Katedra informatiky, FEI VŠB-TUO, Petr Olivka.

4 Správa paměti

Paměť je důležitý systémový prostředek, který musí být pečlivě spravován. I když dnešní průměrný počítač má více než padesátkrát větší paměť, než měl největší počítač na světě šedesátých let - IBM 7094, tak i velikost programů narůstá stejným tempem, jako velikost pamětí. Za zmínku stojí Parkinsův zákon: „Programy se zvětšují tak, aby zaplnili paměť jim danou.“ V této kapitole se dovíme, jak operační systémy paměť spravují.

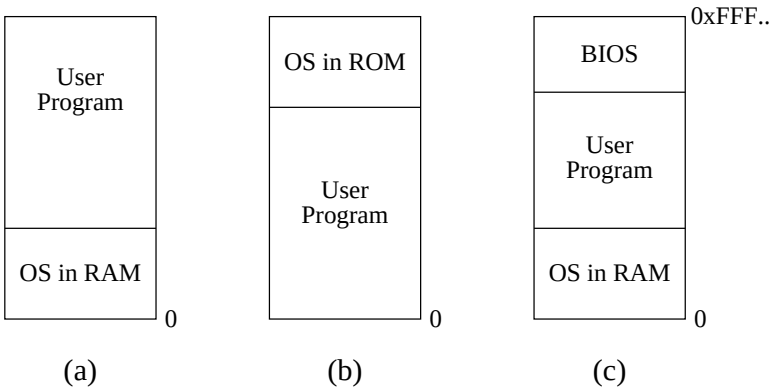
V ideálním případě by si každý programátor přál neomezeně velkou a rychlou paměť, která je *nonvolatile*, tzn. neztrácí svůj obsah při přerušení napájení. Když už jsme u přání, tak proč si také nepřát aby byla levná. Současná technika nám ovšem, bohužel, takovéto paměti neposkytuje. V důsledku toho má většina počítačů **hierarchickou strukturu paměti**, s malým množstvím velmi rychlé, drahé a *volatile* cache paměti, několik megabajtů středně rychlé, středně drahé a *volatile* hlavní paměti (RAM), a stovky či tisíce megabajtů pomalé, levné a *nonvolatile* diskové paměti. Úkolem operačního systému je koordinovat použití těchto pamětí.

Část operačního systému, která se stará o správu hierarchie paměti, se nazývá **správce paměti** (memory manager). Jeho úkolem je sledovat, které části paměti jsou používány a které ne, alokovat paměť procesům když ji potřebují, dealokovat při ukončení a spravovat odkládání mezi hlavní pamětí a diskem, pokud je hlavní paměť příliš malá pro udržení všech procesů.

V této kapitole budeme zkoumat různá schémata správy paměti. Od těch jednoduchých, až po velmi složité. Začneme od začátku a podíváme se na nejjednodušší správu paměti jaká je vůbec možná a postupně se dostaneme k více propracovaným metodám.

4.1 Základní správa paměti

Systémy správy paměti můžeme rozdělit do dvou tříd: těch, které přesouvají procesy tam a zpátky mezi hlavní pamětí a diskem v průběhu vykonávání (odkládání a stránkování), a těch které tak něčiní. Druhá třída je jednodušší, takže začneme od ní. V další části kapitoly budeme zkoumat odkládání (swapping) a stránkování (paging). V průběhu čtení kapitoly by jsme si měli uvědomit, že odkládání a stránkovaní jsou výtvory programátorů, jak obelstít nedostatek hlavní paměti a udržet současně všechny procesy. S klesajícími cenami hlavních pamětí, se některé argumenty hovořící pro určitý typ schématu řízení paměti stávají zastaralými, za podmínky ovšem, že se nezačnou zvětšovat programy rychleji, než budou klesat ceny pamětí.



Obrázek 1: Tři jednoduché metody organizace paměti s operačním systémem a jedním procesem. Jsou možná i jiná řešení.

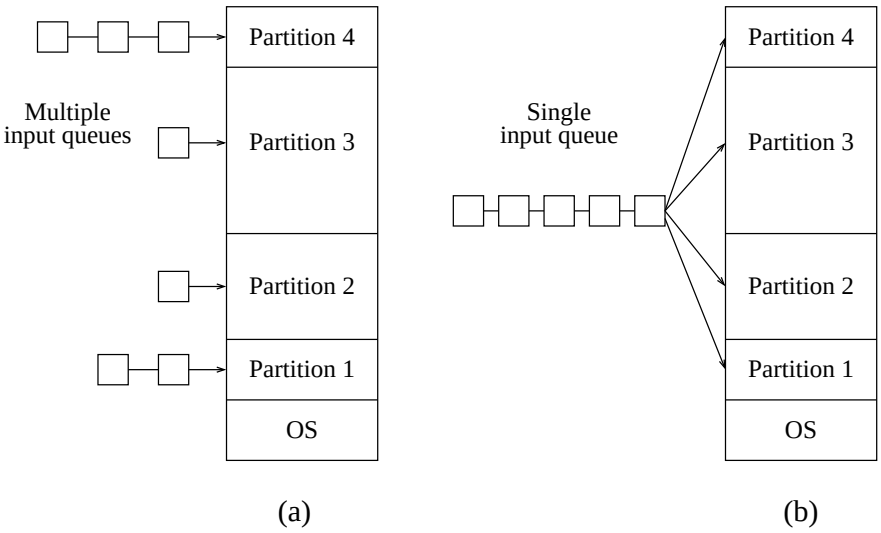
4.1.1 Jednoprocesové programování (monoprogramming) bez odkládání a stránkování

Nejjednodušší možné schéma řízení paměti je spuštění pouze jednoho programu v jednom okamžiku, sdílející paměť jen mezi programem a operačním systémem. Tři variace na toto téma jsou zachyceny na obrázku 1. Operační systém se může nacházet v dolní části paměti RAM (paměť s náhodným přístupem) viz obrázek 1(a), nebo může být uložený v ROM (paměť pouze pro čtení) v horní části paměti, viz obrázek 1(b), nebo řadiče zařízení (device drivers) mohou být v horní paměti v ROM a zbytek systému v RAM pod ní, viz obrázek 1(c). Poslední model se například používá v malých systémech MS-DOS. Na počítačích IBM PC se část systému uložená v ROM nazývá **BIOS**.

Pokud je systém organizován tímto způsobem, tak může v jednu chvíli běžet jen jeden proces Jakmile uživatel napíše příkaz, operační systém zkopíruje vyžádaný program z disku do paměti a spustí jej. Když proces skončí, operační systém zobrazí příkazový řádek a čeká na nový příkaz. Po zadání příkazu nahraje nový program do paměti a přepíše předchozí.

4.1.2 Víceprocesové programování s pevnými oddíly (Multiprogramming with fixed partitions)

Ačkoliv je někdy jednoprocesové programování používáno na malých počítačích s jednoduchým operačním systémem, tak se často požaduje, aby bylo možno spustit více procesů najednou. Na systémech sdílejících čas *(timesharing systems)* s více procesy v paměti to znamená, že když je jeden proces blokován čekáním na ukončení V/V, může jiný proces CPU použít. Takto víceprocesové programování zvyšuje využití CPU. Avšak i na osobních počítačích je mnohdy užitečné mít možnost spuštění dvou nebo více programů



Obrázek 2: Oddíly paměti stejné velikosti. (a) S oddělenými vstupními frontami (b) S jedinou vstupní frontou.

v jedné chvíli.

Nejjednodušší cesta na dosažení multiprocesového programování je jednoduše rozdělit paměť na n (třeba i nestejně velkých) oddílů. K tomuto dělení může dojít například při startu systému.

Když přijde úloha, tak může být vložena do fronty čekající na nejmenší oddíl, dostatečně velký ji uchovat. Protože jsou velikosti oddílů v tomto schématu pevně dané, tak je jakýkoliv nevyužitý prostor programem v oddílu ztracen. Na obrázku 2(a) vidíme, jak vypadá systém s pevnými oddíly a oddělenými vstupními frontami.

Nevýhoda třídění přicházejících úloh do oddělených front se stává viditelná, když fronta na velké oddíly je prázdná, ale fronta na malé oddíly je plná, tak jak je to vidět pro oddíly 1 a 3 na obrázku 2(a) Alternativní organizací paměti je udržovat jednu frontu, viz obrázek 2(b). Kdykoliv se oddíl stává prázdný, tak úloha nejblíže výstupu fronty, která se hodí do oddílu, může být nahrána do prázdného oddílu a spuštěna. Protože je nežádoucí plýtvat velké oddíly na malé úlohy, tak jiná strategie prohledává celou vstupní frontu a kdykoliv se oddíl stává prázdný, vybírá největší hodící se úlohu. Poznamenejme, že druhý zmíněný algoritmus znevýhodňuje malé úlohy, jako nezasluhující si celý oddíl, zatímco je obvykle žádoucí, aby malým úlohám byla poskytnuta nejlepší obsluha a ne právě nejhorší.

Jedním z možných východisek je, mít připraven nejméně jeden malý oddíl. Takovýto oddíl umožní malým úlohám, aby byly spuštěny bez potřeby, alokovat pro ně velký oddíl.

Jiný přístup je, mít stanoveno pravidlo, které zaručuje, že úloha při-

pravená k běhu nemůže být přeskočena více než k krát. Pokaždé když je přeskočena, získává bod. Po dosažení k bodů již přeskočena být nemůže.

Tento systém s pevnými oddíly, které operátor nastaví ráno a pak už je nemění, byl používán mnoho let v OS/360 na velkých sálových počítačích (mainframes) IBM. Nazýval se **MFT** (víceprocesové programování s pevným počtem úloh nebo také OS/MFT). Je jednoduchý k pochopení a stejně jednoduchý k implementaci: přicházející úlohy jsou zařazeny do fronty, dokud nebude k dispozici vhodný oddíl, a poté je úloha nahrána a spuštěna, dokud neskončí. V dnešní době umožňuje tento model jen několik, ne-li žádný, operační systém.

Přemístění a ochrana (relocation and protection)

Víceprocesové programování představilo dva zásadní problémy, které se musely vyřešit - přemístění a ochrana. Podívejme se na obrázek 2. Z obrázku je jasné, že různé úlohy budou spuštěny z různých adres. Když je program spojen (linked) (tedy hlavní program, uživatelské procedury a knihovny procedur) do jednoho adresového rozsahu, spojovací program (linker) musí vědět, na jaké adrese bude začátek programu.

Předpokládejme například, že první instrukce je volání procedury na absolutní adrese 100 v rámci binárního souboru vyrobeného spojovacím programem. Pokud je tento program nahrán do oddílu 1, pak tato instrukce skočí na absolutní adresu 100, která je uvnitř operačního systému. Je zapotřebí zavolat $100K+100$. Pokud je program nahrán do oddílu 2, pak se musí zajistit skok na $200K + 100$ atd. Tento problém je znám jako **přemístění** (relocation).

Jedno možné řešení relokace je vlastní modifikace instrukce v době nahrávání programu do paměti. Programům nahraným do oddílu 1 přidáme $100K$ ke každé adrese, programům nahraným do oddílu 2 přidáme $200K$ atd. Pro vykonání přemístění v průběhu nahrávání, jak jsme popsali, musí spojovací program přidat do binárního kódu seznam nebo bitmapu, říkající, která programová slova jsou adresy pro přemístění a která jsou naopak instrukcemi, konstantami a jinými věcmi, které se nesmí přemístit. Takto fungoval OS/MFT. Takto fungují i některé mikropočítače. Přemístění během nahrávání neřeší problém ochrany. Záludný (malicious) program může vždy vytvořit novou instrukci a skočit na ni. Protože programy v tomto systému používají absolutní paměťovou adresaci, namísto relativní adresy v registru, není zde možnost zamezit konstrukci instrukce, která čte nebo zapisuje do jakéhokoliv slova paměti. Ve víceuživatelských systémech je nežádoucí, aby bylo možno číst či zapisovat paměť, patřící jinému uživateli či procesu.

Řešení, které si vybralo IBM pro ochranu systému 360, bylo dělení paměti na bloky velikosti 2 kB a přiřazení 4 ochranných bitů každému bloku. PSW (Programm Status Word) obsahovalo 4 bitový klíč. Hardware IBM 360 zachytil jakýkoliv přístup běžícího procesu do paměti, jehož ochranný kód se

lišil od klíče PSW. Poněvadž jen operační systém mohl měnit ochranné kódy a klíče, tak uživatelským procesům bylo zamezeno ovlivňovat chod jiného programu a operačního systému, jako takového.

Alternativním řešením obou problémů, přemístění a ochrany, je vybavit stroj dvěma speciálními hardwarovými registry: bázový (base) a regist pro stanovení meze (limit). V průběhu plánování procesu, je bázový registr nastaven na adresu začátku oddílu a mezní registr je nastaven na délku oddílu. Ke každé adrese paměti, generované automaticky, je přičtena hodnota bázového registru před tím, než je poslána do paměti. Takže pokud je bázový registr nastaven na $100K$, pak volání `CALL 100` instrukce je okamžitě převedeno na instrukci `CALL 100K + 100`, bez nutnosti modifikovat instrukci samotnou. Adresy jsou kontrolovány pomocí mezního registru, zda nedochází k pokusu o přístup do paměti mimo oddíl. Hardware chrání bázový a mezní registr před přepsáním uživatelským programem.

CDC 6600 - první superpočítač na světě - tento postup používal. Procesor Intel 8088, užitý v původních strojích IBM PC, používal slabší verzi tohoto řešení - bázový registr, ale bez mezního registru. Počínaje 80286, bylo přijato lepší řešení .

4.2 Odkládání (swapping)

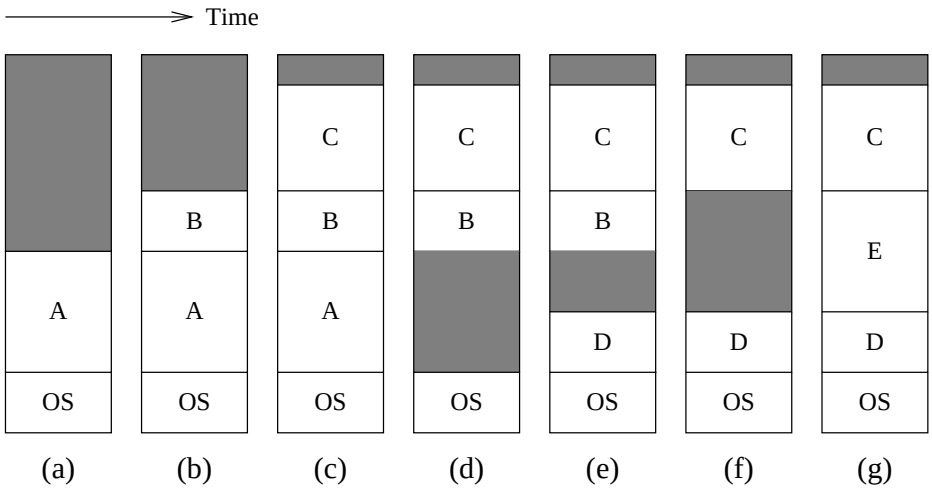
V dávkových systémech je organizace paměti do pevných oddílů jednoduchá a efektivní. Každá úloha je nahrána do oddílu, jakmile se dostane do čela fronty. Zůstává v paměti dokud neskončí. Dokud je dostatek úloh, které je možno uchovávat v paměti, a které zaměstnají CPU po celou dobu, není žádný důvod používat něco komplikovanějšího.

V systémech sdílejících čas, nebo u graficky orientovaných počítačů, je situace jiná. Občas se stává, že není dostatek paměti pro udržení všech aktivních procesů, takže nadbytečné procesy musí být uchovávány na disku a spouštěny dynamicky.

Existují dva obecné přístupy správy paměti, které mohou být použity v závislosti na dostupném hardware. Nejjednodušší strategie, nazývaná **odkládání** (swapping), se skládá z úplného přenesení procesu, jeho spuštění po jistou dobu, a pak následného odložení na disk. Jiná strategie, nazývaná **virtuální paměť**, umožňuje programům být spuštěn, i když se nacházejí v hlavní paměti jen částečně. V následujícím textu se podíváme na odkládání, v pak se budeme zabývat virtuální pamětí.

Operace odkládání je ilustrována na obrázku 3 Na začátku je v paměti jen proces A. Poté jsou vytvořeny procesy B a C, nebo jsou opětovně nahrány z disku. Na obrázku 3(d) proces A končí, nebo je odložen na disk. Poté se objevuje D a mizí B. Nakonec se objevuje E.

Hlavní rozdíl, mezi pevně danými oddíly na obrázku 2 a proměnnými oddíly na obrázku 3 je, že v druhém případě se může počet, umístění a velikost oddílu měnit dynamicky tak, jak procesy vznikají a zanikají, na rozdíl



Obrázek 3: Změny alokace paměti při spouštění procesů. Šedá barva znamená volnou paměť.

