Les 5: Geheugenbeheer

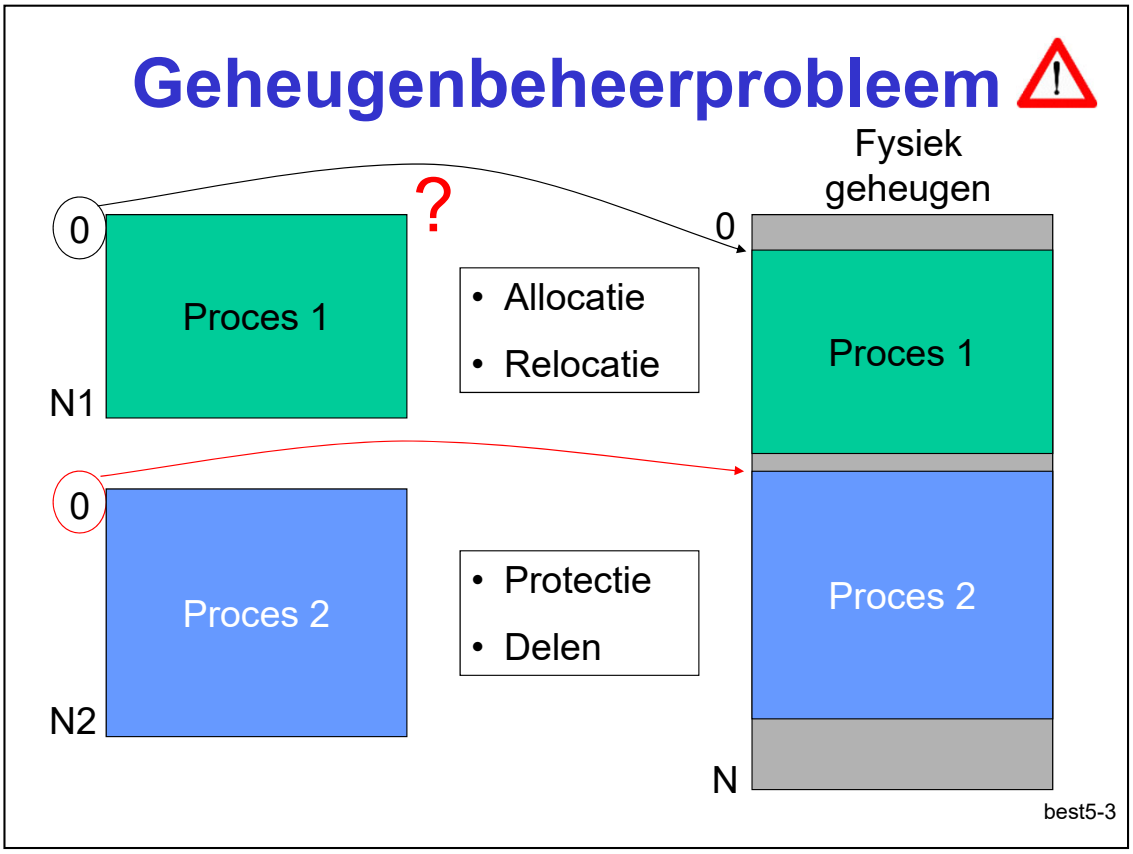
“If we wish to count lines of code, we should not regard them as ‘lines produced’ but as ‘lines spent.’”
– Edsger Dijkstra

best5-1

Overzicht

- Logische vs. fysieke adressen
- Contigue allocatie van geheugen
- Niet-contigue allocatie van geheugen
  - Paginering
  - Segmentering
  - Segmentering + paginering

best5-2

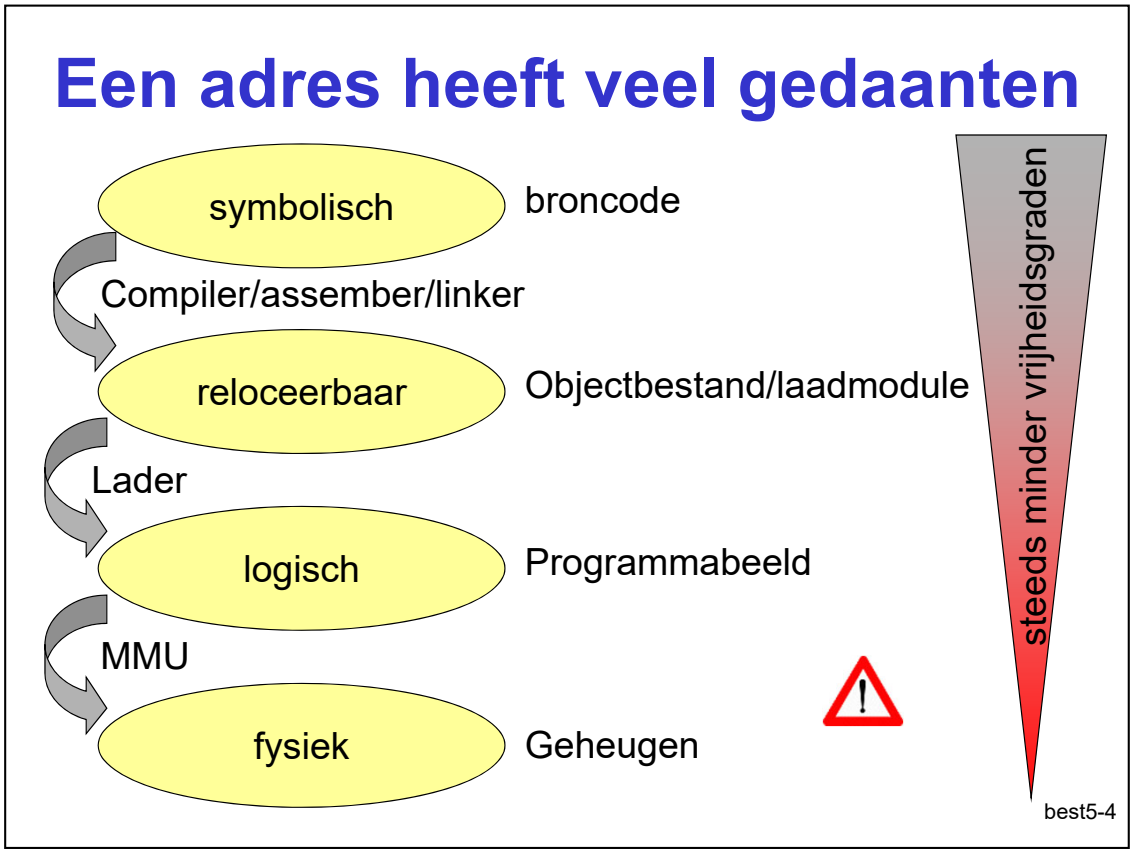


Een fundamenteel probleem dat in deze les behandeld wordt is op welke manier we de hoeveelheid beschikbaar geheugen zo efficiënt mogelijk kunnen verdelen over een aantal processen. Daarbij moeten 2 problemen opgelost worden.

1. Op welke manier kunnen we ervoor zorgen dat de beschikbare hoeveelheid geheugen zo goed mogelijk gebruikt wordt; dit is het probleem van het efficiënt alloceren van geheugen.
2. Op welke manier kunnen we verschillende processen die ervan uitgaan dat ze op een gegeven adres zullen geladen worden (b.v. 0) simultaan in het geheugen houden zonder ze te moeten aanpassen aan hun uiteindelijke adres. Dit is het probleem van het reloceren van adressen.

Verder zullen we aandacht moeten hebben voor een tweetal specifieke kwesties die opduiken van zodra er meer dan 1 proces actief is in het geheugen.

1. Op welke manier kunnen we ervoor zorgen dat die processen elkaar niet kunnen schaden (al dan niet moedwillig). Dit is het probleem van de onderlinge afscherming of protectie van de processen.
2. Op welke manier kunnen we ervoor zorgen dat processen die bepaalde stukken geheugen met elkaar willen delen (hetzij bibliotheken, hetzij gegevens), dit effectief ook kunnen doen. Dit is het probleem van de gedeelde code en data.



