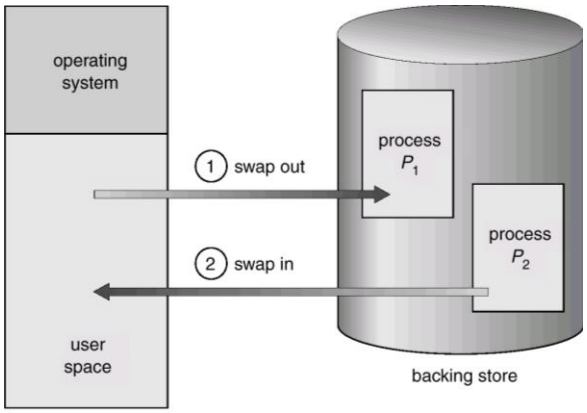
Geheugenbeheer

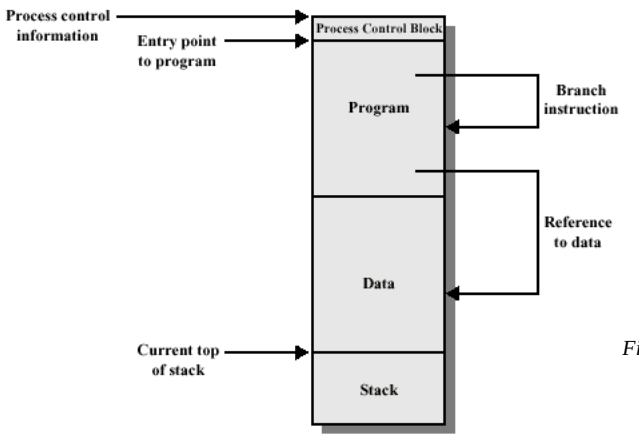
4.1 Vereisten voor geheugenbeheer

Geheugenbeheer is één van de belangrijkste en meest complexe taken van een besturingssysteem. Hierbij worden een aantal problemen gesteld waarvoor hardwareondersteuning absoluut vereist is. Het hoofdgeheugen wordt behandeld als een voorziening, die moet toegewezen worden aan een aantal actieve processen. Het geheugenbeheer staat ondermeer in voor het verplaatsen van informatie tussen het hoofdgeheugen en het secundaire geheugen. Het binnenhalen van programma's in het hoofdgeheugen, zodat de processor ze kan uitvoeren, is hierbij de belangrijkste bewerking. In eerste instantie (tot §4.4) veronderstellen we nog steeds dat een programma volledig in het hoofdgeheugen moet geladen zijn om uitgevoerd te kunnen worden. Elk geheugenbeheersysteem moet voldoen aan een aantal belangrijke randvoorwaarden:

1) Om de efficiëntie van de processor en van de I/O-voorzieningen te verhogen, is het noodzakelijk dat zoveel mogelijk processen in het hoofdgeheugen geladen zijn. Alhoewel de kosten van geheugen drastisch zijn gedaald, is er in de praktijk nooit voldoende geheugen om alle programma's en gegevens op te slaan die nodig zijn voor de actieve processen van gebruikers en van het besturingssysteem zelf. Daarom blijft het noodzakelijk om blokken code en gegevens naar en uit het secundaire geheugen te *swappen* (figuur 4.1). Dit is echter een langzame bewerking, waarvan de snelheid steeds verder achterloopt op die van de processoren. Swapping verhoogt hierdoor de tijdsduur die nodig is voor een proceswisseling. Het is dan ook een complexe taak voor het geheugenbeheersysteem om de swapping zodanig te organiseren dat de processor zoveel mogelijk bezig gehouden wordt. Tegenwoordig wordt swapping bij weinig systemen als standaardoplossing gebruikt. Aangepaste vormen (zoals virtueel geheugen, §4.4) zijn echter in de meeste besturingssystemen terug te vinden.



Figuur 4.1



Figuur 4.2

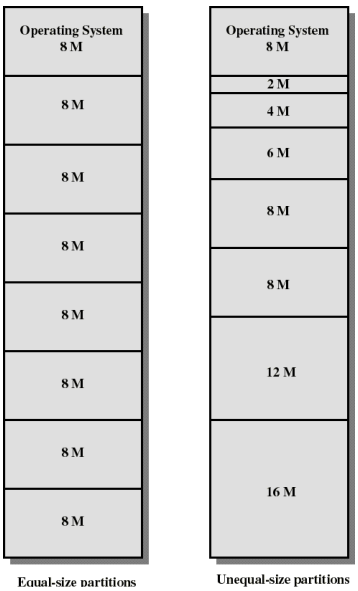
2) De hardware van de processor en de software van het besturingssysteem moeten de verwijzingen in de code van het programma op een of andere wijze vertalen in adressen van het fysieke geheugen, die overeenkomen met de huidige locatie van het programma in het hoofdgeheugen. Het geheugenbeheer moet toestaan dat bij het terugswappen het geheugenbeeld binnen het hoofdgeheugen wordt verplaatst. Aandachtspunten zijn hierbij (figuur 4.2) niet alleen de locatie van het procesbesturingsblok, van de huidige top van de uitvoeringsstack en van het ingangspunt naar het programma, maar ook geheugenverwijzingen binnen het programma zelf, zoals sprongінstructies en gegevensverwijzingen.

3) Enerzijds mogen processen niet zonder toestemming geheugenlocaties van andere processen kunnen lezen of schrijven. Dergelijke instructies moeten afgebroken worden van zodra ze uitgevoerd worden. Ondermeer moeten ook de code en de gegevensstructuren van het besturingssysteem zelf doeltreffend kunnen afgeschermd worden van gebruikersprocessen. Anderzijds moet dit beveiligingsmechanisme voldoende flexibel zijn om verschillende processen toegang te kunnen geven tot gedeelde stukken van het hoofdgeheugen. Het is bijvoorbeeld aangewezen om processen, die hetzelfde programma uitvoeren, niet elk een eigen kopie van de programmainstructies te laten gebruiken. Ook is het dikwijls noodzakelijk dat processen die samenwerken dezelfde gegevensstructuren delen.

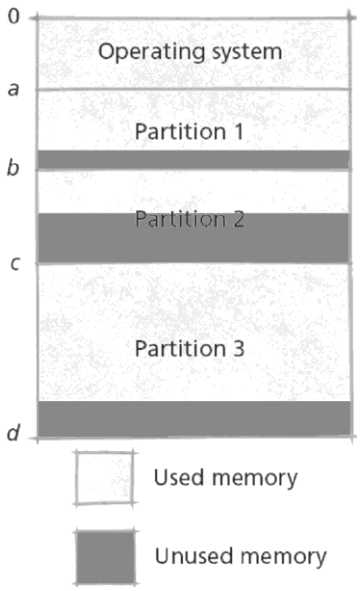
4) Zowel het hoofdgeheugen als het secundaire geheugen zijn doorgaans georganiseerd als een ééndimensionale adresruimte. De meeste programma's daarentegen zijn logisch ingedeeld in modules, met andere karakteristieken (code of gegevens, al dan niet wijzigbaar, …). Modules kunnen onafhankelijk van elkaar geschreven en gecompileerd worden en kunnen andere beschermingsniveaus of gedeelde toegang krijgen. Het is de taak van het geheugenbeheer en van de computerhardware om deze modules te vertalen naar de ééndimensionale adresruimte. De segmentatie techniek (§4.3.2 en §4.4.3) zal hierbij het meest geschikte hulpmiddel blijken.

4.2 Partitionering

Bij de meeste systemen voor geheugenbeheer bezet het besturingssysteem een vast deel van het hoofdgeheugen (figuur 4.1), meestal in het uiterst lage of in het uiterst hoge gebied, omdat de processor de interruptvector (§1.1.3) daar verwacht. De eenvoudigste techniek om de rest van het hoofdgeheugen te beheren, *vaste partitionering* genoemd, is het te verdelen in gebieden met vaste begrenzing. Als dit geheugenbeeld eenmaal gedefinieerd is, verandert het niet meer. Systemen met vaste partitionering vereisen nauwelijks besturingssysteemsoftware: het besturingssysteem moet enkel in een tabel bijhouden welke partities nog beschikbaar zijn, en welke reeds bezet. Het aantal partities dat werd ingesteld bij het genereren van het systeem beperkt echter het aantal actieve processen. Het wordt dan ook in de praktijk niet meer toegepast, tenzij in *batch*-omgevingen. Toch wordt het hier besproken, aangezien het de basis vormt voor de geheugenmodellen in §4.3 en §4.4 .



Figuur 4.3

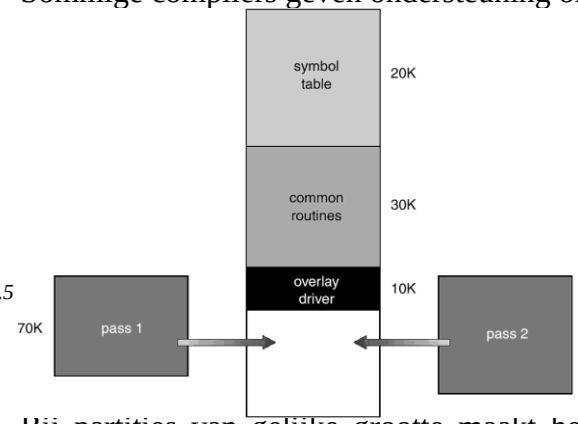


Figuur 4.4

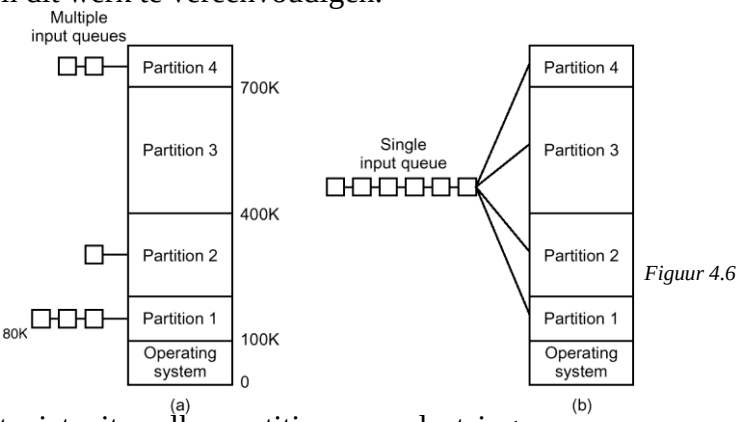
Bij vaste partitionering veronderstelt men dat de maximale hoeveelheid geheugen die een proces zal vereisen, a priori gekend is. Ofwel gebruikt men partities van gelijke, ofwel van

ongelijke grootte (figuur 4.3). Vooral partities met een vaste grootte zijn gekenmerkt door twee problemen. Enerzijds wordt het hoofdgeheugen weinig efficiënt gebruikt: elk procesbeeld, hoe klein ook, bezet een ganse partitie. De partitiegrootte komt zelden precies overeen met de geheugenbehoefte van de aanwezige processen. Deze verspilling van beschikbare geheugenruimte wordt *interne fragmentatie* genoemd: geheugen wordt aan een proces toegekend, maar kan niet gebruikt worden (figuur 4.4). Deze interne fragmentatie wordt aanzienlijk als er veel kleine processen zijn. Een verkeerde keuze van de geheugenverdeling bij de initialisatie van het systeem, kan leiden tot een zeer slechte geheugenbenutting in de toekomst.
Anderzijds kan een procesbeeld nog steeds te groot zijn om in één partitie te passen. In dat geval moet het gebruikersprogramma zelf *overlay* technieken inbouwen, zodat zich altijd maar een deel van het programma in de toegewezen partitie bevindt. Bij overlays worden alleen die instructies en gegevens in het geheugen bewaard die op dat moment nodig zijn. Het proces voert zelf code in een *overlaydriver* (figuur 4.5) uit om partiële geheugenbeelden naar behoefte in te lezen. Er is hierbij geen enkele ondersteuning vereist door het besturingssysteem. De overlaystructuur moet door de programmeur zelf ontworpen worden, wat veel kennis vraagt van de structuur van het programma en van de gebruikte datastructuren. Sommige compilers geven ondersteuning om dit werk te vereenvoudigen.

Figuur 4.5



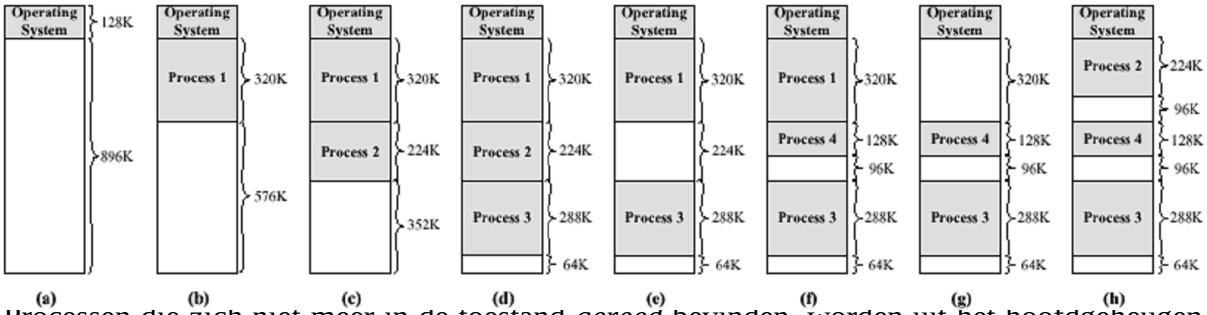
Figuur 4.6



Bij partities van gelijke grootte maakt het niet uit welke partitie voor plaatsing van een proces wordt gebruikt. Partities van ongelijke grootte bieden een grotere flexibiliteit: men kan elk proces toewijzen aan de kleinste partitie waarin het past. Het geheugenbeeld van het besturingssysteem ligt nu dichter bij de werkelijke behoeften van de processen. Voor elke partitie is dan wel een schedulingwachtrij nodig. In deze benadering wordt het verspilde geheugen binnen een partitie weliswaar geminimaliseerd, doch vanuit het globale systeemstandpunt is deze techniek verre van volmaakt. In het voorbeeld dat in figuur 4.6 geïllustreerd wordt, blijft partitie 3 ongebruikt, ook al kan daaraan een kleiner proces toegewezen worden.
Het toepassen van één enkele wachtrij voor alle processen, waarbij telkens de kleinste beschikbare partitie wordt geselecteerd die het proces kan bevatten, is dan ook een betere benadering.

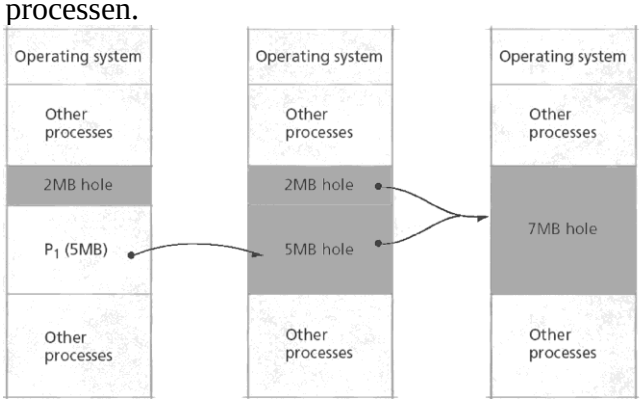
Dynamische partitionering poogt deze beperkingen op te lossen door een variabel aantal partities met een variabele grootte te gebruiken. Dynamische partitionering creëert een geheugenbeeld dat meer aan de eisen van processen voldoet. Wordt een proces overgebracht naar het hoofdgeheugen, dan krijgt het precies de hoeveelheid geheugen die het nodig heeft. Figuur 4.7 illustreert hiervan een voorbeeld, dat uitgaat van 1 MB hoofdgeheugen.

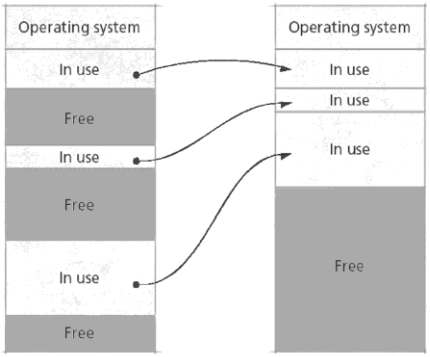
Figuur 4.7



Processen die zich niet meer in de toestand *gereed* bevinden, worden uit het hoofdgeheugen geswapt om plaats vrij te maken voor beschikbare processen, in de toestand *gereed-onderbroken*. Aangrenzende vrije partities kunnen hierbij worden samengevoegd (*coalescing*, figuur 4.8). Dit voorbeeld toont aan dat deze methode uiteindelijk leidt tot een situatie waarin het geheugen buiten de partities steeds meer gefragmenteerd en steeds minder benut wordt. Dit fenomeen wordt *externe fragmentatie* genoemd. Er is misschien wel genoeg geheugen vrij om aan een nieuwe aanvraag te voldoen, maar niet in een aaneengesloten blok. In het ergste geval is er een blok verspild geheugen tussen elke twee opeenvolgende processen.

Figuur 4.8





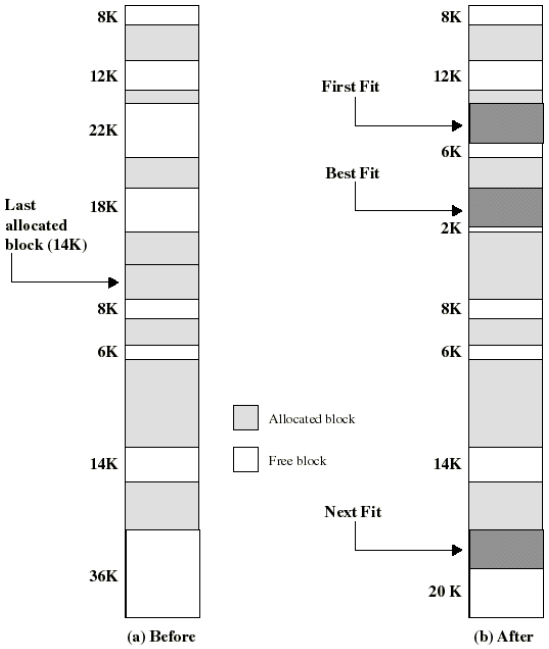
Figuur 4.9

4.2.1 Plaatsingsalgoritmen

Externe fragmentatie kan opgelost worden door *compactie* (figuur 4.9): het besturingssysteem verschuift alle processen, zodat ze aaneengesloten worden, en al het vrije geheugen terug één enkel blok vormt. Zowel coalescing als compactie worden soms *garbage collection* genoemd. Compactie is een proces dat processortijd verslindt. Meer complexe compactie algoritmen proberen dan om ook om een minimaal aantal processen te verplaatsen, en toch een relatief grote opeenvolgend vrije geheugenruimte te bekomen.

Om regelmatige compactie te vermijden, moet oordeelkundig te werk gegaan worden bij toewijzing van processen aan het geheugen (het vullen van de gaten). Vier elementaire *plaatsingsalgoritmen* kunnen hiervoor gebruikt worden (Figuur 4.10):

- *Best-Fit* kiest het blok waarvan de grootte het meest overeenkomt met die van het nieuwe proces,
- *First-Fit* zoekt het hoofdgeheugen sequentieel af vanaf het begin, en kiest het eerste beschikbare blok dat voldoende groot is,
- *Next-Fit*, ook wel *Rotating-First-Fit* genoemd, zoekt het hoofdgeheugen sequentieel af vanaf het punt van de laatste plaatsing, en kiest opnieuw het eerste beschikbare blok dat voldoende groot is,
- *Worst-Fit* wijst het grootst beschikbare blok toe, in de hoop relatief grote gaten over te houden.



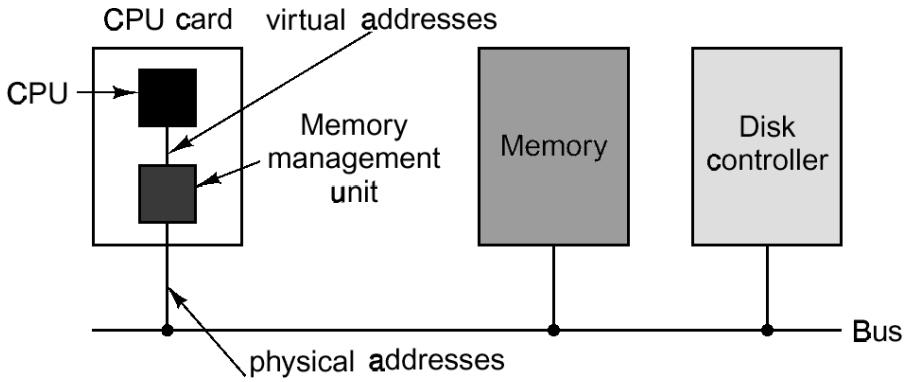
Figuur 4.10

Alhoewel geen globaal geldende vergelijking mogelijk is, blijkt First-Fit niet alleen de eenvoudigste, maar meestal ook de snelste en de beste oplossing. Toch heeft statistische analyse aangetoond dat zelfs bij First-Fit gemiddeld ⅓ van het geheugen uiteindelijk onbruikbaar wordt. Bij Next-Fit worden doorgaans grotere geheugenblokken op het einde van het geheugen vlugger opgesplitst in kleinere fragmenten, waardoor compactie eerder noodzakelijk wordt. Best-Fit daarentegen kent verrassend de slechtste resultaten en is bovendien trager: het geheugen wordt snel verdeeld in blokken die te klein zijn om aan nieuwe verzoeken te beantwoorden. Meer complexere *Optimal-Fit* strategieën schakelen dynamisch over en weer tussen de vier basisalgoritmen, afhankelijk van het dynamische patroon van procescreaties.
Onafhankelijk van welk algoritme gekozen wordt, is er steeds een reële mogelijkheid dat alle processen in het hoofdgeheugen zich in een geblokkeerde toestand bevinden, en dat er zelfs na compactie onvoldoende plaats is om een nieuw proces te laden. Het besturingssysteem moet dan beslissen welk proces uit het hoofdgeheugen zal worden geswapt. De *vervangings-algoritmen* die hiervoor gebruikt kunnen worden, worden in §4.5.6 behandeld.