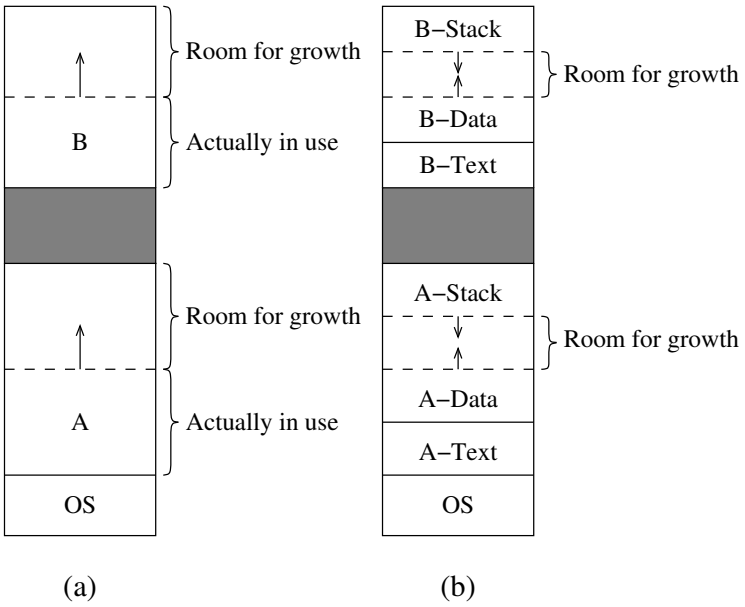
od pevných oddílů v prvním případě. Přizpůsobivost toho, že nejsme vázáni pevným počtem oddílů, které by mohly být příliš velké nebo malé, zlepšuje využití paměti, ale také komplikuje alokaci a dealokaci paměti, stejně tak jako její sledování.

Když proces odkládání vytvoří několik mezer v paměti, je možné je spojit do jedné velké oblasti pomocí přesunutí všech procesů na začátek paměti, tak jak je to jen možné. Tato technika je také známá jako **setřásání paměti** (memory compaction). Obvykle se ale neprovádí, protože vyžaduje hodně procesorového času. Například na stroji s 32 MB, který kopíruje 16 bajtů za mikrosekundu, by trvalo 2 sekundy setřásání celé paměti.

Jedním z bodů, který si zasluhuje naší pozornost, je, jak velká paměť by měla být alokována pro proces, který je vytvořen nebo odložen. Jsou-li procesy vytvářeny s pevnou délkou, která se nemění, pak je alokace jednoduchá: alokuje se přesně tolik, kolik je třeba, nic víc nic míň.

Jestliže však mohou datové segmenty procesů narůstat, například dynamickou alokací pamětí z haldy *(heap)*, jak je to zvykem v mnoha programovacích jazycích, nastává problém vždy, když se proces snaží zvětšit. Pokud mezera přiléhá k procesu, pak může být prostor naalokována a procesu je povoleno narůst do mezery. Na druhou stranu, pokud proces přiléhá k jinému procesu, tak narůstající proces bude buď přemístěn do mezery v paměti dostatečně velké, nebo jeden nebo více procesů bude odloženo, aby se vytvořila dostatečně velká mezera. Jestliže proces nemůže narůst v paměti a odkládací prostor na disku je plný, pak musí proces počkat nebo bude ukončen *(killed)*.

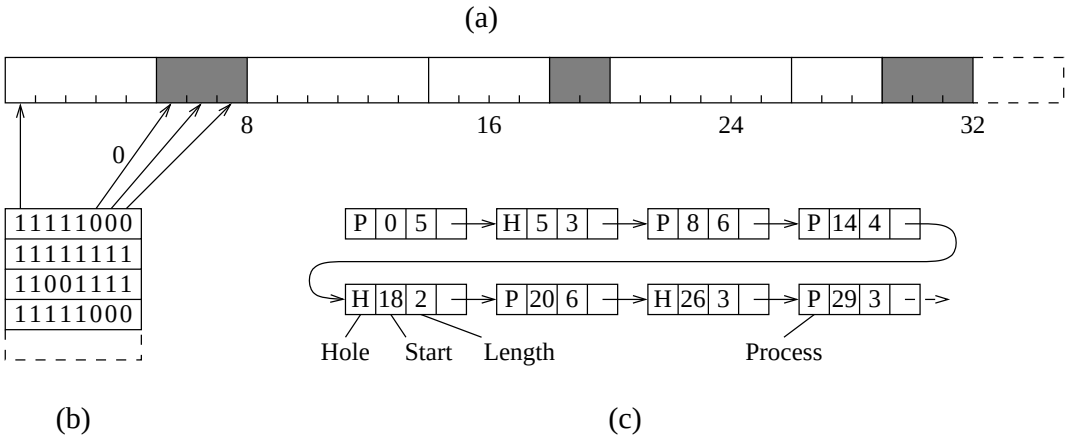
Očekává-li se, že procesy budou v průběhu běhu narůstat, je pravdě-



Obrázek 4: (a) Prostor pro alokaci datového segmentu. (b) Prostor pro alokaci datového segmentu i zásobníku.

podobně dobrým nápadem alokovat navíc trochu paměti kdykoliv, když je proces odložen nebo přesunut, aby se snížily režijní náklady spojené s přesunem a odkládáním procesů, které se již nevlezou do jejich alokované paměti. Nicméně, při okládání procesů na disk se odkládá jen pamět skutečně používáná. Bylo by plýtvání odkládat i paměť alokovanou navíc. Na obrázku 4(a) vidíme uspořádání paměti s dvěma procesy, ve které byl alokován prostor v paměti pro růst.

Pokud procesy mohou mít dva rostoucí segmenty, například používají-li datový segment jako haldu (heap) pro dynamicky alokované proměnné a zásobníkový segment (stack) pro normální lokální proměnné a návratové adresy, pak je vhodné použít alternativní uspořádání, konkrétně takové, jaké je na obrázku 4. Na obrázku vidíme, že každý znázorněný proces má zásobník na vrcholu alokované paměti, která roste směrem dolů a datový segment přesně na druhé straně, který roste nahoru. Paměť mezi nimi může být tedy použita pro oba segmenty. Zaplní-li se tato paměť úplně, pak se musí proces přesunout to větší mezery v paměti, nebo musí být odložen ven z paměti, dokud se nevytvoří dostatečně velká mezera, nebo se musí ukončit.



Obrázek 5: (a) Příklad paměti s pěti procesy a třemi mezerami (šedá barva). (b) Odpovídající bitmapa. (c) Stejná informace jako seznam.

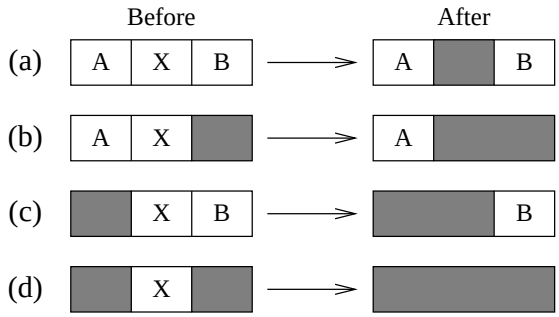
4.2.1 Správa paměti bitmapami (Memory Management with Bit Maps)

Přiřazuje-li se paměť dynamicky, musí jí operační systém spravovat. Obecně existují dva způsoby, jak sledovat využití paměti: bitmapami (bit maps) a seznamem (free lists). V této a následující části se postupně podíváme na obě metody.

U bitmapy je paměť rozdělena na alokační jednotky (allocation units), které mohou mít velikost od několika slov, až po několik kilobajtů. Ke každé alokační jednotce přísluší jeden bit v bitmapě, kde 0 znamená, že jednotka je volná a 1, když je obsazená (nebo naopak). Obrázek 5 ukazuje část paměti a odpovídající bitmapu.

Velikost alokační jednotky je důležitým parametrem pro návrh. Čím jsou alokační jednotky menší, tím větší je bitmapa. Nicméně i s alokační jednotkou velikosti pouhých 4 bajtů potřebuje 32 bitů paměti jeden bit v bitmapě. Paměť velikosti $n * 32$ bitů bude potřebovat n bitovou bitmapu, takže bitmapa zabere pouze 1/33 celkové velikosti paměti. Zvolíme-li velikost alokační jednotky větší, bitmapa bude menší, ale značná část paměti může být ztracena v poslední jednotce, pokud velikost procesu není přesný násobek alokační jednotky.

Bitmapa poskytuje jednoduchou cestu, jak sledovat slova paměti v pevně (velikostně) dané paměti, vzhledem k tomu, že velikost bitmapy závisí jen na velikosti paměti a velikosti alokační jednotky. Hlavní problém nastává, když se rozhodneme načíst proces délky k jednotek do paměti, pak musí správce paměti najít v bitmapě celistvý úsek nul délky k. Prohledávání bitmapy na úseky dané délky je pomalá operace (protože sekvence může být rozložena mezi slovy v bitmapě). Toto je také hlavní argument proti používání bitmap.



Obrázek 6: Čtyři možné kombinace sousedních procesů končícího procesu X.

4.2.2 Správa paměti s propojenými seznamy (Memory management with linked lists)

Jinou cestou jak spravovat paměť, je udržovat propojený seznam alokovaných a volných paměťových segmentů, kde segment je buď proces nebo mezera mezi procesy. Paměť na obrázku 5(a) je reprezentována na obrázku 5(c) jako propojený seznam. Každý záznam v seznamu specifikuje mezeru (hole - H), nebo proces (P), počáteční adresu, délku a ukazatel na další záznam.

V příkladu je seznam segmentů seřazen podle adresy. Seřazení tímto způsobem má výhodu, že když se proces ukončí nebo je odložen, aktualizace seznamu je přímá. Ukončený proces má normálně dva sousedy (kromě případů kdy je na úplném začátku nebo konci paměti). Tito sousedé mohou být buďto procesy nebo mezery, což nás vede ke čtyřem kombinacím na obrázku 6. Na obrázku 6(a) aktualizace seznamu vyžaduje náhradu P za H. Na obrázku 6(b) a (c), se dvě položky spojí do jedné, čímž se seznam stane o jednu položku kratší. Na obrázku 6(d) se tři položky spojí dohromady a dvě položky jsou odstraněny se seznamu. Poněvadž tabulka procesů (process table slot) končícího procesu bude normálně ukazovat na záznam samotného procesu, bylo by daleko vhodnější mít seznam jako oboustranně propojený seznam, než jen jako jednostranně propojený seznam na 5(c). Tato struktura umožňuje jednodušeji najít předchozí záznam a tak zjistit, zda je možné sloučení.

Máme-li seřazeny procesy a mezery podle adresy, můžeme použít několik algoritmů pro alokaci paměti pro nově vzniklé, nebo odložené procesy. Předpokládáme, že správce paměti ví, kolik paměti má alokovat. Nejjednodušší algoritmus se nazývá **první vhodný** (first fit). Správce paměti prochází seznam, dokud nenajde dostatečně velkou mezeru. Mezera je poté rozdělena na dvě části, jedna část pro proces a druhá bude nevyužita, kromě případů, kdy se proces přesně vejde do mezery. Tento algoritmus je rychlý, protože prohledává jen do prvního úspěchu.

Malou odchylkou od prvního vhodného je **další vhodný** (next fit). Fun-

guje stejně jako první vhodný s tím rozdílem, že si uchovává záznam o poloze použité mezery. Při příštím hledání mezery pokračuje hledáním v seznamu od místa, kde naposled skončil, na rozdíl od prvního vhodného, který začíná vždy od začátku. Simulace prováděné p. Baysem (1977) ukazují, že další vhodný dává o trošku horší výsledky než první vhodný.

Další známý algoritmus se nazývá **nejlepší správný** (best fit). Nejlepší správný prohledává celý seznam a vybírá nejmenší vhodnou mezeru. Raději, než rozdělovat velké mezery, které se mohou ještě hodit, se nejlepší správný snaží najít mezeru velikosti nejbližší požadavku.

Podívejme se znova na příklady první a nejlepší vhodný na obrázku 5. Potřebujeme-li blok velikost 2, první vhodný nám alokuje díru na pozici 5, ale nejlepší správný alokuje díru na pozici 18. Nejlepší správný je pomalejší než první vhodný, protože musí prohledat celý seznam pokaždé když, je použit. Překvapivé ovšem je, že vede k většímu plýtvání pamětí, než první vhodný a další vhodný, protože má sklon rozmělnit paměť na malé mezery. První vhodný vytváří průměrně větší mezery.

Abychom odstranili problém rozbíjení téměř vyhovujících mezer na proces a nepoužitelné malé mezery, mohly bychom uvažovat o algoritmu **nejhůře vyhovujícím** (worst fit), který vždy vybere největší mezeru, která po rozbití bude dostatečně velká, aby byla ještě použitelná. Simulace ovšem ukázaly, že nejhůře vyhovující také není nejlepším řešením.

Všechny čtyři algoritmy mohou být urychleny udržováním oddělených seznamů procesů a mezer. Tímto způsobem mohou všechny čtyři algoritmy věnovat celou svou energii prohledáváním mezer a ne procesů. Nevyhnutelnou cenu, kterou musíme zaplatit za toto urychlení, je zvýšená složitost a zpomalení při dealokaci paměti, protože uvolněný segment musí být odstraněn ze seznamu procesů a vložen do seznamu mezer.

Je-li seznam mezer oddělen od seznamu procesů, máme možnost drobné optimalizace. Místo toho, abychom měli oddělené soubory datových struktur pro udržení seznamu mezer, tak jak tomu je na obrázku 5(c), můžeme k tomu využít mezery samotné. Prvním slovem každé mezery by byla velikost mezery a druhým slovem ukazatel na následující položku. Seznam na obrázku 6(c), který potřebuje 3 slova a jeden bit (P/H), již není třeba.

Existuje ještě jeden algoritmus, nazývaný **rychlý vhodný** (quick fit), který si udržuje oddělené seznamy pro některé často používané velikosti. Může mít například tabulku n položek, ve které první záznam ukazuje na hlavičku seznamu mezer velikosti 4 kB, druhý záznam ukazuje na seznam mezer velikostí 8 kB, třetí na seznam 12 kB mezer atd. Mezery velikosti řekněme 21 kB mohou být buď zařazeny do seznamu 20 kB mezer nebo na zvláštní seznam mezer liché velikosti. U rychlého vhodného je hledání díry dané velikostí extrémně rychlé, ale má to také některé nevýhody, stejně jako všechna schémata, která třídí podle velikosti mezer. Konkrétně když proces skončí nebo je odložen, nalezení jeho sousedů a zjištění zda se vůbec můžou spojit, je časově náročné. Pokud nedojde ke spojení, tak se paměť velmi

rychle fragmentuje na malé mezery, do kterých se již žádný proces nevejde.

4.3 Virtuální paměť (Virtual Memory)

Před mnoha lety byli návrháři poprvé postaveni před problém, kdy programy byly příliš velké na to, aby se vešly do dostupné paměti. Řešení, které se obvykle užívalo, bylo rozdělit program na několik částí, nazývaných překryvnými moduly (overlays). Modul 0 začínala běžet jako první. Když skončí, zavolalá další modul. Některé systémy modulů byly velmi složité, umožňující v jednom okamžiku existenci několika modulů v paměti. Moduly se uchovávaly na disku a byly odkládany z a do paměti dynamicky operačním systémem podle potřeby.

Ačkoliv skutečnou práci s odládáním modulů prováděl systém, proces dělení programu na části, musel provádět programátor. Dělení velkých programů na malé modulární části, bylo časově náročné a nezáživné. Netrvalo dlouho a někdo přišel na myšlenku, aby se tato činnost převedla na počítač.

Metoda, která pro to byla vymyšlena (Fortheringham, 1961), se dostala do povědomí jako **virtuální paměť**. Základní myšlenka, na které je založena virtuální paměť, je ta, že celková velikost programů, dat a zásobníků může překročit velikost dostupné fyzické paměti. Operační systém uchovává v hlavní paměti jen ty části, které jsou právě používány, a zbytek má na disku. Například 16MB program může běžet na stroji s 4MB paměti díky tomu, že se podle potřeby pečlivě vybírá, které 4MB části programu uchovávat v paměti a které části má odkládat mezi pamětí a diskem.

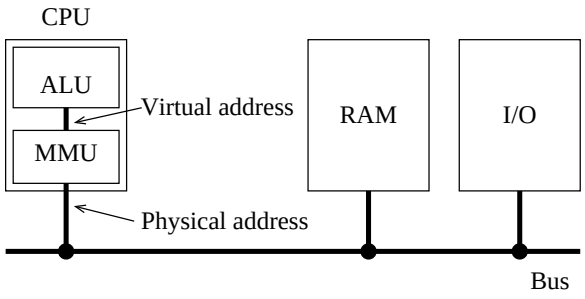
Virtuální paměť může fungovat i na víceprocesových systémech, kdy jsou v jedné chvíli v paměti kousky i větší části mnoha programů. V průběhu doby, kdy program čeká na načtení vlastních části, čeká tak na V/V a nemůže běžet, může být CPU využita jiným procesem, stejným způsobem jako u jiných víceprocesových systémů.

4.3.1 Stránkování *(paging)*

Většina systémů virtuální paměti používá techniku zvanou **stránkování** (paging), kterou si nyní popíšeme. Na každém počítači existují soubory adres paměti, které můžou programy vytvářet. Když program použije instrukci typu

`MOVE REG, 1000`,

kopíruje obsah paměti na adrese 1000 do `REG` (nebo naopak v závislosti na použitém počítači). Adresy mohou být vytvářeny pomocí indexových (indexing), bázových (base), segmentových (segment) registrů, nebo jinými způsoby. Tyto programově vygenerované adresy se nazývají **virtuálními adresami** a vytváří **virtuální adresový prostor** (virtual address space). Na počítačích bez virtuální paměti, se virtuální adresy posílají přímo na paměťovou sběrnici, což způsobí, že se do fyzické paměti zapíše nebo přečte



Obrázek 7: Umístění a funkce MMU.

slovo z dané adresy. Používá-li se virtuální paměť, pak virtuální adresa nejde přímo na paměťovou sběrnici. Místo toho jde do **jednotky správy paměti** (Memory Management Unit - MMU), realizovanou obvodem nebo obvody, které mapují virtuální adresy na fyzické adresy, jak je ukázáno na obrázku 7.