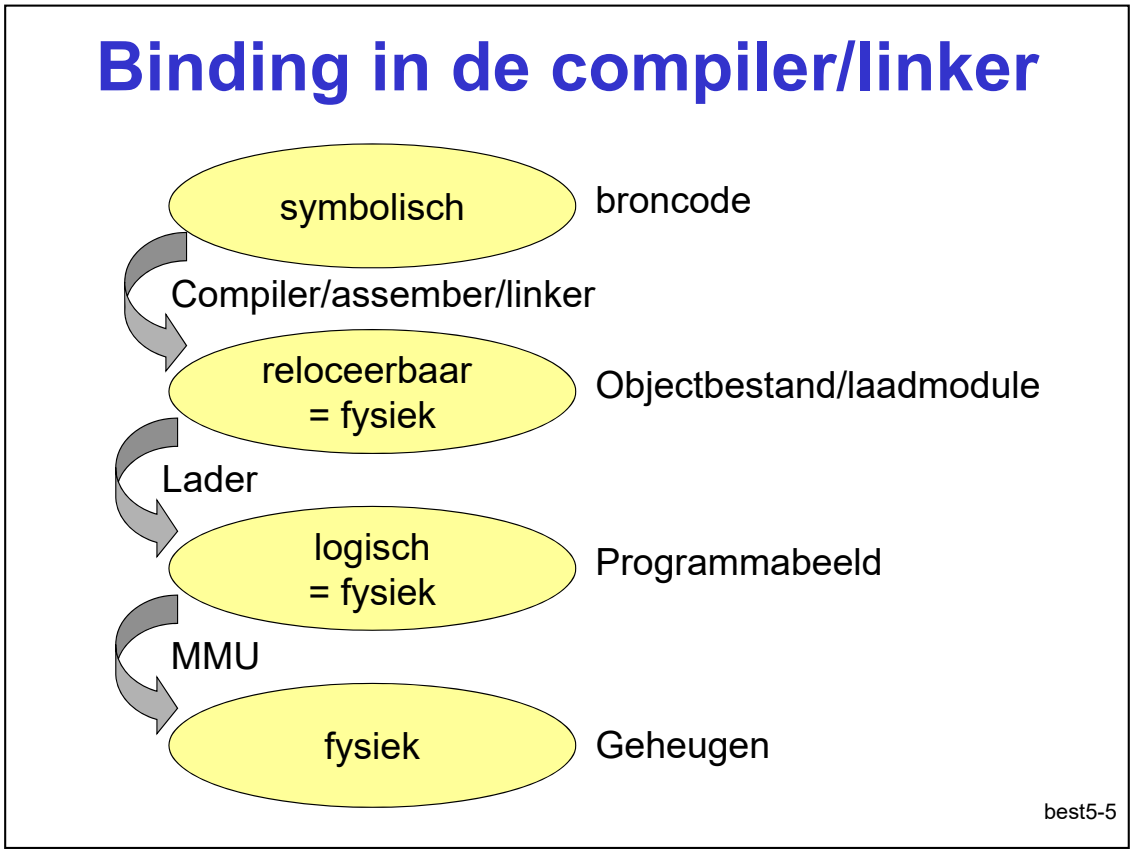
Doordat het geheugen gedeeld wordt door verschillende processen zal een programma niet steeds op dezelfde plaats in het geheugen ingeladen kunnen worden om uitgevoerd te worden. De compiler en de linker zullen dus het programma dusdanig moeten aanmaken dat een (beperkte) verplaatsing in het geheugen mogelijk wordt. Daarvoor maakt men een onderscheid tussen verschillende soorten adressen.

Symbolische adressen in het bronprogramma, b.v. count. Deze laten ons toe om – lang voordat het adres van een object vastligt in het geheugen – toch al naar dit object te refereren.

Reloceerbare adressen in de objectbestanden (b.v. 508 t.o.v. begin van een module), dit wordt een relatief adres genoemd. Aangezien compilers één module per keer compileren en de andere modules nog niet gekend zijn, kunnen adressen nog geen vaste waarde hebben. Een mogelijke oplossing bestaat er in om relatieve adressen t.o.v. het begin van een module te gebruiken. Eenmaal het programma gelinkt is, kunnen de adressen binnenin het programma vastgelegd worden. Echter, aangezien de processen die simultaan met dit programma zullen uitgevoerd worden nog niet gekend zijn, kunnen geen fysieke adressen gebruikt worden. Het resultaat van het linken is dus nog steeds een programma dat reloceerbare adressen gebruikt.

Ter gelegenheid van het inladen van het programma in het geheugen worden de reloceerbare adressen gefixeerd. Vanaf dan noemt men ze **logische of virtuele adressen**. Deze adressen gaan ervan uit dat de hele adresruimte van de processor ter beschikking staat van het proces.

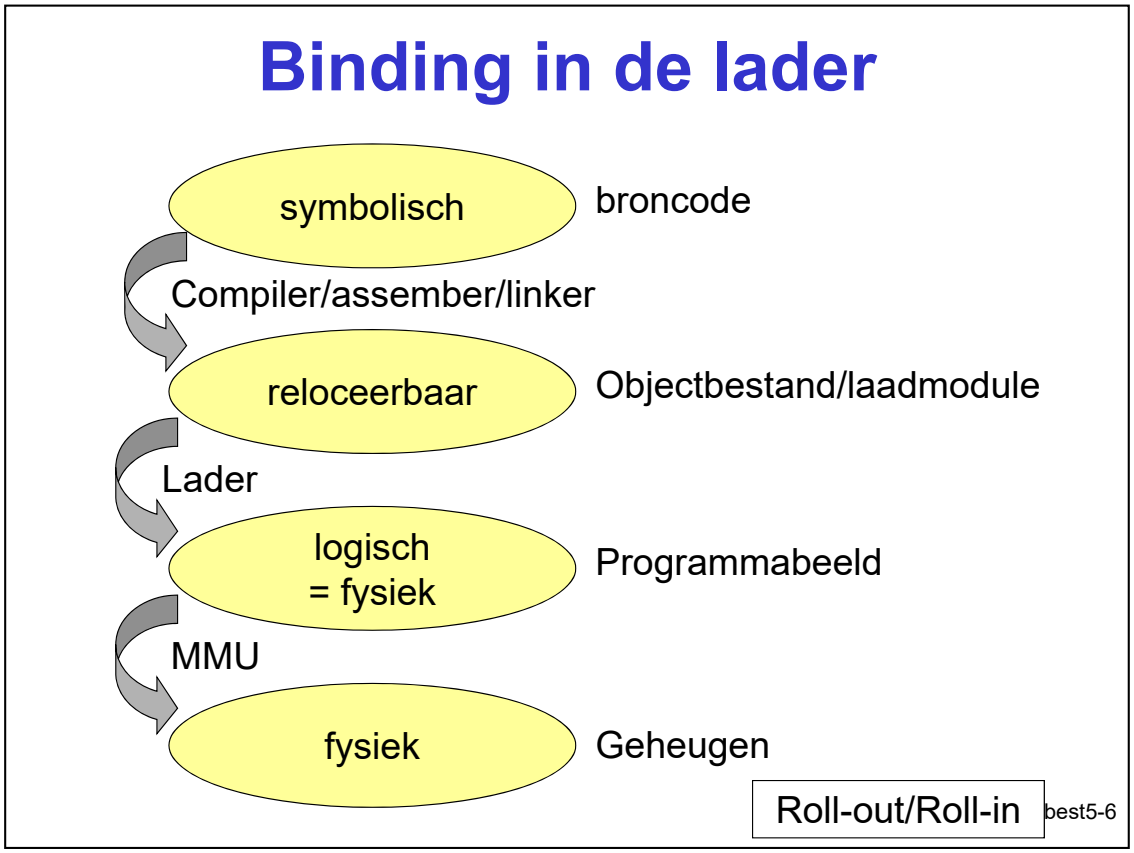
Finaal zullen tijdens de uitvoering van het proces de logische adressen omgezet worden in fysieke adressen. Dit gebeurt door de geheugenbeheerseenheid (MMU).



Indien de compiler of de linker weet waar in het geheugen een proces zal uitgevoerd worden, kunnen er door hen reeds fysieke adressen gegenereerd worden. Men spreekt in dit verband ook van absolute code en van statische binding.

Concreet betekent dit dat de reloceerbare adressen meteen zullen verwijzen naar de fysieke adressen, en dat de logische adressen dezelfde zullen zijn als de fysieke adressen. In dat geval kan de MMU gewoon komen te vervallen.

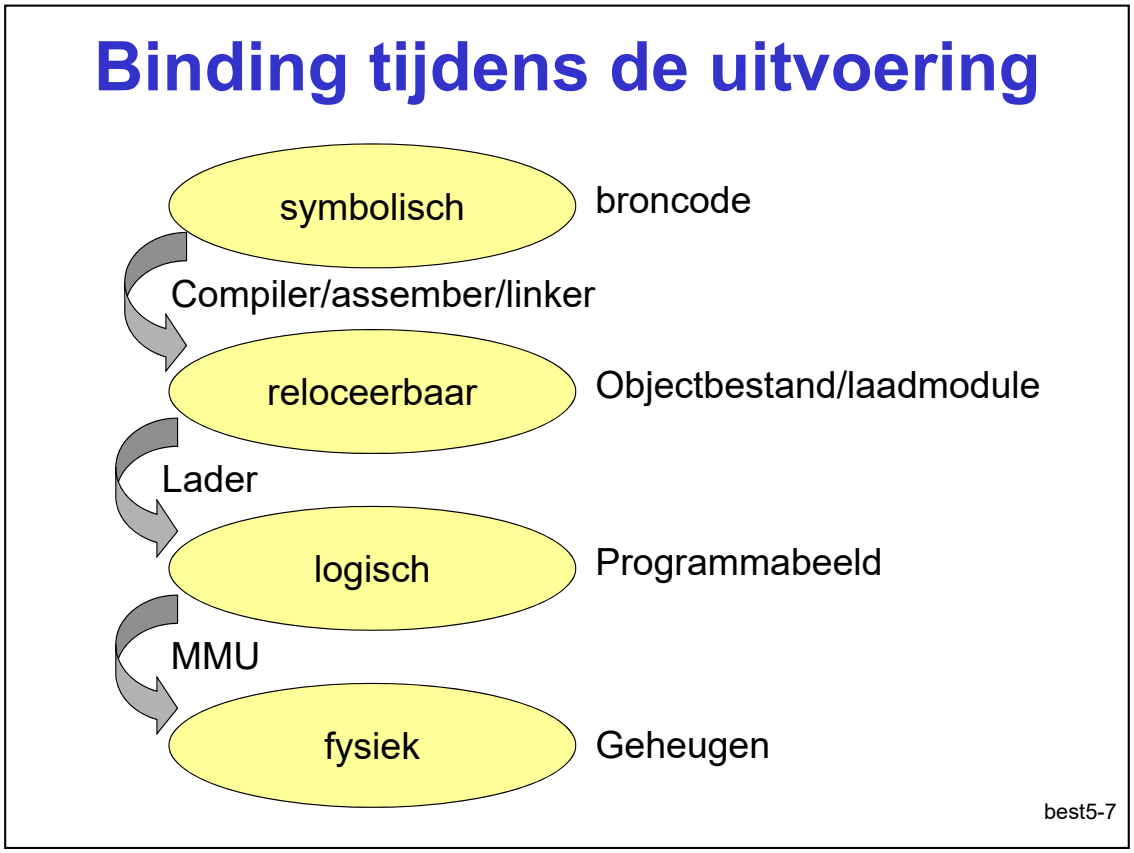
Statische binding wordt soms nog gebruikt in kleine gespecialiseerde computersystemen zoals microcontrollers. Voor grote toepassingen is het te onhandig. Als programmeur wenst men niet geconfronteerd te worden met de problematiek van de toekenning van fysieke adressen aan programmaobjecten.