4.2.2 Adresvertaling

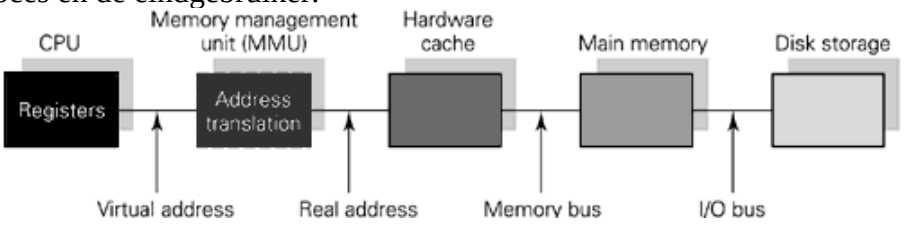
Zowel bij vaste als bij dynamische partitionering kan een proces in de loop van de tijd verschillende partities bezetten. Wordt compactie gebruikt, dan worden processen bovendien verplaatst terwijl ze zich in het hoofdgeheugen bevinden. De locaties waarnaar een proces verwijst zijn daarom niet vast. Om dit probleem op te lossen wordt onderscheid gemaakt tussen:

- *Logische adressen*, verwijzingen naar geheugenlocaties onafhankelijk van de huidige toewijzing van het proces aan het geheugen. Logische adressen stemmen overeen met het geheugenbeeld dat de programmeur heeft. Voor software ontwikkelaars is het een belangrijk voordeel er niet voor te hoeven zorgen dat een proces in een bepaald geheugengebied wordt geladen om correct te worden uitgevoerd. In systemen met partitionering kan men gebruik maken van bijzondere gevallen van logische adressen: *relatieve adressen*. Hierbij kunnen alle adressen uitgedrukt worden ten opzichte van één bepaald punt, doorgaans het begin van het programma.
- *Fysieke* of *absolute adressen*, de werkelijke locaties in het hoofdgeheugen. Deze adressen stemmen overeen met het geheugenbeeld van het besturingssysteem en de processor.



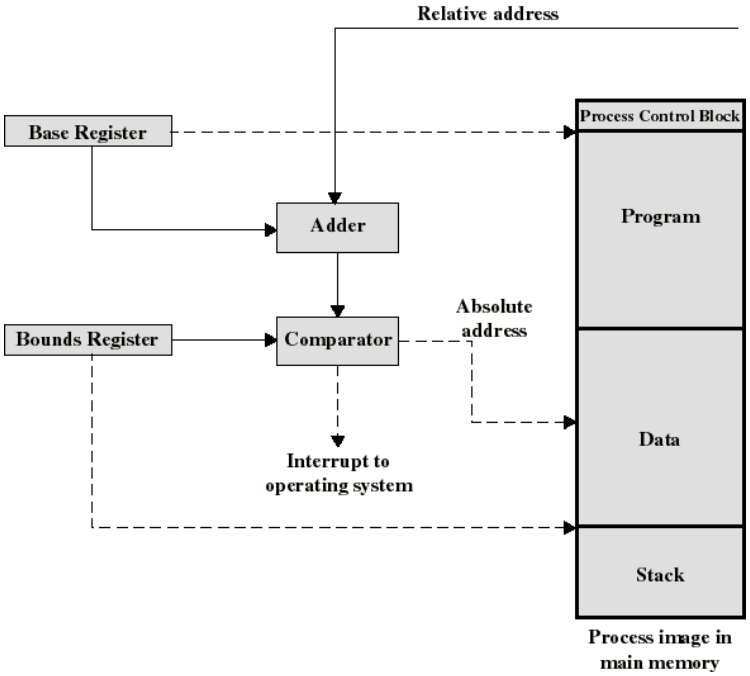
Figuur 4.11

Logische adressen moeten vertaald worden in fysieke adressen vooraleer geheugentoegang mogelijk is. Op het moment dat elke instructie uitgevoerd wordt, staat een RISC processor op de processorkaart, de *memory management unit* (MMU, figuur 4.11), in voor de vertaling van logische in fysieke adressen. Dit wordt *dynamische adresvertaling* genoemd. Gesitueerd in het kader van de geheugenhiërarchie (§1.1.4), bevindt de MMU zich tussen de processor-registers en L2 caches in (figuur 4.12). Dynamische adresvertaling is volledig transparant voor het proces en de eindgebruiker.



Figuur 4.12

Figuur 4.13 illustreert hoe de adresvertaling uitgevoerd wordt in een systeem met dynamische partitionering: wanneer het proces in het hoofdgeheugen wordt geladen, geswapt of verschoven, worden de aan het proces toegekende absolute begin- en eindadressen geladen in het *basisregister* en het *begrenzingregister* van de processorregisters, als onderdeel van de contextwisseling. Tijdens de uitvoering worden enkel relatieve adressen aangewend. Elk adres ondergaat hierbij twee bewerkingen: een optelling om het overeenkomstige fysieke adres te bekomen, en een vergelijking met het begrenzingregister. Ligt het adres niet binnen de begrenzing, dan wordt een interrupt gegenereerd naar het besturingssysteem.

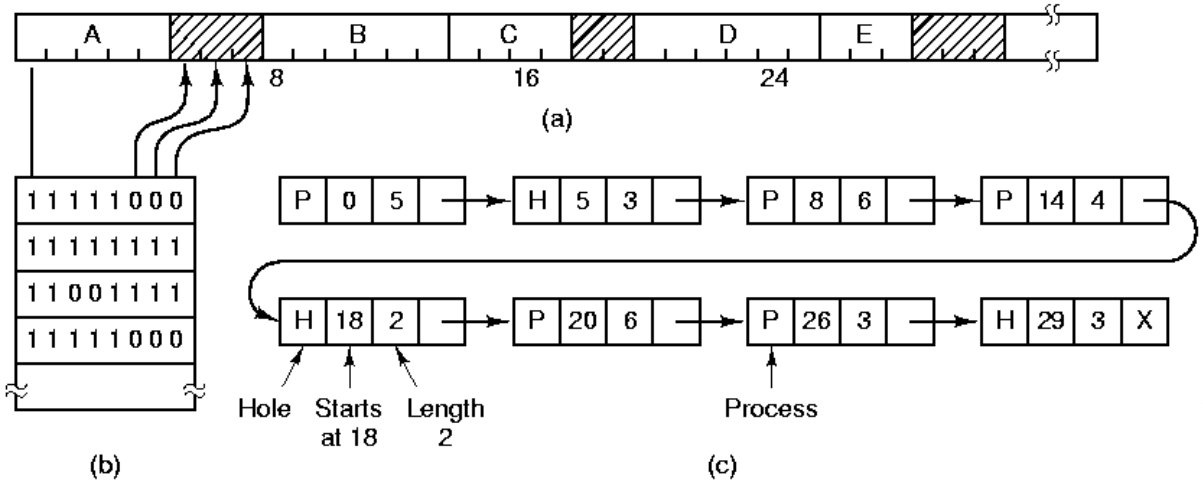


Figuur 4.13

4.2.3 Administratie van het geheugengebruik

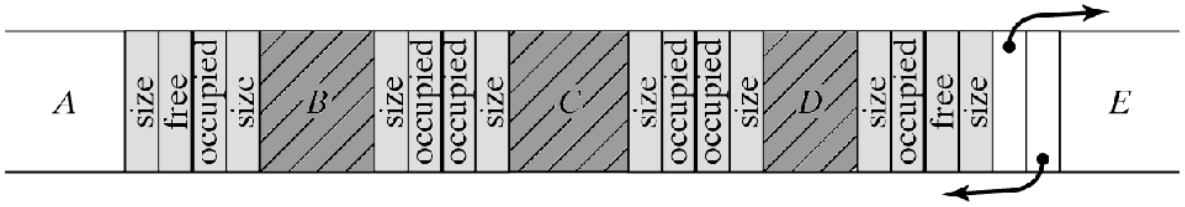
Er zijn drie manieren waarop een besturingssysteem kan bijhouden welke geheugenpartities reeds in gebruik zijn, en welke nog beschikbaar zijn.

1) Bij het gebruik van bitmaps wordt het geheugen verdeeld in kleine allocatie-eenheden van gelijke grootte. Bij elke allocatie-eenheid hoort een bit die aangeeft of de eenheid vrij is of bezet (figuur 4.14b). Hoe kleiner de allocatie-eenheid is, des te groter wordt de bitmap. Als de allocatieeenheid te groot wordt gekozen, verspilt elk proces veel geheugen in zijn laatst gealloceerde eenheid. In de praktijk worden bitmaps niet veel gebruikt: wanneer een nieuw proces een aantal, *N*, aaneensluitende allocatie-eenheden vereist, dan moet het geheugenbeheersysteem immers in de bitmap op zoek gaan naar een rij van *N* opeenvolgende nullen. Dit is een tijdsverslindend proces.



Figuur 4.14

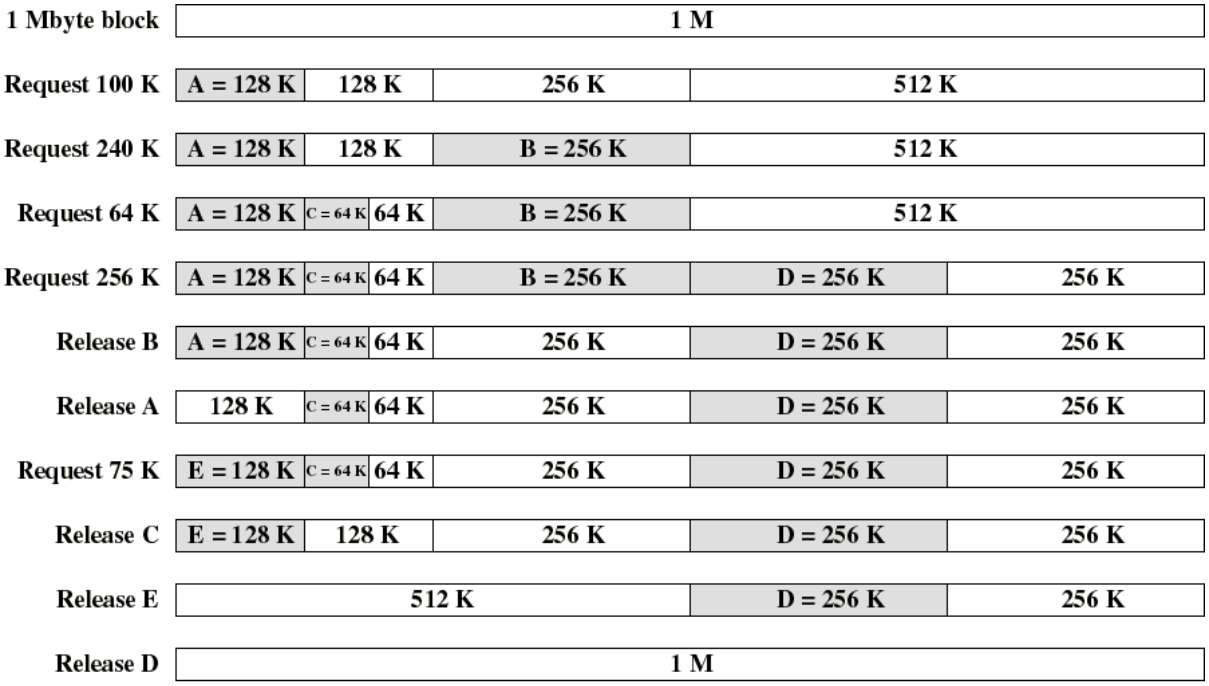
2) *Gekoppelde lijsten* vormen een alternatieve manier om de vrije allocatie-eenheden bij te houden (figuur 4.14c): het geheugen wordt voorgesteld als een gekoppelde lijst segmenten die ofwel in gebruik zijn door een proces, ofwel beschikbaar zijn. Indien bovendien voor het vrije en het gebruikte geheugen aparte lijsten worden bijgehouden, kunnen de lijsten worden gesorteerd op grootte. Dit doet men echter meestal niet, aangezien het bijwerken van deze lijsten complexer en langduriger is, in het bijzonder om de buren van de allocatie-eenheden te vinden wanneer deze vrijgegeven worden. De blijkbaar meest efficiënte oplossing, voorgesteld door Donald Knuth, plaatst zowel aan het begin als aan het einde van elk geheugenblok informatie met de status en de grootte van het blok (figuur 4.15), en neemt enkel de vrije geheugenblokken op in een dubbelgekoppelde lijst.



Figuur 4.15

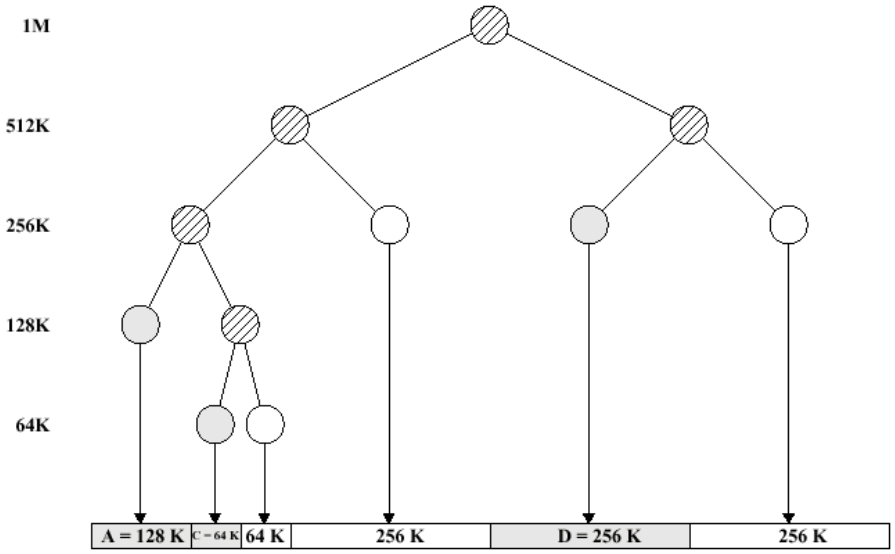
3) Het *buddysysteem*, eveneens ontwikkeld door Knuth, is een algoritme voor geheugenbeheer dat spitsvondig gebruik maakt van het feit dat computers binaire getallen gebruiken om te adresseren. In een buddysysteem wordt de gehele geheugenruimte, die beschikbaar is voor toewijzing, behandeld als één blok met een grootte 2^y, en worden geheugenblokken beschikbaar gesteld met een grootte 2^x ($x \leq y$). Heeft een eerste verzoek om een geheugenblok een grootte *X*, $2^{y-1} < X \leq 2^y$, dan wordt het volledige blok 2^y toegewezen. Anders wordt het blok gesplitst in twee gelijke *buddy's* met grootte 2^{y-1}. Als $2^{y-2} < X \leq 2^{y-1}$, dan wordt het

verzoek toegewezen aan één van deze twee buddy's. Zoniet wordt het proces recursief verder gezet tot wanneer buddy's ontstaan met grootte 2^z, waarbij $2^{z-1}<X\leq 2^z$. Het buddysysteem houdt voortdurend een gekoppelde lijst *x* bij van alle niet toegewezen blokken met grootte 2^x. Een blok kan worden verwijderd uit de lijst *x* door het in twee buddy's te splitsen met grootte 2^{x-1}. Het *buddysysteem* versnelt het samenvoegen van naast elkaar liggende gaten, wanneer een proces eindigt of wordt uitgeswapt: zijn beide buddy's van één paar in de lijst *x* vrij, dan worden ze verwijderd uit lijst *x* en samengevoegd tot één blok met grootte 2^{x+1}.



Figuur 4.16

Figuur 4.16 toont een voorbeeld van een door een buddysysteem beheerd geheugenblok van 1 MB, waarin achtereenvolgens blokken A (100 kB), B (240 kB), C (64 kB), D (256 kB) worden toegekend, B en A worden vrijgegeven, E (75 kB) wordt toegekend, en C, E en D worden vrijgeven. Figuur 4.17 illustreert de overeenkomstige boomstructuur, onmiddellijk na het vrijgeven van blok B.

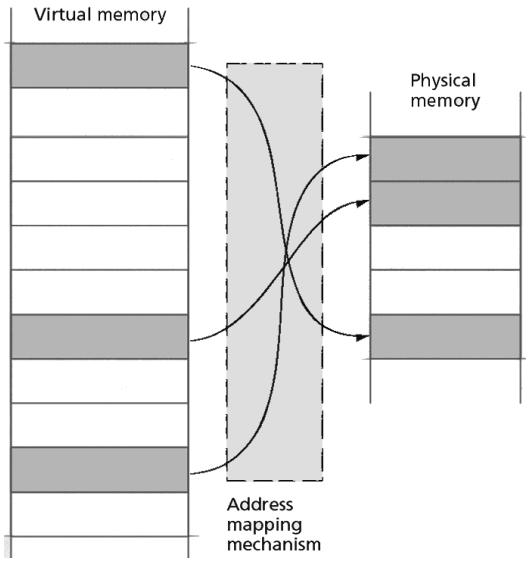


Figuur 4.17

Het buddysysteem omzeilt enkele nadelen van vaste en dynamische partitionering: wanneer er een blok van 2^x wordt vrijgegeven, dan hoeft de geheugenmanager alleen maar in de lijst *x*, met beschikbare blokken van 2^x, te zoeken om vast te stellen of samenvoegen mogelijk is. Bij andere algoritmen, waarin geheugenblokken op willekeurige manieren mogen worden gesplitst, moeten daarentegen alle gatenlijsten worden doorzocht. Het buddysysteem wordt dan ook in de praktijk dikwijls gebruikt, bijvoorbeeld voor de geheugentoewijzing van de kernelstructuren in Unix. In Unix wordt wel een aangepaste vorm gehanteerd, waarbij het samenvoegen van buddy's uitgesteld wordt tot wanneer niet onmiddellijk voldaan kan worden aan de aanvraag van een nieuw blok. Helaas is de graad van interne fragmentatie van buddysystemen zeer hoog: alle aanvragen moeten immers omhoog worden afgerond naar een macht van 2. De rest van het toegewezen blok wordt verspild. Voor toewijzing van het kernelgeheugen van Unix is dit minder een probleem, aangezien de Unix kernel tijdens de uitvoering veelvuldig, vooral kleine, tijdelijke tabellen en buffers vereist, in het bijzonder voor I/O-operaties.

4.3 Paginering en segmentatie

Systemen met vaste partitionering leiden tot interne fragmentatie, systemen met dynamische partitionering tot externe fragmentatie. Deze moeilijkheden hebben één gemeenschappelijke hoofdoorzaak: de poging ervoor te zorgen dat het vrije geheugen steeds voldoend grote aaneengesloten blokken vormt, zodat het maximaal kan worden benut. Misschien is dit de verkeerde methode. In plaats van het geheugen zodanig in te delen dat het aan de behoefte van een proces voldoet, moeten we de vraag stellen hoe een proces willekeurige stukken vrij geheugen zou kunnen benutten (figuur 4.18). Afhankelijk of deze vraag beantwoord wordt met betrekking tot systemen met vaste of dynamische partitionering, bekomen we twee nieuwe geheugenmodellen.



