký na to, abychom tento proces mohli alokovat. Blok je rozdělen na alokovaný proces 417 kB a volný prostor **183 kB**.

Nyní alokujeme třetí proces o velikosti 112 kB. Procházíme paměť od jejího počátku. Blok 100 kB není dostačující pro alokování tohoto procesu, blok 288 kB už je dostačující pro alokování procesu. Blok je opět rozdělen na místo zabrané alokovaným procesem a volný prostor, který činí **176 kB**.

Poslední proces je proces o velikosti 426 kB. Procházíme paměť opět od jejího počátku. Blok 100 kB není dostačující, blok 176 kB také ne, to samé platí i pro následující bloky volné paměti. Ani do jednoho z těchto bloků nemůže být proces alokován. Proces musí čekat, dokud se neuvolní dostatečně velký prostor na to, aby mohl být proces alokován.

5.2.1.8 Next fit

U této animace se zaměříme na chování algoritmu next fit, který je velmi podobný algoritmu first fit. U tohoto algoritmu ovšem neprohledává správce paměti vždy celou paměť od jejího počátku, ale při dalším pokusu o alokaci procesu začíná tam, kde v předchozím kroku skončil.



Obr. 44. Paměť – algoritmus next fit
Zdroj: vlastní zpracování

Mějme takto rozdělenou paměť a chtějme alokovat procesy o velikosti 212 kB, 417 kB, 112 kB, 426 kB **v tomto pořadí**.

Algoritmus next fit začíná při prvním hledání na začátku paměti. Chceme alokovat proces o velikosti 212 kB, správce paměti bude hledat dostatečně velký prostor pro alokování procesu. Při prvním nalezení je proces alokován a blok paměti je rozdělen na alokovaný proces a zbytek volného prostoru, který nebyl procesem využit. První blok je o velikosti 100 kB, není

pro náš proces dostačující. Další blok o velikosti 500 kB již vyhovuje požadavkům procesu a proces je následně alokován. Blok je rozdělen na dvě části, alokovaný proces a zbytek **288 kB** volného prostoru.

Při alokování druhého procesu o velikosti 417 kB nezačíná správce paměti skenovat paměť od počátku, ale od místa, kde při předchozí alokaci skončil. Nyní se správce paměti nachází v 288kB bloku (rozdělený 500kB blok procesem a volným prostorem). 288 kB není dostatečně velký blok paměti pro alokování tohoto procesu. Stejná situace se opakuje u 200kB bloku, 300kB bloku. Správce paměti narazí na blok paměti o velikosti 600 kB, kde je proces o velikosti 417 kB alokován. Vzniká opět nový, menší blok o velikosti **183 kB**.

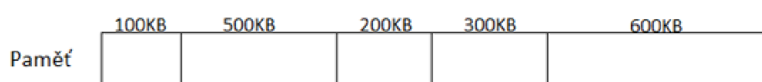
Při alokování třetího procesu o velikosti 112 kB správce paměti skenuje paměť od místa, kde skončil. Naposledy správce paměti skončil na bloku o velikosti 183 kB. Tento blok je dostatečně velký pro alokování tohoto procesu. Blok o velikosti 183 kB je nyní rozdělen na dvě části, alokovaný proces a zbytek volného prostoru **71 kB**.

Pro alokování posledního procesu o velikosti 426 kB opět správce paměti skenuje paměť od místa, kde skončil. Jedná se o blok o velikosti 71 kB. Tento blok není dostatečně velký pro alokování tohoto procesu. Správce paměti došel na konec paměti. Začíná skenovat paměť opět od jejího počátku. Žádný dostupný volný prostor v paměti není dostatečně velký pro alokování procesu o velikosti 426 kB.

Tento proces musí čekat, dokud se neuvolní dostatečně velký prostor na to, aby mohl být alokován.

5.2.1.9 Best fit

Tato animace znázorňuje fungování dalšího algoritmu, a tím je algoritmus best fit. Tento algoritmus funguje jinak než předchozí dva algoritmy. Správce paměti zde při každém pokusu o alokování procesu do paměti skenuje celou paměť a vybírá nejmenší možný prostor pro alokování procesu. Algoritmus best fit třídí bloky paměti podle velikosti a následně vybere nejmenší adekvátní prostor pro alokování procesu.



Obr. 45. Paměť – algoritmus best fit
Zdroj: vlastní zpracování

Budou se postupně alokovat procesy o velikosti 212 kB, 417 kB, 112 kB, 426 kB **v tomto pořadí**.