Een reloceerbaar programma bevat geen absolute adressen maar offsets t.o.v. een fictief beginadres (programma begint b.v. op adres 0). Bovendien bevat het programma een relocatietabel. Deze tabel bevat informatie over de plaatsen in het programma die moeten aangepast worden, indien men het programma zou willen laden op een ander adres, en over de aard van de aanpassingen die dan nodig zijn (meestal zal het verschuiven van de adressen over een zekere afstand voldoende zijn).

Indien de adressen gebonden worden tijdens het laden van het programma, laadt de lader het programma op de gewenste locatie, en overloopt vervolgens de relocatietabel om alle nodige aanpassingen in het programma te doen. Eenmaal het programma gereloceerd werd, ligt zijn plaats in het geheugen vast. Eenmaal de uitvoering van het programma gestart is, is relocatie niet meer mogelijk omdat we geen informatie hebben over dynamisch aangemaakte adressen, en we deze dus niet kunnen aanpassen.

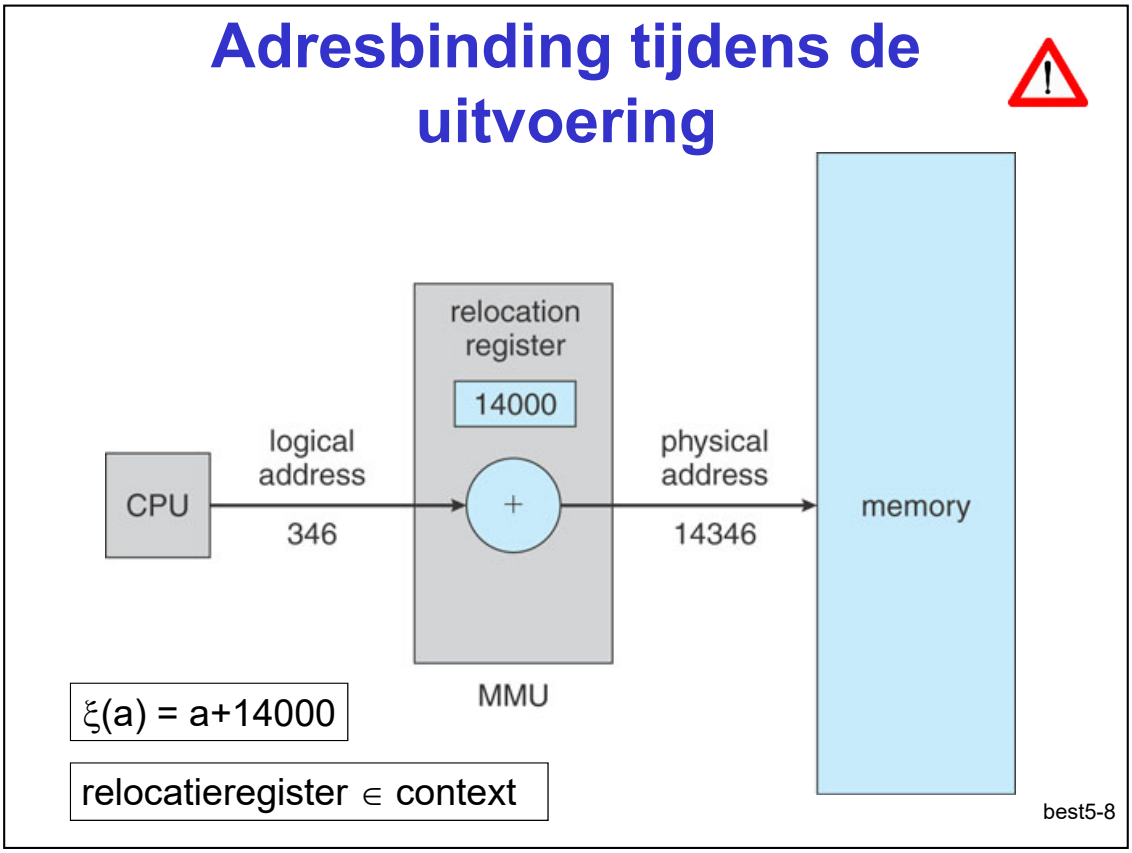
Dit impliceert dat indien we het programma tijdens zijn uitvoering even zouden uitswappen naar schijf, we het programma op precies dezelfde plaats in het geheugen terug zullen moeten plaatsen (dit wordt roll-out/roll-in genoemd). Zoniet zal het niet correct meer uitvoeren.



Indien men de binding nog verder wil uitstellen, moet de binding met de fysieke adressen gebeuren tijdens de uitvoering van het programma zelf. Dit zal toelaten dat programma's nog tijdens hun uitvoering in het geheugen verplaatst worden (het is bijzonder nuttig om een uitgeswapt programma niet noodzakelijk terug te moeten inswappen op dezelfde geheugenlocaties). Binding tijdens de uitvoering vereist speciale hardware. Het is precies de werking van deze hardware die in deze les uitgebreid aan bod zal komen.

Reloceerbare programma's zijn flexibeler dan programma's met fysieke adressen, programma's die pas tijdens hun uitvoering aan de fysieke adressen gebonden worden zijn nog flexibeler dan reloceerbare programma's.

De totale ontkoppeling tussen de logische en fysische adressen heeft voordelen voor de programmeur en voor het besturingssysteem: voor de programmeur beginnen alle programma's conceptueel op adres 0 en het besturingssysteem behoudt niettemin de volle vrijheid om zelfs nog tijdens de uitvoering van het programma te beslissen welk deel van het geheugen gebruikt zal worden.