Jednoduchá ukázka, jak mapování funguje, je na obrázku 8. Na tomto příkladu máme počítač, který je schopný generovat 16bitové adresy, od 0 do 64 kB. To jsou ale virtuální adresy. Ač má tento počítač jen 32 kB fyzické paměti, tak přesto můžeme napsat 64 kB programy, ale i tak tyto programy nemůžou být celé nahrány do paměti a spuštěny. Úplná kopie jádra programu, až do velikosti 64 kB, musí být přítomna na disku tak, aby mohly být jednotlivé části nahrány, až bude potřeba. Virtuální adresní prostor se dělí na několik jednotek zvaných **stránky** (pages). Odpovídající jednotky ve fyzické paměti se nazývají **rámce stránek** (page frames). Stránky a rámce stránek jsou vždy stejné velikosti. V našem příkladu mají 4 kB, obecně se však na současných systémech používají stránky velikosti od 512 bytů do 64 kB. S 64 kB virtuální paměti a 32 kB fyzické paměti máme 16 virtuálních stránek a 8 rámců stránek. Přenosy mezi pamětí a diskem jsou vždy realizovány po stránkách.

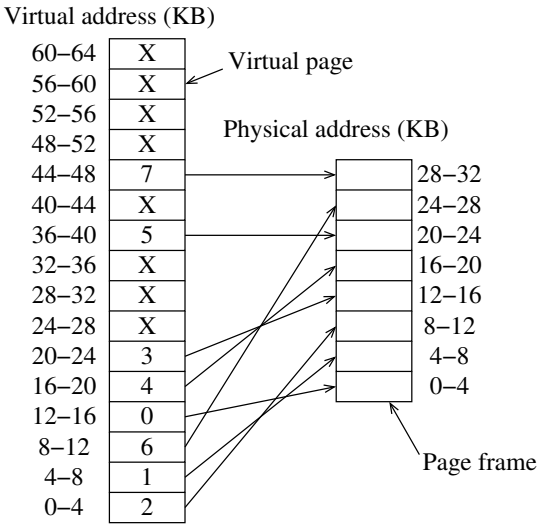
Když se program například pokusí o přístup na adresu použitím instrukce

```
MOVE REG, 0,
```

tak se virtuální adresa 0 pošle do jednotky MMU. Jednotka MMU vidí, že virtuální adresa spadá pod stránku 0 (0-4095), která odpovídá podle odpovídajícího mapování rámci stránky 2 (8192-12287). Takto transformuje adresu na 8192 a pošle adresu 8192 na sběrnici. Paměťová jednotka neví vůbec nic o MMU a vidí jen požadavek na zápis či čtení z adresy 8192, kterou zná. Tak právě jednotka MMU efektivně přemapovala celé virtuální adresy mezi 0-4095 na fyzické adresy 8192-12287.

Podobně instrukce

```
MOVE REG, 8192
```



Obrázek 8: Vztah mezi virtuální adresou a fyzickou pamětí je dán tabulkou stránek.

je fakticky transformována na

`MOVE REG, 24576`,

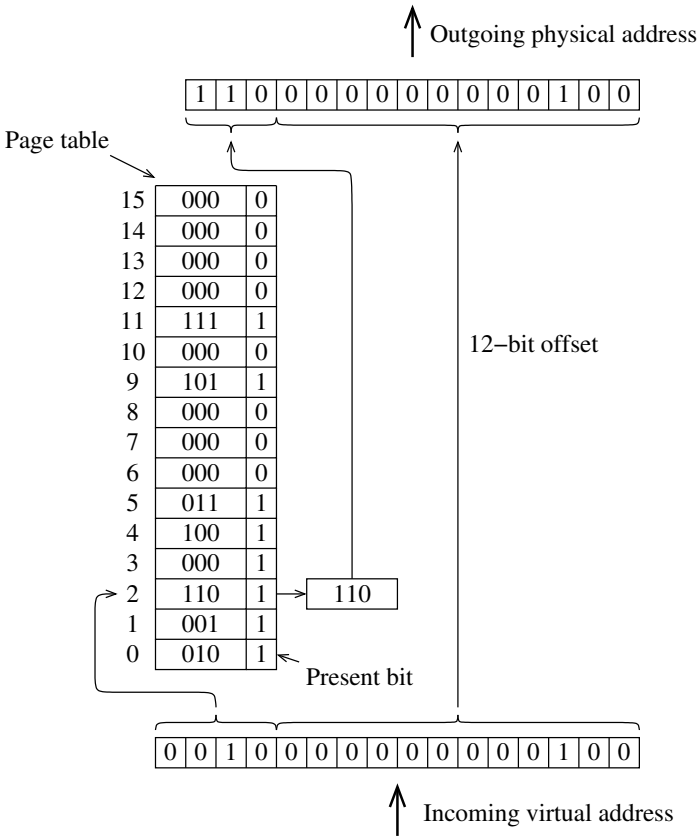
protože virtuální adresa 8192 je ve virtuální stránce 2 a tato stránka je mapována na fyzický rámec stránky 6 (fyzické adresy mezi 24576 až 28671). Jako třetí příklad uveďme virtuální adresu 20500, která je 20 bajtů od začátku virtuální stránky 5 (virtuální adresy $20480 - 24575$) a mapujeme ji na fyzickou adresu $12288 + 20 = 12308$.

Možnost mapovat 16 virtuálních stránek na 8 rámců stránek příslušným nastavením MMU nám samo o sobě neřeší problém, že virtuální adresový prostor je větší než fyzická paměť. Protože máme jen 8 fyzických rámců stránek, tak pouze 8 virtuálních stránek je mapovaných do fyzické paměti. Zbylé, na obrázku označené křížkem, nejsou mapované. Ve skutečném hardwaru je ke každému záznamu přiřazen bit informující o **přítomnosti nebo nepřítomnosti** mapování dané stránky (present/absent bit).

Co se stane, když se program pokusí použít nemapovanou stránku, například použitím instrukce

`MOVE REG, 32780`

což je 12. bajt na virtuální stránce 8 (začínající na 32768)? Jednotka MMU oznámí, že stránka není namapovaná (v obrázku indikováno křížkem), a CPU toto předá operačnímu systému. Tento odchyt se nazývá **výpadek stránky** (page fault). Operační systém vezme nejméně používaný rámec stránky a zapíše jeho obsah zpátky na disk. Poté přenese právě odkazovanou stránku do právě uvolněného rámce stránky, změní mapu a spustí znova



Obrázek 9: Vnitřní operace MMU s $16 \times 4kB$ stránkami

odchycenou instrukci. Pokud se například operační systém rozhodne vyklidit rámec stránky 1, nahraje virtuální stránku 8 na fyzickou adresu 4 kB a provede dvě změny do mapy MMU. Zaprvé označí záznam virtuální stránky 1 jako nemapovaný, aby odchytil případné budoucí přístupy na virtuální adresy v rozsahu 4 a 8 kB. Poté nahradí křížek na virtuální stránce 8 za 1, takže poté, co je odchycená instrukce znovu spuštěna, bude namapována virtuální adresa 32780 na fyzickou adresu 4108.

Nyní se podívejme dovnitř MMU, abychom se dozvěděli, jak to funguje a proč musíme vybírat velikosti stránek jako mocninu 2. Na obrázku 9 vidíme příklad virtuální adresy 8196 (0010000000000100 v binární podobě), mapované použitím mapy MMU na obrázku 8 Příchozí 16-bitová adresa je rozdělena na 4-bitové číslo stránky a 12-bitový offset. Čtyři bity na číslo stránky může reprezentovat 16 stránek a s 12-bitovým offsetem můžeme adresovat celých 4096 bytů v rámci stránky. Číslo stránky je použito jako ukazatel do tabulky stránek *page table* a převádí číslo rámce stránky, podle odpo-

vídající virtuální stránky. Je-li bit přítomnosti stránky nastaven na nulu, pak je toto odchyceno operačním systémem. Je-li tam nastavena jednička, pak je číslo rámce stránky nacházející se v tabulce stránek zkopírováno do horních 3 bitů výstupního registru společně s 12-bitových offsetem, který je beze změny zkopírován z příchozí virtuální adresy. Společně vytvářejí 15-ti bitovou fyzickou adresu. Hodnota výstupního registru je poté poslána na paměťovou sběrnici, jako fyzická adresa paměti.

4.3.2 Tabulky stránek (page tables)

Teoreticky funguje mapování virtuální paměti na fyzické adresy tak, jak jsme právě popsali. Virtuální adresa je rozdělena na číslo virtuální stránky (vyšší bity) a offset (nižší bity). Číslo virtuální stránky se použije jako ukazatel do tabulky stránek, aby se nalezl záznam virtuální stránky. Ze záznamu v tabulce stránek se dohledá číslo rámce stránky (pokud existuje). Číslo rámce stránky je připojeno za konec (vyšší pozice) offsetu, nahrazuje tak číslo virtuální stránky a vytváří fyzickou adresu, která může být poslána do paměti.

Účelem tabulky stránek je, mapovat virtuální stránky na rámce stránek. Řečeno matematicky, tabulka stránek je funkce, která má jako argument číslo virtuální stránky a výsledkem je číslo fyzického rámce. Užitím výsledku této funkce, může být nahrazeno pole virtuální stránky ve virtuální adrese za pole rámce, což vytvoří fyzickou adresu paměti.

Navzdory jednoduchému popisu se můžeme potýkat s dvěma podstatnými problémy:

1. Tabulka stránek může být extrémně velká.

2. Mapování musí být rychlé.

První bod vychází z faktu, že moderní počítače užívají minimálně 32-bitové virtuální adresy. Což, řekněme se stránkami velikosti 4 kB a 32-bitovým adresným prostorem znamená 10^6 stránek a s 64 bitovým adresním prostorem více, než si dokážeme představit. S miliónem stránek ve virtuálním adresním prostoru musí mít tabulka stránek 1 milión položek. A nezapomínejme také na to, že každý proces má vlastní tabulku stránek.

Druhý bod je důsledkem faktu, že mapování virtuální adresy na fyzickou adresu musí být prováděno při každém přístupu do paměti. Typická instrukce má instrukční slovo a často i paměťový operand. V důsledku toho, je třeba provádět 1, 2, nebo někdy i více odkazů v tabulce stránek během jedné instrukce. Jestliže instrukce trvá, řekněme 10ns, hledání v tabulce stránek musí být dokončeno během několika nanosekund, aby se nastalo úzkým hrdlem.

Potřeba vytvářet rychlé mapování pro velké paměti je významným bodem ve vývoji počítačů. I když je tento problém nejvíce řešen u špičkových

strojů, je podstatný i u strojů nižší třídy, kde je cena vzhledem k výkonu rozhodující. V této a následující části se podrobně podíváme na návrh tabulek stránek a ukážeme si řadu hardwarových řešení, která se v současné době používají.

Nejjednodušší návrh (co do koncepčnosti) je mít jednu tabulku stránek, skládající se z pole rychlých hardwarových registrů s jednou položkou pro každou virtuální stránku, které jsou seřazeny podle čísla virtuální stránky. Když je proces spuštěn, operační systém naplní registry tabulkou stránek procesu, která je převzata z kopie uchovávané v hlavní paměti. V průběhu procesu nejsou třeba žádné další odkazy do paměti pro tabulku stránek. Výhodou této metody je její jednoduchost a to, že nepotřebuje v průběhu mapování žádné odkazy do paměti. Nevýhodou je, že může být potenciálně nákladná (pokud jsou tabulky stránek velké). Povinnost nahrát tabulku stránek v každém přepnutí kontextu, může také negativně ovlivnit výkon.

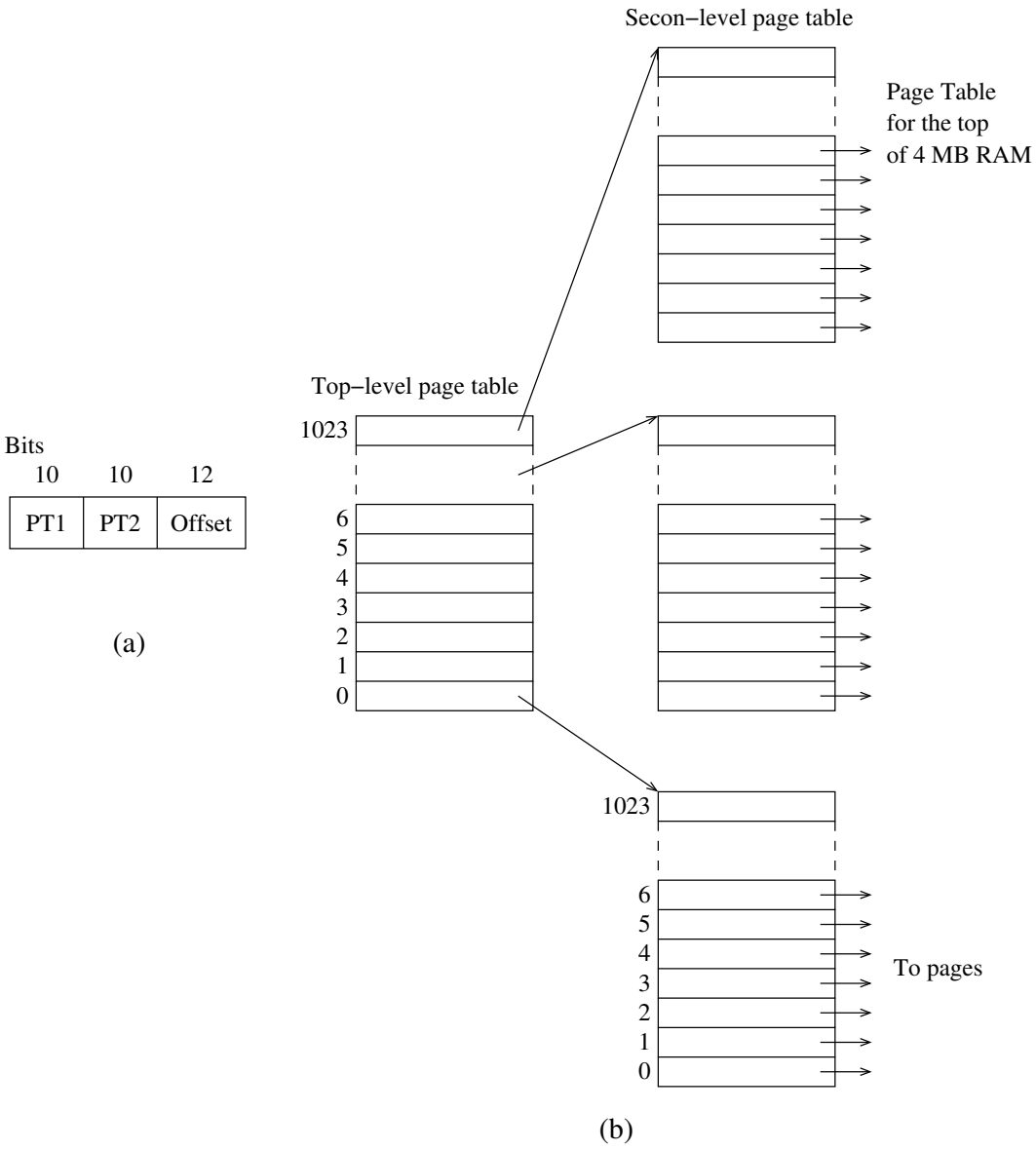
Jiným extrémem je, že tabulka stránek může být celá v hlavní paměti. Veškerý hardware, který je k tomu třeba, je jeden registr ukazující na začátek tabulky stránek. Tento návrh umožňuje přepnout kontext mapy paměti pouhou změnou jednoho registru. Samozřejmě i to má své nevýhody, například potřebu jednoho nebo více odkazů do paměti pro načtení záznamů tabulky stránky během vykonávání každé instrukce. Z tohoto důvodu je tento přístup jen zřídka používán ve své čisté podobě a my se budeme dále zabývat některými obměnami, které mají vyšší výkon.

Víceúrovňové tabulky stránek (Multilevel Page Tables)

Abychom se vyhnuli problému nutnosti držet po celou dobu velké tabulky stránek v paměti, používá mnoho počítačů víceúrovňové tabulky stránek. Jednoduchý příklad je ukázán na obrázku 10. Na obrázku 10(a) máme 32-bitové virtuální adresy rozděleny na 10 bitů pole PT1, 10 bitů pole PT2 a 12 bitů pole offsetu. Protože offsety mají 12 bitů, stránky mají 4 kB, takže jich je celkem 2^{20}.

Tajemstvím metody víceúrovňových tabulek stránek je, předcházet tomu, aby nebyly po celou dobu všechny tabulky v paměti. Zvláště aby ty, které nejsou potřeba, nebyly udržovány v paměti. Předpokládejme, že proces potřebuje 12 MB, dolní 4 MB pro text programu, dalších 4MB pro data, a horní 4 MB pro zásobník. Mezi vrcholem dat a spodkem zásobníku je obrovská mezera, která není využita.