Jako první proces má být alokován proces o velikosti 212 kB. Správce paměti skenuje celou paměť a seřadí dostupné volné prostory podle velikosti: 100 kB, 200 kB, 300 kB, 500 kB, 600 kB. Správce paměti nyní vyhledává nejmenší možný prostor pro alokování procesu. Proces o velikosti 212 kB je alokován do volného prostoru o velikosti 300 kB, tento prostor je opět rozdělen na dvě části, alokovaný proces a zbytek volného prostoru **88 kB**.

Druhý proces pro alokování je proces o velikosti 417 kB. Správce paměti skenuje celou paměť a seřadí dostupné prostory podle velikosti: 88 kB, 100 kB, 200 kB, 500 kB, 600 kB. Je hledán nejmenší dostupný prostor pro alokování procesu. Tento proces je alokován do volného prostoru paměti o velikosti 500 kB, který je následně rozdělen na dvě části. Alokovaný proces a zbytek **83 kB** volného prostoru.

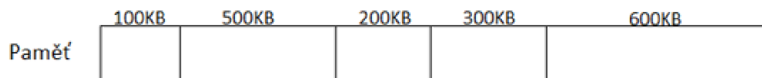
Jako třetí proces bude alokován proces o velikosti 112 kB. Správce paměti skenuje celou paměť a seřadí dostupné prostory podle velikosti od nejmenšího: 83 kB, 88 kB, 100 kB, 200 kB, 600 kB. Jako nejmenší adekvátní prostor je v tomto případě vybrán blok o velikosti 200 kB. Tento blok je následně rozdělen na dvě části. Jedná se o alokovaný proces 112 kB a zbytek volného prostoru **88 kB**.

Jako poslední proces je alokován proces o velikosti 426 kB. Správce paměti skenuje celou paměť a setřídí dostupné prostory podle velikosti od nejmenšího: 83 kB, 88 kB, 88 kB, 100 kB, 600 kB. Jako nejmenší adekvátní prostor je nalezen prostor o velikosti 600 kB. V tomto prostoru je alokován proces o velikosti 426 kB a zbytek je **174 kB** volný prostor.

Algoritmus best fit je značně pomalejší oproti předchozím dvěma, protože při každém pokusu o alokování, prohledává celou paměť od počátku do konce, což značně zpomaluje rychlost zpracovávání procesu. Mezi nevýhody tohoto procesu také patří vytváření malých kousků paměti volného prostoru, které nebudou využity.

5.2.1.10 Worst fit

Tato animace zachycuje fungování algoritmu worst fit. Tento algoritmus je podobný s předchozím algoritmem best fit. Při každém volání prohledává celou paměť a tentokrát je vybrán největší blok volné paměti.



Obr. 46. Paměť – algoritmus worst fit
Zdroj: vlastní zpracování

Následně se do paměti budou alokovat procesy o velikosti 212 kB, 417 kB, 112 kB, 426 kB **v tomto pořadí**.

Při prvním pokusu o alokování procesu o velikosti 212 kB, je prohledána celá paměť a dostupné volné prostory jsou setříděny podle velikosti od největšího: 600 kB, 500 kB, 300 kB, 200 kB, 100 kB. Jako největší volný prostor je prostor o velikosti 600 kB, tento prostor je rozdělen na alokovaný proces a **388 kB** volný prostor.

Při alokování dalšího procesu o velikosti 417 kB skenuje správce paměti celou paměť a seřadí dostupné volné prostory podle velikosti od největšího: 500 kB, 388 kB, 300 kB, 200 kB, 100 kB. Jako největší volný prostor je prostor o velikosti 500 kB. V tomto místě je alokován proces o velikosti 417 kB a zbytek prostoru je volný prostor o velikosti **83 kB**.

Při alokování třetího procesu o velikosti 112 kB správce paměti skenuje celou paměť a seřadí volné prostory podle velikosti od největšího: 388 kB, 300 kB, 200 kB, 100 kB. Největší prostor je zde o velikosti 388 kB, kde bude alokován proces o velikosti 112 kB. Vzniká nový volný prostor o velikosti **276 kB**.

Pro alokování posledního procesu o velikosti 426 kB skenuje opět správce paměti celou paměť a seřadí dostupné volné prostory od největšího: 300 kB, 276 kB, 200 kB, 100 kB, 83 kB. Pro proces o velikosti 426 kB, není v paměti žádný adekvátní prostor. Tento proces musí čekat, dokud se neuvolní dostatečně volný prostor na to, aby mohl být proces alokován.