Figuur 4.18

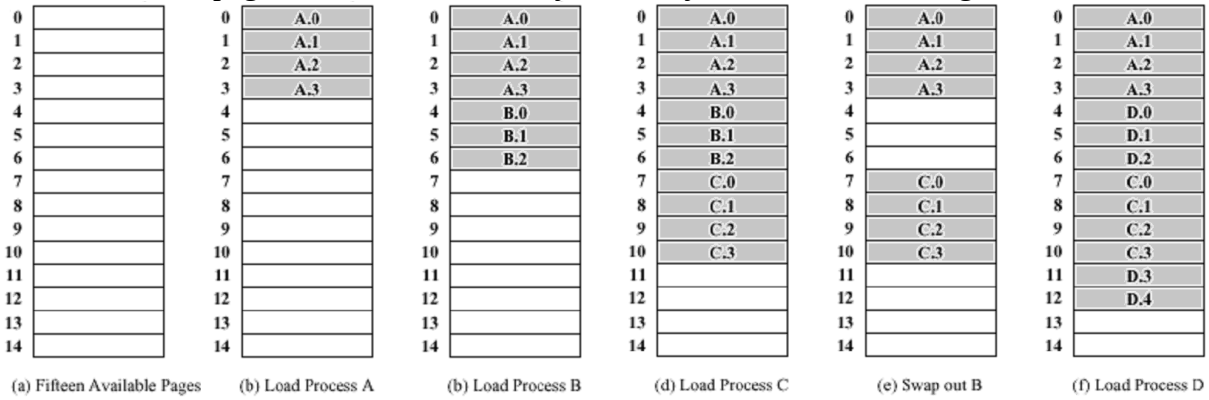
4.3.1 Paginering

Bij de *paginering* techniek verdeelt men daarom:

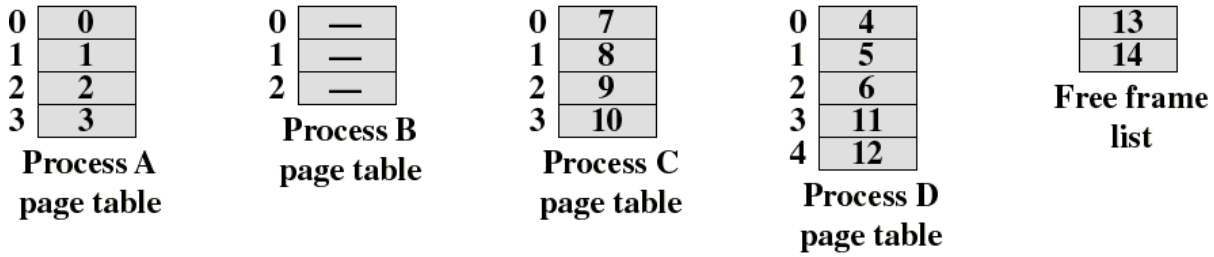
- het hoofdgeheugen in *frames*, stukken met een gelijke, relatief kleine, grootte,
- en alle procesbeelden in *pagina's* ter grootte van de frames.

De procespagina's worden toegewezen aan frames van het hoofdgeheugen. Kleinere processen vereisen minder frames, grotere processen vereisen er meer. Het besturingssysteem houdt een lijst van vrije frames bij, en een paginatabel voor elk proces (in

het procesbeeld van het proces zelf). Deze bevat de framelocatie van elke pagina van het proces: elke ingang in de paginatabel bevat het nummer van het frame in het hoofdgeheugen dat de corresponderende pagina bevat. Paginering is verwant aan vaste partitionering: de partities zijn echter veel kleiner, en processen bezetten verschillende partities, die niet aaneengesloten hoeven te zijn. Er is geen externe fragmentatie. De interne fragmentatie blijft beperkt tot een fractie van de laatste pagina van elk proces. Figuur 4.19 illustreert paginering tijdens het achtereenvolgens laden van processen A (4 pagina's), B (3 pagina's), C (4 pagina's), het uitswappen van B, en het laden van D (5 pagina's). Figuur 4.20 toont de toestand van de paginatabellen en van de lijst met vrije frames, nadat D is geladen.

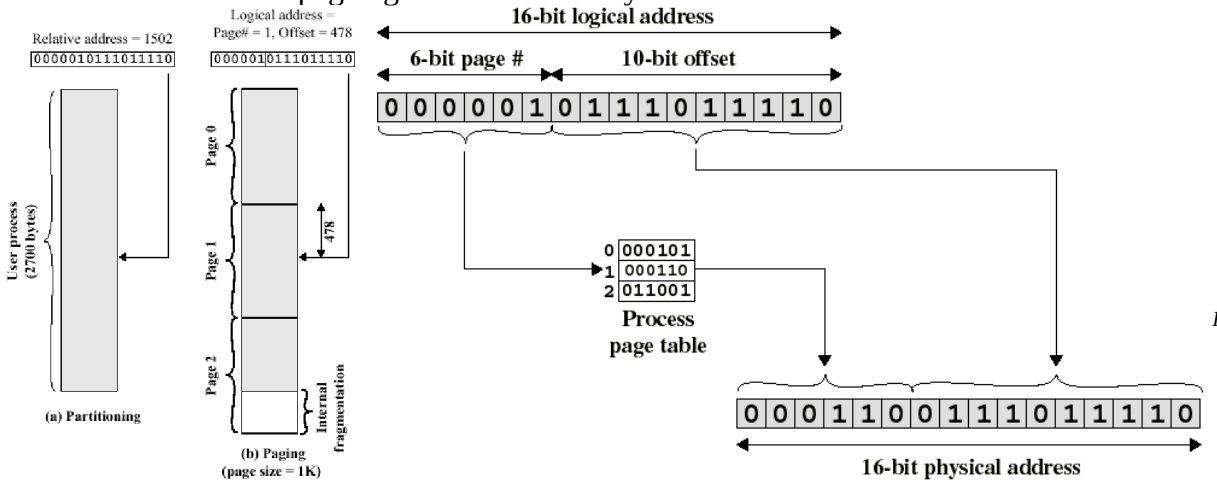


Figuur 4.19



Figuur 4.20

Bij paginering bestaat elk logisch adres uit een paginanummer en een relatieve positie binnen de pagina. Ook hier gebeurt de vertaling van logische in fysieke adressen door de processor-hardware, meer bepaald door de memory management unit (MMU). Voor de grootte van pagina's en frames wordt steeds een macht van 2 gekozen. Hierdoor kunnen de logische adressen beschouwd blijven worden als relatieve adressen, verwijzend naar de oorsprong van het programma. Als de grootte van de logische adresruimte 2^m is, en de paginagrootte is 2^n woorden, dan bevatten de $m-n$ meest significante bits van een adres het paginanummer, en de n minst significante bits de offset ten opzichte van het begin van de pagina. Figuur 4.21, illustreert dit voor een paginagrootte van 1024 bytes.



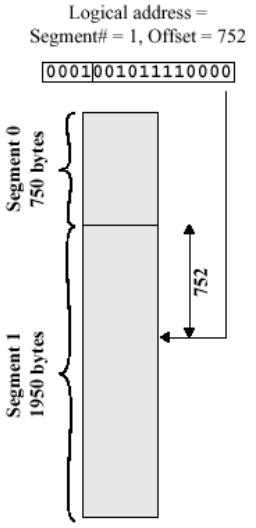
Figuur 4.21

Figuur 4.22

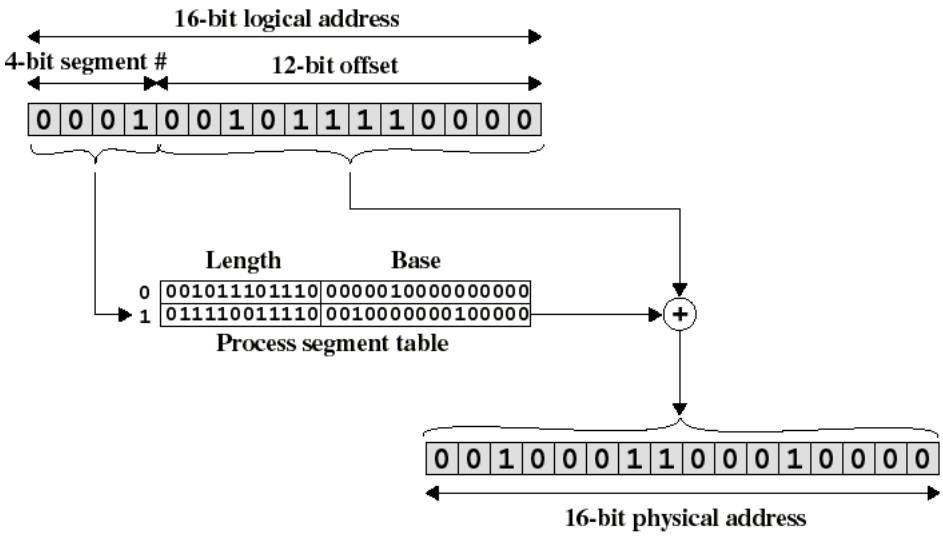
Dit vereenvoudigt de dynamische adresvertaling tijdens uitvoering (figuur 4.22): het paginanummer wordt gehaald uit de linkerbits van het logische adres, gebruikt als een index in de procespaginatabel om het framenummer te vinden, waarna het fysieke adres wordt gevonden door dit framenummer en de positie binnen de pagina aan elkaar toe te voegen. Dit komt er op neer dat de MMU tijdens de instructiecyclus niets anders moet doen dan het paginanummer door het framenummer te vervangen, waardoor de geheugenverwijzing automatisch naar de correcte fysieke locatie verwijst. Aangezien de paginatabel van een proces enkel de eigen pagina's bevat, en de paginatabel enkel door het besturingssysteem kan aangepast worden, is een proces niet in staat om geheugen te benaderen dat het niet bezit.

4.3.2 Segmentatie

Segmentatie lijkt op dynamische partitionering, zoals paginering lijkt op vaste partitionering. Ook hier worden het programma en de gegevens verdeeld in blokken, nu echter niet noodzakelijk met uniforme grootte. Elk logisch adres bestaat uit twee onafhankelijke dimensies: een segmentnummer en een relatieve positie binnen het segment (figuur 4.23). Het hoofdgeheugen blijft uiteraard een eendimensionale, lineaire, rij adressen, vertrekkend van 0. De koppeling tussen het tweedimensionale en eendimensionale geheugenbeeld gebeurt aan de hand van een segmenttabel. Het besturingssysteem houdt een segmenttabel bij voor elk proces, die de fysieke locatie van elke segment van het proces bevat. In tegenstelling tot dynamische partitionering kunnen processen bij segmentatie meerdere partities in het hoofdgeheugen bezetten, die niet aaneengesloten hoeven te zijn. Segmentatie vermijdt interne fragmentatie, maar is onderhevig aan externe fragmentatie, een verschijnsel dat hier *checkerboarding* wordt genoemd. Omdat echter het procesbeeld wordt opgebroken in veel kleinere stukken, is het waarschijnlijker dat ze passen in de beschikbare geheugenblokken: de externe fragmentatie is minder groot dan bij dynamische partitionering
Ten gevolge van de ongelijke grootte van segmenten, is er, in tegenstelling tot paginering, geen eenvoudige relatie tussen logische en relatieve adressen. Hierdoor is de dynamische adresvertaling tijdens uitvoering van logische in absolute adressen minder elementair (figuur 4.24): het *segmentnummer* wordt gehaald uit de linkerbits van het logische adres, gebruikt als een index in de *processegmenttabel* om het fysieke beginadres en de lengte van het segment te vinden, waarna het fysieke adres gevonden wordt door dit beginadres en de positie binnen het segment bij elkaar op te tellen. De specifieke hardware van de MMU zorgt ervoor dat dit veel sneller gebeurt dan indien de CPU dit zou moeten uitvoeren. De lengte van het segment moet groter zijn dat de rechterbits van het logische adres aangeven, zoniet is het logische adres ongeldig, en wordt een interrupt gegenereerd.



Figuur 4.23

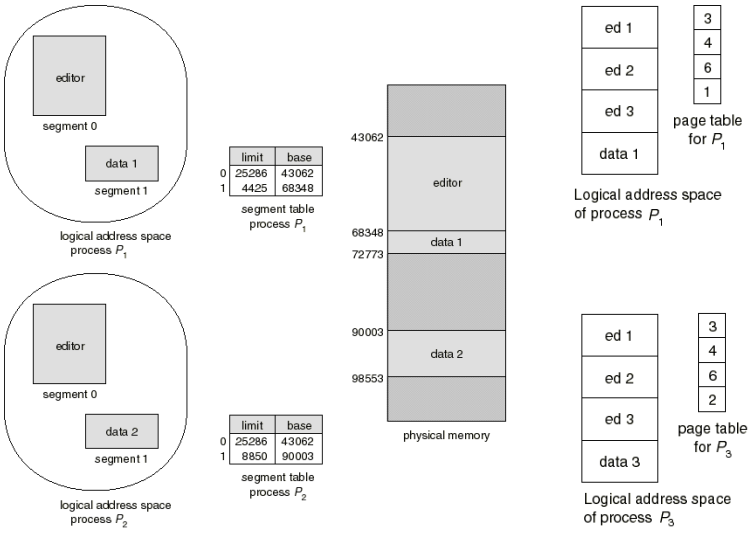


Figuur 4.24

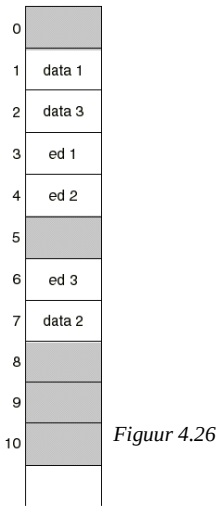
Paginering blijft verborgen voor de programmeur. Segmentatie daarentegen wordt bewust zichtbaar gemaakt: de programmeur of de compiler kan zowel individuele modules van het programma als individuele gegevensstructuren toewijzen aan verschillende segmenten. De principes van gestructureerd ontwerpen leiden tot modulaire software. Modules kunnen onafhankelijk van elkaar worden gewijzigd en gecompileerd. Segmentatie biedt een manier om de geheugenbehoeften van een proces logisch te bekijken, in plaats van als een verzameling fysiek aparte delen. De compiler kan automatisch segmenten aanmaken die met de modules van het programma corresponderen, en aparte segmenten invoeren voor het aan objecten toegewezen geheugen (de *heap*) en de door elke thread gebruikte stack.

Segmentatie vergemakkelijkt ook *sharing* van procedures of data tussen meerdere processen. Verschillende segmenten kunnen hierbij een verschillend type protectie hebben. In figuur 4.25 bijvoorbeeld verwijzen de segmenttabellen van twee gebruikersprocessen, die dezelfde editor gebruiken, naar een identiek blok in het hoofdgeheugen voor wat de programmainstructies betreft. Dit segment kan bijvoorbeeld alleen uitvoerbaar worden gemaakt, om te voorkomen dat erin wordt geschreven. Beide processen blijven echter voor hun gegevens beschikken over afzonderlijke datasegmenten, waarin kan gelezen en geschreven worden. Dit delen van procedures of data tussen meerdere processen is ook in zuivere pagineringsystemen wel mogelijk, maar dit is ingewikkelder: in feite wordt dan segmentatie gesimuleerd (figuur 4.26)

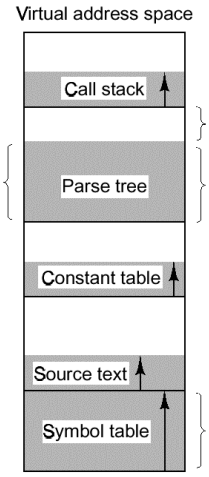
Figuur 4.25



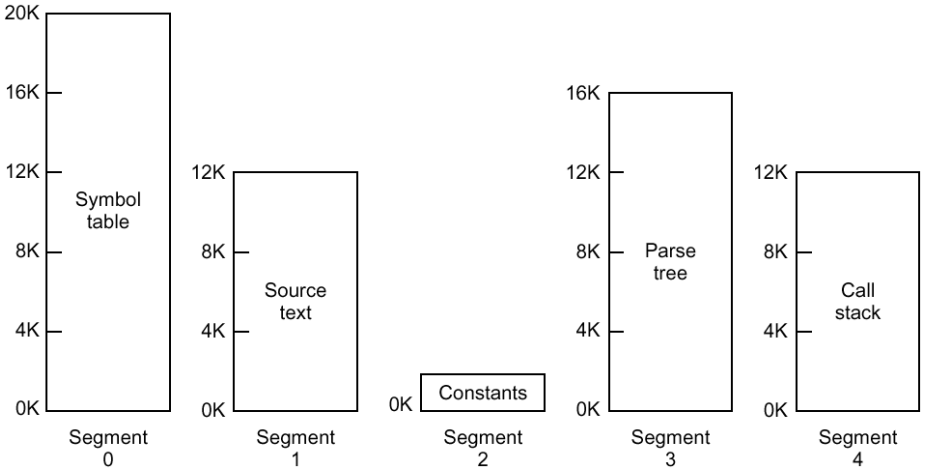
Figuur 4.26



Verder kan, omdat elk segment een aparte adresruimte vormt, de lengte van de segmenten in de loop van de uitvoering van het proces veranderen. Bij het compileren van programma's bijvoorbeeld moet een compiler beschikken over tabellen in het hoofdgeheugen, die tijdens de compilatie steeds groter worden: de broncode van het programma, een symbooltabel met de namen en attributen van alle variabelen, een tabel met de gebruikte integer en floating-point constanten, en een parse-boom met de syntactische analyse van het programma. Eveneens roept de compiler voor procedureaanroepen een stack aan, die tijdens compilatie onvoorspelbaar kan groeien of krimpen. Indien geen segmentatie gebruikt wordt (figuur 4.27), dan kan het zijn dat de compilatie moet onderbroken worden, omdat één tabel volloopt, terwijl er in de andere tabellen nog veel ruimte is. Indien segmentatie wordt aangewend, dan kunnen de verschillende segmenten onafhankelijk van elkaar groeien of krimpen, zonder logische adresconflicten (figuur 4.28).



Figuur 4.27



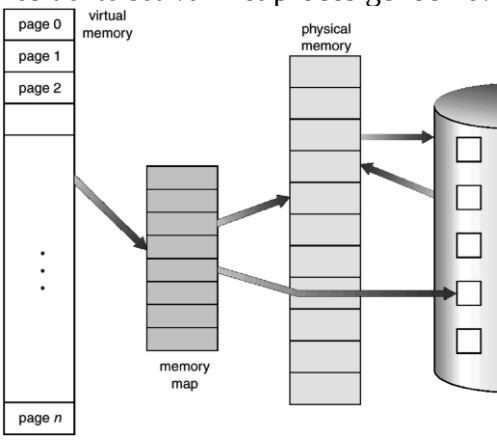
Figuur 4.28

4.4 Virtueel geheugen

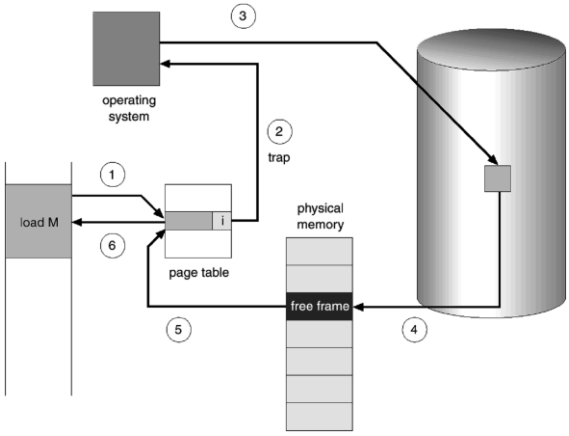
Twee fundamentele kenmerken vallen op indien men paginering en segmentatie vergelijkt met vaste of dynamische partitionering:

- Alle geheugenverwijzingen binnen een proces zijn logische adressen, die tijdens de uitvoering dynamisch vertaald worden in absolute adressen. Hierdoor kan het proces, na uit en terug in het hoofdgeheugen swappen, verschillende gebieden in het hoofdgeheugen bezetten.
- Door het gebruik van pagina- of segmenttabellen, kan het procesbeeld opgebroken worden in meerdere stukken, die niet aaneengesloten hoeven te zijn in het hoofdgeheugen.

Tot nu hebben we verondersteld dat het gehele proces in het hoofdgeheugen moet zijn geladen om uitgevoerd te kunnen worden. Gebruik makend van ofwel paginering (figuur 4.29), ofwel segmentatie, kan deze eis versoepeld worden: de uitvoering van een proces kan tijdelijk worden verder gezet indien de pagina's (of de segmenten) met de volgende op te vragen instructies en gegevenslocaties zich in het hoofdgeheugen bevinden, ook al zijn andere pagina's of segmenten van het procesbeeld niet aanwezig. Dit concept noemt men *virtueel geheugen*. Het deel van een proces dat zich op een bepaald moment werkelijk in het hoofdgeheugen (dat nu reëel geheugen wordt genoemd) bevindt, wordt de *residente set* van het proces genoemd.