Na obrázku 10(b) vidíme, jak na tomto příkladu funguje dvouúrovňová tabulka stránek. V levé části vidíme tabulku stránek nejvyšší úrovně s 1024 položkami, odpovídajících 10 bitům pole PT1. Když je virtuální adresa předána jednotce MMU, je nejdříve vyňato pole PT1 a jeho hodnota se použije jako ukazatel do tabulky stránek nejvyšší úrovně. Každá z těchto 1024 položek reprezentuje 4 MB, protože celé 4 GB adresního prostoru jsou rozděleny na díly velikosti 1024 bajtů. Položka nalezená pomocí ukazatele v tabulce nejvyšší úrovně udává adresu nebo číslo rámce stránky tabulky strá-



Obrázek 10: (a) 32 bitová adresa se dvěma indexy do tabulek stránek. (b) Dvoúrovňové stránkování.

nek druhé úrovně. Položka 0 v tabulce nejvyšší úrovně ukazuje na tabulku stránek programového textu, položka 1 ukazuje na tabulku stránek dat a položka 1023 ukazuje na tabulku stránek zásobníku. Ostatní (šedé) položky nejsou použity. Pole PT2 se nyní použije jako ukazatel do vybrané tabulky stránek druhé úrovně, aby se nalezlo číslo rámce stránky pro stránku samou.

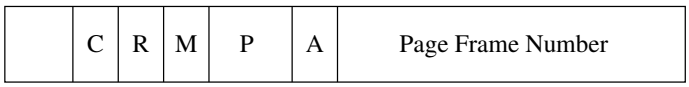
Předpokládejme například 32 bitovou virtuální adresu 0x00403004 (dekadicky 4206596), která je 12292 bytem v datech. Tato adresa odpovídá $PT1 = 1$, $PT2 = 3$ a $offset = 4$. Jednotka MMU použije nejprve PT1 jako ukazatel do tabulky stránek nejvyšší úrovně a získá záznam 1, který odpovídá adresám od 4 MB do 8 MB. Poté použije PT2 jako ukazatel do tabulky stránek druhé úrovně a vyjme záznam 3, který odpovídá adresám 12288 až 16383 v rámci 4 MB úseku (tj. absolutní adresy 4206592 do 4210687). Tato položka obsahuje číslo rámce stránky obsahující stránku virtuální adresy 0x00403004. Jestliže tato stránka není v paměti, bit přítomnosti stránky je nastaven na nulu, způsobí to výpadek stránky. Je-li stránka v paměti, tak se číslo rámce stránky přečtené z tabulky stránek druhé úrovně složí s offsetem 4 a vytvoří tak fyzickou adresu. Tato adresa je vložena na sběrnici a odeslána do paměti. Můžeme si všimnout jedné zajímavé věci na obrázku 10, přestože adresní prostor obsahuje více než milión stránek, ve skutečnosti potřebujeme jen 4 tabulky stránek: tabulku nejvyšší úrovně a tabulky druhé úrovně v rozsazích 0-4 MB, 4-8 MB a nejvyšších 4 MB.

Bit přítomnosti stránky v 1021 položkách tabulky nejvyšší úrobně je nastaven na nulu a vytváří tak výpadek stránky, pokud je nějaká z nich použita. Pokud se toto stane, operační systém zaregistruje, že se proces snaží dostat do paměti, kam by neměl a provede odpovídající akce, například pošle aplikaci příkaz k ukončení (killing signal). V tomto příkladu jsme vybrali celá čísla pro různé velikosti a vybrali jsme shodná PT1 a PT2, ale v praxi jsou samozřejmě možné i jiné hodnoty.

Dvouúrovňový systém tabulek stránek může být rozšířen na tři, čtyři nebo více úrovní. Další úrovně umožňují větší flexibilitu, ale dá se pochybovat o tom, že zvýšená složitost více než třech úrovní bude mít smysl.

Přejděme teď od obecné struktury tabulek stránek k podrobnostem záznamů na jednotlivých stránkách. Přesná struktura záznamu je sice silně hardwarově závislá, ale typ informací v ní obsažených je zhruba stejný u všech počítačů. Na obrázku 11 je ukázka záznamu v tabulce stránek. Velikosti se liší podle typu počítače, ale 32 bitů je nejobvyklejší. Nejdůležitější pole je číslo rámce stránky (Page frame number). Konec konců, cílem mapování je nalezení právě této hodnoty. Další položkou je bit přítomnosti stránky. Je-li tento bit nastaven na jedničku, záznam je platný a může být použit. Je-li nulový, virtuální stránka ke které přísluší, není právě v paměti. Přístup k záznamu v tabulce stránky, který má hodnotu toho bitu na nule, vyvolá výpadek stránky.

Ochranné bity (Protection bits) nám řeknou, jaký typ přístupu je povolen. V nejjednodušším případě tato položka obsahuje jeden bit, nastaven na



C – Disable Caching
R – Referenced
M – Modified
P – Protection
A – Present/Absent

Obrázek 11: Typický záznam v tabulce stránek.

0 pro čtení/zápis a 1 pouze pro čtení. Složitější uspořádání je se 3 bity, po bitu pro povolení čtení, zápisu a vykonání stránky.

Bit modifikace (Modified) a odkazu (Referenced) uchovává informace o používání stránky. Je-li proveden zápis do stránky, hardware automaticky nastaví bit modifikace. Tento bit nabývá významu, když se operační systém rozhodne vrátit zpět rámec stránky. Byla-li stránka modifikována (tj. dotčena - dirty), musí být zapsána zpět na disk. Pokud nebyla modifikovaná, může se zrušit, protože její kopie na disku je stále platná. Tento bit se občas nazývá **bit použití** (dirty bit), protože odráží stav stránky.

Bit odkazu (referenced) se nastavuje, kdykoliv je na stránku odkázáno, buď při čtení nebo zápisu. Jeho hodnota pomáhá operačnímu systému rozhodnout, kterou stránku vyklidit když dojde k výpadku stránky. Stránky, které nejsou používané, jsou lepšími kandidáty, než stránky používané a tento bit hraje také podstatnou úlohu v několika algoritmech náhrad stránek, které budeme probírat později v této kapitole.

A konečně poslední bit umožňuje vypnout rychlou vyrovnávací paměť (caching) pro tuto stránku. Tato vlastnost je důležitější pro stránky mapující zařízení na registry, než pro paměť. Čeká-li operační systém ve smyčce na odpověď příkazu zaslaného nějakému V/V zařízení, je nezbytné, aby hardware načítal slovo ze zařízení a nepoužíval starou kopii ve vyrovnávací paměti. Pomocí toho bitu můžeme vyrovnávací paměť vypnout. Stroje, které mají oddělený V/V prostor a nepoužívají pamětí mapovaný V/V, tento bit nepotřebují.

Zmiňme se ještě, že diskové adresy užívané k ukládání stránek, když není místo v paměti, nejsou součástí tabulky stránek. Důvod je prostý. Tabulka stránek udržuje jen informace, které hardware potřebuje pro překlad virtuální adresy na fyzickou. Informace, které potřebuje operační systém k obsloužení výpadku stránky, jsou uchovávány v softwarových tabulkách v rámci operačního systému.

4.3.3 Překlad s nahlédnutím do bufferu (TLB - Translation Lookaside Buffers)

Ve většině stránkovacích mechanismů jsou tabulky stránek uchovávané v paměti z důvodů jejich velkých rozměrů. Tento návrh má obrovský vliv na výkonnost. Uvažme například instrukci, která kopíruje obsah jednoho registru do druhého. V případě chybějícího stránkování, vykoná tato instrukce jen jeden odkaz do paměti, konkrétně pro načtení instrukce. V případě stránkování jsou potřeba další odkazy do paměti pro přístup k tabulce stránek. Protože rychlost provádění je obecně limitovaná rychlostí, kterou může CPU získávat instrukce z paměti, tak dva odkazy do paměti navíc sníží výkon o 2/3. Za těchto podmínek by se toto nikdy nedalo používat.

Návrháři počítačů o tomto problému věděli již léta a přišli s řešením. Jejich návrh vychází z pozorování, že většina programů se snaží dělat velký počet odkazů do malé části stránek, a ne naopak. Tak je tedy malý zlomek záznamů tabulek stránek často čten a zbytek se téměř vůbec nepoužívá.

Řešení, které navrhli, spočívá ve vybavení počítačů malým hardwarovým zařízením, umožňujícím mapování virtuálních adres na fyzické adresy, bez potřeby procházet tabulku stránek. Zařízení, nazývané TLB (Translation Lookaside Buffers - česky bychom mohli volně přeložit jako překlad s nahlédnutím do bufferu), někdy také nazývané plně asociativní paměť (associative memory), je ilustrována v tabulce 1. Obvykle se nachází uvnitř MMU a skládá se z malého počtu záznamů, v našem příkladě 8, ale obvykle několik set. Každý záznam obsahuje informace o jedné stránce, konkrétně obsahuje číslo virtuální stránky, bit modifikace, který je nastaven při každé změně, ochranný kód (práva čtení, zápisu a provádění), a fyzický rámec stránky, ve kterém se stránka nachází. Tyto záznamy odpovídají jedna ku jedné záznamům v tabulce stránek. Další bit indikuje, zda-li je položka platná (tj. je používána) nebo ne.

Tabulka 1: Využití TLB k urychlení stránkování

Platnost	**Virt. stránka**	**Modifikace**	**Ochrana**	**Rámec str.**
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

Příklad, který může vygenerovat TLB, je v tabulce 1. Jde proces ve smyčce, který uchopil (spans) virtuální stránky 19, 20 a 21, takže záznamy

v TLB mají nastavené ochranné kódy pro čtení a vykonání *execution*. Hlavní používaná data jsou (řekněme zpracovávaný řetězec) na stránkách 129 a 130. Stránka 140 obsahuje rejstříky, používané během výpočtů nad řetězci. Na konci se nachází zásobník na stránkách 860 a 861.

Podívejme se tedy jak TLB funguje. Potom, co je virtuální adresa předána MMU k překladu, se hardware podívá, zda-li se číslo virtuální stránky nevyskytuje v TLB, porovnávajíc všechny záznamy paralelně. Pokud se najde platný záznam a přístup neodporuje ochranným bitům, je rámec stránky převzat přímo z TLB, bez přístupu do tabulky stránek. Vyskytuje-li se číslo virtuální stránky v TLB, ale instrukce se snaží zapsat do stránky označené jen pro čtení, tak je generována chyba ochrany (protection fault), stejně jako by se tomu dělo v tabulce stránek samotné.

Zajímavý případ nastává, když se číslo virtuální stránky nevyskytuje v TLB. Jednotka MMU detekuje selhání a provede obvyklé prohledání tabulky stránek. Poté odstraní jeden záznam z TLB a nahradí ho položkou z tabulky stránek právě nalezené. Proto, jeli stránka po krátké chvíli znova použita, vyhledávání se podruhé již neselže, ale uspěje. Když je záznam vyčištěn (purged) z TLB, je i bit modifikace zkopírován zpět do tabulky stránek v paměti. Ostatní hodnoty tam jsou již přítomny. Když je TLB nahráno z tabulky stránek, berou se všechny hodnoty z paměti.

Programová správa TLB (Software TLB Management)

Dosud jsme předpokládali, že každý stroj se stránkovanou virtuální pamětí má tabulky stránek zkoumané hardwarem a navíc má TLB. V tomto návrhu je veškerá správa TLB a odchytávání výpadku TLB řešeno kompletně pomocí hardware MMU. Odchytávání událostí operačním systémem se děje, jen pokud se stránka nevyskytuje v paměti.

V minulosti byl tento předpoklad správný. Nicméně u některých moderních RISC strojů, včetně MIPS, Alpha a HP PA, se děje téměř veškerá správa stránek programově. Na těchto strojích jsou položky TLB explicitně nahrávány operačním systémem. Dojde-li k výpadku TLB, tak místo toho, aby MMU hledala a vybrala potřebný odkaz stránky, generuje se pouze výpadek TLB a problém je předán režii operačního systému. Systém musí najít stránku, odstranit záznam z TLB, vložit nový a restartovat instrukci, která selhala. A to vše se samozřejmě děje u mnoha instrukcí, protože výpadek TLB se stává častěji, než výpadek stránky.

Může být překvapivé, že pokud máme TLB dostatečně velké (řekněme 64 položek), tak abychom snížili frekvenci výpadků, tak se programová správa TLB jeví celkem efektivní. Hlavní výhodou je mnohem jednodušší MMU, přičemž se uvolní celkem velká plocha na čipu CPU, která se dá využít pro cache nebo jiné funkce, které mohou zvýšit výkon. Programová správa TLB je podrobně probrána v literatuře.

Byly vyvinuty rozličné strategie pro vylepšení výkonu na strojích, které používají programovou správu TLB. Jedním z přístupů je pokus snížit počet

selhání TLB a zároveň snížit cenu selhání TLB, když k ní dojde. Abychom snížili počet selhání TLB, můžeme využít operační systém, aby se snažil předpovědět, kterou stránku bude potřebovat příště a v předstihu tak nahrál záznam do TLB. Například, pokusí-li se klient o RPC na proces serveru stejného stroje, je velmi pravděpodobné, že se za chvíli rozběhne i server. Při této znalosti může systém v průběhu zpracování odchytávání RPC, zkontrolovat, kde se nachází kód serveru, stránky dat a zásobníku a namapovat je před tím, než mohou způsobit výpadek TLB.

Normální postup, jak zpracovat výpadek TLB, ať od hardware nebo software, je, jít do tabulky stránek a provést prohledávací operace, které najdou odkazovanou stránku. Problém spojený s takovýmto vyhledáváním programem je, že stránky které udržují tabulku stránek, nemusí být v TLB, což povede k dalším TLB výpadkům během zpracování. Tyto výpadky mohou být redukovány udržováním si velké (např. 4 kB) programové vyrovnávací paměti TLB záznamů na pevně dané pozici, jejíž stránka je vždy uchovávána v TLB. Prvořadou kontrolou programové vyrovnávací paměti může operační systém podstatně snížit počet výpadků TLB.

4.3.4 Převrácené tabulky stránek (Inverted Page Tables)

Obvyklé tabulky stránek, tak jak jsme je dosud popisovali, měli jeden záznam na každou virtuální adresu, poněvadž jsou seřazeny podle čísla virtuální stránky. Skládá-li se adresní prostor z 2^{32} bytů a má-li 4096 bytů na stránku, pak je třeba přes 1 milión záznamů v tabulce stránek. Jako holé minimum je potřeba minimálně 4 MB pro tabulku stránek. Na větších systémech se dá tato velikost ještě tolerovat.

Ovšem čím více se stávají 64 bitové systémy běžnější, situace se drasticky mění. Máme-li adresní prostor velikost 2^{64} bytů s 4 kB stránkami, pak potřebujeme více než 10^{15} bytů pro tabulku stránek. Vynutit si více než 1 milión GB jen pro tabulku stránek, je už neakceptovatelné, jak dnes, tak i v příštích desetiletích. V důsledku toho je třeba najít řešení pro 64 bitové stránkované virtuální adresní prostory.

Jedno řešení se nazývá **převrácená tabulka stránek** (inverted page table). V tomto návrhu existuje jen jeden záznam na každý rámec skutečné paměti, na rozdíl od jednoho záznamu pro každý virtuální adresní prostor. Takže například s 64 bitovým adresním rozsahem, 4 kB stránkami a 32 MB RAM má převrácená tabulka stránek jen 8192 položek. Položky uchovávají informace o tom, která virtuální stránka nebo proces je umístěna ve stránkovém rámci.

Přestože převrácené tabulky stránek ušetří obrovské množství paměti, tak v případě, kdy je virtuální adresní prostor o hodně větší, než fyzická paměť, mají vážnou vadu: překlad virtuální na fyzickou adresu je o dost těžší. Když proces n odkazuje na virtuální stránku p, tak hardware již nemůže nalézt fyzickou stránku za pomoci p jako ukazatele do tabulky stránek. Místo

toho musí prohledat celou převrácenou tabulku stránek na záznam (n, p). Ale co víc, toto hledání se musí provést při každém odkazu do paměti, a ne pouze při výpadku stránky. Prohledávat 8 kB tabulku při každém odkazu do paměti, určitě není cestou, jak si urychlit počítač.

Únikovou cestou z tohoto dilema je použít TLB. Pokud je TLB schopno udržet všechny často používané stránky, může se překlad dít stejně rychle jako s normální tabulkou stránek. Při výpadku TLB se ovšem musí prohledat celá převrácena tabulka stránek, a toto prohledávání se musí dít přiměřeně rychle.

Převrácené tabulky stránek se nyní používají na většině počítačů.

4.4 Algoritmy výměny stránky

Když se objeví chyba stránky, operační systém musí vybrat stránku k vyjmutí z paměti, aby vytvořil prostor pro stránku, která musí být vložena. Jestliže tato stránka k vyjmutí byla během pobytu v paměti upravena, musí být přepsána na disku, aby byla disková kopie udržována aktuální. Jestliže stránka nebyla pozměněna (například stránka obsahuje kód programu), disková kopie je tedy aktuální a žádný přepis není nutný. Načítaná stránka pouze přepisuje stránku odkládanou.

Přestože je možné vybrat náhodnou stránku k náhradě každé chybné stránky, výkon systému je daleko lepší, je-li vybrána stránka nepříliš používaná. Jestliže je vyjmuta hodně používaná stránka, bude pravděpodobně zapotřebí vrátit ji rychle zpět, jinak dojde k vysokému přetěžování (extra overhead). Bylo uděláno mnoho prací na téma algoritmů náhrady stránky, teoretických i experimentálních. Níže si popíšeme některé z nejdůležitějších.