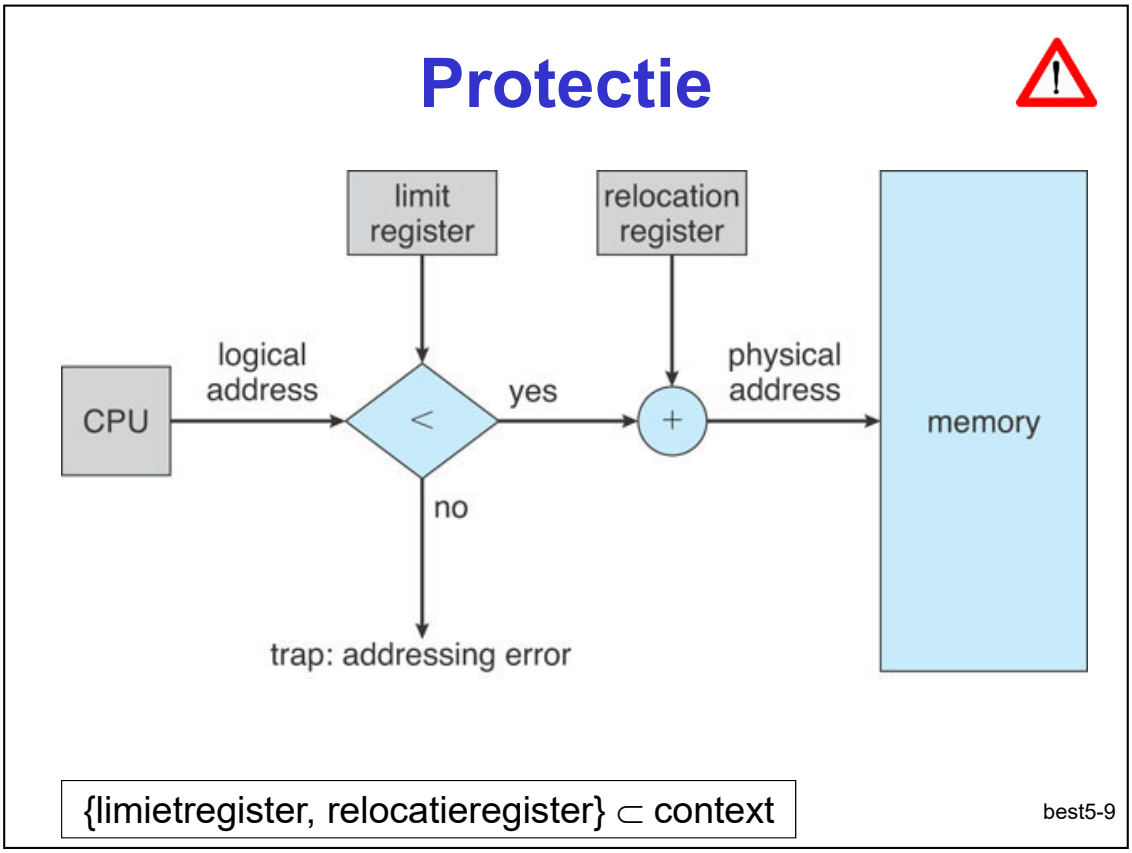


Adresbinding tijdens de uitvoering vereist de aanwezigheid van gespecialiseerde hardware. Deze hardware wordt de geheugenbeheerseenheid genoemd (memory management unit of MMU). Deze eenheid zal de vertaling van logische adressen naar fysieke adressen moeten doorvoeren tijdens de uitvoering van het programma. In zijn meest eenvoudige vorm bestaat een MMU uit een relocatieregister dat bij elk adres dat door de CVE gegenereerd wordt, opgeteld wordt. De som is dan het fysiek adres dat gebruikt wordt om het geheugen aan te spreken. De CVE voert het programma volledig uit in de logische adresruimte (b.v. 4 GiB te beginnen vanaf adres 0). Adressen worden pas vertaald net voor ze op de adresbus van het geheugen geplaatst worden. Dit wil zeggen dat wijzers naar datastructuren allemaal in hun logische vorm bijgehouden worden, niet de fysieke. Als programmeur zal men in gebruikersmode nooit met de fysieke adressen geconfronteerd worden.

Men zegt dat de logische adresruimte van het programma door de MMU afgebeeld wordt op de fysieke adresruimte. Wiskundig gezien implementeert de MMU de afbeeldingsfunctie ξ die een logisch adres afbeeldt op zijn fysiek adres. In het bovenstaande geval wordt de functie als volgt gedefinieerd $\xi(a) = a+14000$. Later in deze les volgen aanzienlijk complexere functies.

Om deze eenvoudige geheugenbeheerseenheid te kunnen gebruiken is het noodzakelijk dat een proces ingeladen wordt in een aaneengesloten (contigu) stuk geheugen en dat de compiler/linker ervoor zorgt dat het programma op het logisch adres 0 begint. In dat geval volstaat het om het werkelijke beginadres van het blok geheugen (in het fysiek geheugen) in het relocatieregister in te laden en ervoor te zorgen dat bij elke geheugenreferentie het relocatieregister opgeteld wordt bij het gegenereerde logisch adres. Hierna bespreken we een aantal manieren om aaneengesloten blokken geheugen te

alloceren.



Om ervoor te zorgen dat processen niet kunnen schrijven in elkaars geheugengebied kan de CVE ervoor zorgen dat de adressen die kunnen gegenereerd worden hardwarematig beperkt worden tot een zeker bereik. Dit gebeurt aan de hand van een limietregister. De gegenereerde logische adressen worden eerst met het limietregister vergeleken en enkel indien ze kleiner zijn dat de waarde van het limietregister, worden ze samengeteld bij het relocatieregister. Indien ze buiten het toegelaten bereik van het proces vallen, dan wordt er een adresseringsfout gegenereerd.

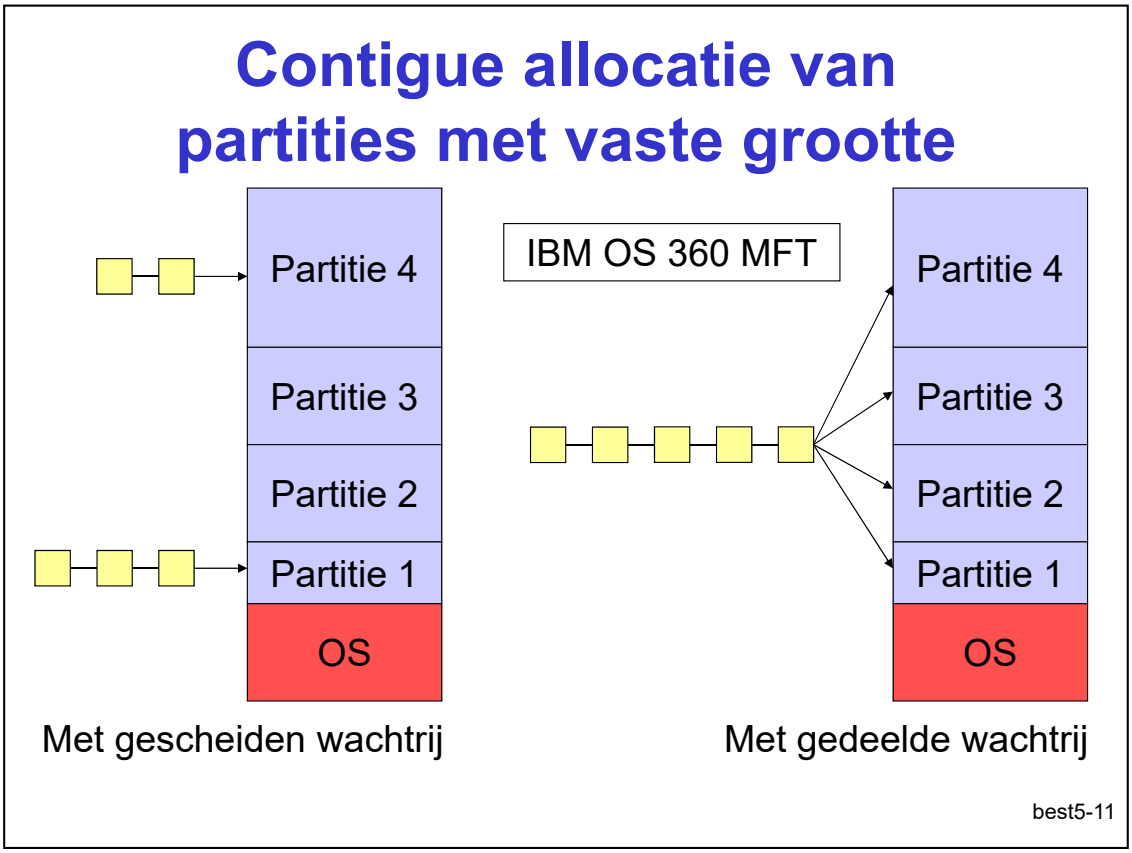
Op deze manier kan effectief afgedwongen worden dat een proces enkel adresseert binnen zijn eigen adresbereik. Door de waarde van het limietregister te veranderen kan het adresbereik van een proces ook effectief uitgebreid worden (dit is uiteraard enkel zinvol indien de partitie ook kan uitgebreid worden).

Bij een contextwisseling zal het relocatieregister en het limietregister samen met de andere registers bewaard worden in de contextrecord.

Overzicht

- Logische vs. fysieke adressen
- Contigue allocatie van geheugen
- Niet-contigue allocatie van geheugen
  - Paginering
  - Segmentering
  - Segmentering + paginering

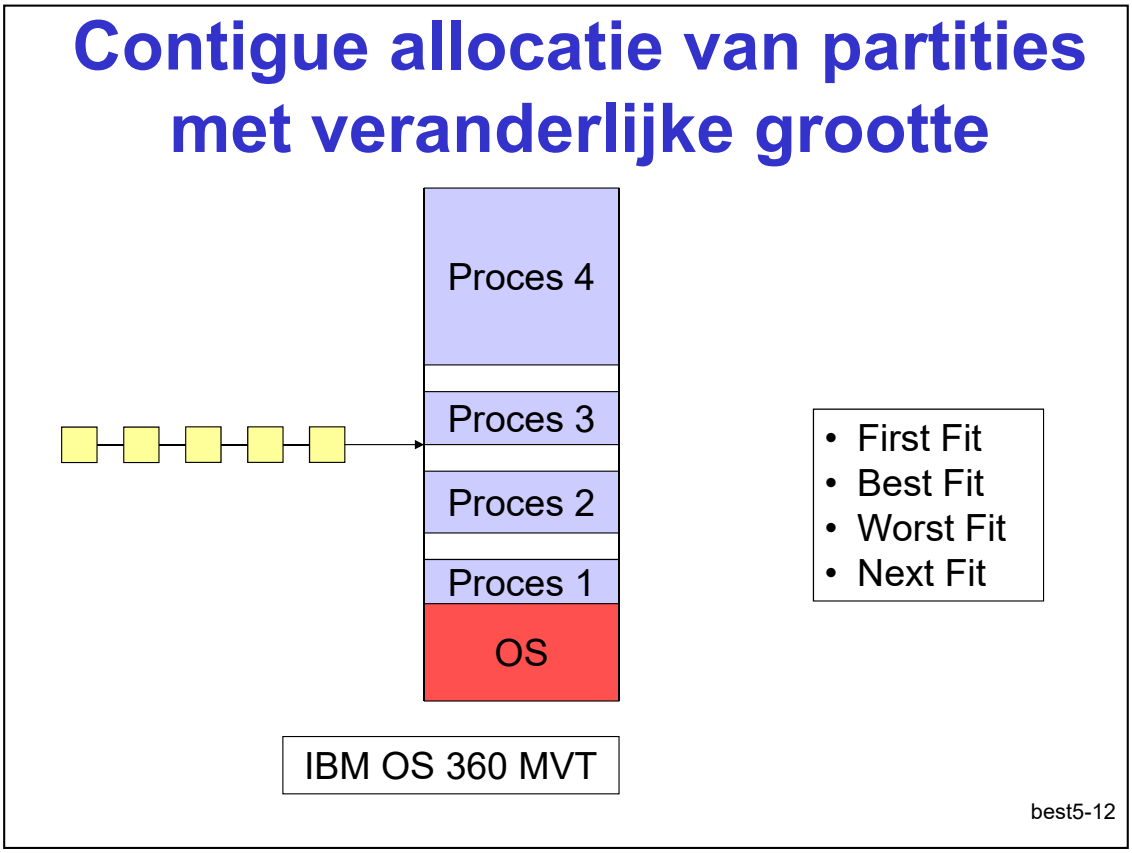
best5-10



Het efficiënt alloceren van aangesloten blokken geheugen is niet altijd eenvoudig omwille van de versnippering van de geheugenblokken die na verloop van tijd ontstaat.