Figuur 4.29



Figuur 4.30

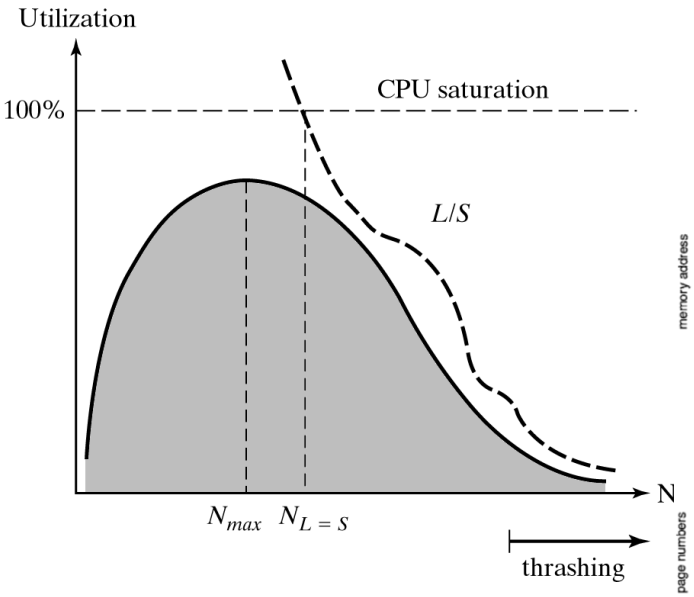
Op basis van de paginatabel (of van de segmenttabel) kan de processor vaststellen (① in figuur 4.30) of alle geheugenverwijzingen betrekking hebben op locaties die zich in de residente set bevinden. Wordt een logisch adres (nu *virtueel adres* genoemd) tegengekomen dat zich niet in het hoofdgeheugen bevindt, dan genereert de processor een interrupt die een

geheugentoegangsfout aangeeft (*paginafout* of *page fault*, ② in figuur 4.30). Het besturingssysteem neemt op dat ogenblik de besturing over, en plaatst een I/O-verzoek (③) om het stuk van het procesbeeld met het virtuele adres dat de toegangsfout veroorzaakte, binnen te halen in het hoofdgeheugen. Terwijl de schijf-I/O plaatsvindt, kan het besturingssysteem een ander proces activeren. Is het gewenste stuk eenmaal binnengehaald in het hoofdgeheugen (④), dan wordt opnieuw een interrupt gegenereerd, nu een *I/O-interrupt*, die opnieuw het besturingssysteem activeert. Het besturingssysteem kan hierdoor de paginatabellen bijwerken (⑤), en het betrokken proces terug in de wachtrij *gereed* plaatsen (⑥), of misschien zelfs onmiddellijk activeren. Omdat de toestand van het onderbroken proces bij de paginafout werd opgeslagen in het PCB blok, kan het proces op precies dezelfde instructie en in dezelfde toestand herstarten, behalve dan dat de gewenste geheugenverwijzing nu wel kan opgehaald worden.

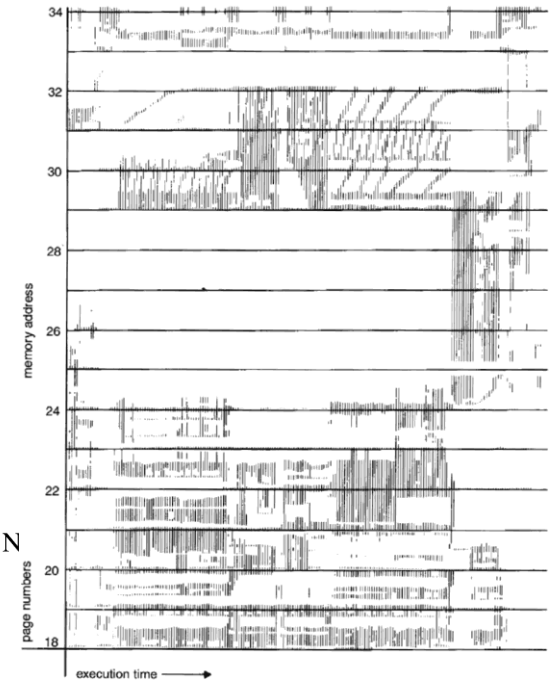
Enerzijds leidt het gebruik van virtueel geheugen tot een verbetering van het systeemgebruik:

- Omdat de processen slechts partieel in het hoofdgeheugen moeten geladen worden, is er ruimte voor meer processen. Hierdoor is de kans groter dat minstens één proces zich in de toestand *gereed* bevindt.
- Het wordt mogelijk dat de geheugenhoeveelheid die elk individueel proces aanspreekt groter is dan het totale hoofdgeheugen. Voor wat de programmeur betreft, wordt het hoofdgeheugen enkel beperkt door de beschikbare schijfruimte. De programmeur moet zich niet langer zorgen maken over de hoeveelheid beschikbaar fysiek geheugen. Hij moet ook niet langer denken aan overlay- of andere benaderingen om grote programma's in stukken te splitsen, die afzonderlijk in het geheugen kunnen worden geladen. Die taak wordt (terecht) overgelaten aan het besturingssysteem.
- Het laden van het volledige proces (indien geen virtueel geheugen wordt gebruikt) is duidelijk een verspilling als er slechts enkele stukken effectief gebruikt worden vooraleer het programma wordt onderbroken en terug uit het hoofdgeheugen geswapt wordt. Het geheugen wordt efficiënter benut door slechts enkele stukken in te laden. Bovendien worden I/O-operaties en bijgevolg tijd bespaard omdat ongebruikte stukken niet in en uit het hoofdgeheugen moeten geswapt worden.

Anderzijds veroorzaakt het gebruik van virtueel geheugen heel wat overhead: telkens naar een logisch adres gerefereerd wordt dat zich niet in het hoofdgeheugen bevindt, moet niet alleen het besturingssysteem tweemaal (na de paginafout, en na de I/O-interrupt) de controle overnemen, maar moet eveneens een ander stuk uitgeswapt worden.
Gebeurt dit te vaak, dan treedt een situatie op die *trashing* wordt genoemd (figuur 4.31): de processor besteedt meer tijd aan het swappen van stukken, dan aan het uitvoeren van instructies. Om de optimale situatie zo goed mogelijk te benaderen, is het zinvol om twee graadmeters te vergelijken: de gemiddelde tijd tussen opeenvolgend paginafouten (L), en de gemiddelde tijd die nodig is om een pagina te vervangen (S). De disk waarop geswapt wordt, blijft onderbenut indien $L \gg S$. Indien echter $L \ll S$, dan zijn er meer paginafouten dan het besturingssysteem en het I/O-mechanisme aankan.
Daarom is het essentieel dat het besturingssysteem, op basis van de historiek in het recente verleden, oordeelkundig kan schatten welke stukken niet meer en welke stukken waarschijnlijk wel nog zullen gebruikt worden in de nabije toekomst. Gelukkig hebben verwijzingen naar gegevens en instructies de neiging zich op te hopen tot clusters (*beginsel van lokaliteit*, figuur 4.32), waardoor de basisveronderstelling van virtueel geheugen, dat slechts enkele stukken van een proces nodig zijn in elke tijdsperiode, in de praktijk wordt bevestigd.



Figuur 4.31

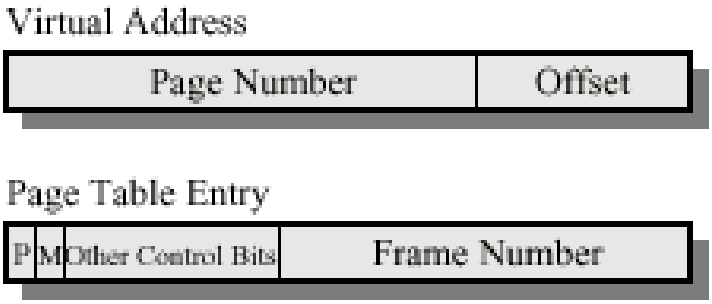


Figuur 4.32

Zowel objectgeoriënteerde technieken, als toepassingen met multithreading bevorderen echter het gebruik van veel kleine programma- en gegevensmodules, en hebben de neiging de lokaliteit van verwijzingen binnen een proces te verminderen.

4.4.1 Virtueel geheugen met paginering

Aan elk proces wordt een unieke paginatabel gekoppeld en in het hoofdgeheugen geladen. Het besturingssysteem is verantwoordelijk voor een consistente inhoud van de paginatabel. Elke ingang in de paginatabel bevat ruimte voor het framenummer van de bijbehorende pagina in het hoofdgeheugen (figuur 4.33). Bovendien geeft een bit ('P', van *present*) aan of de pagina al dan niet aanwezig is in het hoofdgeheugen. Een tweede bit ('M', van *modified*, meestal *dirty bit* genoemd) geeft aan of de inhoud van de bijbehorende pagina gewijzigd is sinds de pagina het laatst in het hoofdgeheugen werd geladen. Dit maakt het mogelijk om te weten of het al dan noodzakelijk is om de pagina weg te schrijven, wanneer de pagina moet worden vervangen. Nog aanvullende besturingsbits kunnen gebruikt worden indien bescherming of *sharing* worden beheerd op paginaniveau. Elke pagina kan op die manier verschillende beveiligingsniveaus toelaten. In Windows bijvoorbeeld geeft een controlebit voor elke pagina aan welke processormodus vereist is om toegang te hebben tot de pagina. Een schending van de ingestelde geheugenbescherming veroorzaakt een trap naar het besturingssysteem.

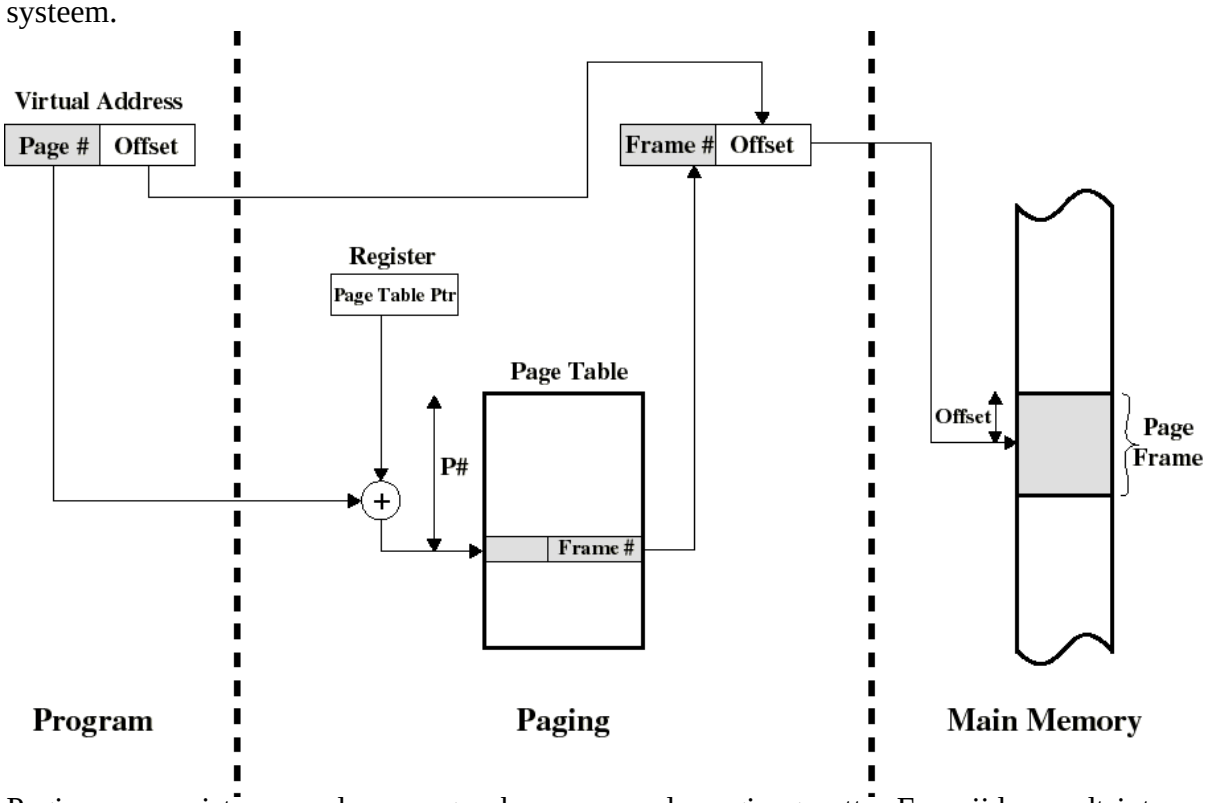


Figuur 4.33

Het schijfadres dat gebruikt wordt om de pagina te bewaren, wanneer deze niet in het geheugen staat, maakt daarentegen geen deel uit van de paginatabel: de paginatabel bevat alleen de informatie die de hardware nodig heeft om een virtueel adres te vertalen in een fysiek adres. Informatie die het besturingssysteem nodig heeft om paginafouten af te

handelen, staat in andere tabellen binnen het besturingssysteem. De paginatabel moet steeds toegankelijk zijn in het hoofdgeheugen (tenzij het volledige procesbeeld uitgeswapt zou zijn).

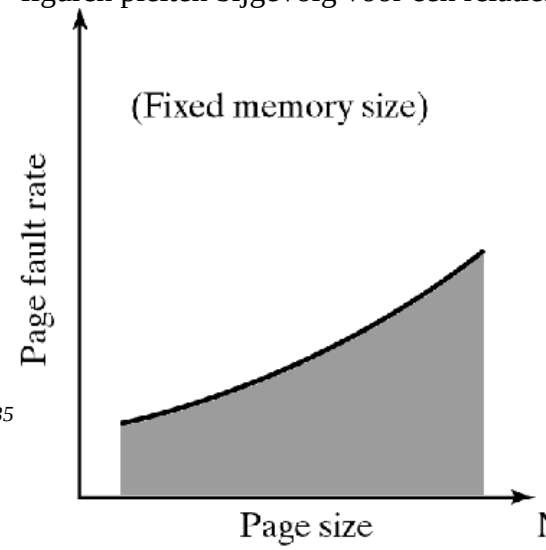
Om een systeem voor virtueel geheugen met paginering te kunnen toepassen is hardware-ondersteuning vereist. Tijdens uitvoering van het proces bevat een register van de processor, het *Page Table Base Register* (PTBR), het beginadres van de paginatabel voor dat proces. Een ander register, het *Page Table Length Register* (PTLR), kan aangeven hoe groot de paginatabel is. Het is opnieuw de verantwoordelijkheid van het besturingssysteem om er voor te zorgen dat deze registers correct ingevuld worden, vanuit de processortoestandsinformatie in het PCB blok. Het paginanummer van het virtueel adres (de meest linkse bits) wordt door de MMU gebruikt als een index in de paginatabel voor het opzoeken van het corresponderende framenummer (figuur 4.34). Dit wordt gecombineerd met het offsetdeel van het virtuele adres om het gewenste reële adres te vinden. De enige aanvullende operatie die de MMU moet uitvoeren ten opzichte van een systeem zonder virtueel geheugen, is het controleren van de 'P' bit. Is de pagina niet aanwezig, of wordt een paginanummer gerefereerd groter dan het PTLR aangeeft, dan geeft een interrupt de controle over aan het besturingssysteem.



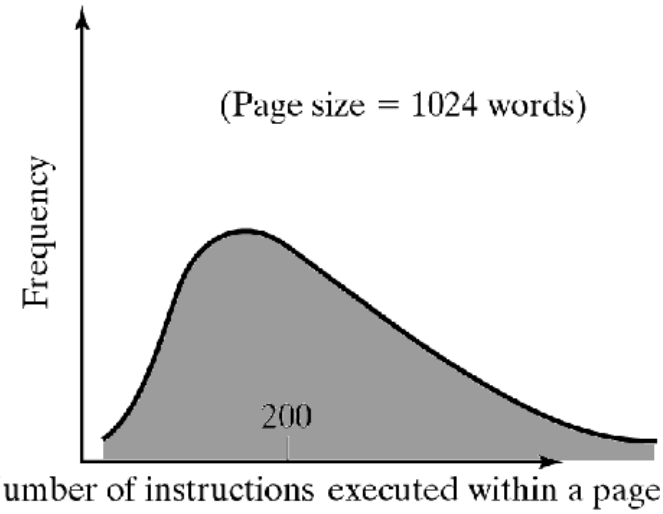
Figuur 4.34

Pagineren vereist een weloverwogen keuze voor de paginagrootte. Enerzijds wordt interne fragmentatie beperkt door kleinere pagina's te gebruiken. Anderzijds vereist een kleine paginagrootte een groot aantal pagina's per proces, en bijgevolg grotere paginatabellen, waardoor minder geheugen voor het proces overblijft. Grotere pagina's genieten ook de voorkeur voor een meer efficiënte blokoverdracht van gegevens uit secundaire opslag. De tijd die nodig is voor een gegevensoverdracht hangt immers meer van het aantal aanvragen af, dan van de hoeveelheid data. Bij een constante hoeveelheid toegewezen geheugen, heeft de paginagrootte invloed op de frequentie waarmee paginafouten optreden: is de paginagrootte klein, dan zijn een groot aantal pagina's voor het proces beschikbaar in het hoofdgeheugen. Naarmate de grootte van de pagina toeneemt (en het aantal toegewezen pagina's verkleint), begint de paginafoutfrequentie op te lopen (figuur 4.35). Het blijkt eveneens dat

gemiddeld minder dan 200 instructies uit dezelfde pagina kunnen uitgevoerd worden (figuur 4.36), vooraleer instructies uit andere pagina's moeten opgehaald worden. Beide figuren pleiten bijgevolg voor een relatief kleine paginagrootte.



Figuur 4.35



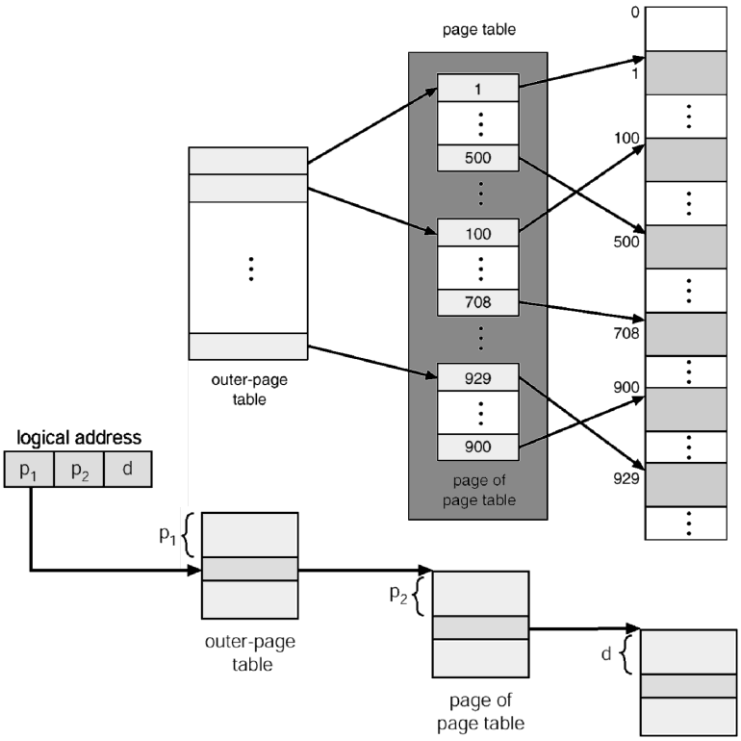
Figuur 4.36

In de praktijk wordt meestal een paginagrootte in de orde kilobyte gekozen, gaande van 0.5 kB (IBM AS/400) tot 8 kB (Alpha). Recentere systemen tonen een tendens naar grotere pagina's en ondersteunen zelfs simultaan meerdere paginagrootten.