5.2.2 Animace pro 2. téma – virtuální paměť

Animace z druhého tématu se věnují především problematice stránkování a využití jednotky procesoru MMU.

5.2.2.1 Stránka, rámec stránky, počítání

V této animaci se jedná o rychlý přehled důležitých pojmů k pochopení a správné orientaci v celém tématu virtuální paměť. Jsou zde zmiňovány dva adresní prostory. Jedná se o virtuální adresní prostor (dále jen VAD) a fyzický adresní prostor (dále jen FAP). VAP se

skládá z jednotlivých bloků, které jsou stejné velikosti (např. 4 kB) a tyto bloky nazýváme stránky. FAP se skládá z bloků, které mají z pravidla stejnou velikost jako stránky ve VAP a tyto bloky nazýváme rámce stránek.

VAP je mapován na FAP. Čili jednotlivé stránky jsou mapovány na rámce stránek. Rámce stránek obsahují konkrétní adresy fyzické paměti, které jsou odesílány na paměťovou sběrnici.

Co se týká počtu stránek ve VAP a rámců stránek ve FAP, musíme brát v potaz tři důležité informace. Jedná se o velikost VAP a velikost FAP. Poslední údaj udává velikost jedné stránky, resp. jednoho rámce stránky. Pokud si uvedeme situaci na příkladě, máme VAP 64 kB, FAP 32 kB a stránku o velikosti 4 kB. Počet stránek se zjistí jednoduchým výpočtem: $\frac{VAP}{stránka} = \frac{64}{4} = \frac{2^6}{2^2} = 2^4 = 16$. Zjistíme, že VAP bude rozdělen na 16 stránek. Počet rámců stránek se zjistí obdobným způsobem: $\frac{FAP}{rámec} = \frac{32}{4} = \frac{2^5}{2^2} = 2^3 = 8$. Z výpočtu zjistíme počet rámců stránek ve FAP, jedná se o 8 rámců stránek.

Jednotlivé stránky VAP jsou poté mapovány, resp. mají referenci na rámec stránky.

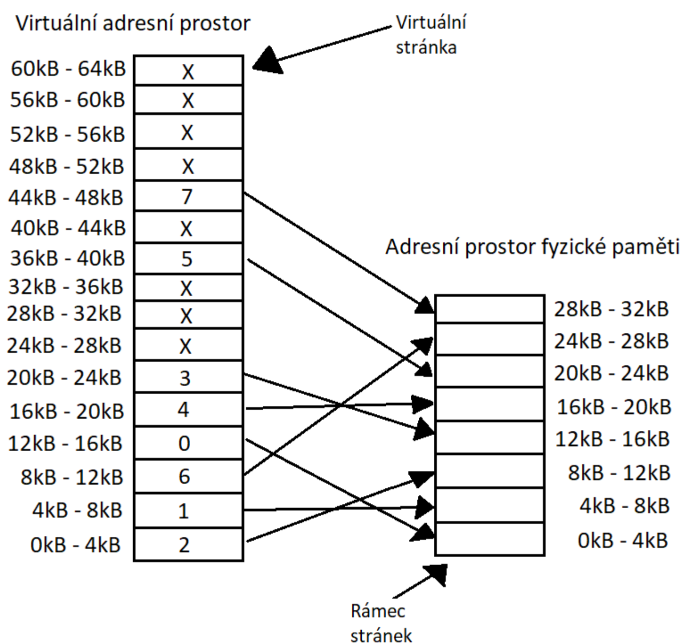
5.2.2.2 Ukázka VA

Tato animace zobrazuje smysl, či znázornění nového pojmu, který je v tématu virtuální paměť důležitý. Jedná se o jednotku procesoru MMU. MMU je součást mikroprocesoru, která umožňuje přístup do virtuální paměti. Obstarává především překlad virtuální adresy na fyzickou.

CPU přijímá virtuální adresu, která je předána MMU pro zpracování. MMU virtuální adresu zpracuje a mapuje tuto adresu na správný rámec stránky ve FAP, z kterého se získá fyzická adresa, která je poslána na paměťovou sběrnici.

5.2.2.3 Mapování virtuální adresy na fyzickou adresu

V této animace je zachyceno mapování virtuální adresy na fyzickou adresu. Jedná se o animaci, která se snaží objasnit problém z pohledu získání fyzické adresy, když na CPU přichází virtuální adresa. Animace je provedena na konkrétním příkladě, který zde bude následně popsán.