De meest eenvoudige (maar niet de meest bruikbare methode) bestaat erin om het geheugen in te delen in een aantal blokken van gelijke vaste grootte. Bij het inladen van een proces wordt er dan gezocht naar een blok dat het proces kan herbergen. Uiteraard zal dit proces zelden het volledige blok nodig hebben, en gaat er op die manier potentieel toch heel wat geheugenruimte verloren. Het oude IBM OS 360 MFT werkte op deze manier.

Om het geheel nog wat te optimaliseren kan men de verschillende partities een verschillende grootte geven. Op die manier zullen kleine processen minder geheugenruimte verspillen. Processen zullen staan aanschuiven (in de zgn. input queue) om ingeladen te worden in het geheugen (dit is de taak van de jobplanner). Hierbij kan men processen op voorhand indelen in een bepaalde wachtrij of kan men de processen in een gedeelde wachtrij laten wachten. Men kan ook beslissen om een klein proces toch in een grotere partitie te laten uitvoeren indien alle partities van de gepaste grootte bezet zijn, en een grotere partitie toch nog beschikbaar blijkt te zijn.



Om de verspilling van geheugen bij de allocatie van partities met vaste grootte tegen te gaan kan men ook partities alloceren die precies de gepaste grootte hebben. Hiervoor zal men dan wel gebruik moeten maken van een complexer geheugenallocatiealgoritme. Initieel zal het geheugen bestaan uit één groot blok dat gaandeweg verkaveld wordt. Hier en daar zullen stukken opnieuw vrijgegeven en zullen er gaten ontstaan in de lay-out van het geheugen.

Om een geschikt blok te alloceren voor een nieuw proces kan men gebruik maken van de volgende allocatiestrategieën:

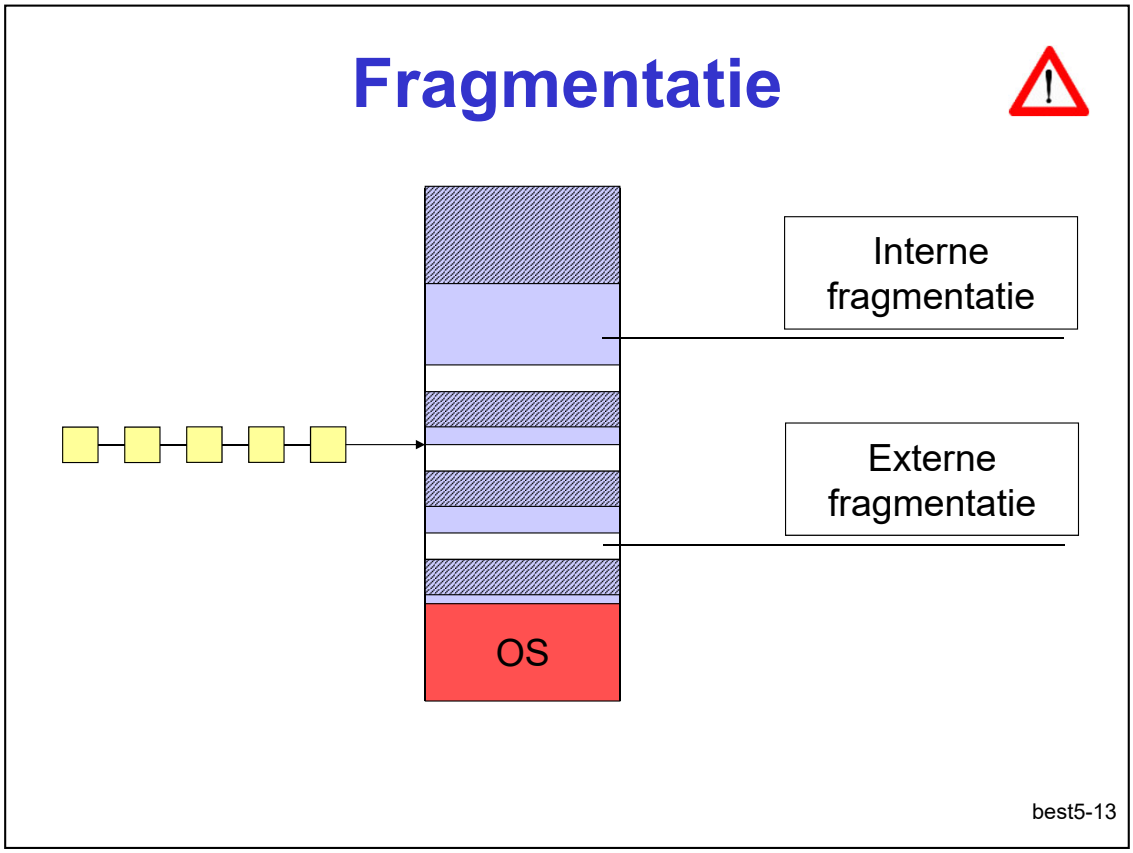
First fit: het eerste vrije blok dat groot genoeg is wordt gekozen. Bij het zoeken kan men steeds vooraan in de lijst met vrije blokken beginnen, of verder zoeken vanaf waar men de vorige keer gekomen was (ook **next fit** genoemd). Van het gekozen blok wordt net zoveel gealloceerd als nodig. De rest blijft bestaan als vrij blok.

Best fit: men kiest voor het kleinste blok dat nog net groot genoeg is. Deze strategie vereist dat de volledige lijst doorlopen wordt. Het overblijvende stuk zal wel minimaal (en daarom misschien niet langer bruikbaar) zijn. Indien dit het geval zou zijn, kan men beslissen om toch het gehele blok te alloceren.

Worst fit: om te verhinderen dat het overblijvende stuk onbruikbaar zou worden, kan men er ook voor kiezen om een stuk te nemen van het grootste blok. Op die manier verhindert men het ontstaan van veel kleine gaten, maar men offert aan de andere kant wel de vrije ruimte op.

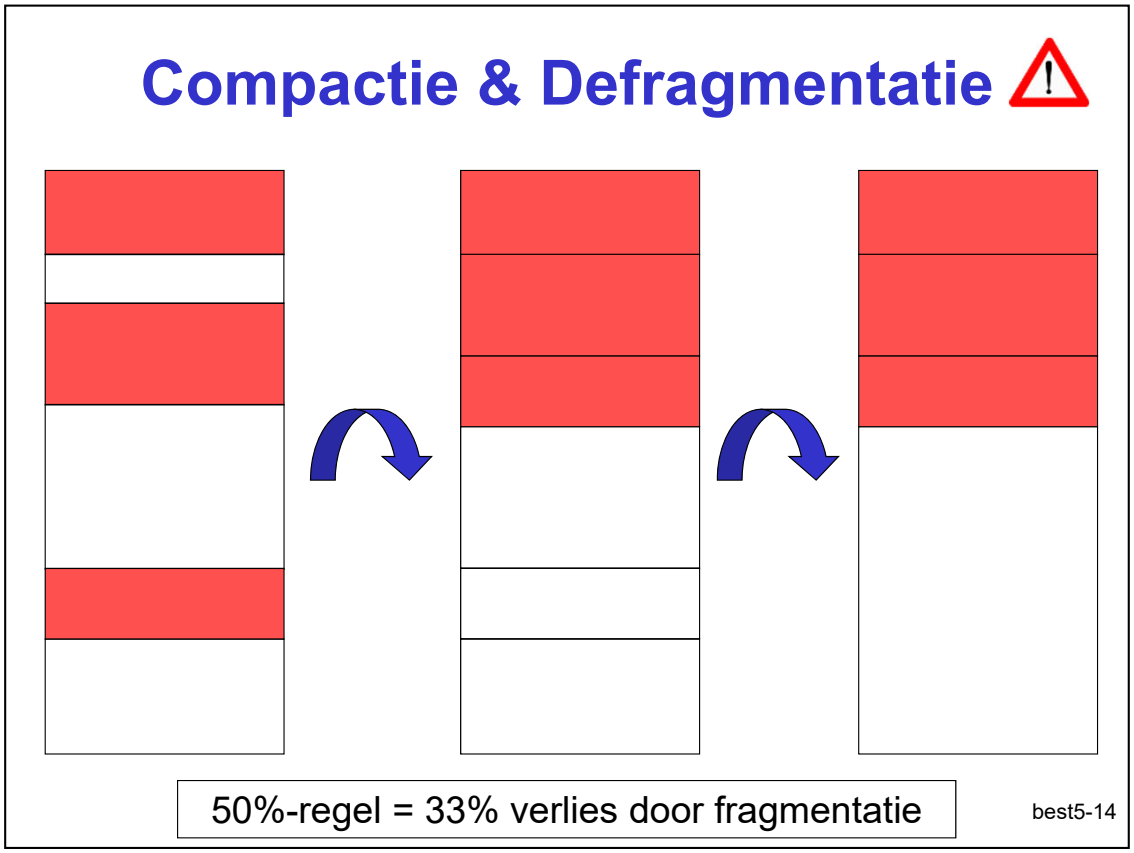
In het algemeen blijkt uit simulaties dat first en best fit beter presteren dan worst fit dat op termijn aanleiding zal geven tot veel versnippering. Een implementatie die de vrije blokken bijhoudt in sublijsten van een bepaalde grootte, wordt soms ook quick fit

genoemd.