4.4.2 Structuur van paginatabellen

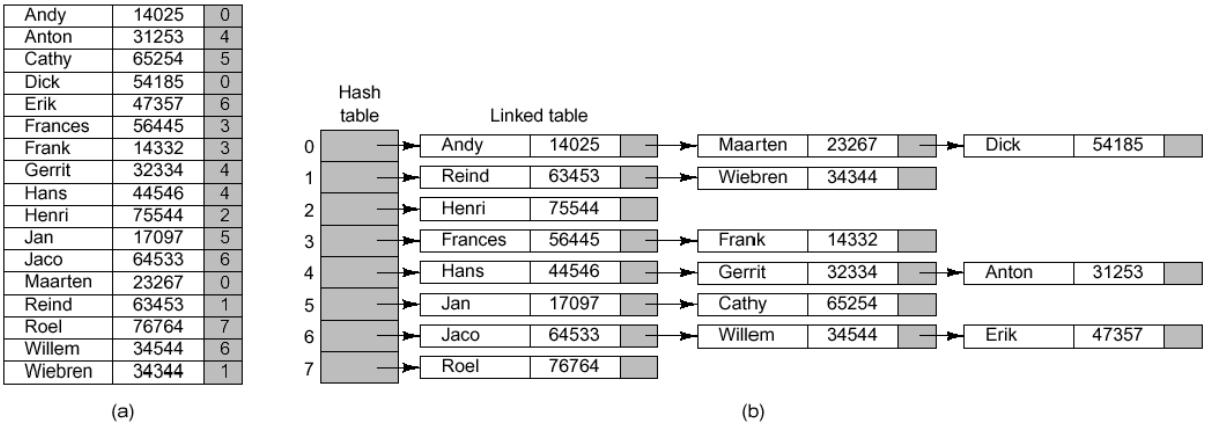
In veel computersystemen is de virtuele adresruimte die elk proces kan aanspreken groter dan de beschikbare hoeveelheid reëel geheugen. Elk gebruikersproces in Windows ziet bijvoorbeeld een afzonderlijke 32-bits adresruimte, wat 4 GB geheugen per proces mogelijk maakt (waarvan 1 of 2 GB, afhankelijk van de configuratie, is voorbehouden voor het besturingssysteem; hiervan kan een gedeelte, de *nonpaged pool* niet uitgepagineerd worden). Hierdoor wordt de hoeveelheid geheugen die per proces alleen al voor paginatabellen vereist is te groot om in het hoofdgeheugen te kunnen worden geladen. Hiervoor bestaan twee oplossingen:

1) Een aantal systemen voor virtueel geheugen, zoals de 80386, slaan de paginatabellen zelf ook op in virtueel geheugen, in plaats van in het hoofdgeheugen. Hiervoor kan een systeem met twee niveaus, *forward-mapped page table* genoemd (figuur 4.37), gebruikt worden. Het paginanummer in het logische adres wordt hierbij opgesplitst in twee segmenten: het meest significante segment is een index voor de buitenste (steeds residente) paginatabel, en verwijst naar een specifieke pagina (deelverzameling) van de paginatabel. Het minst significante deel bevat een index in die pagina, en verwijst naar de fysieke frame. Bij een verwijzing naar het geheugen kan dan een dubbele paginafout optreden: één voor het binnenhalen van het benodigde deel van de paginatabel, en een tweede voor het binnenhalen van de procespagina. Dit systeem met hiërarchische paginatabellen kan uitgebreid worden tot meerdere niveaus. De SPARC van Sun ondersteunt bijvoorbeeld drie niveaus, terwijl op de Motorola 68030 processor het aantal niveaus programmeerbaar is van 0 tot 4. Verder is het aantal bits op elk niveau ook nog instelbaar. Voor systemen met een 64-bits logische adresruimte voldoet zelf paginering met meerdere niveaus niet langer. De UltraSPARC zou bijvoorbeeld zeven niveaus paginering nodig hebben, met een veel te groot aantal geheugenbenaderingen voor de vertaling van elk logisch adres tot gevolg.



Figuur 4.37

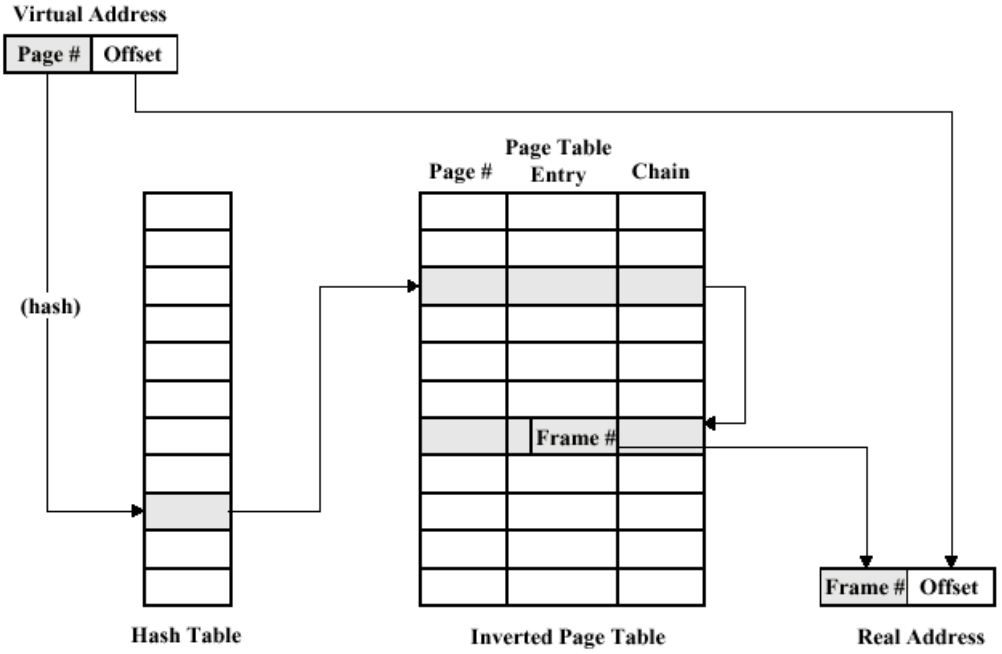
2) Andere systemen, zoals de IA-64, de UltraSPARC en de PowerPC, gebruiken één enkele *geïnverteerde paginatabel* voor het ganse systeem, in plaats van een paginatabel voor elk individueel proces. Een geïnverteerde paginatabel bevat een ingang per geheugenframe, in plaats van per virtuele pagina. Daardoor is een vaste grootte vereist voor de tabel, enkel bepaald door de hoeveelheid geheugen, ongeacht het aantal processen en het aantal virtuele pagina's. Omdat de geïnverteerde paginatabel niet is gesorteerd op paginanummer, maar op framenummer, en om te vermijden dat het paginanummer sequentieel moet opgezocht worden, past men *hashing* toe. Figuur 4.38 illustreert het algemene concept van hashing: een verzameling van een aantal, *N*, elementen *A*[*i*], moet worden opgeslagen in een gegevens-structuur.



Figuur 4.38

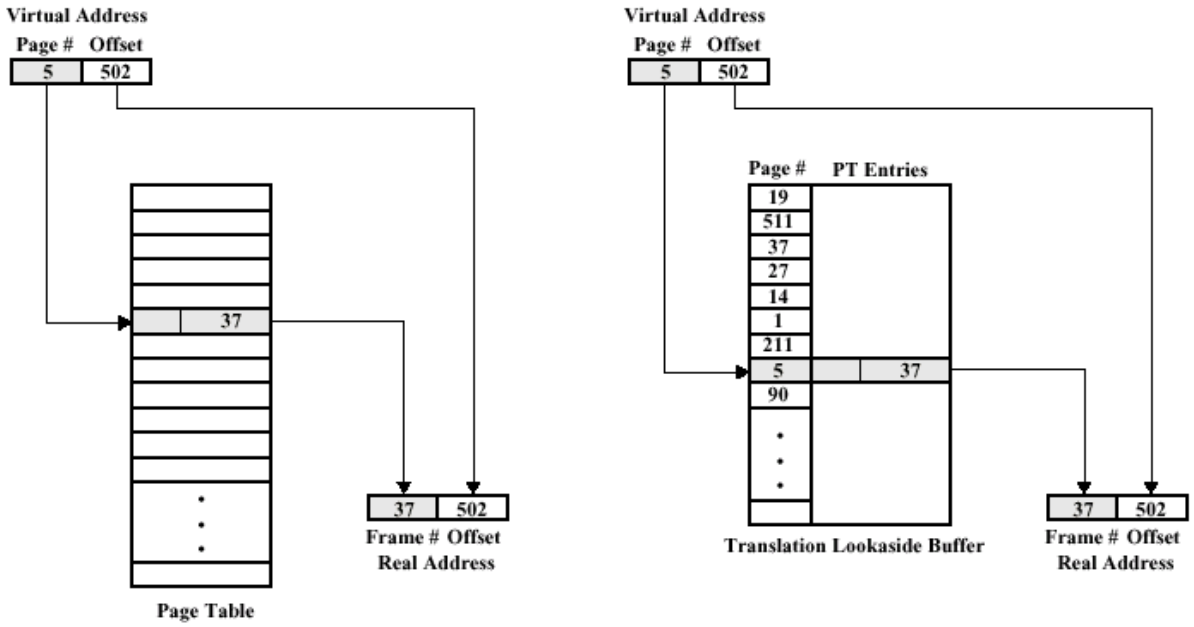
Elk element *A*[*i*] bestaat uit een label k_i (in figuur 4.38a, een inschrijvingsnummer) en aanvullende informatie (in figuur 4.38a, de voornaam van de student). Een eenvoudige vertaalfunctie *I*(*k*) converteert labels naar een *hashwaarde*, een vrijwel willekeurig getal *n* tussen 0 en *M-1* (in het voorbeeld is *M*=8). Het is nu de bedoeling de elementen op te slaan in één van *M* gekoppelde lijsten. Een *hashtabel* met lengte *M* verwijst naar de diverse gekoppelde lijsten van elementen met telkens dezelfde hashwaarde. Om de gegevens op te zoeken behorend bij een bepaald label *k*, wordt de hashwaarde ervan, *I*(*k*), gebruikt als index

in de hashtabel, en worden de labels van alle elementen in de corresponderende gekoppelde lijst, sequentieel vergeleken met de gezochte waarde *k*.
Toegepast op virtueel geheugenbeheer, wordt de combinatie van het paginanummer van een virtueel adres en de procesidentificatie (de *Address Space Identifier*, ASID) vertaald naar de hashwaarde, die gebruikt wordt als een index in een hashtabel. Deze tabel bevat een verwijzing naar één van de elementen in de geïnverteerde paginatabel (figuur 4.39) die de specifieke hashwaarde opleveren. Alle elementen met dezelfde hashwaarde zijn in de geïnverteerde paginatabel aan elkaar gekoppeld in een circulaire lijst. Het framenummer in figuur 4.39 kan berekend worden op basis van het adres van het element (ten opzichte van het beginadres van de geinverteerde paginatabel), en hoeft bijgevolg niet expliciet opgeslagen te worden. Hoe groter de hashtabel, hoe kleiner doorgaans de gekoppelde lijsten. Dikwijls heeft een hashtabel evenveel ingangen als de geïnverteerde paginatabel zelf. Zonder extra voorzieningen kunnen systemen met geïnverteerde paginatabellen, gedeeld geheugengebruik niet implementeren, omdat er maar plaats is voor één paginanummer voor elke frame.



Figuur 4.39

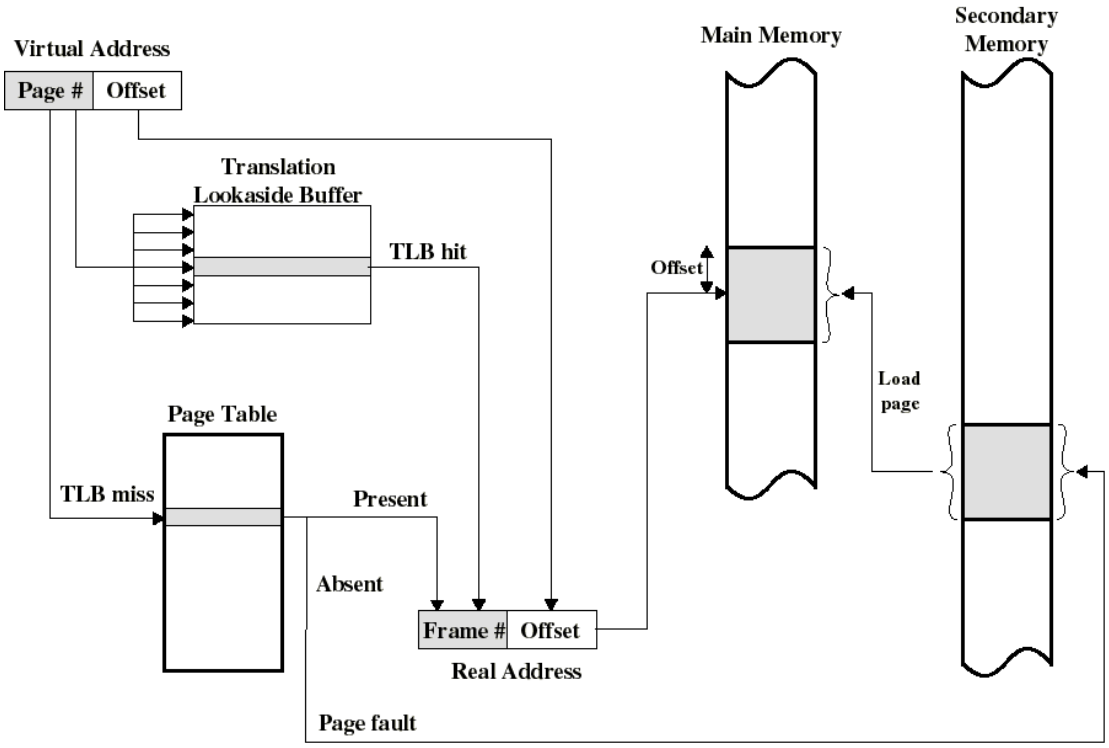
3) In beide technieken veroorzaakt elke verwijzing naar een virtueel adres minimaal twee geheugentoegangen: één voor het opvragen van de juiste paginatabelingang, en één voor het opvragen van de gewenste gegevens. Om een verdubbeling van de geheugentoegangstijd te vermijden, gebruiken de meeste virtuele geheugensystemen een hardwarecache in de MMU die de meest recent gebruikte paginatabelingangen bevat: de *translation lookaside buffer* (TLB). Die is gebaseerd op het feit dat de meeste programma's meestal verwijzen naar een klein aantal pagina's. Daarbij wordt slechts een klein aantal paginatabelelementen dikwijls gelezen, terwijl de rest nauwelijks gebruikt wordt. De processor is hierbij uitgerust met hardware die het mogelijk maakt om tegelijkertijd meerdere TLB-ingangen te onderzoeken. Deze techniek wordt *associatieve vertaling* (figuur 4.40b) genoemd, en is de tegenhanger van indexering of directe vertaling (figuur 4.40a). Meestal is het aantal ingangen in een TLB klein, tussen 64 en 1024. Toch zorgt het lokaliteitsprincipe ervoor dat dit dikwijls genoeg is om meer dan 90 % van alle gevraagde paginatabelingangen in de TLB terug te vinden.



Figuur 4.40

Bij vertaling van een virtueel adres wordt eerst de TLB onderzocht (figuur 4.41):

- Is de gewenste paginatabelingang aanwezig, dan wordt aan de hand van het framenummer het reëel adres onmiddellijk gevormd.
- Is de gewenste paginatabelingang niet beschikbaar in de TLB, dan gebruikt de processor het paginanummer als index voor de procespaginatabel, en wordt de corresponderende paginatabelingang onderzocht. Is de bit 'P' ingeschakeld, dan kan het reëel adres in het hoofdgeheugen gevormd worden. Indien niet, dan wordt via een paginafout het besturingssysteem aangeroepen, die de pagina laadt en de paginatabel bijwerkt.



Figuur 4.41

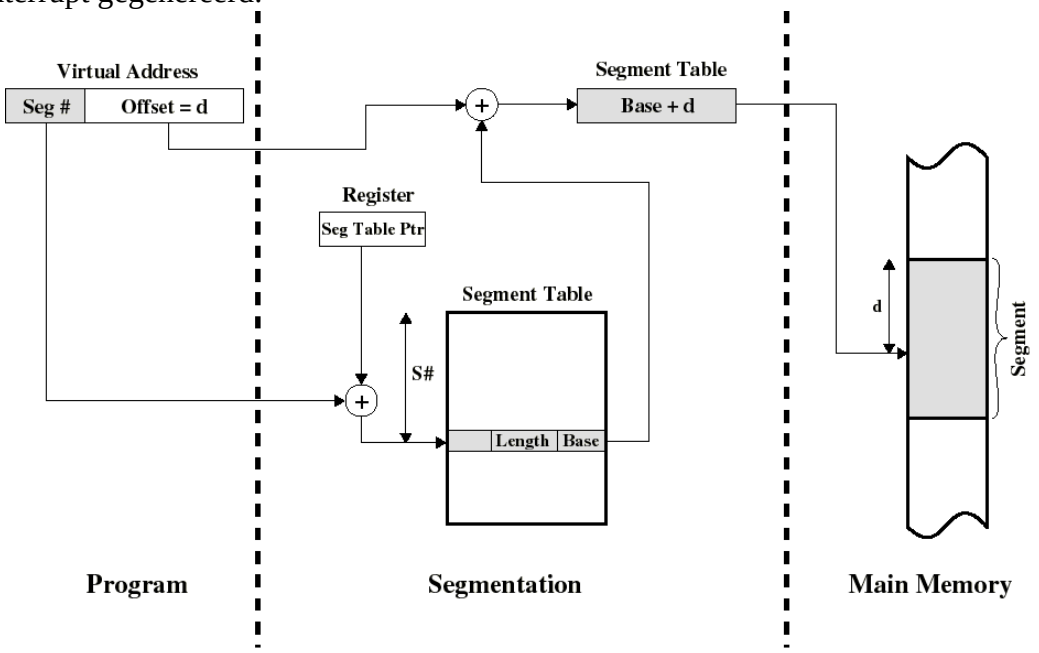
Indien de paginatabelingang via de paginatabel moest worden opgezocht, dan wordt één van de elementen uit de TLB vervangen door het element dat zojuist werd opgezocht, zodat het sneller kan worden gevonden bij een eventuele volgende verwijzing. Bij sommige TLB's kunnen ingangen vastgelegd worden, zodat ze niet kunnen verwijderd worden uit de TLB. Meestal gebeurt dit bij verwijzingen naar kernelcode. Sommige TLB's identificeren elke ingang met een identificatie van het proces (de ASID), en kunnen hierdoor ingangen bevatten voor meer processen tegelijkertijd. Als de TLB geen ASID's ondersteunt, dan moet de TLB worden leeggemaakt bij elke proceswisseling. Door de volledige adresruimte van de kernel expliciet op te nemen in de adresruimte van elk individueel gebruikersproces, vermijden besturingssystemen zoals Linux en NT dat de TLB ook zou moeten leeggemaakt worden bij elke contextwisseling (ondermeer bij elke systeemaanroep).

4.4.3 Virtueel geheugen met segmentatie

Bij een virtueel geheugensysteem met segmentatie wordt aan elk proces een unieke segment-tabel verbonden. Net als bij virtueel geheugen met paginering geven de 'P' en 'M' bits aan (figuur 4.42) of het segment zich in het hoofdgeheugen bevindt, en is gewijzigd sinds het in het hoofdgeheugen werd geladen. De ingang bevat naast het absolute beginadres ook de lengte van het segment. Tijdens uitvoering van het proces bevat een register van de processor het beginadres van de segmenttabel voor dat proces.

Figuur 4.42

Het segmentnummer van het virtueel adres (de meest linkse bits) wordt gebruikt als een index in de segmenttabel, voor het opzoeken van de 'P' bit, de grootte en het geheugenadres van het begin van het corresponderende segment (figuur 4.43) . Dit laatste wordt opgeteld bij het offsetdeel van het virtuele adres om het gewenste reële adres te vinden. Indien het segment niet resident is, of het offsetdeel is groter dan de grootte van het segment, dan wordt een interrupt gegenereerd.



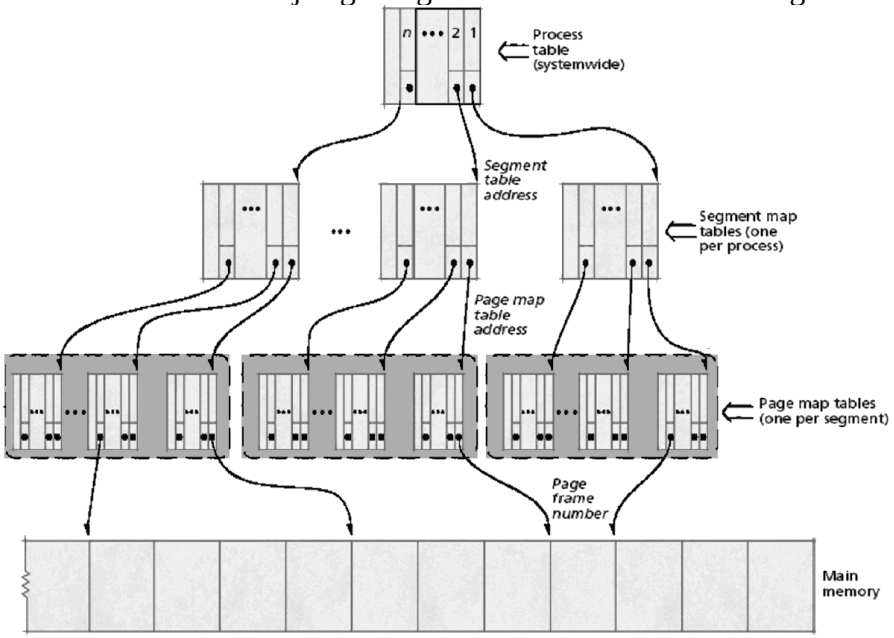
Figuur 4.43

De hardware controleert eveneens de beschermingsbits die in de segmenttabel zijn opgenomen, om illegale geheugentoegang te voorkomen. Sommige programmeerfouten zullen dus door hardware gedetecteerd worden vooraleer ze schade kunnen berokkenen.