4.4.1 Optimální algoritmus náhrady stránky

Nejlepší možný algoritmus náhrady stránky je lehké popsat, ale nemožné naimplementovat. Funguje to následovně. Ve chvíli, kdy se objeví chyba stránky, je v paměti určitý soubor stránek. Jedna z těchto stránek bude odkazována na nejbližší instrukci (stránku obsahující tuto instrukci). Ostatní stránky mohou být odkazovány až o 10, 100, nebo možná 1000 instrukcí později. Každá stránka může být označena počtem instrukcí, které budou vykonány, než bude stránka poprvé odkazována.

Optimální stránkový algoritmus říká, že stránka s nejvyšším označením by měla být vyjmuta. Jestliže jedna stránka nebude použita během 8 miliónů instrukcí a jiná nebude použita během 6 miliónů instrukcí, vyjmutí té první posune chybu stránky, která se vrátí zpět za co nejdelší dobu. Počítače se snaží, stejně jako lidé, odkládat nepříjemné věci tak dlouho, jak jen to jde.

Jediným problémem tohoto algoritmu je jeho nerealizovatelnost. V době, kdy dojde k chybě stránky, nemá operační systém šanci vědět, kdy bude každá stránka příště odkazována. Podobnou situaci jsme zde měli i u al-

goritmu nejkratší dávka jako první - jak může systém říct, která dávka je nejkratší? Stálým udržováním si záznamů o všech stránkových odkazech za běhu programu na simulátoru, je možné naimplementovat optimální náhradu stránky při druhém běhu. K tomu použijeme informace získané o stránkových odkazech během prvního spuštění.

Takto je možné porovnat výkon realizovatelných algoritmů s tím nejlepším. Jestliže operační systém docílí výkonu dejme tomu pouze o 1 procento horšího, než u optimálního algoritmu, pak úsilí vynaložené na nalezení lepšího algoritmu přinese maximálně jednoprocentní zlepšení.

K vyhnutí se jakémukoliv možnému nedorozumění bychom si měli vyjasnit, že tento záznam o stránkových odkazech odkazuje pouze na jeden právě aktuální program. Algoritmus náhrady stránky z toho odvozený je tak specifický právě pro ten jeden program. Ačkoliv je tato metoda užitečná k vytvoření algoritmu náhrady stránky, v praktických systémech se nepoužívá. Dále se budeme zabývat algoritmy užitečnými v reálných systémech.

4.4.2 Náhrada dříve nepoužité stránky (Not Recently Used - NRU)

Abychom umožnili operačnímu systému sbírat užitečné statistiky o tom, které stránky jsou používány a které ne, je většina počítačů s virtuální pamětí vybavena dvěma stavovými bity, asociovanými s každou stránkou. R je nastaveno, kdykoliv je stránka odkazována (čtena nebo zapisovana). M je nastaveno, když je stránka upravována. Bity jsou obsaženy v každé položce tabulky stránky, jak je vidět na obrázku 11. Je velmi důležité si uvědomit, že tyto bity musí být aktualizovány při každém odkazu do paměti, takže je nezbytné, aby byly nastavovány hardwarově. Jakmile je jednou bit nastaven na 1, zůstává 1, dokud ho operační systém v programu nevynuluje.

Pokud hardware nedisponuje těmito bity, mohou být simulovány následovně. Když je proces zahájen, všechny položky stránkové tabulky jsou označeny jako mimopaměťové. Jakmile je některá stránka odkazována, objeví se chyba stránky. Operační systém poté nastaví R bit (ve svých interních tabulkách), změní položku tabulky stránky, aby ukazovala na správnou stránku s módem READ ONLY a znovu spustí instrukci. Jestliže je stránka následně modifikována, objeví se další chyba stránky a je operačnímu systému umožněno nastavit M bit a změnit mód stránky na READ/WRITE.

R a M bity mohou být následovně použity k vytvoření jednoduchého stránkovacího algoritmu. Když je proces zahájen, oba stránkové bity pro všechny jeho stránky jsou operačním systémem nastaveny na 0. Pravidelně (např. na každé hodinové přerušení), je R bit vymazán, aby se odlišily stránky, které dosud nebyly odkazovány od těch, které již byly.

Pokud se objeví chyba stránky, operační systém projde všechny stránky a rozdělí je do čtyř kategorií, založených na momentálních hodnotách jejich R a M bitů:

- Třída 0: neodkazovány, nepozměněny
- Třída 1: neodkazovány, pozměněny
- Třída 2: odkazovány, nepozměněny
- Třída 3: odkazovány, pozměněny

Ačkoliv se na první pohled zdají stránky třídy 1 nemožné, objeví se, když stránka třídy 3 má svůj R bit vymazán hodinovým přerušením. Hodinová přerušení nevymazávají M bit, jelikož tato informace je potřebná k tomu, abychom věděli, jestli stránka musí být přepsána na disk či nikoliv.

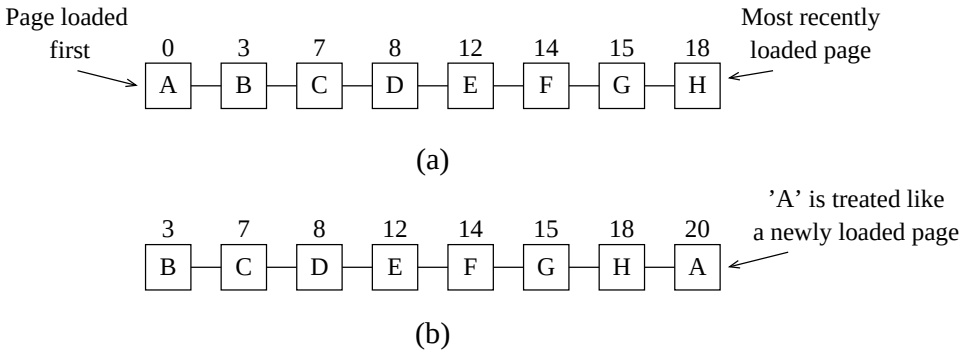
NRU algoritmus vyjímá stránku náhodně z neprázdné třídy s nejnižším číslem. Je lepší vyjmout pozměněnou stránku, která nebyla odkazována v nejméně jednom hodinovém cyklu (typicky 20 milisekund), nežli hojně používanou prázdnou stránku. Hlavním kladem NRU je jeho snadná pochopitelnost, efektivní implementace a výkon, který jistě není optimální, ale často dostatečný.

4.4.3 FIFO (First-In, First-Out) algoritmus náhrady stránky

Jiný nenáročný stránkovací algoritmus je FIFO algoritmus. K objasnění, jak pracuje, si představte supermarket, kde mají tolik regálů, aby mohli vystavit přesně k druhů různého zboží. Jednoho dne nějaká společnost představí novou předpřipravenou potravinu - instantní, sušený, zmrazený, přírodní jogurt, který se dá udělat v mikrovlnné troubě. Výrobek má okamžitý úspěch, takže náš supermarket se musí zbavit nějakého starého zboží, aby jej mohl umístit.

Jedna možnost je, najít zboží, které supermarket prodává nejdéle (např. něco, co začali prodávat před 120 lety) a zbavit se jej na základě toho, že již o něj nikdo nejeví zájem. V podstatě si supermarket udržuje provázaný seznam všech momentálně prodávaných produktů v pořadí, v jakém byly do obchodu uvedeny. Nový produkt jde na konec tohoto seznamu; nejvýše umístěný na seznamu je vyřazen.

U algoritmu náhrady stránky je použitelná stejná myšlenka. Operační systém si udržuje seznam všech stránek, které se momentálně nacházejí v paměti. V čele seznamu je nejstarší stránka, na jeho konci je pak stránka, která byla umístěna do paměti poslední. Při chybě stránky je vyjmuta stránka z čela seznamu a na konec seznamu je přidána nová stránka. Když opět použijeme srovnání s obchodem, FIFO může odstranit vosk na knírek, ale stejně tak může odstranit mouku, sůl nebo máslo. U počítačů se objevuje stejný problém. Z tohoto důvodu se FIFO ve své jednoduché podobě používá jen zřídka.



Obrázek 12: Princip druhé šance. (a) Stránky v pořadí FIFO. (b) Pokud v čase 20 dojde k výpadku stránky, je A přesunuta nakonec seznamu a vynulován R bit.

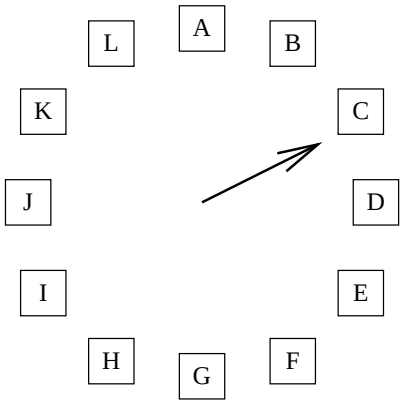
4.4.4 Náhrady stránky s druhou šancí (The Second Chance)

Jednoduchá modifikace FIFO, která umožní vyhnutí se problému s vyřazováním často používaných stránek, je ošetření R bitu nejstarší stránky. Jestliže je 0, stránka je stará a nepoužívaná, takže je okamžitě odstraněna. Pakliže je R bit 1, je tento bit vymazán a stránka je umístěna na konec seznamu. Čas jejího umístění do seznamu je aktualizován tak, jako by tato stránka byla umístěna do paměti teprve nyní. Poté vyhledávání pokračuje.

Postup tohoto algoritmu, nazvaného **druhá šance** (second chance), je ukázán na obrázku 12. Na obrázku 12(a) vidíme stránky A až H umístěné v provázaném seznamu a setříděné podle času umístění do paměti.

Předpokládejme, že se chyba stránky objeví v čase 20. Nejstarší je stránka A, která byla umístěna v čase 0, když proces začal. Jestliže A má R bit vymazán, je z paměti vyjmuta buď přepsáním na disk nebo jen prostě opuštěním. Na druhou stranu je-li R bit nastaven, A je umístěna na konec seznamu a její čas umístění je seřízen na současný čas. R bit je vymazán. Hledání vhodné stránky pokračuje s B.

Princip druhé šance je hledání staré stránky, která nebyla odkazována v předchozím hodinovém intervalu. Jestliže byly odkazovány všechny stránky, druhá šance degeneruje v čistou metodu FIFO. Specificky, představte si, že všechny stránky na obrázku 12(a) mají své R bity nastaveny. Operační systém přemisťuje stránky jednu po druhé na konec seznamu, a když je na konci seznamu připojí, vymaže R bit. Nakonec opět dojdeme ke stránce A, která již má nyní svůj R bit vymazán a je tedy odstraněna. Takto je jisté, že algoritmus vždy skončí.



When a page fault occurs, the page the hand is pointing to is inpected.
The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Obrázek 13: Hodinový algoritmus výměny stránek.

4.4.5 Hodinový algoritmus náhrady stránky

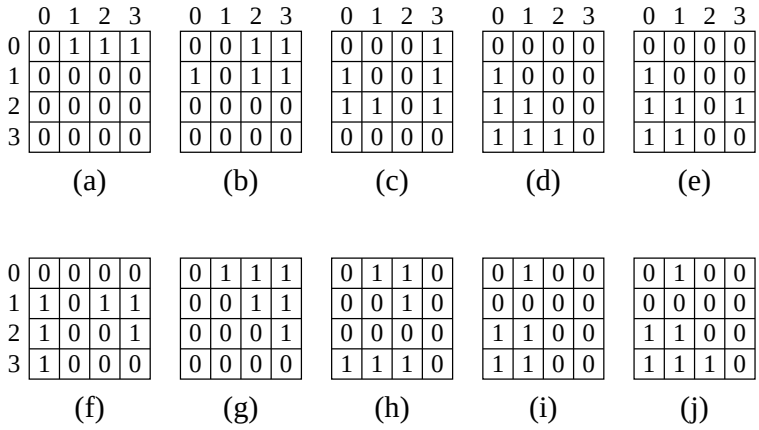
Ačkoliv je druhá šance rozumný algoritmus, je zbytečně neefektivní, jelikož neustále přesouvá stránky seznamem. Lepší přístup je udržovat všechny stránky v kruhovém seznamu ve tvaru hodin, jak je ukázáno na obrázku 13. Ručička ukazuje na nejstarší stránku.

Když se objeví chyba stránky, je vybrána stránka, na kterou ukazuje ručička. Jestliže je její R bit 0, stránka je odstraněna a na její místo je vložena stránka nová. Ručička se posune o jednu pozici. Pakliže je R bit 1, je vynulován a ručička se posune na další stránku. Tento proces se opakuje, dokud není nalezena stránka s bitem $R = 0$. Ne příliš překvapivě je tento algoritmus nazýván hodinový. Od druhé šance se liší pouze implementací.

4.4.6 Náhrada nejdéle nepoužívané stránky (LRU - The Least Recently Used)

Největší přiblížení k optimálnímu algoritmu vychází z pozorování, že stránky hojně využívané v několika posledních instrukcích budou pravděpodobně hojně využívány i v několika příštích. Naopak stránky již dlouho nepoužívané, zůstanou nejspíše nepoužité dlouho i nadále. Tato myšlenka vyústila v realizovatelný algoritmus: když se objeví chyba, vyžadí se stránka, která nebyla používaná nejdelší čas. Tato strategie se nazývá LRU stránkování.

Ačkoliv je LRU teoreticky realizovatelný, není zas až tak snadný. Ke kompletní implementaci LRU je nezbytné udržovat provázaný seznam všech stránek v paměti s nejdávněji používanou stránkou vepředu a s naposledy



Obrázek 14: LRU s použitím matic.

používanou vzadu. Obtížné je, že seznam musí být aktualizován při každém paměťovém odkazu. Vyhledání stránky v seznamu, její smazání a následný přesun do čela seznamu je operace, která spolyká mnoho času, dokonce i hardwarového (za předpokladu, že by takový hardware mohl být vyroben).

Nicméně jsou zde jiné možnosti, jak naimplementovat LRU se použitím speciálního hardware. Uvažujme nejdříve nejjednodušší způsob. Tento vyžaduje vybavení hardwaru 64 bitovým čítačem C, který je automaticky inkrementován po každé instrukci. Dále ještě každá položka tabulky stránky musí mít dostatečně velké pole k pojmutí čítače. Po každém paměťovém odkazu je současná hodnota C uložena v položce tabulky stránky pro právě odkazovanou stránku. Když se objeví chyba stránky, operační systém vyzkouší všechny čítače v tabulce stánky, aby našel ten s nejnižším údajem. Tato stránka je naposledy používaná.

Nyní se podívejme na druhý hardwarový LRU algoritmus. Pro stroj s n stránkovými rámci umí LRU hardware udržovat matici o $n \times n$ bitech, zpočátku všech nulových. Kdykoliv je stránkový rámec k odkazován, hardware nejdříve nastaví všechny bity řádku k na 1, poté nastav všechny bity sloupce k na 0. V každém okamžiku je řádek, jehož binární hodnota je nejmenší, naposledy používaný, řádek s druhou nejmenší binární hodnotou je druhý naposledy používaný a tak dále. Fungování tohoto algoritmu je na obrázku 14 pro čtyři stránkové rámce a stránkové odkazy v pořadí

$$0 - 1 - 2 - 3 - 2 - 1 - 0 - 3 - 2 - 3$$

Poté, co je odkazována stránka 0, máme situaci obrázku 14(a) atd.

4.4.7 Programová simulace LRU

Ačkoliv jsou oba předchozí LRU algoritmy v principu realizovatelné, jen málo strojů, pokud vůbec nějaké, disponuje tímto hardwarem, takže mají pouze malé využití pro návrháře operačních systémů, který vytváří systém pro stroje, které tento hardware nemají. Místo toho je zapotřebí řešení, které může být implementováno programově. Jednou možností je **málo používaný** (Not Frequently Used - NFU) algoritmus. Ten vyžaduje programový čítač asociovaný s každou stránkou na počátku nastavený na 0. Při každém hodinovém přerušení prohlédne operační systém všechny stránky v paměti. Pro každou stránku je R bit, který může být 0 nebo 1, přičten do čítače. V podstatě jsou čítače jakousi snahou o udržování záznamu o tom, jak často byla každá stránka odkazována. Když se objeví chyba stránky, je k výměně vybrána stránka s nejnižším čítačem.