Bij het gebruik van contigue allocatie zal men af te rekenen krijgen met het probleem van fragmentatie, dit zijn stukken geheugen die vrij zijn, en toch niet gebruikt kunnen worden. Bij het gebruik van vaste partities heeft men te maken met **interne fragmentatie**. Dit wil zeggen dat de partitie groter kan zijn dan het proces, maar doordat de volledige partitie aan het proces toegewezen werd, kan er van de vrije geheugenruimte geen gebruik gemaakt worden door de andere processen.

Bij het gebruik van partities van veranderlijke grootte zal men af te rekenen hebben met **externe fragmentatie**, dit is de geheugenruimte die overblijft tussen de verschillende processen in het geheugen, en waarvan de stukken individueel te klein blijken te zijn om een proces in onder te brengen, maar die opgeteld wel voldoende groot zijn om het proces in te herbergen.

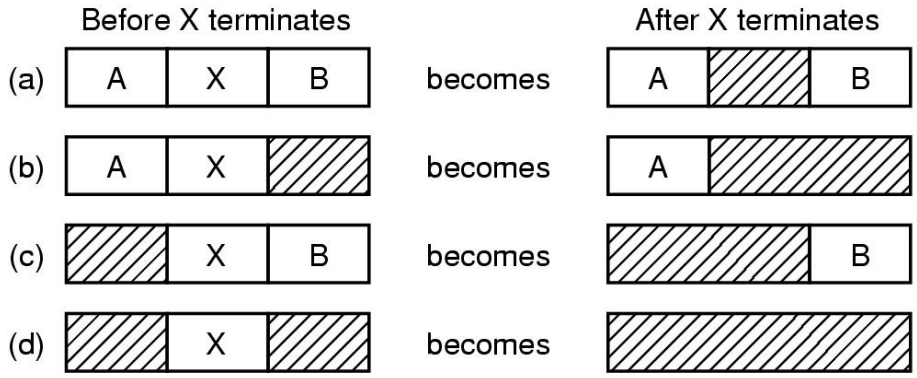


Externe fragmentatie ontstaat doordat partities die vrijgegeven worden enkel kunnen gecombineerd worden met partities die erbij aansluiten. In het slechtste geval kan men tussen elke twee gebruikte partities een ongebruikte partitie hebben. De individuele partities kunnen elk te klein zijn om een gegeven proces te bevatten terwijl hun som ruimschoots voldoende kan zijn. In dit verband hanteert men de 50%-regel die zegt dat voor elke gebruikte geheugencel er een gemiddeld 0.5 geheugencel verloren gaat aan fragmentatie. Hieruit mag blijken dat fragmentatie geen onbelangrijk probleem is.

Een mogelijke oplossing voor fragmentatie is het aanwenden van compactie. Compactie houdt in dat de processen in het geheugen zodanig verplaatst worden dat er grotere stukken vrij geheugen ontstaan. Een mogelijke manier om dit te realiseren is om bijvoorbeeld alle processen zoveel mogelijk naar voor of naar achter te schuiven. In het slechtste geval zal men de totale hoeveelheid reeds gealloceerde geheugen moeten kopiëren. Een alternatieve methode kan erin bestaan om individuele processen te kopiëren naar vrije blokken om aldus hier en daar grotere blokken te krijgen.

Het is duidelijk dat compactie enkel mogelijk is indien processen tijdens hun uitvoering kunnen gereloceerd worden. In de praktijk zal dit enkel mogelijk zijn indien we gebruik maken van adresbinding tijdens de uitvoering en op voorwaarde dat het noch het programma, noch het besturingssysteem intern fysieke adressen bijhouden (tenzij op een goed gedocumenteerde manier – zoals b.v. in de paginatabellen). Indien dit niet het geval zou zijn, dan zou het kunnen gebeuren dat er na het verschuiven van het proces toch nog hier en daar naar de oorspronkelijke fysieke adressen geschreven wordt, met onaanvaardbare effecten als gevolg.

Boekhouding geheugen: vrijgeven geheugen



best5-15

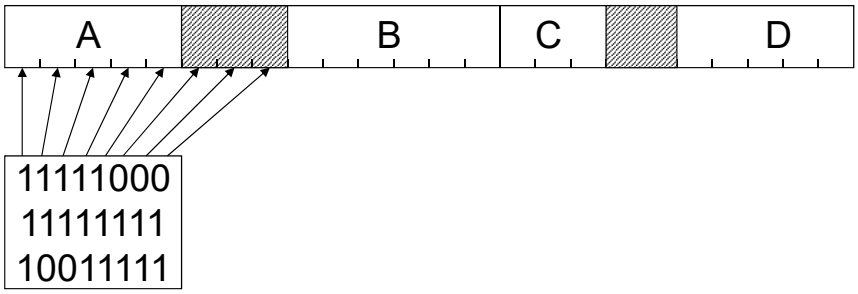
Bij het vrijgeven van blokken geheugen is het van belang dat deze in de mate van het mogelijke terug samengevoegd worden met belendende vrije blokken om aldus de versnippering van het geheugen in kleine stukjes zoveel mogelijk tegen te gaan. Zeker indien het geheugen verkaveld werd in duizenden kleine stukjes, kan het terug samenvoegen van vrije blokken redelijk wat tijd in beslag nemen.

Voor het geheugenbeheer van de geheugenpartities is dit van minder belang omdat het aantal partities doorgaans niet zo heel hoog oploopt, en dat er ter gelegenheid van het termineren van een proces wel wat tijd beschikbaar is om een dergelijke bewerking uit te voeren. Indien men dit echter toepast op het beheer van het grabbelgeheugen waar er per seconde vele honderden blokken kunnen gealloceerd worden en terug kunnen vrijgegeven worden, wordt dit aspect wel belangrijk.

Het is dan ook belangrijk om een geschikte gegevensstructuur te gebruiken.

Boekhouding geheugen: bitmaps

Geheugen opgedeeld in eenheden vaste grootte



- vrijgeven geheugen eenvoudig
- k opeenvolgende vrije blokken zoeken is trage operatie

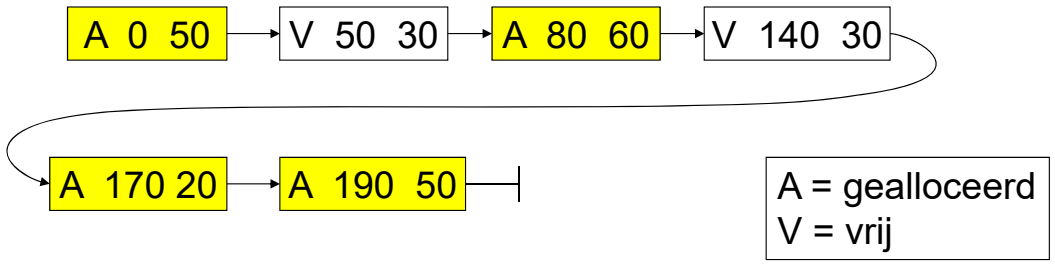
best5-16

Het gebruik van een bitmap is een populaire manier om aan geheugenbeheer te doen. Het geheugen wordt opgedeeld in blokken van b.v. 4 KiB (een partitie zal in deze visie dus steeds uit een veelvoud van 4 KiB bestaan). Per blok houdt men 1 bit bij om aan te geven of het blok al dan niet gealloceerd werd. Dit is een zeer eenvoudig systeem dat weinig overhead met zich meebrengt (1 bit per 4 KiB).

Het vrijgeven gebeurt door de corresponderende bits op 0 te zetten. Het samenvoegen van de blokken gebeurt dus automatisch.

Het op zoek gaan naar een aaneengesloten stuk van k blokken is echter een heel andere opgave. De bitmap moet nu overlopen worden op zoek naar een aaneengesloten rij van k nulletjes. Dit is een vrij ingewikkelde bewerking.