4.4.4 Virtueel geheugen met paginering en segmentatie

Virtueel geheugen met segmentatie vereenvoudigt het afhandelen van groeiende structuren: bevindt een segment dat moet worden uitgebreid zich ergens in het hoofdgeheugen, en is er daar onvoldoende ruimte, dan kan het besturingssysteem het segment verplaatsen naar een groter gebied in het hoofdgeheugen, als dat beschikbaar is, of kan het segment uit het hoofdgeheugen geswapt en teruggeswapt worden van zodra dat mogelijk is. Virtueel geheugen met paginering daarentegen kent een aantal andere sterke punten, die reeds in §4.3.2 aan bod kwamen: onzichtbaarheid voor de programmeur en eliminatie van externe fragmentatie. Bovendien zijn de algoritmen voor geheugenbeheer, die in §4.5 zullen beschreven worden, eenvoudiger te implementeren indien virtueel geheugen met paginering gebruikt wordt.

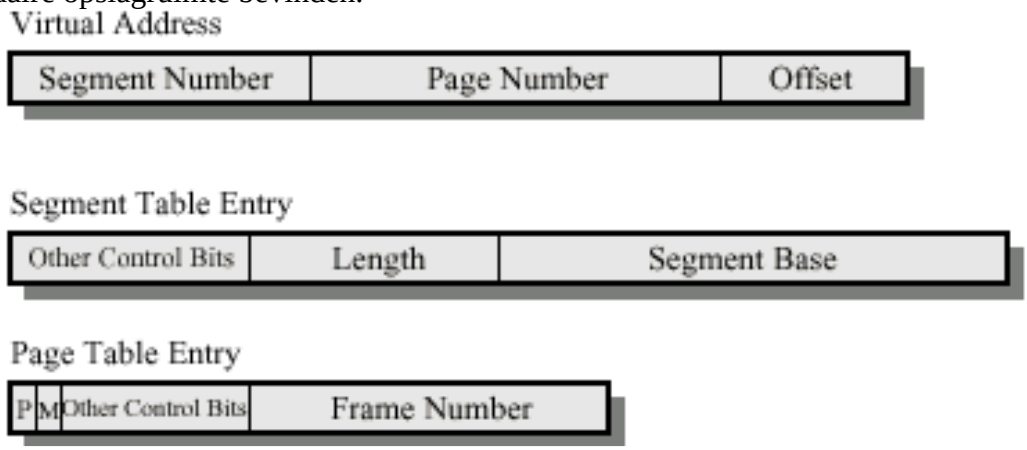
Om de voordelen van beide systemen te combineren, worden de meeste moderne systemen uitgerust met processorhardware en besturingssystemen die virtueel geheugen implementeren met zowel paginering als segmentatie (figuur 4.44). Systemen met uitsluitend segmentatie, zoals de 80286, komen praktisch niet meer voor. In een gecombineerd paginering-segmentatie virtueel geheugensysteem wordt de adresruimte van een proces opgebroken in enkele segmenten, die gekozen worden door de programmeur. Elk segment wordt verdeeld in pagina's met een vaste grootte, gelijk aan de grootte van een frame in het hoofdgeheugen. Vanuit het standpunt van het proces wordt hierdoor een geheugenbeeld behouden waarin het virtuele geheugen gesegmenteerd is. Vanuit het standpunt van het systeem daarentegen blijft het beeld van kracht dat het werkelijke geheugen uit frames met een vaste grootte bestaat.



Figuur 4.44

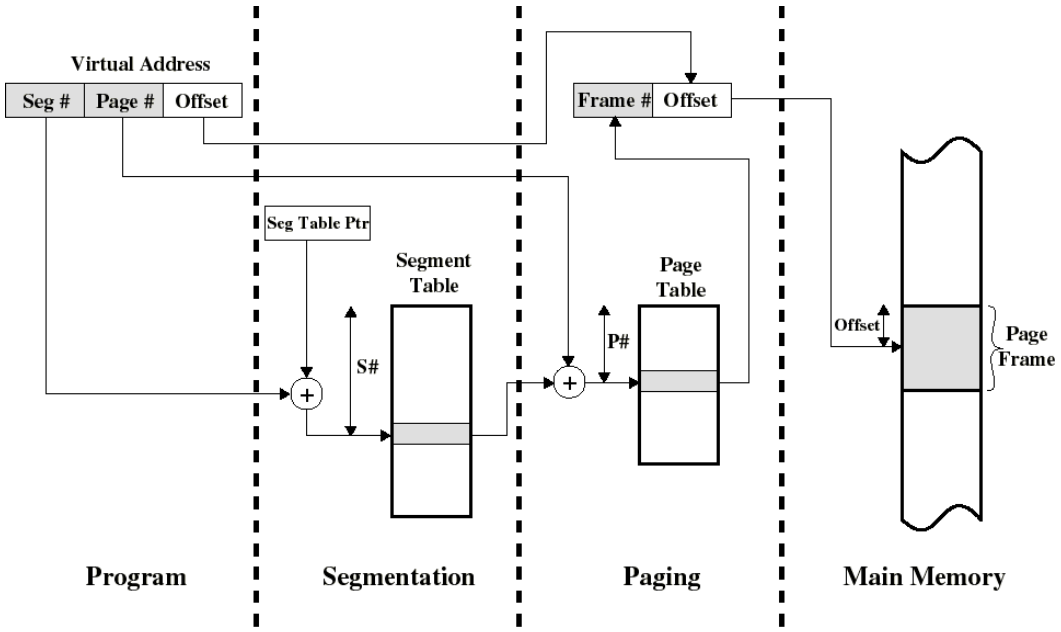
Een logisch adres bestaat uit een segmentnummer (meestal *selector* genoemd) en een segmentpositie, die op zijn beurt bestaat uit een paginanummer en een paginapositie. Met elk proces correspondeert een segmenttabel (of *descriptor table*), terwijl voor elk processegment een paginatabel, al dan niet in meerdere hiërarchische niveaus, gebruikt wordt. Figuur 4.45 toont de indeling van de segmenttabelingang en van de paginatabelingang. De bits 'P' en 'M' worden afgehandeld op paginaniveau, terwijl andere besturingsbits (voor bescherming en

sharing) meestal op segmentniveau worden ingesteld. Om de mogelijkheid te behouden om geheugen per frame toe te wijzen, wordt de eis niet gesteld dat alle pagina's van een segment in het hoofdgeheugen staan: sommige pagina's van een segment kunnen zich enkel in de secundaire opslagruimte bevinden.



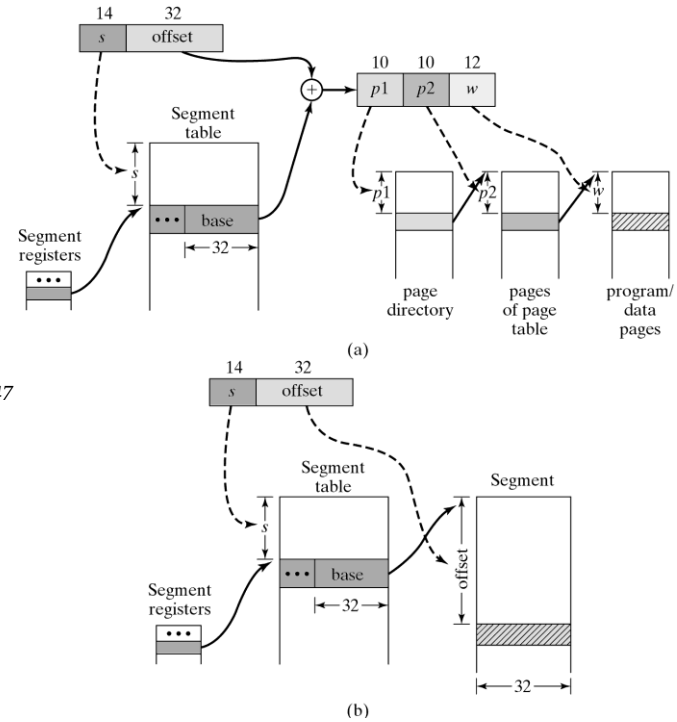
Figuur 4.45

Figuur 4.46 illustreert de corresponderende vertaling van virtuele in reële adressen. Hierbij wordt een paginatabel op slechts één niveau verondersteld. Men kan ook hier de techniek van TLB's en associatieve vertaling gebruiken om de combinatie van segmentnummer en paginanummer efficiënter te vertalen in framenummers.

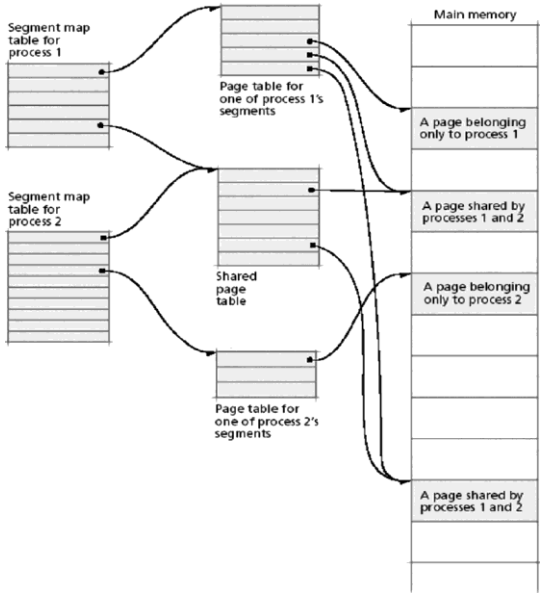


Figuur 4.46

Er zijn veel varianten mogelijk op het schema van figuur 4.46. Zo converteert de Pentium een 48-bits logisch adres eerst via een segmentabel naar een 32-bits *lineair adres* (figuur 4.47a), dat vervolgens geïnterpreteerd wordt in een pagineersysteem met twee niveaus van elk 1024 elementen. Elk proces kan tot 2^{13} individuele segmenten aanspreken. Bovendien zijn er nog 2^{13} segmenten die gedeeld worden door alle processen, ondermeer voor functies van het besturingssysteem (figuur 4.48). Ondermeer om compatibiliteit met de 80286 mogelijk te maken, ondersteunt de Pentium eveneens een systeem met zuivere segmentatie (figuur 4.47b): het lineair adres wordt hierbij onmiddellijk geïnterpreteerd als een fysiek adres.



Figuur 4.47



Figuur 4.48

Besturingssystemen zoals Linux, die ontworpen zijn om ondersteund te worden op uiteenlopende hardware architecturen, waarvan sommige slechts een beperkte ondersteuning voor segmentatie bieden, maken een minimaal gebruik van segmentatie. Zo gebruikt Linux slechts zes segmenten:

- één segment voor kernelcode,
- één segment voor kernelgegevens,
- één segment voor gebruikerscode, waarbij alle processen dezelfde adresruimte gebruiken,
- één segment voor gebruikersdata, opnieuw gedeeld door alle processen,
- een *Tast State Segment* (TSS) segment, individueel per proces, gebruikt om tijdens proceswisselingen de hardwarecontext op te slaan,
- het *Local Descriptor Table* (LDT) segment, standaard één globaal segment, alhoewel elk proces een eigen LDT kan maken.

Het gebruik van een zeer kleine segmenttabel heeft het voordeel dat die steeds in de processorregisters kan opgeslagen worden. Linux gebruikt tot drie niveaus voor de paginatabel van elk segment. Het middelste niveau wordt uitgeschakeld op hardware platforms, zoals de Pentium, die maar twee niveaus ondersteunen.

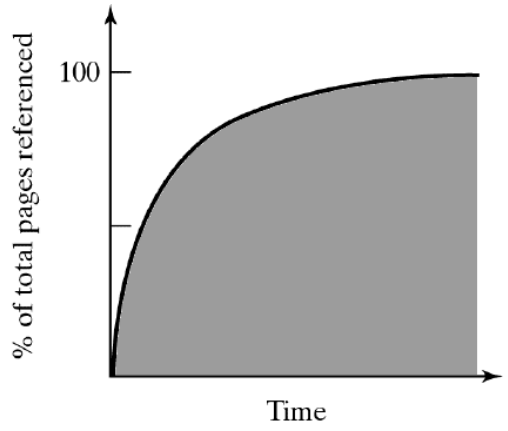
4.5 Strategieën voor geheugenbeheer

Net zoals tussen elke twee opeenvolgende lagen in de geheugenhiërarchie (§1.1.4), moeten beslissingen genomen worden in verband met de verplaatsing van pagina's of segmenten tussen het secundaire geheugen en het hoofdgeheugen. Op dit niveau van de hiërarchie is het besturingssysteem verantwoordelijk voor deze beslissingen en de uitwerking ervan. De gebruikte technieken moeten hierbij volledig transparant zijn voor de gebruiker. Aangezien in de meeste systemen hetzij zuivere paginering gebruikt wordt (Windows), hetzij segmentatie gecombineerd wordt met paginering (Linux), spitst het geheugenbeheer zich toe op de kwesties die samenhangen met pagineren, meer bepaald op het minimaliseren van de frequentie van paginafouten. Paginafouten leiden immers tot een aanzienlijke overhead van

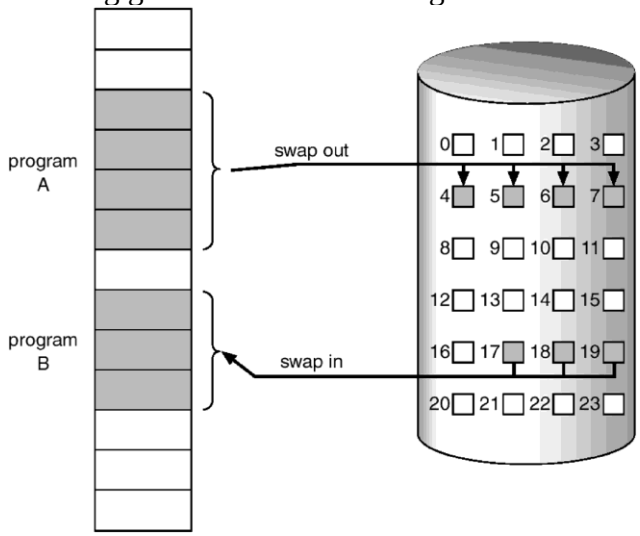
het besturingssysteem. De belangrijkste problemen hierbij worden in elk van volgende deelparagrafen besproken.

4.5.1 Ophaalstrategieën

Wanneer moet een pagina worden overgebracht naar het hoofdgeheugen ? Hiervoor zijn twee alternatieven. Bij *vraagpaginering* (*paging on demand*) wordt een pagina pas overgebracht naar het hoofdgeheugen indien hierom gevraagd wordt. Dit leidt meestal tot een groot aantal paginafouten onmiddellijk na opstarten van het proces (figuur 4.49), totdat elke benodigde pagina in het geheugen is geladen. Unix volgt deze strategie. Bij *prepaginering* (*prepaging* of *prefetching*) worden meerdere aaneengesloten pagina's, zoals die zijn opgeslagen in het secundaire geheugen, ineens binnengehaald (figuur 4.50), ook al is strikt genomen slechts één pagina vereist. Dat kan in sommige gevallen een voordeel opleveren. Het kan echter ook zijn dat veel van de extra pagina's niet gebruikt worden. NT volgt de prepaginering strategie, noemt het *clustering*, en laat het aantal pagina's afhangen van de fysieke hoeveelheid geheugen, en van de aard van de pagina (code of data): is bijvoorbeeld meer dan 64 MB geheugen aanwezig, dan worden 4 datapagina's of 8 codepagina's ineens ingelezen. Is minder geheugen aanwezig, dan wordt de clustering gereduceerd of zelfs uitgeschakeld.



Figuur 4.49



Figuur 4.50

4.5.2 Plaatsingsalgoritmen

Op welke plaats in het hoofdgeheugen moeten de pagina's worden geladen ? Deze vraag blijkt enkel relevant in zuivere segmentatiesystemen, en werd reeds behandeld in §4.2.1 (cfr. *Best-Fit*, *Next-Fit*, *First-Fit* en *Worst-Fit*).

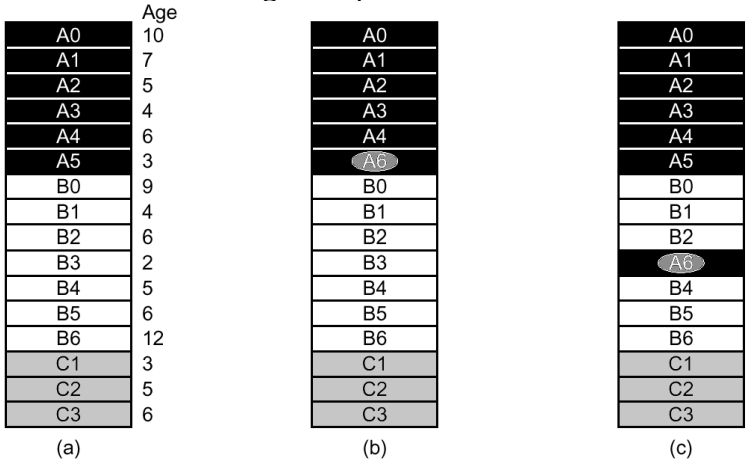
4.5.3 Vaste of variabele toewijzing

Hoeveel paginaframes moeten worden toegewezen aan elk actief proces ? Hoe kleiner de hoeveelheid geheugen is die wordt toegewezen aan elk proces, hoe groter het aantal processen dat zich tegelijkertijd in het hoofdgeheugen kan bevinden, en hoe groter de kans dat ten minste één proces zich in de toestand *gereed* bevindt. Een te klein aantal pagina's per proces zal echter de frequentie van paginafouten doen toenemen, terwijl boven een bepaalde drempelwaarde, *overtoewijzing* van hoofdgeheugen aan een bepaald proces geen merkbare verbetering meer geeft. Bovendien bevinden zich te weinig processen in het hoofdgeheugen wanneer de geheugenhoeveelheid per proces onnodig groot zijn. Twee mogelijke strategieën zijn mogelijk: *vaste toewijzing*, waarbij het aantal pagina's wordt toegekend bij creatie van

het proces, en *variabele toewijzing*, waarbij dynamisch paginaframes worden toegewezen of onttrokken aan processen om de paginafoutfrequentie te verlagen. Hierbij moet echter het gedrag van actieve processen worden beoordeeld, wat onvermijdelijk gepaard gaat met overhead. Een ander probleem bij een variabele toewijzing is dat een proces heel verschillend kan presteren, alleen al door externe omstandigheden (de paginafouten veroorzaakt door andere processen). Toch leidt variabele toewijzing meestal tot een grotere globale doorvoer van het systeem, en is het daarom gebruikelijker.
Vooral in systemen met vaste toewijzing (maar ook in systemen met variabele toewijzing, zoals Unix en Windows) maakt men dikwijls gebruik van de *copy-on-write* techniek. Als een kindproces wordt aangemaakt door het ouderproces (bijvoorbeeld via een *fork()* variant, §2.6.1), dan delen de processen aanvankelijk dezelfde pagina's: de pagina's worden gemapt naar hetzelfde frame. Hierdoor kan een kindproces snel worden aangemaakt. Pas indien één van de processen naar een gedeelde pagina schrijft, wordt een individuele kopie van de gedeelde pagina gemaakt. Men minimaliseert hierdoor het aantal nieuwe pagina's dat moet toegewezen worden: alleen de pagina's worden gekopieerd die door een proces worden gewijzigd.

4.5.4 Lokale of globale vervanging

Moet de verzameling pagina's die in aanmerking komt voor vervanging, wanneer een nieuwe pagina moet worden binnengehaald, *lokaal* worden beperkt tot de pagina's van het proces dat de paginafout heeft veroorzaakt, of *globaal* alle paginaframes in het hoofdgeheugen bevatten, ongeacht welk proces een bepaalde pagina bezit ? Veronderstel bijvoorbeeld dat in het voorbeeld van figuur 4.51a een paginafout wordt gegenereerd door proces A, en dat de vervangingsstrategie van het besturingssysteem gebaseerd is op het uitswappen van de pagina's met de kleinste leeftijdswaarde: figuur 4.51b illustreert de lokale paginavervanging, terwijl figuur 4.51c de globale benadering toont. Lokale benaderingen kunnen de gevolgen van trashing beperken: een proces kan, als het zelf begint te trashen, geen frames afnemen van een ander proces, zodat dat ook gaat trashen. Bij vaste toewijzing van het aantal pagina's per proces wordt enkel een lokale strategie gebruikt. Bij variabele toewijzing daarentegen kunnen zowel globale als locale strategieën optreden:

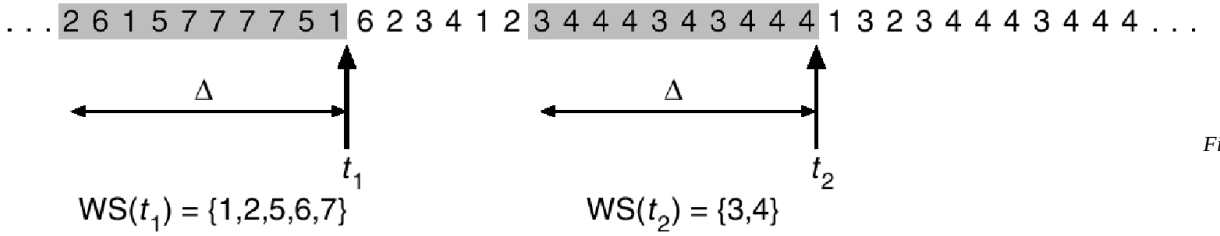


Figuur 4.51

1) De combinatie variabele toewijzing met een globale vervangingsstrategie, ondermeer gebruikt in UNIX, is het meest eenvoudig te implementeren, omdat ze tegelijkertijd de aan elk proces toegewezen hoeveelheid hoofdgeheugen dynamisch regelt (§4.5.3). Meestal, doet het besturingssysteem hierbij een beroep op een techniek die *paginabuffering* wordt genoemd. Paginabuffering wordt echter ook, zoals in Windows, gebruikt bij de combinatie variabele toewijzing met een locale vervangingsstrategie.