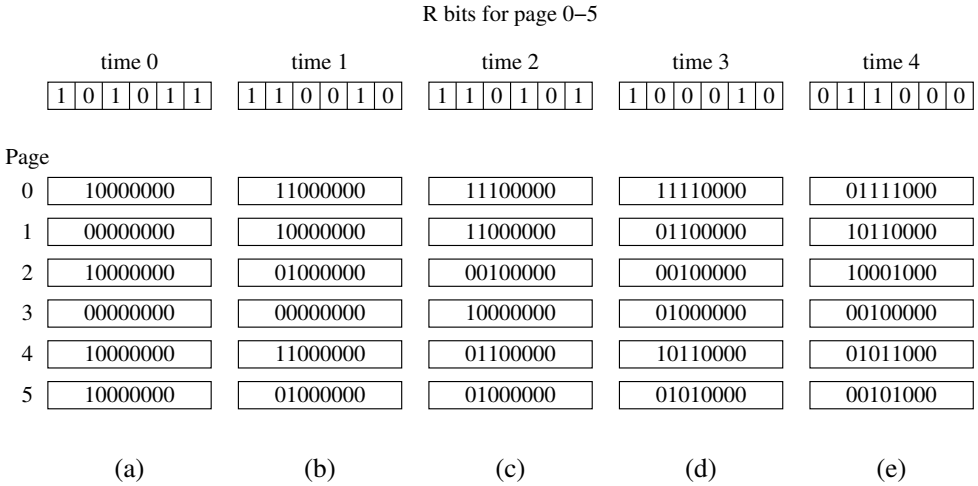
Hlavním problémem NFU je, že nikdy nezapomene všechno. Například u víceprůchodového překladače mohou mít stránky, které byly hojně používané při průchodu 1, stále vysokou hodnotu přenesenou do dalších průchodů. Pokud se stane, že průchod 1 má nejdelší vykonávací čas, stránky obsahující kód pro následující průchody mohou vždy mít nižší čítače než stránky průchodu 1. Následně operační systém odstraní užitečné stránky namísto stránek, které se již nepoužívají.

Naštěstí malá úprava umožní simulovat NFU docela dobře. Úprava má dvě části. V první části jsou všechny čítače posunuty doprava o jeden bit dříve, nežli je přičten R bit. V druhé části je R bit přičten raději do nejlevějšího, než do nejpravějšího bitu.

Obrázek 15 ukazuje, jak upravený algoritmus, známý jako **stárnutí**, pracuje. Předpokládejme, že po prvním hodinovém cyklu mají R bity stránek 0 až 5 hodnoty 1, 0, 1, 0, 1 a 1 (tedy stránka 0 má 1, stránka 1 má R bit 0, stránka 2 má 1 atd.). Jinými slovy, mezi cykly 0 a 1 byly stránky 0, 2, 4 a 5 odkazovány a nastavily své bity na 1, zatímco ostatním zůstaly 0. Poté, co bylo posunuto šest odpovídajících čítačů a R bity byly vloženy nalevo, mají čítače hodnoty ukázané na obrázku 15(a). Čtyři zbývající sloupce ukazují šest čítačů po dalších čtyřech hodinových cyklech.

Když se objeví chyba, je odstraněna stránka, jejíž čítač je nejnižší. Je zřejmé, že stránka, která nebyla odkazována po řekněme čtyři hodinové cykly, bude mít v čítači na začátku čtyři nuly, a tak bude mí nižší hodnotu než čítač, který nebyl odkazován po tři hodinové cykly.

Tento algoritmus se liší od LRU dvěma způsoby. Vezměme stránky 3 a 5 na obrázku 15(e). Žádná nebyla odkazována po dva hodinové cykly, obě stránky byly odkazovány v cyklu dřívějším. Podle LRU, pokud stránka musí být vyměněna, měli bychom vybrat jednu z těchto dvou. Problém je, že nevíme, která z těchto dvou byla odkazována poslední v intervalu mezi cyklem 1 a cyklem 2. Zaznamenáváním pouze jednoho bitu pro časový interval jsme ztratili možnost rozlišit dřívější odkazy v časovém intervalu od těch pozděj-



Obrázek 15: Programová simulace LRU. Šest stránek v pěti hodinových cyklech (a) až (e).

ších. Jediné, co můžeme udělat, je odstranit stránku 3, protože stránka 5 byla odkazová také o dva cykly dříve, zatímco stránka 3 ne.

Druhým rozdílem mezi LRU a stárnutí je, že při stárnutí mají čítače konečný počet bitů, v tomto případě 8. Předpokládejme, že dvě stránky mají obě hodnotu čítače 0. Jediné, co můžeme dělat, je vybrat jednu z nich náhodně. Ve skutečnosti to může být tak, že jedna z těchto stránek byla naposledy odkazována o 9 cyklů dříve a druhá o 1000 cyklů dříve. To nemůžeme vědět. Obecně je však v praxi 8 bitů dostačujících, jestliže je hodinový cyklus okolo 20 milisekund. Jestliže stránka nebyla odkazována 160 milisekund, pravděpodobně není tak důležitá.

4.5 Stránkovací systémy a problémy navrhování (design issues)

V předešlých sekcích jsme vysvětlili, jak stránkování funguje a uvedli jsme několik základních algoritmů náhrady stránky. Ale znalost prostých mechanismů není dostatečná. K navrhování systému toho musíte znát mnohem více, aby fungoval správně. Je to podobný rozdíl jako mezi tím vědět, jak pohybovat s pěšcem, králem a dalšími figurkami ze šachů a být dobrým šachistou. V následujících sekcích se podíváme na další oblasti, které musí návrháři operačních systémů pečlivě zvažovat, aby dostali dobrý výkon ze stránkovacího systému.

4.5.1 Model pracovní sady

V nejčistší podobě stránkování jsou procesy zahájeny bez stránek v paměti. Jakmile CPU zkouší vykonat první instrukci, objeví se chyba stránky způso-

bující, že operační systém předkládá stránku obsahující první instrukci. Další chyby stránky pro globální proměnné a zásobník většinou rychle následují. Po chvíli má proces většinu z potřebných stránek a začne běžet s relativně málo chybami stránek. Tato strategie se nazývá **stránkování na požádání** (demand paging), protože stránky jsou načteny pouze na požádání, nikoliv předem.

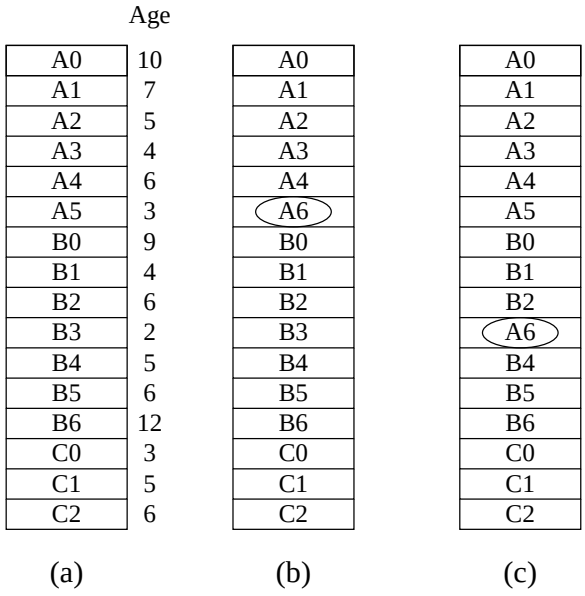
Samozřejmě je celkem lehké napsat testovací program, který systematicky čte všechny stránky ve velkém adresovém prostoru a způsobuje tak mnoho chyb stránek, že není dostatek paměti k uchování všech. Naštěstí takto většina procesů nepracuje. Vystavují **prostor odkazů** ve smyslu toho, že během jakékoliv fáze vykonávání proces odkazuje pouze relativně malý zlomek svých stránek. Například každý průchod víceprůchodového překladače odkazuje pouze zlomek všech stránek.

Soubor stránek, které proces zrovna používá se nazývá **pracovní sada** (working set). Jestliže je celá pracovní sada v paměti, proces poběží bez způsobování mnoha chyb, dokud se nepřesune do jiné vykonávací fáze (např. do dalšího průchodu překladačem). Pokud je dostupná paměť příliš malá na to, aby se v ní dala uchovávat celá pracovní sada, proces bude způsobovat mnoho chyb stránek a poběží pomalu. Vykonání instrukce často zabere několik nanosekund a načtení stránky z disku typicky zabere desítky milisekund. Při tempu jedné nebo dvou instrukcí za 20 milisekund bude trvat věky, než skončí. Program, způsobující chyby stránek vždy po několika instrukcích, způsobuje **zbytečné stránkování** (thrashing).

V systému sdílejícím čas jsou procesy často přesouvány na disk (všechny jejich stránky jsou odloženy z paměti), aby se na řadu v CPU dostaly i další procesy. Vyvstává otázka, co dělat, když se proces dostane opět zpátky. Technicky se nemusí dělat nic. Proces pouze způsobí chyby stránek, dokud nebude načtena celá jeho pracovní sada. Problémem je, že je pomalé, když se vyskytne 20, 50, nebo dokonce 100 chyb stránky pokaždé, když je proces načítán. Taktéž to plýtvá významným časem CPU, jelikož vyřízení chyby stránky vezme operačnímu systému několik milisekund z CPU času.

Proto se většina stránkovacích systémů snaží o udržování záznamů pracovních sad všech procesů a ještě předtím, než spustí proces, se ujistí, že je pracovní sada v paměti. Tato metoda se nazývá **model pracovní sady**. Je navržen k ohromnému snížení míry chyb stránek. Načítání stránek před spuštěním procesu se nazývá **předstránkování**.

K implementování modelu pracovní sady je pro operační systém nezbytné, aby si udržoval záznam o stránkách, které jsou v pracovní sadě. Jednou možností, jak monitorovat tuto informaci, je použití stárnoucího algoritmu probíraného dříve. Jakákoliv stránka obsahující 1 bit mezi vyššími n bity čítače, je považována za člena pracovní sady. Jestliže stránka nebyla odkazována v n nepřetržitých hodinových cyklech, je z pracovní sady vyjmuta. Parametr n musí být určen experimentálně pro každý systém, ale výkon systému většinou není příliš ovlivněn přesnou hodnotou.



Obrázek 16: Lokální versus globální výměna stránek. (a) PŮvodní stav. (b) Výměna stránek lokálně. (c) Vyměna stránek globálně.

Informace o pracovní sadě může být použita ke zlepšení výkonu hodinového algoritmu. Normálně, když ukazuje ručička na stránku, jejíž R bit je 0, je tato stránka vyjmuta. Vylepšením je zkontrolovat, zdali je tato stránka členem pracovní sady daného procesu. Jestliže ano, stránka je nahrazena. Tento algoritmus se nazývá **WSCLOCK** (working set clock model).

4.5.2 Lokální versus globální způsoby alokace

V předešlých sekcích jsme diskutovali několik algoritmů, sloužících k výběru stránky k výměně, když se objeví chyba stránky. Hlavní téma asociované s touto možností (dosud jsme jej pečlivě zametli pod rohožku) je, jak by měla být přidělována paměť mezi soupeřící spustitelné procesy.

Podívejme se na obrázek 16(a). Na něm tvoří sadu spustitelných procesů tři procesy - A, B a C. Předpokládejme, že A získá chybu stránky. Měl by se algoritmus náhrady stránky pokusit najít naposledy používanou stránku zvažujíce pouze šest stránek momentálně přidělených procesu A, nebo by měl zvažovat všechny stránky v paměti? Když bude brát pouze stránky procesu A, stránkou s nejnižším věkem je A5, takže dostáváme situaci z obrázku 16(b).

Na druhou stranu, je-li odstraněna stránka s nejnižší hodnotou věku bez rozdílu příslušnosti k procesu, pak bude vybrána B3 a my dostaneme situaci z obrázku 16(c). Algoritmus z obrázku 16(b) se nazývá **lokální** algoritmus

náhrady stránky, zatím co ten z obrázku 16(c) se nazývá **globální**. Lokální algoritmus přiděluje každému procesu pevnou část paměti. Globální algoritmus dynamicky přiděluje rámce stránek mezi spustitelné procesy. Tak se počet rámců stránek přiřazených každému procesu v čase mění.

Obecně pracují lépe globální algoritmy, obzvláště když se velikost pracovní sady může měnit během života procesu. Jestliže je použit lokální algoritmus a pracovní sada roste, výsledkem bude zbytečné stránkování, dokonce i když bude k dispozici dostatek volných rámců stránek. Když se pracovní sada zmenší, mrhají lokální algoritmy pamětí. Jestliže je použit globální algoritmus, systém musí neustále rozhodovat, kolik rámců přidělit každému procesu. Jednou možností je sledovat velikost pracovní sady, jak ji ukazují stárnoucí bity, ale tato metoda nemusí zamezit zbytečnému stránkování. Pracovní sada může změnit velikost během mikrosekund, zatímco stárnoucí bity jsou primitivní prostředek, rozprostřený přes mnoho hodinových cyklů.

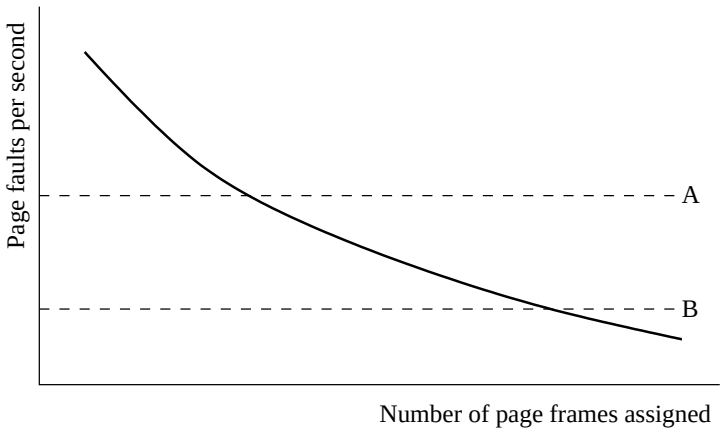
Jiný postup je, mít algoritmus pro přidělování rámců procesům. Jednou možností je pravidelně určovat počet běžících procesů a každému přidělit stejný díl. Tak když máme 475 dostupných rámců a 10 procesů, každý proces dostane 47 rámců. Zbylých 5 rámců jde do uložiště a budou použity, když se objeví chyba stránky.

Ačkoliv se tato metoda zdá přijatelná, nedává příliš smysl přidělovat stejný díl paměti procesu, který má 10 kB, jako procesu, který má 300 kB. Stránky mohou být raději alokovány v poměru k celkové velikosti každého procesu. Takto dostane 300 kB proces 30× větší příděl než 10 kB proces. Je nejspíše rozumné dát každému procesu nějaké minimální množství, takže může běžet bez ohledu na to, jak malý je. Například na některých strojích může jednoduchá instrukce potřebovat i šest stránek, protože samotná instrukce, zdrojový operand a cílový operand mohou ohraničovat hranice stránky. S přidělením pouhých 5 stránek nemůže program, obsahující takovou instrukci nic vykonat.

Ani rovné přidělování, ani metoda proporcionálního přidělování, se nezabývá přímo problémem informování operačního systému. Přímější cesta k jeho kontrole je použití alokačního algoritmu **četnosti chyby** stránky PFF (Page Fault Frequency). Pro velké třídy algoritmů náhrady stránky, obsahující LRU, je známo, že míra chyby klesá, čím více stránek je vyhrazeno. Tato vlastnost je ilustrována na obrázku 17.

Přerušovaná čára A značí míru chyby stránky, která je nepřípustně vysoká, takže je chybovému procesu dáno více rámců ke zmenšení míry chyby. Přerušovaná čára B značí míru chyby stránky tak nízkou, že může být považováno, že má proces k dispozici příliš mnoho paměti. V tomto případě mohou být rámce odebrány. Tak se PFF snaží udržet míru stránkování v přijatelných mezích.

Když se zjistí, že v paměti je tak mnoho procesů, že je nemožné udržet všechny pod hranicí čáry A, tak jsou nějaké procesy z paměti odstraněny a jejich rámce jsou rozděleny mezi zbývající procesy, nebo jsou vloženy do



Obrázek 17: Počet výpadků stránek jako funkce počtu přiřazených stránkových rámců.

úložiště dostupných stránek, které mohou být použity pro následující chyby stránek. Rozhodnutí o odstranění procesu z paměti je formou regulace zátěže (load control). Ukazuje, že u stránkování je ještě stále potřebné i odkládání, i když nyní je vyměňování použito spíše ke snížení potencionálních požadavků na paměť, nežli ke kultivaci jejich bloků k okamžitému použití.

4.5.3 Velikost stránky

Velikost stránky je často parametr, který může být zvolen operačním systémem. Dokonce i když je hardware navržen se stránkami o velikosti například 512 bytů, operační systém může lehce považovat stránky 0 a 1, 2 a 3, 4 a 5 atd. za stránky velikosti 1 kB díky přidělení dvou sdružených rámců stránek velikosti 512 bytů.

Určení optimální velikosti stránky vyžaduje zvážení několika soupeřících faktorů. Pro začátek, náhodně vybraný text, data nebo část zásobníku nezaplní základní počet stránek. V průměru polovina z výsledných stránek bude prázdná. Další prostor na této stránce je vyplýtván. Tomuto plýtvání se říká **interní fragmentace**. S n segmenty v paměti a stránkou velikosti p bajtů bude na interní fragmentaci vyplýtváno $np/2$ bajtů. Tato úvaha argumentuje pro malé velikosti stránek.

Další argument pro malé velikosti stránek je zřejmý, pokud uvažujeme o programu, sestávajícího z osmi souvislých částí, každá velikosti 4 kB. S 32 kB velikostí stránky musí být programu stále přiděleno 32 kB. Se stránkou velikosti 16 kB stačí pouze 16 kB. Se stránkou velikosti 4 kB nebo menší, vyžaduje v každém okamžiku pouze 4 kB. Obecně velká velikost stránek způsobí více nepoužívaných programů v paměti, než malá velikost.

Na druhou stranu malé stránky znamenají, že program bude potřebo-

vat mnoho stránek a proto i velkou tabulku stránek. Program velikosti 32 kB potřebuje pouze čtyři 8 kB stránky, ale 64 stránek 512 bajtových. Při přesunu z disku a na disk jde většina ztraceného času na vrub vyhledání a rotaci, takže přesun malé stránky zabere téměř stejně času, jako přesun velké stránky. Načtení 64 stránek 512 bajtových stránek může zabrat 64×15 ms, ale načtení čtyř 8 kB stránek zabere pouze 4×25 ms.