Obr. 47. Mapování
Zdroj: vlastní zpracování

Jako první krok je, že CPU přijme virtuální adresu 20 500. Tato adresa je následně předána MMU, jak bylo znázorněno na předchozí animaci (5.2.2.2). MMU zde podle mapování zjistí, že se jedná o 5. stránku. Čísluje od nuly a hledáme takový rozsah, do kterého bude spadat virtuální adresa 20 500. V tomto případě se jedná o rozsah 20 kB – 24 kB, čemuž odpovídá 5. stránka ve VAP. MMU podle mapování zjistí, že se jedná o stránku, která je mapována na 3. rámec stránky ve FAP. Tento rámec stránky odpovídá fyzickým adresám od 12 kB do 16 kB. Následně je zjištěna fyzická adresa, která bude MMU odeslána na paměťovou sběrnici.

Od adresy 20 500 musíme vzít její počátek. Jedná se o adresu 20 480 + 20 kB od jejího počátku. Proč 20 480? Stránka je v rozsahu 20 kB do 24 kB. Jedna stránka má rozsah adres od 0 do násobku čísla 4096 (velikost stránky 4 kB). Jedná se o 5. stránku ve VAP, tudíž $5 * 4096 = 20\,480$. 5. stránka odpovídá virtuálním adresám v rozsahu 20 480 – 24 575.

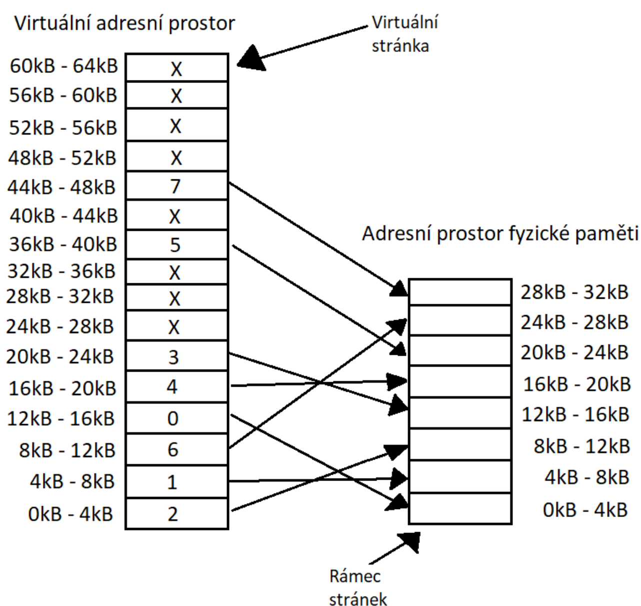
Stejný případ je i u FAP. 5. stránka je mapována na 3. rámec stránky ve FAP. 3. rámec stránky odpovídá fyzickým adresám v rozsahu 12 kB – 16 kB. Opět velikost jednoho rámce stránky odpovídá fyzickým adresám v rozsahu od 0 do násobku čísla 4096. 3. rámec stránky obsahuje fyzické adresy v rozsahu $3 * 4096 = 12\,288\ do\ 16\,383$.

Tím máme zjištěno, jaké je pokrytí virtuálních adres 5. stránky ve VAP a fyzických adres 3. rámce stránky ve FAP. Pracujeme s adresami od jejich počátku. Vybíráme tudíž 3. rámec stránky a jeho počáteční adresu 12 288. K tomuto ovšem nesmíme zapomenout přičíst

počet kB od počátku v paměti. V tomto příkladě se jedná o 20 kB. Získáme tedy výslednou fyzickou adresu 12 288 + 20 = 12 308. Tato fyzická adresa je následně poslána na paměťovou sběrnici.

5.2.2.4 Chyba stránky

Chyba stránky nastane v případě, že do MMU přijde virtuální adresa, která není mapována na fyzickou adresu. To znamená, že se ve VAP nachází stránka, která není mapována na rámec stránky ve FAP. Tato animace znázorňuje situaci, u které dojde k chybě stránky a jak je celá situace dále řešena.