Boekhouding geheugen: gelinkte lijsten

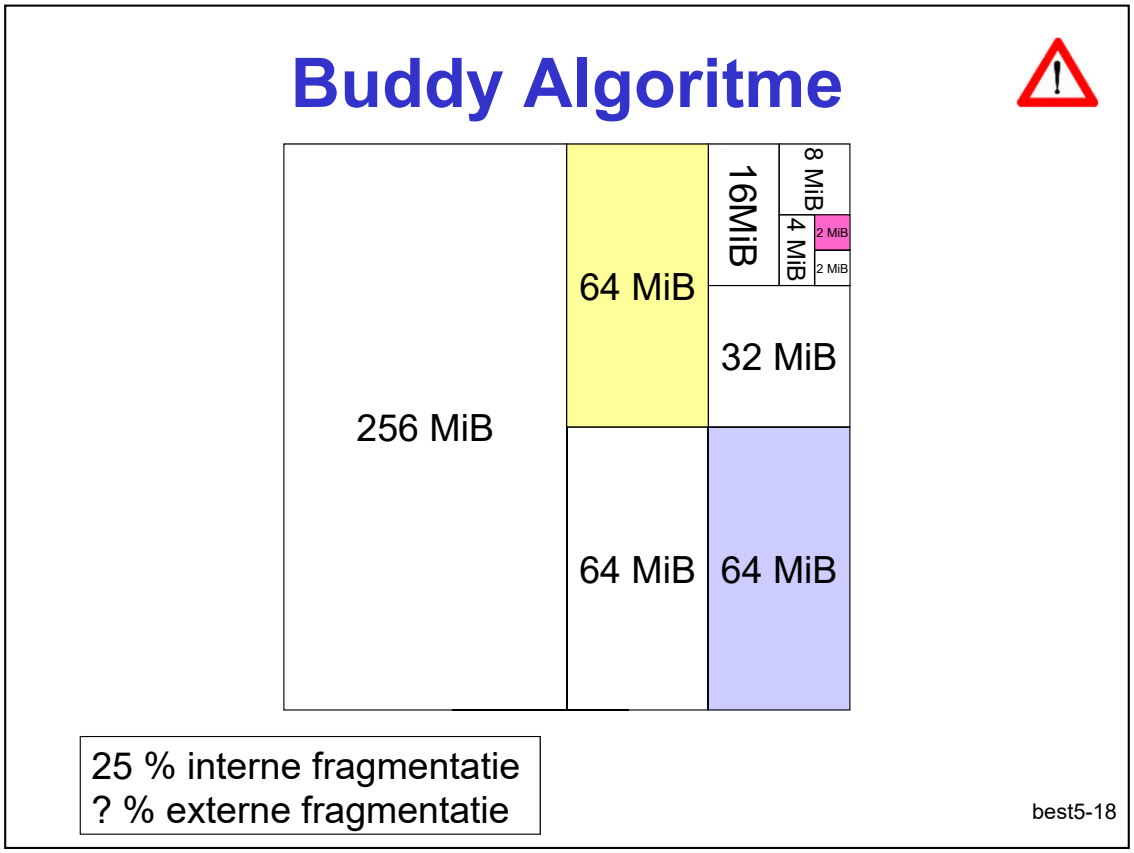


- k opeenvolgende blokken zoeken gaat snel
- boekhouding ingewikkelder bij vrijgeven geheugen

best5-17

Een alternatieve methode bestaat uit het bijhouden van de gealloceerde en de vrije blokken in een gelinkte lijst. De blokken kunnen een variabele grootte hebben. Het opzoeken van een stuk vrij geheugen van een gegeven grootte is eenvoudiger dan bij een bitmap. Eén vergelijking per vrij blok volstaat en men kan stoppen van zodra men een vrij blok van minstens de gevraagde grootte tegenkomt (first fit). Indien het gevonden blok te groot is, zal er een bijkomend element in de gelinkte lijst moeten aangemaakt worden die de rest van het vrije blok bijhoudt.

Het vrijgeven van een blok is ook eenvoudig. Het volstaat om de toestand van het blok van gealloceerd naar vrij te veranderen. Om vrije blokken samen te voegen moet er naar de twee aanliggende blokken gekeken worden (eenvoudig indien een dubbel gelinkte lijst gebruikt wordt).

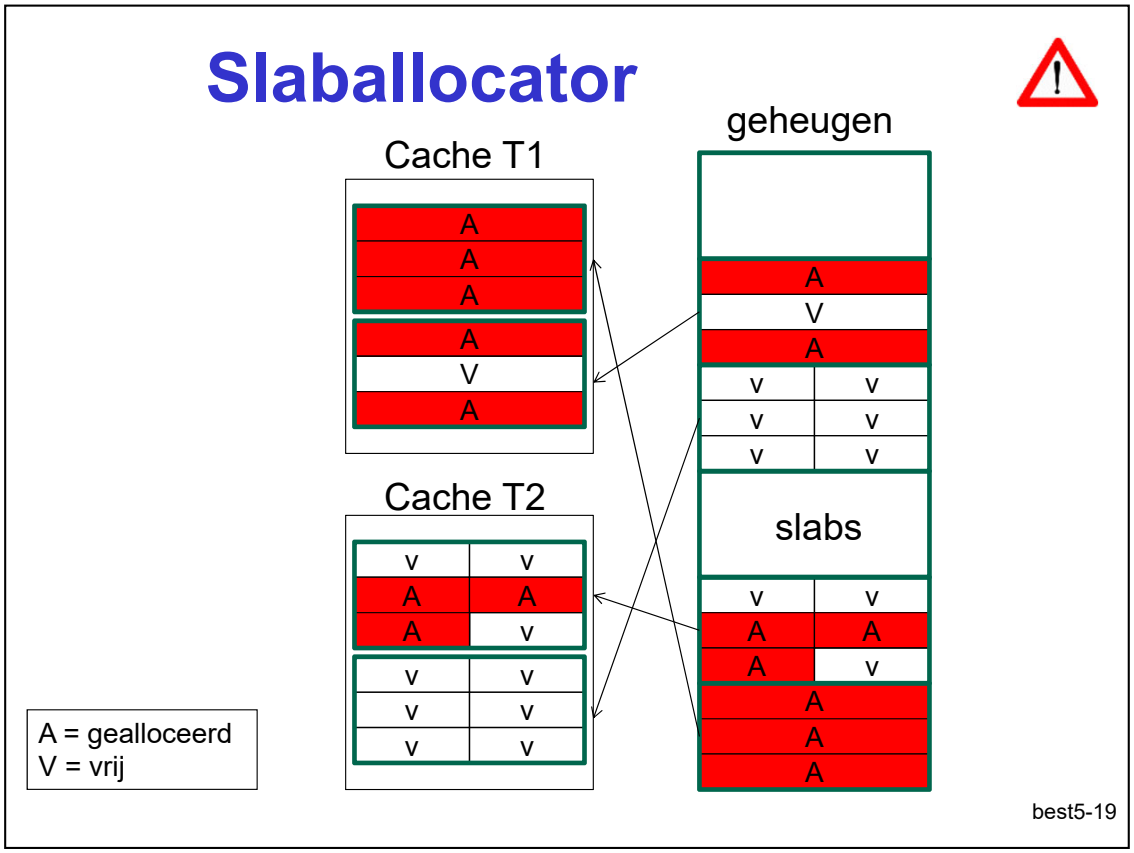


Een populair geheugenbeheeralgoritme is het zgn. Buddy Algoritme. In grote lijnen komt het erop neem dat men enkel maar blokken alloceert die een macht van 2 zijn en dat men de totale hoeveelheid geheugen hiërarchisch opsplitst (door telkens in 2 gelijke delen te splitsen, de zgn buddies) totdat men een blok heeft dat net groot genoeg is om de gevraagde allocatie te kunnen doen.

Het grootste voordeel van dit algoritme is dat het opsplitsen en het terug samenvoegen van twee buddies zeer efficiënt kan gebeuren omdat twee blokken die afkomstig zijn van de opsplitsing van een groter blok gemakkelijk herkenbaar zijn.

Het grootste nadeel van dit algoritme is dat in het slechtste geval slechts 50% van de gealloceerde geheugenruimte gebruikt zal worden door het proces. Gemiddeld zal slechts 75% van de gealloceerde geheugenruimte effectief door het proces gebruikt worden.

Verder is er ook nog de externe fragmentatie omdat de vrije blokken verspreid kunnen liggen over het volledige fysiek geheugen. Het is zelfs mogelijk dat twee belendende blokken niet samengevoegd kunnen worden omdat ze oorspronkelijk niet van hetzelfde grotere blok afstammen. In dat geval kunnen ze dan ook niet samen gealloceerd worden.



De buddyallocator heeft als nadeel dat de interne fragmentatie hoog kan oplopen. De slaballocator werkt deze interne fragmentatie weg. De basisidee is dat geheugenallocaties gegroepeerd worden per allocatiegrootte. In een objectgeoriënteerde programmeertaal is dit vrij eenvoudig implementeerbaar door te alloceren per objecttype. Alle allocaties worden bijgehouden in caches per type (of per grootte). De caches groeien niet per individuele allocatie maar per ‘slab’, dit is een contigu blok geheugen waarin verschillende allocaties kunnen plaatsvinden. Bij opname van een slab in een cache wordt het blok verkaveld in verschillende objecten die ‘vrij’ gemarkeerd worden. Bij een allocatieaanvraag wordt er in de cache naar een vrij object gezocht en wordt dit teruggegeven. Bij het zoeken wordt er eerst gezocht in partieel gevulde slabs. Pas dan wordt er gealloceerd uit een lege slab, of wordt er een slab toegevoegd. Bij het vrijgeven van een object, wordt er gecontroleerd over de slab waartoe het behoort eventueel leeg geworden is. Dan kan beslist worden om de slab (eventueel) uit het cache te verwijderen. De slaballocator wordt in Linux gebruikt.