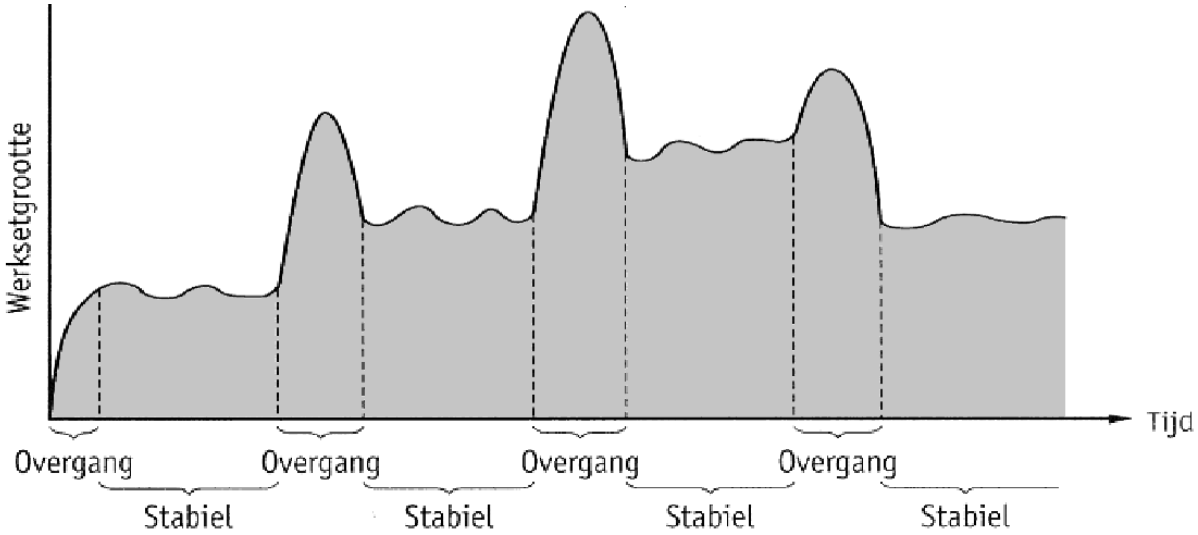
Parallel met de paginatabellen per proces worden op systeemniveau twee intermediaire lijsten van *slachtofferframes* bijgehouden: *de lijst van vrije frames*, en de *lijst van vrije frames met gewijzigde pagina's*, die als cache werken voor te vervangen pagina's. Beide lijsten verwijzen naar de eerstvolgende frames die zullen gebruikt worden voor het inlezen van nieuwe pagina's. Beslist het besturingssysteem dat een pagina moet worden uitgeswapt, dan wordt de pagina nog niet onmiddellijk fysiek verwijderd of verplaatst naar het secundaire geheugen. In de plaats daarvan wordt enkel de ingang in de paginatabel voor deze pagina verwijderd, en achteraan één van de lijsten slachtofferframes geplaatst, al naargelang de pagina gewijzigd is of niet gedurende het verblijf in het hoofdgeheugen. Moet een extra pagina worden ingelezen, dan wordt het frame aan het begin van één van de lijsten slachtofferframes gebruikt. De corresponderende pagina die zich daar bevindt wordt dan pas effectief vernietigd. Uiteraard doet men liever geen beroep op de lijst van vrije frames met gewijzigde pagina's, aangezien deze eerst nog naar de schijf moeten weggeschreven worden, en de benodigde tijd voor de afhandeling van de paginafout hierdoor sterk verhoogd wordt. Het belangrijkste voordeel van de intermediaire lijsten is dat de pagina die moet vervangen worden tijdelijk in het geheugen behouden blijft. Verwijst een proces dan toch noch naar een dergelijke pagina, dan kost het terugplaatsen van de pagina daardoor weinig tijd, zonder I/O. De schade blijft beperkt wanneer aanvankelijk het verkeerde slachtoffer werd geselecteerd.

2) De combinatie variabele toewijzing met een locale vervangingsstrategie selecteert de te vervangen pagina uit de residente set van het actieve proces dat de fout heeft veroorzaakt. Hierdoor moeten alternatieve technieken gebruikt worden om de waarschijnlijke toekomstige behoeften, meer bepaald de ideale grootte van de residente set, van deze actieve processen in te schatten. Hiervoor wordt soms (zoals bijvoorbeeld in Windows) de *werksetbenadering* toegepast.



Figuur 4.52

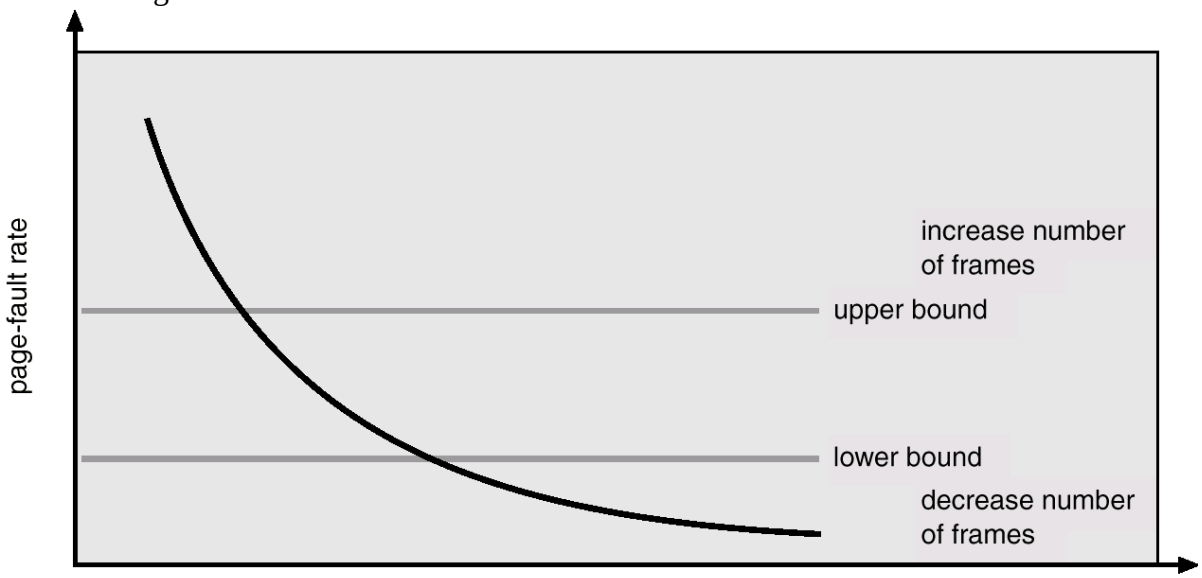
Op elk ogenblik *t* is de *werkset W*(*t*,Δ) met parameter Δ (*venstergrootte*) van een proces, de verzameling van pagina's van dat proces waarnaar is verwezen in het laatste tijdsinterval Δ (figuur 4.52), gemeten in tijd die de processor werkelijk aan het proces heeft besteed. De werkset is de verzameling pagina's die nodig is voor de uitvoering van een proces gedurende een bepaalde periode. Als deze set in het hoofdgeheugen staat, treden er geen paginafouten op. Treedt een paginafout op, dan is de werkset blijkbaar aan het veranderen. Het betekent echter niet dat een andere pagina bestaat waaraan geen behoefte meer is. De werksetgrootte is een niet-dalende functie van de venstergrootte Δ, en varieert bij een vaste waarde van Δ in de loop van de tijd: doorgaans wisselen overgangsperioden vrij stabiele perioden af (figuur 4.53). In de overgangsperioden bevat de werkset pagina's die in de beide aangrenzende stabiele perioden aangesproken worden.
In de werksetbenadering wordt een constante waarde Δ vastgelegd, en wordt de grootte van de werkset van elk proces bewaakt. Periodiek worden de pagina's uit de residente set van een proces die niet voorkomen in de werkset, verwijderd: er staan pagina's in het geheugen die daar niet hoeven te staan, en het proces bezet ruimte die een ander proces beter zou kunnen gebruiken.



Figuur 4.53

Een proces wordt anderzijds enkel efficiënt uitgevoerd als de in het hoofdgeheugen residente set de werkset kan bevatten: zijn er te weinig frames, dan zijn er veel meer geheugenverwijzingen die paginafouten veroorzaken. Indien de som van de werksetgroottes van alle processen groter is dan het totaal aantal beschikbare frames, dan treedt trashing op (cfr. figuur 4.31), omdat sommige processen niet genoeg frames kunnen krijgen. Het besturingssysteem kan dan best beslissen om één of meerdere processen volledig uit het geheugen te swappen. De werksetstrategie kan bijgevolg automatisch en dynamisch het aantal actieve programma's en hun geheugenverbruik beheren, en kan bovendien *trashing* vermijden.

Het nauwkeurig meten van de werkset voor elk proces is niet praktisch, en vraagt veel overhead. Daarom wordt soms, ter vervanging van de werksetgrootte, de paginafout-frequentie van elk proces bewaakt (cfr. figuur 4.54): daalt deze onder een drempelwaarde, dan kunnen paginaframes aan andere processen toegekend worden, zonder dat het proces in kwestie daaronder lijdt. Wordt de paginafoutfrequentie te hoog, dan heeft het proces meer frames nodig.



Figuur 4.54

Windows gebruikt nog een andere heuristiek om een raming te bekomen voor de werkset-grootte van elk proces. De werksetgrootte kan hierbij schommelen tussen twee drempels, die afhankelijk van de hoeveelheid geheugen ingesteld worden (ergens tussen 20 en 345 frames). Telkens een paginafout optreedt, wordt de werksetgrootte van dat proces verhoogd, op

voorwaarde dat de bovendrempel niet overschreden wordt. Pas indien de werksetgrootte van een proces de bovendrempel bereikt, leidt een paginafout tot een paginavervanging. Indien de lijst van vrije frames te weinig elementen bevat, dan krijgen alle processen van de *Working Set Manager* de opdracht een aantal pagina's vrij te geven, en hun werksetgrootte aan te passen. Het precieze aantal is hierbij afhankelijk van diverse criteria.

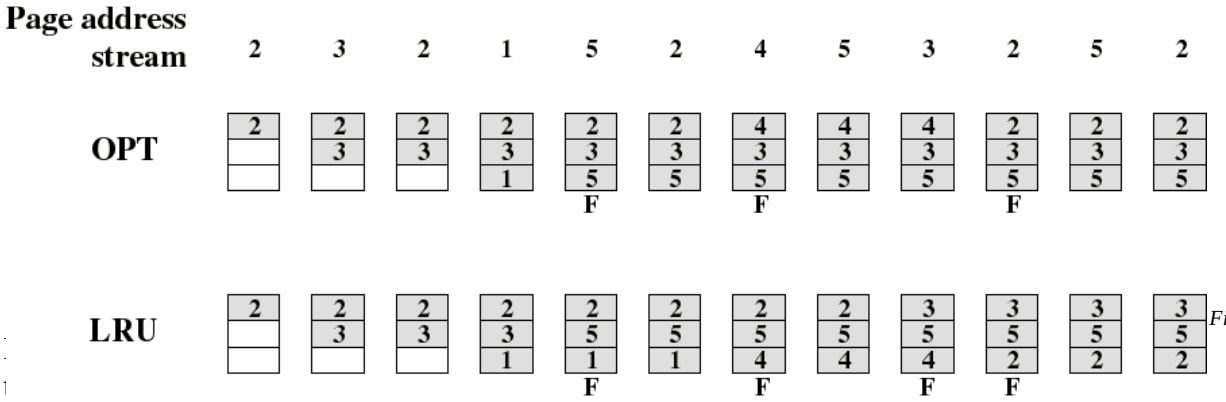
4.5.5 Wegschrijfstrategieën

Wanneer moet een gewijzigde pagina worden weggeschreven naar het secundaire geheugen ? Ook hier zijn twee gangbare strategieën. Bij *vraagopschoning* (*cleaning on demand*) wordt een pagina alleen weggeschreven naar het secundaire geheugen wanneer deze is geselecteerd voor vervanging. Bij *opschoning vooraf* (*precleaning*) worden pagina's in batches weggeschreven (figuur 4.50), ook al zijn hun paginaframes nog niet aan vervanging toe. Dit leidt tot verspilling als de kans groot is dat de pagina's, later tijdens hun verblijf in het hoofdgeheugen, opnieuw worden gewijzigd. Daarom wordt dikwijls een compromis gehanteerd, gebruik makend van de reeds behandelde techniek van paginabuffering: de pagina's in de intermediaire lijst van vrije frames met gewijzigde pagina's, worden periodiek in batches weggeschreven en verplaatst naar de intermediaire lijst van ongewijzigde pagina's. De *Working Set Manager* van Windows bijvoorbeeld voert deze techniek uit van zodra er meer dan 300 pagina's in de intermediaire lijst van vrije frames met gewijzigde pagina's voorkomen, en reduceert dit aantal dan tot minder dan 150.

4.5.6 Vervangingsstrategieën

Welke specifieke pagina's in het hoofdgeheugen moeten worden vervangen wanneer een nieuwe pagina moet worden binnengehaald? Doel van alle strategieën is dat de pagina die wordt verwijderd, juist die pagina is waarvan de kans het kleinst is dat er in de nabije toekomst nog naar zal worden verwezen. De meeste strategieën proberen dit toekomstig gedrag te voorspellen aan de hand van het gedrag in het recente verleden. Hierbij mag niet uit het oog verloren worden dat de overhead die een vervangingsstrategie veroorzaakt tot een aanvaardbaar minimum beperkt moet blijven.

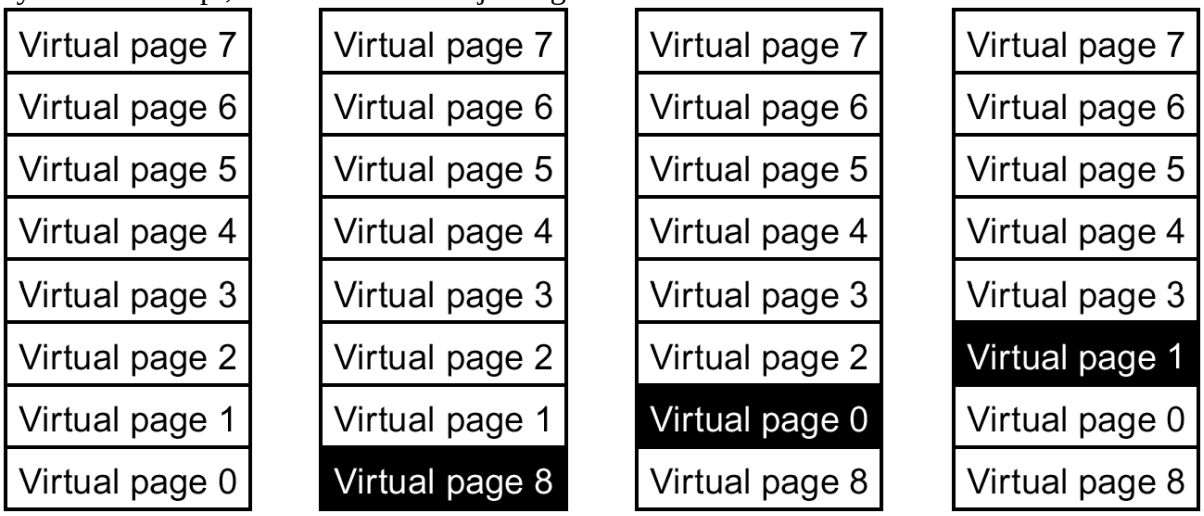
1) De *optimale strategie* vervangt de pagina waarvoor de tijdsduur tot de volgende vervanging het langst is. Aangezien het besturingssysteem hiertoe alle toekomstige gebeurtenissen zou moeten kennen, is deze strategie louter utopisch: de toekomstige activiteit hangt af van de interne logica van een proces, en van zijn data. In het vervolg van deze paragraaf zal deze strategie wel vermeld worden, om de resultaten van de andere, reële, strategieën ermee te vergelijken.



Figuur 4.55

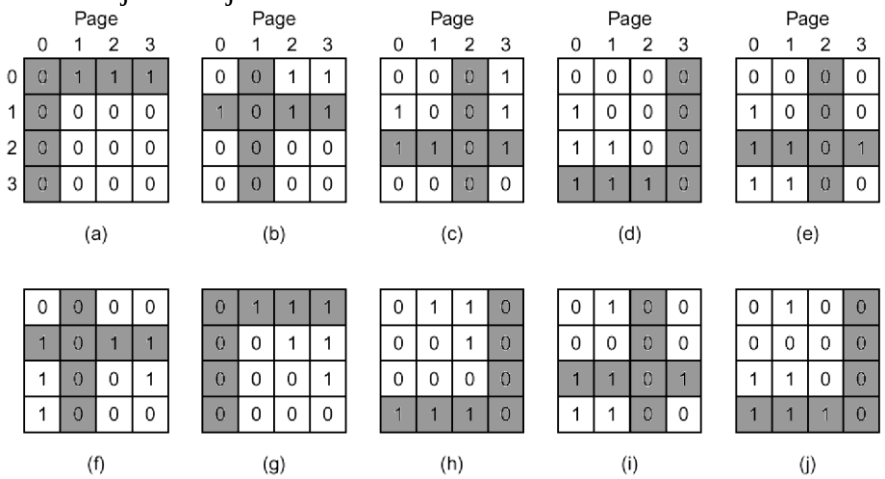
5, 3, 2, 5 en 2. Figuur 4.55 toont aan dat, indien de optimale strategie zou kunnen toegepast worden, het proces tijdens uitvoering slechts drie paginafouten zou genereren.

2) De *Last Recently Used* (LRU) strategie vervangt de pagina in het geheugen waarnaar het langst niet verwezen is. Pagina's waarnaar onlangs verwezen is, worden in het geheugen gehouden, omdat het wegens het beginsel van lokaliteit (§4.4) waarschijnlijk wordt geacht dat naar dergelijke pagina's in de nabije toekomst opnieuw zal verwezen worden. Ook hier zijn de resultaten van deze strategie, toegepast op het voorbeeldproces, in figuur 4.55 weergegeven. LRU lijkt in het voorbeeld een goede benadering voor de optimale strategie. In andere simulaties echter presteert LRU veel slechter. Hebben processen bijvoorbeeld een cyclisch verwijzingspatroon, waarbij het paginabereik groter is dan de hoeveelheid geheugen die toegewezen is aan het proces, dan swapt LRU de pagina's weg in de volgorde waarin ernaar zal worden verwezen. Dit leidt tot een continue, stabiele reeks van paginafouten. Figuur 4.56 toont dit aan in een extreem voorbeeld, waarin aan een proces, dat 9 pagina's cyclisch aanroept, slechts 8 frames zijn toegewezen.



Figuur 4.56

Bovendien is LRU moeilijk in de praktijk te implementeren: het is in principe nodig om een gesorteerde lijst of stapel van pagina's bij te houden, waarbij de recentst gebruikte pagina voorop staat en de minst recent gebruikte achteraan. Deze lijst of stapel moet bij elke verwijzing naar het geheugen bijgewerkt worden, wat veel te lang duurt. Het softwarematig bijwerken van dergelijke datastructuren zou elke geheugenverwijzing minstens een factor tien langzamer maken. Een mogelijke oplossing, die in hardware kan worden uitgevoerd, houdt voor een machine met *N* paginaframes een matrix bij van *N*x*N* bits, waarin alle elementen aanvankelijk nul zijn.