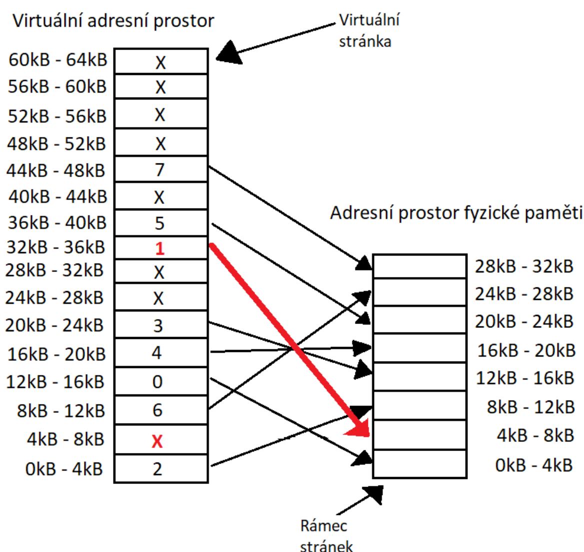
Obr. 48. Mapování – chyba stránky
Zdroj: vlastní zpracování

Zachovejme stejné mapování jako bylo uvedeno v popisu animace 5.2.2.3 (Mapování virtuální adresy na fyzickou adresu). Pro příklad využijeme virtuální adresu 32 780.

Postup je obdobný jako v předchozí animaci. Na CPU přijde virtuální adresa 32 780, tato adresa je předána MMU, která najde odpovídající stránku ve VAP. Pokud známe velikost stránky a virtuální adresu, lehce zjistíme, o jakou stránku se jedná: $\frac{32\,780}{4096} = 8{,}003$. Po zaokrouhlení dolů zjistíme, že se jedná o 8. stránku ve VAP. Její rozsah získáme následovně: $8 * 4096 = 32\,768\ do\ 36\,863$. 8. stránka tedy odpovídá virtuálním adresám v tomto rozsahu: 32 768 – 36 863. Opět pracujeme se stránkou od počátku, kde následně přičteme hodnotu k získání příchozí virtuální adresy. V tomto případě 32 768 + 12.

Podle mapování zjistíme, že 8. stránka (označena křížkem X) není mapována na žádný z rámců stránek ve FAP. Present/absent bit této stránky je nastaven na nule. V tomto případě musí operační systém odložit obsah málo používaného rámce stránky na disk. Zde se jedná o metodu swapování. Pro příklad řekněme, že se operační systém rozhodne přesunout obsah 1. rámce stránky. Následně musí být provedena aktualizace mapování stránek ve VAP na rámce stránek ve FAP.

Podle mapování víme, že 1. rámec stránky ve FAP má referenci s 1. stránkou ve VAP. Po přesunutí obsahu rámce stránky na disk je toto mapování zrušeno. 1. stránka ve VAP je nyní označena křížkem X a její present/absent bit je nastaven na hodnotu 0. 8. stránka ve VAP je nyní mapována na volný rámec stránky. Po aktualizaci mapování proběhne znovu spuštění instrukce, u které nastala chyba stránky.



Obr. 49. Mapování – aktualizace mapování
Zdroj: vlastní zpracování

8. stránka VAP je nyní mapována na 1. rámec stránky FAP. Pracujeme s adresou 32 780. Podle rozsahu adres, který odpovídá 1. rámci stránek ve FAP (4096–8191) je brán počátek a přičten zbytek k získání úplné adresy, která byla poslána na CPU. V našem případě u 8. stránky, to odpovídá adrese 32 780, počátek 32 768 + 12. V rámci stránky se jedná o počátek 4096 + 12 = 4108.

Z těchto kroků můžeme učinit nyní závěr, který vznikne po znovu spuštění instrukce, u které nastala chyba stránky. Při přijetí virtuální adresy 32 780 CPU předá adresu MMU, která následně provede potřebné mapování a na paměťovou sběrnici je odeslána získaná fyzická adresa 4 108.

5.2.2.5 Vnitřní mapování MMU

Tato animace shrnuje mapování adres ve virtuální paměti fází, která je zde zcela zásadní pro pochopení mapování. Jedná se o animaci, která znázorňuje, jak samotná jednotka MMU pracuje s virtuální adresou a dokáže získat fyzickou adresu, která je následně MMU odeslána na paměťovou sběrnici.

Pro příklad na animaci je použita příchozí virtuální adresa 8 196, která je CPU předána MMU pro překlad z virtuální adresy na fyzickou. Pro lepší znázornění nám zde poslouží tabulka stránek, která obsahuje jednotlivé záznamy, resp. jednotlivé stránky, které nám ukazují číslo stránky, jejich mapovaný rámec stránky a present/absent bit, který určuje, zda jsou stránky namapovány na rámec stránky ve FAP.