Na některých strojích musí být tabulka stránek nahrána do hardwarových registrů vždy, když CPU přepne z jednoho procesu na druhý. Na těchto strojích mít malou stránku znamená, že čas, potřebný k nahrání registrů stránky bude větší, čím bude velikost stránky menší. Navíc prostor zabraný tabulkou stránek se bude zvětšovat se snižováním velikosti stránky.

Poslední bod může být analyzován matematicky. Nechť je průměrná velikost procesu s bajtů a velikost stránky p bajtů. Dále předpokládejme, že každá položka stránky potřebuje e bajtů. Přibližný počet stránek potřebných na proces je pak s/p a zabírají se/p bajtů prostoru tabulky stránky. Promrhaná velikost paměti na poslední stránce procesu dle interní fragmentace je $p/2$. Tak je celkový horní odhad (overhead) ztrát tabulky stránky a interní fragmentace dán vztahem

$$overhead = \frac{s \cdot e}{p} + \frac{p}{2}$$

První zlomek (velikost tabulky stránky) je velký, když je velikost stránky malá. Druhý zlomek (interní fragmentace) je velký, když je velikost stránky velká. Optimum musí ležet někde mezi. Když vezmeme první derivaci podle p a položíme ji rovnou nule, dostaneme rovnici

$$\frac{-s \cdot e}{p^2} + 1 = 0$$

Z této rovnice můžeme odvodit vzoreček, který udává optimální velikost stránky (uvažujíce pouze paměť promrhanou fragmentací a velikostí tabulky stránky). Výsledek je:

$$p = \sqrt{2 \cdot s \cdot e}$$

Pro $s = 128$ kB a $e = 8$ bajtů na položku tabulky stránky je optimální velikost stránky 1448 bajtů. Ve skutečnosti by bylo použito velikosti 1 kB nebo 2 kB, to by záleželo na dalších okolnostech (např. na rychlosti disku). Nejvíce obchodně dostupné počítače používají velikosti stránek od 512 bajtů po 64 kB.

4.5.4 Rozhraní virtuální paměti

Až dosud jsme předpokládali, že virtuální paměť je transparentní pro procesy a programátory, že vše co vidíme je velký virtuální adresový prostor na počítači s malou (menší) fyzickou pamětí. Ve většině systémů tomu tak je,

ale v některých vyspělejších systémech mají programátoři jakousi kontrolu nad mapou paměti a mohou ji využít i netradičními směry. Na některé z těchto směrů se stručně podíváme v této sekci.

Jedním z důvodů, proč dát programátorům kontrolu nad paměťovou mapou, je možnost sdílení stejné paměti dvěma nebo více procesy. Jestliže programátoři mohou pojmenovat oblasti paměti, pak může jeden proces případně dát jinému procesu jméno oblasti paměti, takže tento proces si může tuto oblast taktéž namapovat. Se dvěma (nebo více) procesy sdílejícími stejné stránky se stane možnou velká šířka pásma sdílení - jeden proces do sdílené paměti zapíše a ostatní z ní mohou číst.

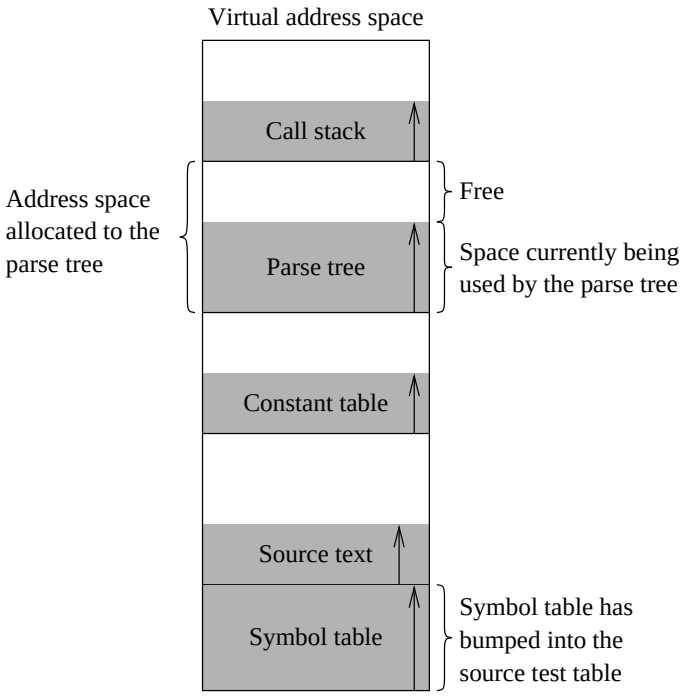
Sdílení stránek může být taktéž použito k implementaci vysokovýkonného předávání zpráv *message-passing* systému. Když jsou normálně zprávy předány, jsou data za výraznou cenu kopírována z jednoho adresového prostoru do jiného. Když mohou procesy kontrolovat svou stránkovou mapu, zpráva může být předána tak, že posílající proces odmapuje stránku či stránky, obsahující zprávu a přijímající proces si ji namapuje. Zde se nemusí kopírovat celá data, stačí zkopírovat pouze jména stránek.

Další vyspělou technikou správy paměti je **distribuovaná sdílená paměť**. Hlavní myšlenkou je umožnit vícenásobným procesům sdílet přes síť sadu stránek, jako jednoduchý sdílený lineární adresový prostor. Když se proces odkazuje na stránku, která není zrovna namapovaná, objeví se chyba stránky. Obsluha chyby stránky, která může být v jádře nebo v prostoru uživatele, poté lokalizuje stroj, vlastnící stránku, a pošle mu zprávu, v níž žádá o odmapování stránky a poslání této stránky do sítě. Když stránka dorazí, je namapována a chybná instrukce je spuštěna znovu.

4.6 Segmentace

Diskutovaná virtuální paměť je jednodimenzionální, neboť virtuální adresy jdou od 0 po danou maximální adresu, pěkně jedna za druhou. Při mnoha problémech by bylo mnohem lepší mít dva nebo více rozdílných virtuálních adresových prostorů nežli pouze jeden. Například překladač má mnoho tabulek, které jsou vytvořeny jako výsledek překladu a nejspíše obsahují:

1. zdrojový text uložený kvůli tištěnému výpisu (u dávkových systémů),
2. tabulku symbolů, obsahující jména a atributy proměnných,
3. tabulku obsahující všechny použité konstanty typu integer a s plovoucí čárkou,
4. rozkladový strom, obsahující syntaktickou analýzu programu,
5. zásobník, použitý pro volání procedur uvnitř překladače.



Obrázek 18: ...

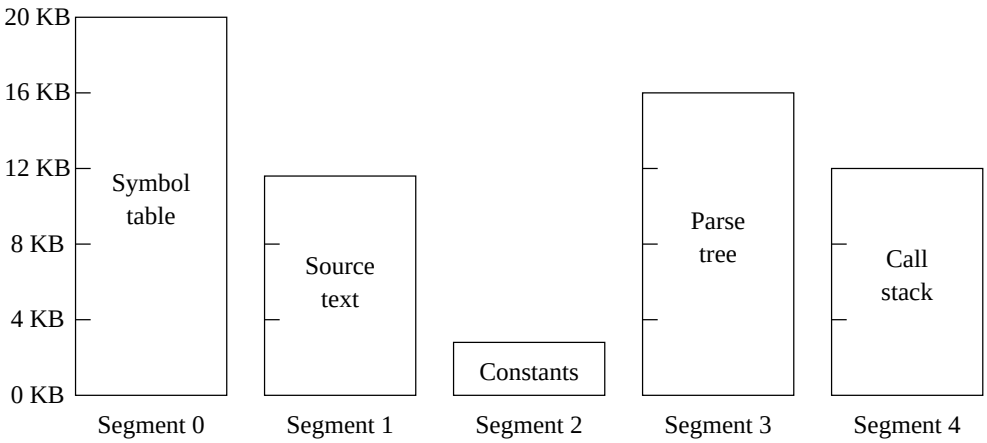
Každá z prvních čtyřech tabulek se vytváří a roste plynule podle toho, jak postupuje překlad. Poslední se během překladu nepředvídatelně zmenšuje a roste. V jednodimenzionální paměti by těchto pět tabulek muselo být alokováno na sousední části prostoru virtuálních adres, jak je tomu na obrázku 18.

Uvažujme, co nastane, má-li program výjimečně vysoký počet proměnných. Část adresového prostoru, přiděleného tabulce symbolů, může být zaplněn, ale mnoho dalšího prostoru může být v ostatních tabulkách. Překladač samozřejmě může jednoduše vydat zprávu, říkající, že překlad nemůže pokračovat z důvodu příliš mnoha proměnných, ale toto se nezdá příliš sportovní, když je nevyužitý prostor v jiných tabulkách.

Jinou možností je hrát si na Robina Hooda a brát prostor tabulkám, které ho mají příliš a dávat jej těm tabulkám, které ho mají nedostatek. Toto promíchání se udělat může, ale je nepříjemnost v nejlepším a otravná a nevděčná práce v nejhorším.

Co je opravdu zapotřebí, je osvobození programátora od nutnosti spravovat zvětšující a zmenšující se tabulky, stejným způsobem virtuální paměť odstraňuje obavy z uspořádání programu do vrstev.

Přímým a extrémně obecným řešením je poskytnout stroj s mnoha absolutně nezávislými adresovými prostory, zvanými **segmenty**. Každý segment



Obrázek 19: Segmentovaná paměť povoluje všem tabulkám zvětšování i zmenšování nezávisle na jiných tabulkách.

se skládá z lineární posloupnosti adres od 0 po dané maximum. Délka každého segmentu může být něco mezi nulou a povoleným maximem. Rozdílné segmenty mohou mít, a většinou mají, rozdílnou délku. Ba co víc, délka segmentu se může měnit během vykonávání. Délka zásobníkového segmentu může vzrůst, kdykoliv je něco uloženo na zásobník a zmenšit se, když je něco ze zásobníku vyjmuto.

Protože každý segment představuje samostatný adresový prostor, mohou různé segmenty růst a zmenšovat se nezávisle, bez ovlivnění jiných. Jestliže zásobník v jistém segmentu potřebuje k růstu více adresového prostoru, může ho mít, protože v jeho adresovém prostoru není nic jiného, co by tomu vadilo. Samozřejmě, segment může být zaplněn, ale většinou jsou segmenty tak velké, že tento jev je velmi řídký. K určení adresy v této segmentované nebo dvojdimenzionální paměti musí program nabízet adresu, tvořenou dvěmi částmi - číslem segmentu a adresou uvnitř segmentu. Obrázek 19 ukazuje segmentovanou paměť použitou pro tabulky překladače, o kterých jsme se bavili dříve.

Zdůrazňujeme, že segment je logická entita, čehož si je programátor vědom a jako logickou entitu jej používá. Segment může obsahovat podprogram, nebo pole, nebo zásobník, nebo soubor skalárních proměnných, ale většinou neobsahuje směsici rozdílných typů.

Segmentovaná paměť má kromě zjednodušení zacházení s datovými strukturami, které rostou, či se zmenšuj, i další výhody. Jestliže každá procedura okupuje samostatný segment s nulou jako počáteční adresou, pak spojení procedur, kompilovaných odděleně, je značně zjednodušeno. Po zkompilování a spojení všech procedur, utvářejících program, zavolá procedura proceduru v segmentu n použitím dvojdílné adresy $(n,0)$ k adresování slova 0

(počáteční bod).

Jestliže je procedura v segmentu n následně pozměněna a znovu zkompilována, nemusí se měnit žádné další procedury (protože nebyla pozměněna počáteční adresa), dokonce ani když je nová verze větší než předchozí. U jednodimenzionální paměti jsou procedury nahuštěné těsně jedna na druhou, bez jakéhokoliv adresového prostoru mezi sebou. Následně změna velikosti jedné procedury může ovlivnit počáteční adresy jiných, nesouvisejících procedur. Toto vyžaduje pozměnění všech procedur, které volají některou z přemístěných procedur, aby se začlenily jejich nové počáteční adresy. Jestliže program obsahuje stovky procedur, může být tento postup velmi nákladný.

Segmentace taktéž usnadňuje sdílení procedur, nebo dat, mezi několika procesy. Běžným příkladem je **sdílená knihovna**. Moderní pracovní stanice, běžící na pokročilých systémech s GUI, mají často extrémně velké grafické knihovny, zakompilované do téměř všech programů. V segmentovaném systému může být grafická knihovna vložena do segmentu a sdílená několikanásobnými procesy, eliminujíce potřebu mít knihovnu v adresovém prostoru každého procesu. V čistě stránkovacích systémech je taktéž možno sdílet knihovny, ale je to mnohem komplikovanější. V podstatě to tyto systémy řeší simulováním segmentace.

Protože každý segment tvoří logickou entitu, se kterou je programátor obeznámen, jako například procedura, nebo pole, nebo zásobník, mohou mít rozdílné segmenty rozdílné druhy ochrany. Procedurální segment může být určen jako pouze vykonávací, chráněný před pokusy o čtení z něj či ukládání do něj. Pole s plovoucí čárkou může být určeno pouze ke čtení či zápisu *(read/write)* bez možnosti vykonávání, což zajistí zachycení pokusů o skok do něj. Taková ochrana je nápomocna při zachytávání programovacích chyb.

Měli bychom se snažit pochopit, proč je ochrana smysluplná u segmentované paměti, ale ne u jednodimenzionální stránkované paměti. U segmentované paměti je uživatel obeznámen s tím, co se v každém segmentu nachází. Normálně segment neobsahuje například proceduru a zásobník, ale buď jedno nebo druhé. Jelikož každý segment obsahuje pouze jeden typ objektu, může mít ochranu vhodnou pro tento konkrétní typ. Stránkování a segmentace jsou porovnány v tabulce 2.

Obsahy stránek jsou v jistém smyslu náhodné. Programátor není obeznámen s faktem, že stránkování zrovna nastává. Ačkoliv uložení několika bitů do každé položky tabulky stránky k určení povolených přístupů je možné, k využití této vlastnosti si musí programátor vést záznam o tom, kde byly v jeho adresovém prostoru hranice stránky. A to je přesně ten typ správy, kvůli které bylo vynalezeno stránkování, aby se potlačil. Protože se uživateli segmentované paměti zdá, že jsou všechny segmenty v hlavní paměti po celou dobu - může je adresovat, jako by byly - a může chránit každý segment samostatně, bez potřeby starat se o správu jejich překrývání. *(overlaying)*.

Tabulka 2: Porovnání stránkování a segmentace.

Vlastnost	**Stránkování**	**Segmentace**
Potřebuje programátor vědět, že se používá příslušná technologie?	Ne	Ano
Kolik lineárních adresních prostorů je k dispozici?	1	Mnoho
Může celkový adresní prostor překročit velikost fyzické paměti?	Ano	Ano
Mohou být oddělena data od kódu a odděleně zabezpečena?	Ne	Ano
Je možné se snadno přizpůsobit blokům proměnné velikosti?	Ne	Ano
Napomáhá technologie sdílení kódu mezi procesy?	Ne	Ano
Proč byla daná technologie vynalezena?	Získat velký adresní prostor bez nutnosti mít více fyzické paměti.	Oddělení adresního prostoru pro data a kód a pro sdílení a ochranu.

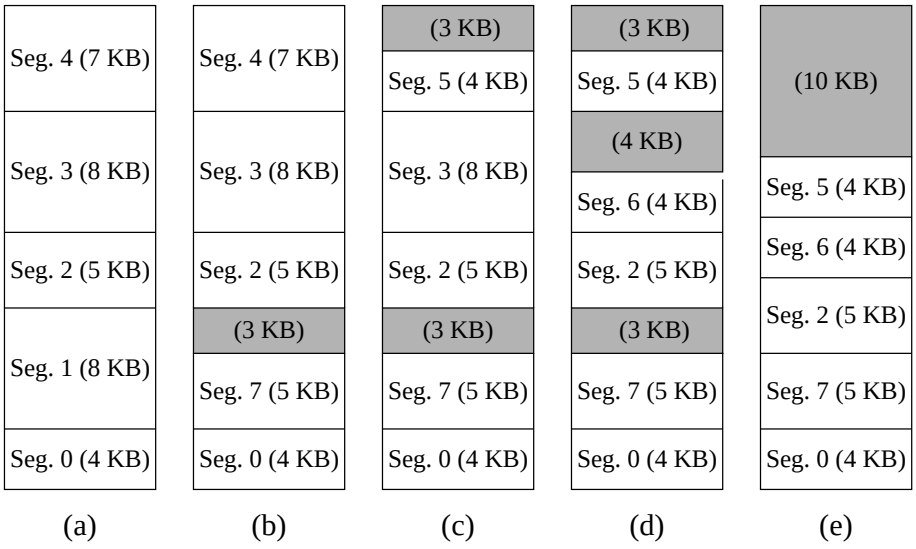
4.6.1 Implementace čisté segmentace

Implementace segmentace se od stránkování liší již v základu: zatímco stránky mají pevnou velikost, segmenty nikoliv. Obrázek 20(a) ukazuje příklad fyzické paměti zpočátku obsahující pět segmentů. Nyní uvažujme, co se stane, když odstraníme segment 1 a na jeho místo umístíme segment 7, který je menší. Dostaneme se k uspořádání paměti tak, jak je tomu na obrázku 20(b). Mezi segmentem 7 a segmentem 1 je nepoužitá oblast - mezera. Poté je jako na obrázku 20(c) nahrazen segment 4 segmentem 5 a segment 3 je nahrazen segmentem 6 dle obrázku 20(d). Po chvíli běhu systému bude paměť rozdělena na mnoho kusů, některé obsahující segmenty a některé mezery. Tento jev, zvaný **šachovnicový** (checkerboarding) nebo **externí fragmentace**, plýtvá pamětí na mezery. To může být vyřešeno komprimací, jak ukazuje obrázek 20(e).