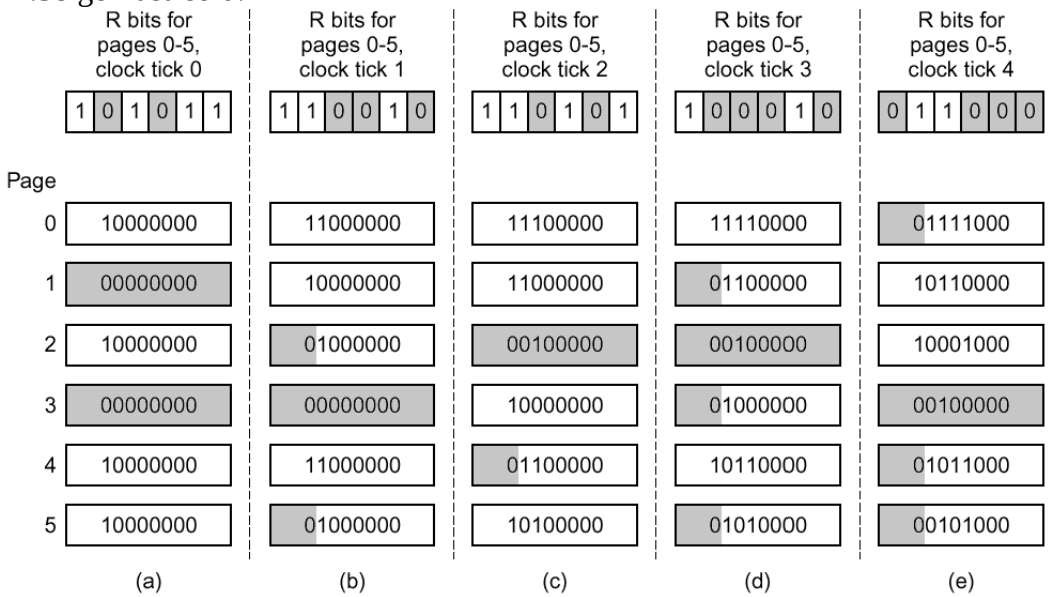


Figuur 4.57

Telkens pag e bits 1, en vervolgens i e rij dan aan

hoe recent de overeenkomstige pagina is gebruikt (de pagina in de rij met de laagste binaire waarde is de minst recent gebruikte, en komt voor vervanging in aanmerking). Figuur 4.57 geeft een voorbeeld, voor een systeem met vier frames waarin achtereenvolgens frame 0, 1, 2, 3, 2, 1, 0, 3, 2 en 3 worden gebruikt. Indien men de vervangingstrategie bekijkt op de niveaus tussen het secundaire geheugen en het hoofdgeheugen, dan is de hoeveelheid hardware die deze techniek vereist niet te realiseren. Ook in de hogere lagen van de geheugenhiërarchie echter moeten analoge beslissingen genomen worden, en is deze oplossing wel realistisch.

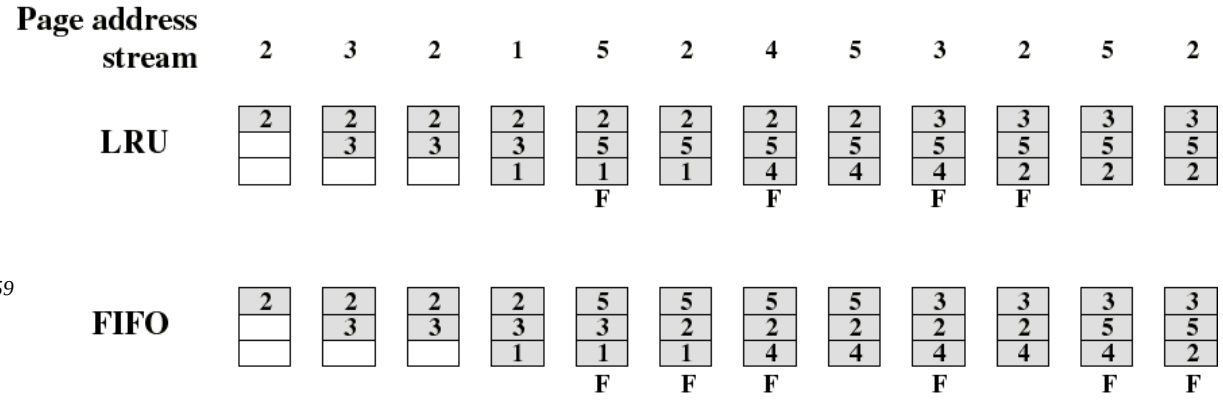
3) Aangezien op dit niveau de speciale LRU hardware niet beschikbaar verondersteld kan worden, moet LRU softwarematig door het besturingssysteem worden gesimuleerd. Meestal wordt een licht gewijzigd LRU algoritme geïmplementeerd: *Not Frequently Used* (NFU). In tegenstelling tot LRU, waar de aanpassing bij elke geheugenverwijzing uitgevoerd wordt, worden tijdsintervallen ingevoerd. NFU maakt geen onderscheid of in een dergelijk tijdsinterval één of meerdere keren naar een specifieke pagina verwezen wordt. Bovendien houdt NFU slechts met een beperkt aantal tijdsintervallen in het verleden rekening. NFU houdt voor elke pagina een softwareteller bij (in de paginatabel), die de geschiedenis van het gebruik van de pagina voorstelt. Deze teller is aanvankelijk nul. Bij elke geheugenverwijzing wordt de meeste linkse bit van de paginateller op 1 geplaatst. Hedendaagse processoren bieden hiervoor hardwarematige ondersteuning. Bij elke klokinterrupt (of een veelvoud ervan) worden alle softwaretellers 1 bit naar rechts verschoven, wat er op neerkomt dat de numerieke waarde van de teller gedeeld wordt door 2. Geheugenverwijzingen in vorige tijdsintervallen krijgen hierdoor een exponentieel dalend gewicht. Wanneer er een paginafout optreedt, dan wordt de pagina verwijderd waarvan de teller het kleinst is. Dit wordt in figuur 4.58 geïllustreerd.



Figuur 4.58

4) De strategie *First In, First Out* (FIFO) behandelt de frames die zijn toegewezen aan een proces als een circulaire buffer, voegt nieuwe pagina's toe aan het einde van de buffer, en verwijdert pagina's die zich het langst in het hoofdgeheugen bevinden. FIFO is dan ook de eenvoudigst uit te voeren strategie voor paginavervanging. De redenering hierbij is dat nieuwere pagina's misschien binnenkort opnieuw zullen gebruikt worden. De tijd die een pagina in het geheugen staat geeft echter niet altijd goed aan hoe vlug het proces opnieuw om die pagina zal vragen: pagina's die langdurig voor het proces beschikbaar moeten blijven (bijvoorbeeld pagina's met code voor veel gebruikte procedures, of pagina's die een

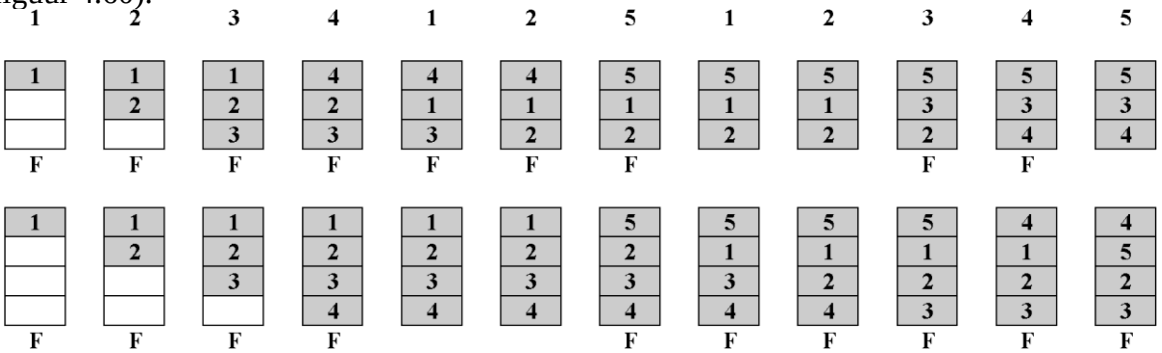
variabele bevatten die constant gebruikt wordt) worden door FIFO dan ook herhaaldelijk uit en terug in het hoofdgeheugen gepagineerd.



Figuur 4.59

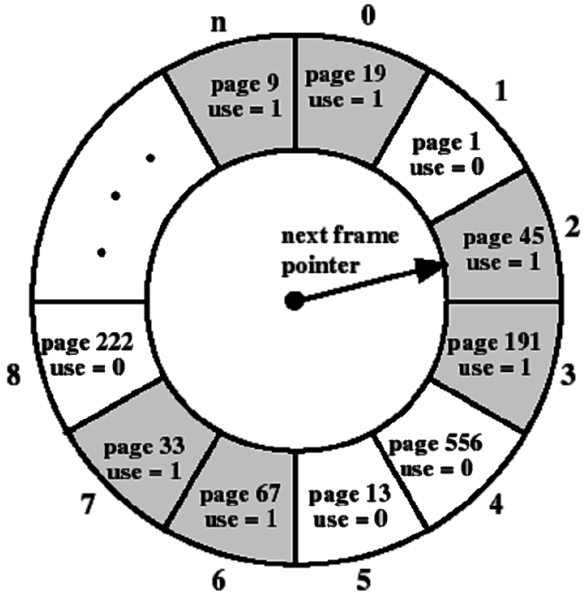
Figuur 4.59 illustreert op het voorbeeldproces hoe de LRU strategie wel detecteert dat pagina's 2 en 5 vaker worden geadresseerd dan de andere pagina's, maar dat FIFO daarentegen dit niet kan opvangen. FIFO resulteert dan ook in zes paginafouten. Globaal presteert FIFO dan ook zeer slecht.

Onder een FIFO-strategie kan eveneens een eigenaardig verschijnsel optreden, dat de *anomalie van Belady* wordt genoemd: een proces kan, afhankelijk van de specifieke volgorde van de verwijzingen, in een grotere adresruimte toch meer paginafouten genereren (figuur 4.60).

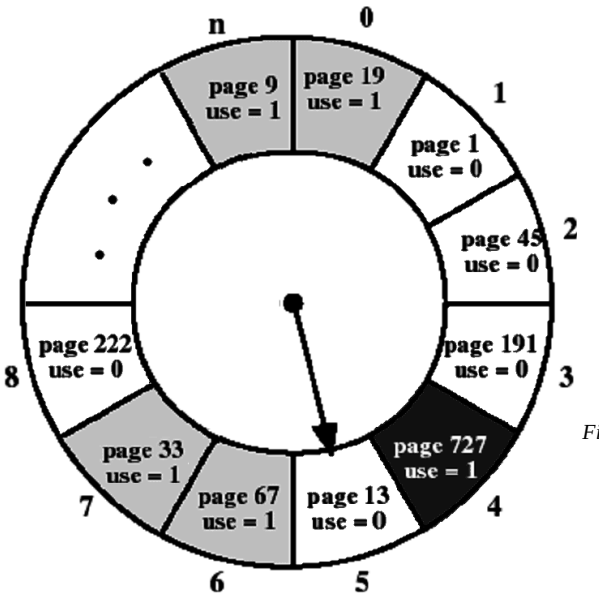


Figuur 4.60

5) De *klokstrategie*, ook wel *tweede-kans-algoritme* genoemd, koppelt aan elk frame een extra *use* bit, die in de (al dan niet geïnverteerde) paginatabel opgeslagen wordt. Het kan dan ook geïnterpreteerd worden als een NFU algoritme, waarvan de tellers maar uit één enkele bit bestaan. De *use* bit wordt ingesteld op 1, bij voorkeur hardwarematig, zowel wanneer de pagina in het hoofdgeheugen wordt geladen, als wanneer er naar wordt verwezen. De frames die kandidaat zijn voor vervanging worden behandeld als een circulaire buffer (figuur 4.61), zoals in de FIFO strategie. Moet een pagina worden vervangen, dan zoekt het besturingssysteem aan de hand van een pointer, die hier *wijzer* wordt genoemd (figuur 4.61), naar een frame waarvan de *use* bit is ingesteld op 0 (figuur 4.62). Komt het besturingssysteem hierbij frames tegen met *use* bit op 1, dan wordt de bit ingesteld op 0, maar wordt de frame niet gebruikt. Hebben alle frames een *use* bit 1, dan maakt de wijzer een volledig rondje door de buffer waarbij hij alle *use* bits instelt op 0. Vervolgens stopt hij bij zijn oorspronkelijke positie, en vervangt hij dat frame. De klokstrategie lijkt op FIFO, met dit verschil dat recent geadresseerde frames worden overgeslagen. De klokstrategie benadert in de praktijk beter de prestaties van LRU of NFU, en dit met een relatief kleine overhead.

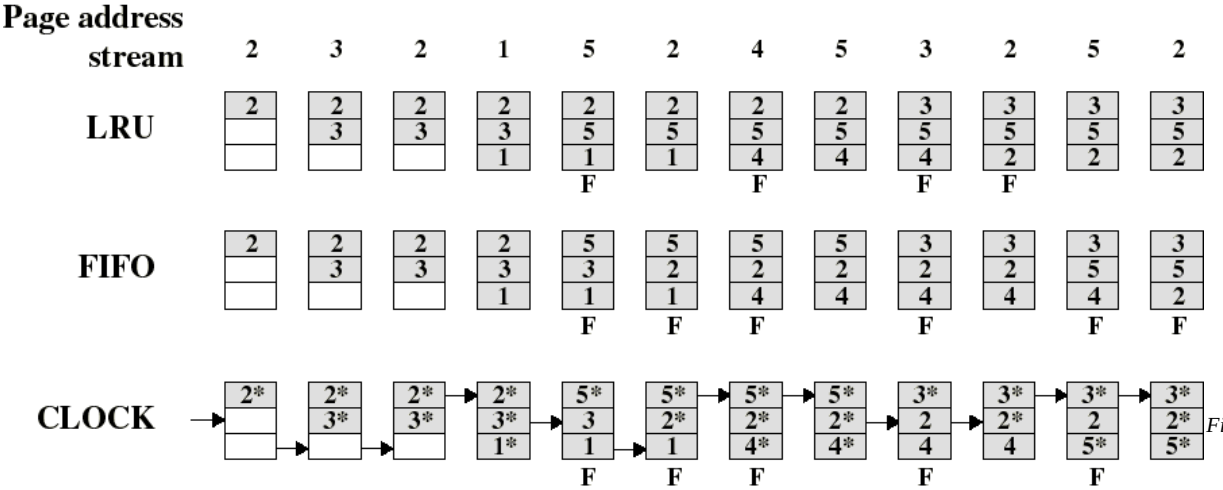


Figuur 4.61



Figuur 4.62

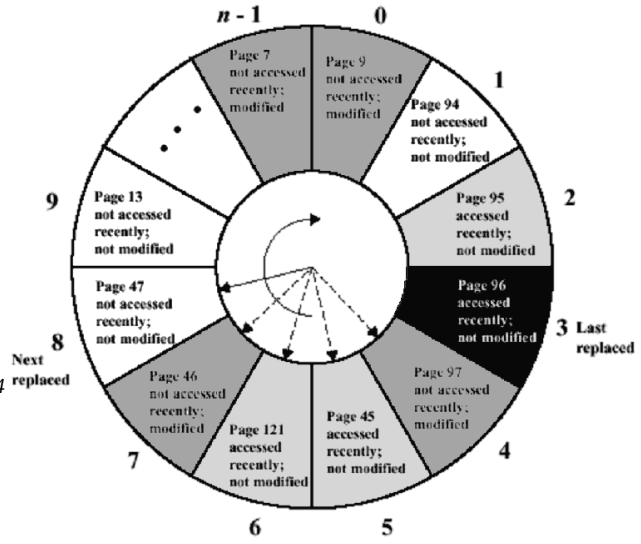
Toegepast op ons voorbeeld, figuur 4.63 waarin de pijl de huidige positie van de wijzer aangeeft, en een asterisk vermeldt dat de *use* bit gelijk is aan 1, slaagt ook de klokstrategie er in pagina's 2 en 5 te beschermen tegen vervanging.



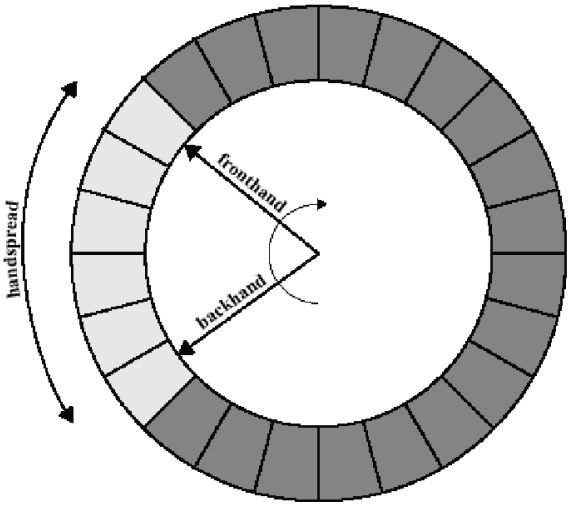
Figuur 4.63

Het klokalgoritme kan geoptimaliseerd worden door het aantal *use* bits te verhogen, waardoor de NFU-strategie beter benaderd wordt. Het mechanisme dat Linux hanteert, komt eigenlijk hierop neer. Ook kan men de *dirty bit* ('M', figuur 4.33) bij de selectie betrekken. In dit laatste geval, dat in de praktijk bijvoorbeeld wordt aangewend in het Mac OS van Apple, worden bij voorkeur, om zo weinig mogelijk I/O te veroorzaken, pagina's vervangen die niet recent zijn gebruikt èn niet zijn gewijzigd sinds paginering in het geheugen (figuur 4.64). Vervanging van deze pagina's heeft immers het voordeel dat deze pagina's niet hoeven weggeschreven te worden naar het secundaire geheugen. Wordt bij deze eerste ronde geen geschikte kandidaat gevonden, dan doorloopt het algoritme de buffer opnieuw, nu op zoek naar een gewijzigde pagina die niet recent is gebruikt. Ook nu kan een volledig rondje opnieuw niets opleveren. Vandaar de naam *derde-kans-algoritme*.

Unix gebruikt nog een andere verfijning van het klokalgoritme als vervangingsstrategie in het virtueel pagineringsysteem voor gebruikersprocessen. Voor het beheren van de geheugentoewijzing voor de kernel daarentegen wordt een niet-virtueel buddysysteem gebruikt: de kernel vereist immers veelvuldig dynamische toewijzing van kleine tabellen en buffers, die veel kleiner zijn dan de standaard paginagrootte.



Figuur 4.64



Figuur 4.65

De paginering in Unix wordt gedeeltelijk uitgevoerd door een specifiek proces, de paginadaemon (*proces 2*). Unix houdt een dubbelgekoppelde lijst bij waarin alle vrije paginaframes staan. Als er een paginafout optreedt, dan verwijdert Unix het eerste paginaframe van de vrije lijst en wordt de benodigde pagina in dat frame ingelezen. De paginadaemon onderzoekt regelmatig (typisch om de 250 ms) of het aantal vrije pagina's minstens gelijk is aan de systeemparameter *lotsfree* (meestal ingesteld op ongeveer een kwart van het geheugen). Als er niet voldoende frames vrij zijn, dan begint de paginadaemon pagina's uit het geheugen naar schijf over te brengen. De daemon houdt hierbij geen rekening met aan welk proces de pagina's toebehoren: de vervangingsstrategie is bijgevolg globaal, en het aantal pagina's die aan een proces zijn toegewezen varieert in de loop van de tijd. Omdat volledige doorgangen bij het normale klokalgoritme vrij lang blijken te duren, wordt een *klokalgoritme met twee wijzers* gebruikt (figuur 4.65). De *hoek* tussen de voorste en de achterste wijzer blijft hierbij constant, maar kan bij generatie van het Unix systeem worden geconfigureerd, op basis van de hoeveelheid fysiek geheugen. De voorste wijzer zet de *use* bit op 0, terwijl de achterste wijzer pagina's voor verwijdering selecteert waarvan de *use* bit op 0 staat. Als de twee wijzers bijna 360° uit elkaar staan, bekomt men het oorspronkelijk klokalgoritme. Als de twee wijzers dicht bij elkaar worden gehouden, zullen alleen zeer frequent gebruikte pagina's, die geadresseerd worden tussen de momenten in dat beide wijzers langskomen, voor vervanging worden genegeerd. Als het systeem merkt dat er te vaak gepagineerd wordt, en het aantal vrije pagina's continu lager is dan de *lotsfree* parameter, dan wordt de swapperdaemon (*proces 0*) geactiveerd om één of meer processen volledig uit het geheugen te swappen.

Windows gebruikt eveneens een klokalgoritme, maar enkel op systemen met één processor. Op systemen met meerdere processoren, waar een TLB aanwezig is per processor, wordt eenvoudigweg FIFO toegepast, zonder rekening te houden met de use bit. Alle activiteiten in verband met geheugenbeheer worden in NT uitgevoerd door een zestal gespecialiseerde systeemthreads, met verschillende prioriteit, binnen de *Virtual Memory Manager* module van de Executive. Deze threads kunnen door diverse gebeurtenissen geactiveerd worden: ondermeer op tijdsbasis (elke seconde), of door de scheduler, indien de paginafoutfrequentie te hoog oploopt, of indien de lijst met vrije frames te klein wordt. Ook Windows kan volledige processen uit het geheugen swappen. Van elke thread die langer dan een aantal seconden (3 tot 7 seconden, afhankelijk van de hoeveelheid geheugen) niet actief is geweest,

wordt het stackgebied uitgepagineerd. Geldt dit voor alle threads van een proces, dan wordt het volledige procesbeeld geswapt.