15	X	0
14	X	0
13	X	0
12	X	0
11	111	1
10	X	0
9	101	1
8	X	0
7	X	0
6	X	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Obr. 50. Tabulka stránek
Zdroj: vlastní zpracování

Pro následující postup budeme potřebovat operovat s binárními čísly s kterými MMU pracuje. Příchozí adresu 8 196 zapíšeme binárně: 0010000000000100. Tato binárně zapsané adresa je nyní rozdělena na číslo stránky a offset. Číslo stránky získáme pomocí počtu stránek, resp. zápisu 2^n. V tabulce stránek se nachází 16 stránek, resp. tabulka obsahuje 16 záznamů. $16 = 2^4$ z tohoto zápisu získáme počet bitů pro číslo stránky, jedná se o 4 bity.

Příchozí virtuální adresa je nyní rozdělena na číslo stránky (4 b) a offset (zbytek 12 b). Offset nám udává, jakého rozsahu může dosahovat jedna konkrétní stránka. V tomto příkladě se jedná o velikost od 0 do násobku čísla 4096. Tudíž $2^{12} = 4096$, jedná se o velikost stránky.

Dostaneme tedy nový zápis 0010 | 000000000100, kde čtyři bity slouží jako index na číslo stránky do tabulky stránek. Z binárního zápisu 0010 dostaneme hodnotu 2. Jedná se tedy o 2. stránku ve VAP. Ve 2. stránce VAP se tedy nachází virtuální adresa 8 196.

Pokud provedeme kontrolu, kde chceme zjistit, jestli je rozsah adres shodný s příchozí virtuální adresou, provedeme následující krok. Rozsah 2. stránky ve VAP je od 8 192 $(2 * 4096)$ do 12 887 $(8\,192 + 4095)$. Příchozí adresa 8 196 opravdu tedy patří do rozsahu 2. stránky ve VAP.

Podle tabulky stránek zjistíme, na jaký rámec stránky odkazuje 2. stránka ve VAP. Podle tabulky stránek získáme binární hodnotu 110. Z toho vyplývá, že 2. stránka VAP má referenci na 6. rámec stránky ve FAP. Nyní se provede krok, kterým se zjistí odchozí fyzická adresa, která bude odeslána na paměťovou sběrnici.

Binárně zapsaný offset se zkopíruje, zůstává po celou dobu neměnný, jelikož velikost stránky je shodná s velikostí rámce stránky. Nyní se přepíše hodnota čísla stránky, která nyní inicializuje číslo stránky rámce, na který je daná stránka mapována. V tomto příkladě tedy dostaneme novou binární adresu 110 | 000000000100. Rozdělena na číslo rámce stránky a offset.

Binární číslo 110000000000100 přepíšeme do desítkové soustavy a získáme odchozí fyzickou adresu na paměťovou sběrnici, jedná se o adresu 24 580.

Pro kontrolu můžeme provést kroky, které jsou znázorněny u animace 5.2.2.3. Na CPU přijde virtuální adresa 8 196, tato adresa je předána MMU. MMU zjistí, že virtuální adresa odpovídá rozsahu 2. stránky ve VAP $(\frac{8\,196}{4096} = 2{,}001)$. Po zaokrouhlení dolů dostaneme 2. stránku VAP. Tato stránka obsahuje rozsah adres od 8 192 – 12 887. Opět bereme počátek adresy 8 192 + 4 kB, které jsou potřeba pro získání úplné příchozí virtuální adresy. MMU zjistí, že stránka je mapována na 6. rámec stránky. Tento rámec stránky obsahuje rozsah adres od 24 576 do 28 671 $(6 * 4096 = 24\,576; 24\,576 + 4095 = 28\,671)$. Pro získání fyzické adresy bereme počátek adresy, na které začíná 6. rámec stránky. Jedná se o adresu 24 576. K této adrese jsou přičteny 4 kB k získání úplné fyzické adresy. Na paměťovou sběrnici MMU odesílá fyzickou adresu 24 580.

5.2.3 Animace pro 3. téma – procesy

5.2.3.1 Plánování procesů

Animace zachycuje přepínání mezi jednotlivými stavy procesu, které řídí plánovači.

Připravené a spuštěné procesy jsou uloženy v operační paměti. Připravené/pozastavené procesy se mohou nacházet na disku. V plánovači procesu máme tři plánovače: krátkodobý plánovač, střednědobý plánovač, dlouhodobý plánovač.

Krátkodobý plánovač vybírá proces z fronty připravených a přidělujeme jim procesor (CPU). Jedná se o přechod mezi stavem připraven a spuštěný. Tento plánovač musí rychle rozhodovat, kterému procesu bude přiděleno CPU. Vybírá tedy proces, který poběží na uvolněném procesoru, přiděluje procesu procesor (CPU).