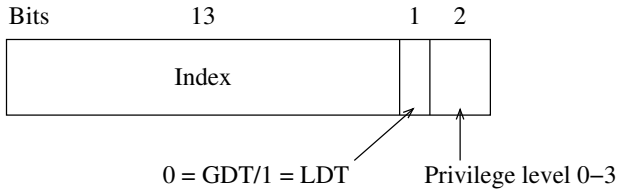
4.6.2 Segmentace se stránkováním: Intel Pentium

Virtuální paměť Pentia (a Pentia Pro) obsahuje segmentaci i stránkování. Pentium má 16 000 nezávislých segmentů, přičemž každý udrží až 1 miliardu 32 bitových slov. Vysoký počet segmentů a jejich velikost je důležitá, jelikož jen málo programů potřebuje více než 1000 segmentů, ale většina potřebuje segmenty, které zvládnou megabajty.

Srdce virtuální paměti Pentia se skládá ze dvou tabulek, **LDT** (Local



Obrázek 20: (a)-(d) Vznik šachovnice. (e) Odstranění šachovnice setřesením.

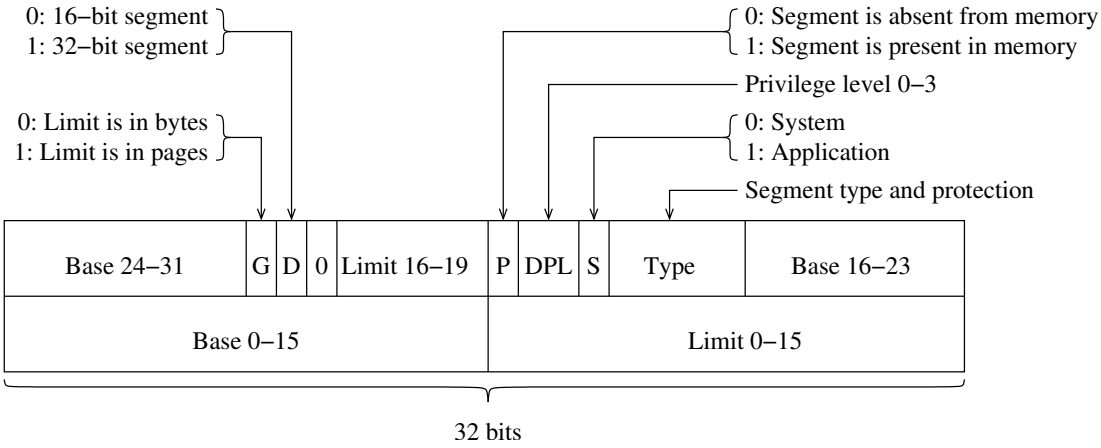


Obrázek 21: Selektor Pentia.

Descriptor Table) a **GDT** (Global Descriptor Table). Každý program má vlastní LDT, ale je pouze jedno GDT, a to sdílejí všechny programy na počítači. LDT popisuje lokální segmenty každého programu, obsahujíce jeho kód, data, zásobník a podobně, zatímco GDT popisuje systémové segmenty, obsahující i vlastní operační systém.

Pro přístup k segmentu nejprve program Pentia načte selektor pro tento segment do jednoho ze šesti segmentových registrů procesoru. Během vykonávání drží CS registr selektor pro kódový segment a registr DS drží selektor pro datový segment. Ostatní segmentové registry jsou méně důležité. Každý selektor je 16-bitové číslo, jak ukazuje obrázek 21.

Jeden z bitů selektoru říká, zda-li se jedná o segment lokální, či globální (tedy je-li v LDT, či GDT). Dalších 13 bitů určuje číslo položky LDT nebo GDT, takže tyto tabulky jsou omezeny na udržování 8 kB segmentových deskriptorů. Zbylé 2 bity souvisejí s ochranou a budeme se jimi zaobírat později. Deskriptor 0 je zakázaný. Může být bezpečně načten do segmentového registru, aby ukázal, že segmentový registr není momentálně dostupný.



Obrázek 22: Deskriptor segmentu Pentia pro kódový segment.

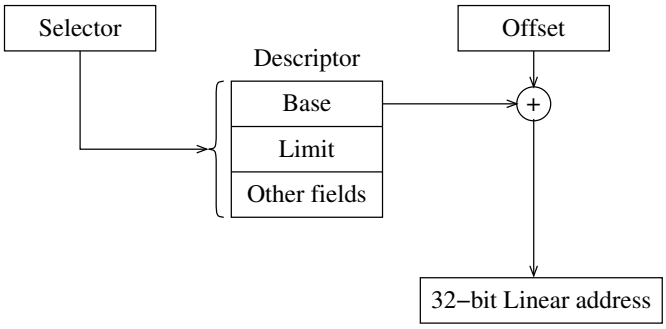
Jestliže je použit, je o tom informován operační systém.

V čase načítání selektoru do segmentového registru je odpovídající deskriptor vyjmut z LDT nebo GDT a uložen v mikroprogramových registrech, takže může být rychle přístupný. Deskriptor se skládá z 8 bajtů, obsahuje základní adresu segmentu, velikost a další informace, jak je popsáno na obrázku 22.

Formát selektoru byl vybrán chytře, aby se dal lehce lokalizovat deskriptor. Nejdříve je dle bitu 2 selektoru vybrána buď LDT nebo GDT, poté je selektor zkopírován do interního SCRATCH registru a 3 jeho nejnižší bity jsou nastaveny na 0. Nakonec je k němu přidána adresa LDT nebo GDT tabulky, aby se získal přímý ukazatel na deskriptor. Například selektor 72 odkazuje na položku 9 v tabulce GDT, která je umístěna na adrese GDT + 72.

Pojďme sledovat kroky, podle kterých je pár (selektor, offset) přeměněn na fyzickou adresu. Hned, jakmile mikroprogram pozná, který segmentový registr je používán, může najít kompletní deskriptor odpovídající tomuto selektoru ve svých interních registrech. Pokud segment neexistuje (selektor 0), nebo je momentálně odložen, objeví se zbytečné stránkování a je vyvoláno přerušení.

Poté se zkontroluje, zda-li je offset před hranicí konce segmentu, ve kterém došlo k přerušení. Logicky by mělo v deskriptoru být 32 bitové pole udávající velikost segmentu, ale k dispozici je pouze 20 bitů, takže je použito jiné schéma. Jestliže granularitní *Gbit* pole je 0, *Limitní* pole je přesně velikosti segmentu, až 1 MB. Jestliže je 1, *Limitní* pole dává velikost segmentu v počtu stránek, nikoliv v bajtech. Velikost stránky Pentia je pevně dána na 4 kB, takže 20 bitů je dostačujících pro segmenty až do velikosti 2^32 bajtů.



Obrázek 23: Převod dvojice selektor:offset na lineární adresu.

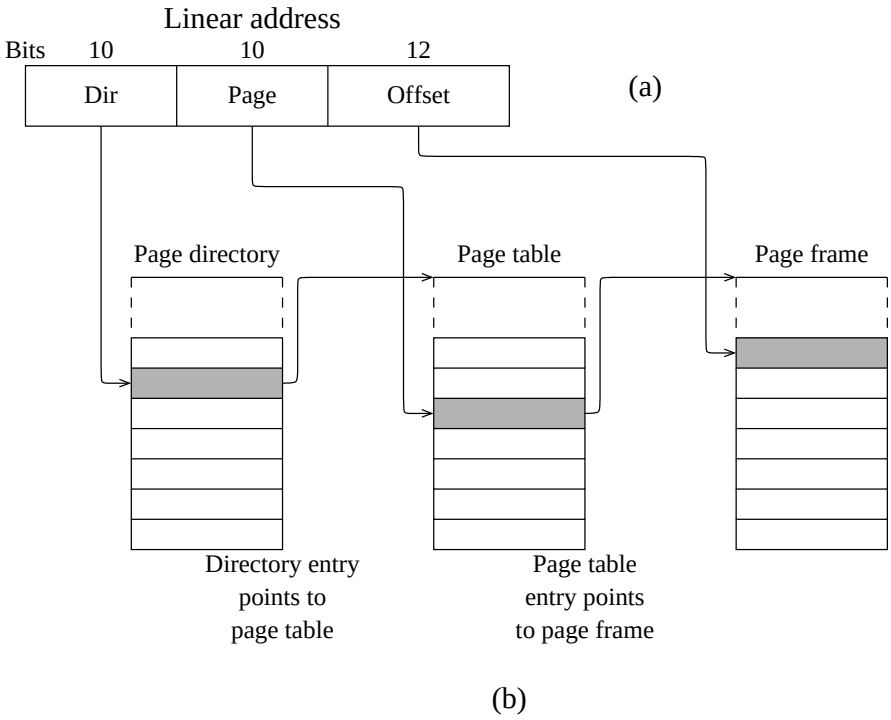
Předpokládejme, že je segment v paměti a offset v dosahu, Pentium poté přidává 32 bitové *základní* pole v deskriptoru k offsetu, aby vytvořilo něco, čemu se říká **lineární adresa**, jak vidno z obrázku 23. *Základní* pole je rozloženo na tři části a rozloženo po celém deskriptoru, aby byla zajištěna kompatibilita s 80286, u které je toto pole pouze 24-bitové. Ve skutečnosti umožňuje *základní* pole každému segmentu začínat na libovolném místě uvnitř 32 bitového lineárního adresového prostoru.

Jestliže je stránkování vypnuto (bitem v globálním kontrolním registru), je lineární adresa pojmuta jakožto fyzická a poslána do paměti ke čtení, či zápisu. Takto s vypnutým stránkováním dostáváme schéma čisté segmentace, se základní adresou každého segmentu poskytovanou v jeho deskriptoru. Segmentům je povoleno překrývání, nejspíše proto, že by bylo příliš obtížné ověřit, že se nepřekrývají, a taktéž by to zabralo mnoho času.

Na druhou stranu, je-li stránkování zapnuto, pak je lineární adresa pojmuta jako virtuální adresa a namapována pomocí tabulek stránek na fyzickou adresu, tak jako v našich dřívějších případech. Jediným opravdovým problémem je, že s 32 bitovou virtuální adresou a 4 kB stránkou může segment obsahovat 1 milion stránek. Takže pro malé segmenty je použito dvojúrovňové mapování k redukci velikosti tabulky stránky.

Každý běžící program má **stránkový adresář** sestávající z 1024 32 bitových položek. Tento adresář je umístěn na adrese, na kterou ukazuje globální registr. Každá položka v tomto adresáři ukazuje na tabulku stránky, která taktéž obsahuje 1024 32 bitových položek. Položky tabulky stránky ukazují na rámce stránek. Schéma je ukázáno na obrázku 24.

Na obrázku 24(a) vidíme lineární adresu rozdělenou do tří polí - *Dir, Page a Off*. Pole *Dir* je použito k indexování stránkového adresáře k umístění ukazatele do správné tabulky stránky. Poté je použito pole *Page* jako index v tabulce stránky k nalezení fyzické adresy rámce stránky. Nakonec je pole *Off* přidáno k adrese rámce stránky, aby se získala fyzická adresa potřebného *byte* nebo *word*.



Obrázek 24: Mapování lineární adresy na fyzickou adresu.

Každá položka tabulky stránky má 32 bitů, 20 z nich obsahuje číslo rámce stránky. Zbylé bity obsahují přístupové a příznakové bity, nastavené v hardware ve prospěch operačního sytému, ochranné bity a další pomocné bity.

Každá tabulka stránky má položky pro 1024 rámce stránek velikost 4 kB, takže jedna tabulka stránky obsluhuje 4 MB. Segment, kratší než 4 MB, bude mít stránkový adresář s jedinou položkou, ukazatelem na tuto jednu jedinou tabulku stránky. Tímto způsobem je režie pro krátké segmenty pouze dvoustránková, na rozdíl od miliónu stránek, které by byly potřeba u jednoúrovňové tabulky stránek.

Aby se netvořily opakované odkazy do paměti, má Pentium malé TLB, které přímo mapuje nejdříve použitou kombinaci *Dir-Page* na fyzickou adresu rámce stránky. Pouze když není aktuální kombinace přítomna v TLB, tak je vykonán mechanismus z obrázku 24(b) a TLB je aktualizován.

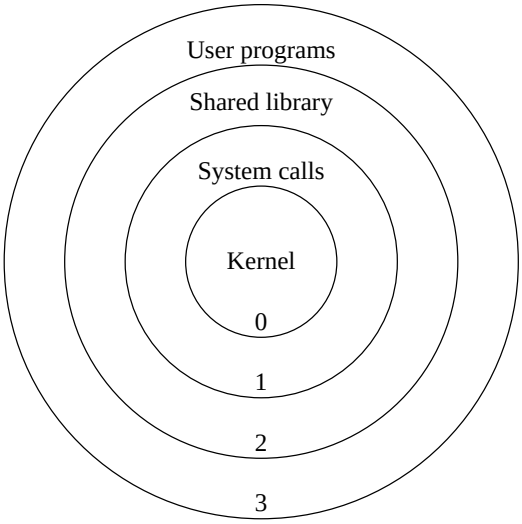
Malé zamyšlení může odhalit fakt, že když je použito stránkování, tak není důvod, aby pole *Base* v deskriptoru bylo nenulové. Vše, co Base dělá je, že způsobuje malé posunutí k použití položky uprostřed stránkového adresáře, namísto na začátku. Skutečný důvod pro zahrnutí *Base* je umožnění čisté (nestránkované) segmentace a kompatibility s 80286, která má stránkování vždy vypnuté (286 má pouze čistou segmentaci, ale ne stránkování).

Taktéž stojí za zmínku, že pokud nějaká aplikace nepotřebuje segmentaci, ale je spokojena s jediným stránkovaným 32 bitovým adresním prostorem, je tento model přijatelný. Všechny segmentové registry mohou být vytvořeny stejným selektorem, jehož deskriptor má $Base = a$ *Limit* nastaven na maximum. Pozice instrukce pak bude lineární adresou, při použití jediného adresového prostoru - tedy ve skutečnosti normální stránkování.

Musíme vzdát hold návrhářům Pentia. Vytyčili si tak protichůdné cíle, jako je implementace čistého stránkování, čisté segmentace a stránkovaných segmentů a zároveň zachování kompatibility s 80286, přičemž vše mělo být děláno efektivně. Výsledek je překvapivě jednoduchý a čistý.

Ačkoliv jsme prošli kompletní architekturu virtuální paměti Pentia, byť jen letmo, stojí za to si říct několik slov o ochraně, neboť toto téma se virtuální paměti blízce dotýká. Pentium podporuje čtyři úrovně ochrany. Úroveň 0 má nejvýsadnější postavení, úroveň 3 postavení nejméně výsadní. Toto je ukázáno na obrázku 25. V každém okamžiku je běžící program v určité úrovni, označené 2-bitovým polem ve svém deskriptoru. Taktéž každý segment v systému má svou úroveň.

Dokud se program omezuje na používání segmentů ze stejné úrovně, všechno pracuje skvěle. Pokus o přístup k datům vyšší úrovně je povolen. Pokus o přístup k datům nižší úrovně je nedovolený a způsobuje přerušní. Pokusy o volání procedur odlišných úrovní (vyšší nebo nižší) jsou dovoleny, ale pouze velmi opatrně a kontrolovaně. K uskutečnění meziúrovňového volání musí instrukce `CALL` obsahovat místo adresy selektor. Tento selektor určuje deskriptor zvaný **brána volání** (call gate), který dává adresu procedury



Obrázek 25: Úrovně ochrany Pentia.

k volání. Tak není možné skočit doprostřed libovolného kódového segmentu v rozdílné úrovni. Mohou být použity pouze oficiální vstupní body. Myšlenka ochranných úrovní a bran volání byla poprvé použita v MULTICS, kde byla představena jako **ochranné kruhy**.

Typické použití těchto mechanismů je ukázáno na obrázku 25. V úrovni 0 najdeme jádro operačního systému *(kernel)*, které obsluhuje vstup a výstup, správu paměti a další kritické záležitosti. V úrovni 1 se nachází obsluha volání systému. Uživatelské programy zde mohou volat procedury, aby byly vykonány systémová volání, ale volán může být pouze určitý a chráněný seznam procedur. Úroveň 2 obsahuje knihovní procedury, možná sdílené mezi mnoha běžícími programy. Uživatelské programy mohou tyto procedury volat a číst jejich data, ale nemohou je pozměňovat. Konečně uživatelské programy běží v úrovni 3, která má nejmenší ochranu.

Přerušení používají mechanismy podobné branám volání. Ony taktéž odkazují deskriptory spíše, než absolutní adresy a tyto deskriptory ukazují na určité procedury k vykonání. Pole *Type* z obrázku 22 rozlišuje mezi kódovými segmenty, datovými segmenty a různými typy bran.