Střednědobý plánovač se stará o swapování procesů. Tento plánovač vybírá procesy ze stavu blokující a odkládá je, to samé platí pro procesy připravené/pozastavené. Procesy, které jsou odložené se nenachází v operační paměti ale na disku. Logicky tento plánovač patří částečně do správy hlavní paměti. Vybírá proces, který je možno zařadit mezi odložené procesy, tím uvolní prostor zabíraný procesem ve FAP. Pracuje i naopak, kdy odloženému procesu opět přidělí prostor ve FAP.

Dlouhodobý plánovač vytváří nové procesy. U uživatelského počítače je tímto plánovačem uživatel, který spouští procesy. U serveru, kde procesy přichází po internetové síti se o ně stará dlouhodobý plánovač. Vybírá, který požadavek na výpočet lze zařadit mezi procesy a definuje tak stupeň multiprogramování.

5.2.3.2 FIFO

Animace zobrazuje plánování FIFO, angl. First In, First Out, také označováno jako FCFS (angl. First Come, First Served). Tento plánovací algoritmus bývá označován jako fronta.

Při plánování v řádném frontovém režimu procesy přicházejí do stavu „připravený" a jsou umisťovány na konec fronty. Při plánování se procesor (CPU) přidělí tomu procesu, který je ve frontě nejdéle, resp. je jako první na řadě. Tuto strategii je možné používat při preemptivním i nepreemptivním plánování procesů. Při preemptivním plánování se někdy nazývá také jako cyklické plánování (Round Robin scheduling).

5.2.3.3 LIFO

Animace zachycuje plánovací algoritmus LIFO, angl. Last In Last Out. Tento plánovací algoritmus bývá označován jako zásobník.

Jedná se o protiklad plánovacího algoritmu FIFO. Zde je zpracován nejdříve vrchol zásobníku. Zde jsou připravený procesy „skládány“ na sebe v podobě zásobníku a procesor (CPU) je přiděleno procesu, který strávil v zásobníku nejkratší dobu.

5.2.3.4 Round Robin

Animace zachycuje plánování round robin. Jedná se o plánování cyklickou obsluhou. Round robin je preemptivním plánováním.

V tomto plánování každý proces dostává procesor (CPU) na malou jednotku času. Tuto jednotku nazýváme časové kvantum procesu. Po uplynutí této doby je běžící proces „předběhnut“ nejstarším procesem ve frontě připravených procesů a zařadí se na konec této fronty. Je-li ve frontě připravených *n* procesů a časové kvantum je *q*, pak každý proces získá 1/n-tinu doby CPU, najednou nejvýše po *q* časových jednotkách. Žádný proces nečeká na přidělení CPU déle než (n-1)q časových jednotek.

6 Shrnutí výsledků

Veškerý obsah bakalářské práce byl konzultován s vedoucím práce a dospělo se k tomuto obsahu. Tento obsah může být dále rozšiřován o další problematiky a mohou být vytvářeny další animace, které poslouží pro podporu výuky operačních systémů a zvýší se tak studijní materiál, který bude během výuky k dispozici. Dále se může pokračovat v nových tématech, ke kterým budou vytvářeny nové animace.

Výukový materiál může být shrnut popisem animace a přiložením vyrenderované animace, která bude sloužit pro lepší znázornění konkrétní problematiky. Vše může být umístěno v kurzu předmětu na portálu Blackboard, kam mají studenti UHK přístup. Vše bude na jednom místě a studenti si nemusí stahovat dodatečný software pro zobrazení animací.

7 Závěry a doporučení

Při vytváření animací jsem narazil na pár problémů, které by bylo dobré zmínit například pro budoucí práce, které by také mohly využít tento animační program, či sloužit jako poznatky, s kterými jsem se během práce s programem setkal.

Renderování do formátu GIF je v programu Synfig Studio celkem pomalé a vyrenderovaný soubor může při větším množství animací a strukturování projektu dosahovat desítek MB. Při renderování, kdy zvolíme v Target magick++, je koncovka souboru změněna na filename.gif a je také možno vyrenderovat soubor gif s menší velikostí a renderování je v porovnání rychlejší. Ovšem při tomto postupu jsem se párkrát setkal s neúspěchem, kdy došlo během renderování k pádu programu. S renderováním do formátu MP4 jsem neměl žádné problémy a vše se renderovalo v pořádku. Při renderování do formátu, kdy bylo ponecháno Target gif jsem se také setkal se špatně vyrenderovaným projektem. Na některých snímcích docházelo k problikávání obrazu a menšího cukání prováděné animace. Po znovu provedení renderování bylo už vše v pořádku. Tudíž jsem narazil na problém, že jsem musel některé animace renderovat víckrát do formátu gif kvůli způsobeným chybám. Opět při formátu MP4 jsem na takové problémy nenarazil.

Při větším množství objektů v projektu je lepší organizovat vše do složek kvůli zachování čistého obrazu, kdy na objekty nepůsobí tolik hodnota z depth, která rozmazává objekty s rostoucí hodnotou z depth. To znamená, že objekty jsou od pozorovatele vzdálenější a působí rozmazaným dojmem.

Měl jsem možnost vyzkoušet program Synfig Studio na dvou platformách operačního systému. Na operačním systému Windows 10, Windows 7 a operačním systému Linux, distribuce Ubuntu 20.04. Program fungoval a reagoval nejlépe na operačním systému Linux. Na operačním systému Windows 10 jsem se setkal několikrát s pády programu, které vedly ke ztrátě neuložené práce. Po aktualizaci na novější verzi programu Synfig Studio jsem se zatím s žádnými pády na operačním systému Windows 10 nesetkal.

8 Seznam použité literatury

[1] STALLINGS, William. Operating systems: Internals and Design Principles. 6. vydání. Upper Saddle River: Pearson/Prentice Hall, 2009. 822 stran. ISBN: 0-13-600632-9

[2] TANENBAUM, Andrew Stuart. Modern operating systems. 3. vydání. Upper Saddle River: Pearson/Prentice Hall, 2008. 1076 stran. ISBN: 978-0-13-600663-3

[3] JELÍNEK, Lukáš. Jádro systému Linux – kompletní průvodce programátora. 1.vydání. Brno: Computer Press, 2008. 688 stran. ISBN: 978-80-251-2084-2

[4] DRÁB, Martin. Jádro systému Windows: kompletní průvodce programátora. 1. Vydání. Brno: Computer Press, 2011. 472 stran. ISBN: 978-80-251-2731-5

[5] KLIMEŠ, Cyril. Principy výstavby počítačů a operačních systémů [online]. Dostupné z: https://publi.cz/books/11/01.html

[6] Instalace programu Synfig Studio [online]. Dostupné z: https://www.synfig.org/

[7] Uživatelský manuál programu Synfig Studio [online]. Dostupné z: https://synfig.readthedocs.io/en/latest/index.html

[8] Tutoriály k programu Synfig Studio [online]. Dostupné z: https://wiki.synfig.org/Category:Tutorials

[9] Study material for CE\IT Students. Darshan: Institute of Engineering & Technology [online]. Dostupné z: http://www.darshan.ac.in/DIET/CE/GTU-Computer-Engineering-Study-Material

[10] BRINCH HANSEN, Per. Operating system principle [online]. 1. vydání. Englewood Cliffs: Prentice Hall, 1973. 380 stran. ISBN: 0-13-637843-9. Dostupné z: https://dl.acm.org/doi/pdf/10.5555/540365

9 Přílohy

[1] Obsah přiloženého ZIP souboru:

- Animace ve formátu GIF
- Animace ve formátu MP4
- Soubory pro program Synfig Studio

[2] Zadání bakalářské práce

Zadání bakalářské práce

Autor:	**Vojtěch Havlík**
Studium:	I1800172
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Grafické animace modelů pro podporu výuky operačních systémů**
Název bakalářské práce AJ:	Graphic animations of models to support the learning of operating systems

Cíl, metody, literatura, předpoklady:

Cílem práce je navrhnout a vytvořit vybrané animace klíčových modelů struktury a mechanismů operačních systémů Linux, UNIX, Android a Windows.

V teoretické části budou popsány techniky a postupy pro tvorbu animací využitelnou pro vzdělávací proces. Dále budou popsány teoretické principy vybraných funkcionalit OS, které budou modelovány.

V praktické části budou na vybranou technikou modelovány vybrané problémy z OS pro podporu výuky.

STALLINGS, William. *Operating Systems: Internals and Design Principles. Harlow, United Kingdom: Pearson Education Limited, 2018. ISBN 9781292214290.*

Garantující pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	Mgr. Josef Horálek, Ph.D.
Datum zadání závěrečné práce:	21.10.2019