Geheugensanering (garbage collection)

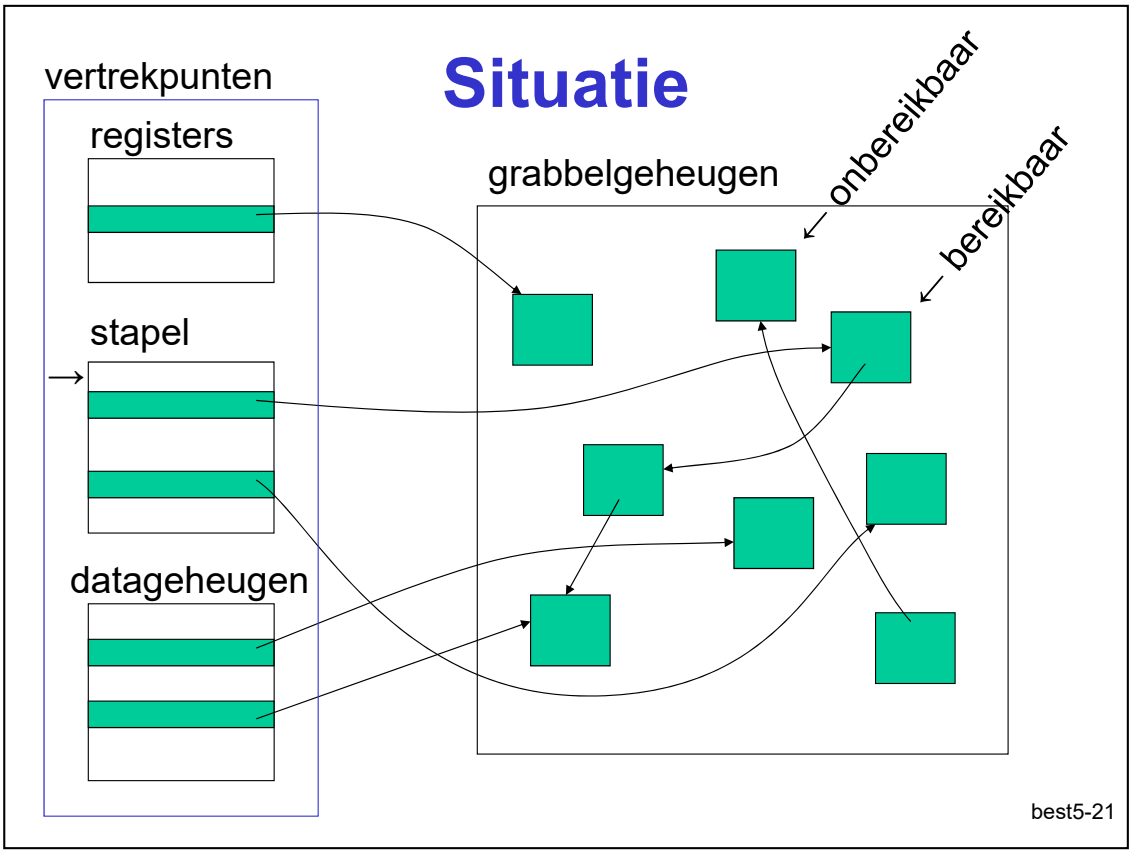
- Het automatisch beheer van dynamisch gealloceerd geheugen
- Meer specifiek het automatisch vrijgeven van objecten die niet langer door een proces gebruikt worden
  - –Voordeel: minder werk dan manueel, foutloos
  - –Nadeel: trager dan manueel

best5-20

Bij geheugensanering hoeft men de gealloceerde blokken geheugen niet langer manueel vrij te geven. Op bepaalde ogenblikken gaat de geheugenbeheerssoftware zelf controleren welke gealloceerde blokken er niet langer in gebruik zijn, en deze blokken worden dan automatisch vrijgegeven.

Dit heeft talrijke voordelen: de programmeur hoeft zich geen zorgen meer te maken over het geheugenbeheer – meer bepaald over het correct vrijgeven van blokken geheugen. Fouten zoals te vroeg vrijgegeven blokken (bengelende wijzers – dangling pointers) of te laat (of nooit) vrijgegeven blokken (geheugenlekken – memory leaks) kunnen in principe niet meer voorkomen. Dit komt de ontwikkelsnelheid en de betrouwbaarheid van de code ten goede.

Het voornaamste minpunt van geheugensanering is dat het doorgaans een stuk trager is dan manueel geheugenbeheer en dat het niet gebruikt kan worden in combinatie met eender welke programmeertaal omdat de taal moet toelaten om automatisch te achterhalen welke blokken geheugen er in gebruik zijn en dewelke niet. In een taal waar men vrij liberaal met adressen kan omspringen zoals in C is dit zeer moeilijk realiseerbaar. In een taal zoals Java is geheugensanering nagenoeg de enige methode om aan geheugenbeheer te doen. In wat volgt zullen we spreken over objecten i.p.v. blokken geheugen en wordt het Java model als basismodel gehanteerd (zonder daartoe beperkt te zijn).



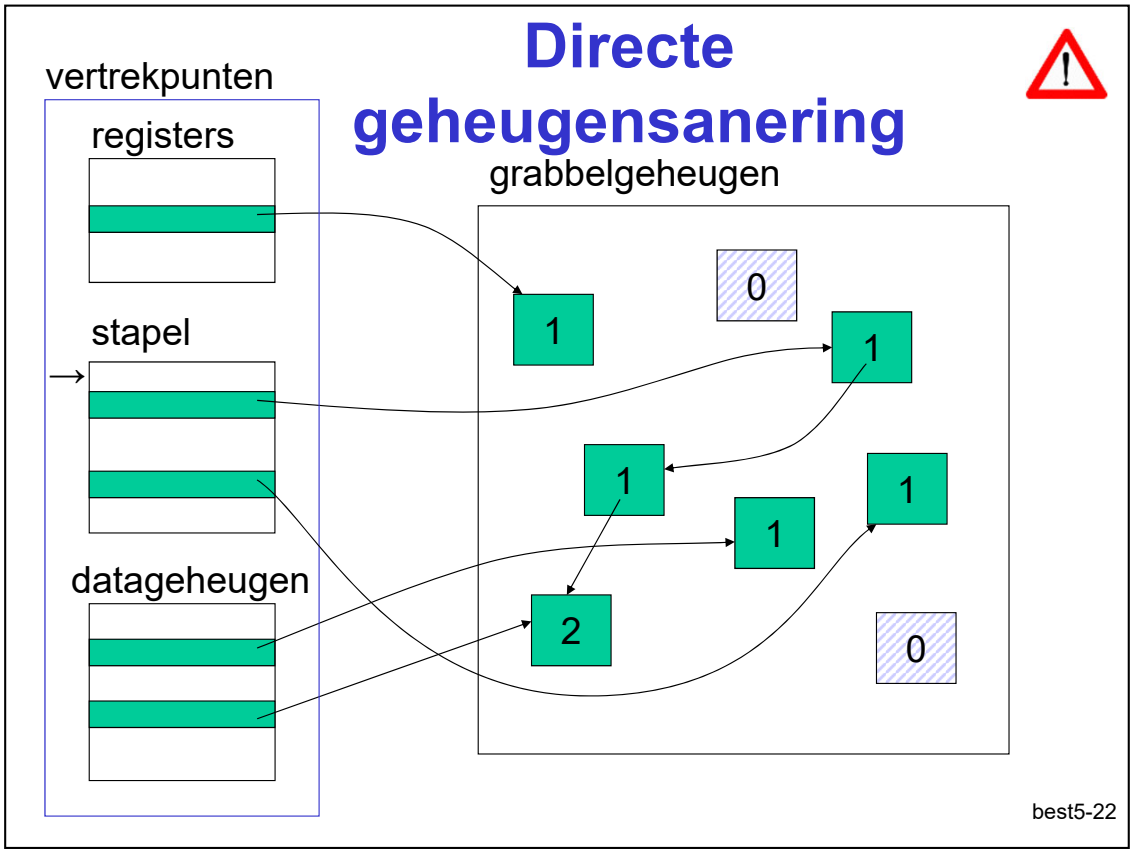
Deze figuur geeft een typische situatie weer van de manier waarop een programma omgaat met dynamisch gealloceerde geheugen uit het grabbelgeheugen. Objecten van veranderlijke grootte (hier allemaal voorgesteld door blokken van dezelfde grootte), worden gealloceerd in het grabbelgeheugen, en de adressen (referenties) van de gealloceerde objecten worden door het programma op verschillende manieren gebruikt:

- Adressen kunnen opgenomen worden in andere objecten (b.v. om een graaf op te bouwen in het geheugen)
- Adressen kunnen opgenomen worden in statisch gealloceerde veranderlijken in het (statische) datagebied van het programma.
- Adressen kunnen voorkomen op de stapel van het programma
- Adressen kunnen voorkomen in de registers van de processor.

Als we nu willen achterhalen welke objecten er allemaal in gebruik zijn, dan moeten we in eerste instantie proberen achterhalen of ze wel bereikbaar zijn voor het programma. Daarvoor kan men vertrekken van de ‘vertrekpunten’ (root set) en via het aflopen van de wijzers kijken welke objecten er allemaal bereikbaar zijn. De objecten die niet bereikbaar zijn mogen in de praktijk als niet meer in gebruik beschouwd worden en mogen gerecycleerd worden.

In de praktijk zijn er twee manieren om te achterhalen welke objecten er bereikbaar zijn, en welke objecten niet:

- Door het voeren van een geheugenboekhouding die continu bijhoudt of een object bereikbaar is – directe geheugensanering
- Door af en toe alle wijzers af te lopen en op die manier (on-)bereikbaarheid vast te stellen: afscannende geheugensanering.

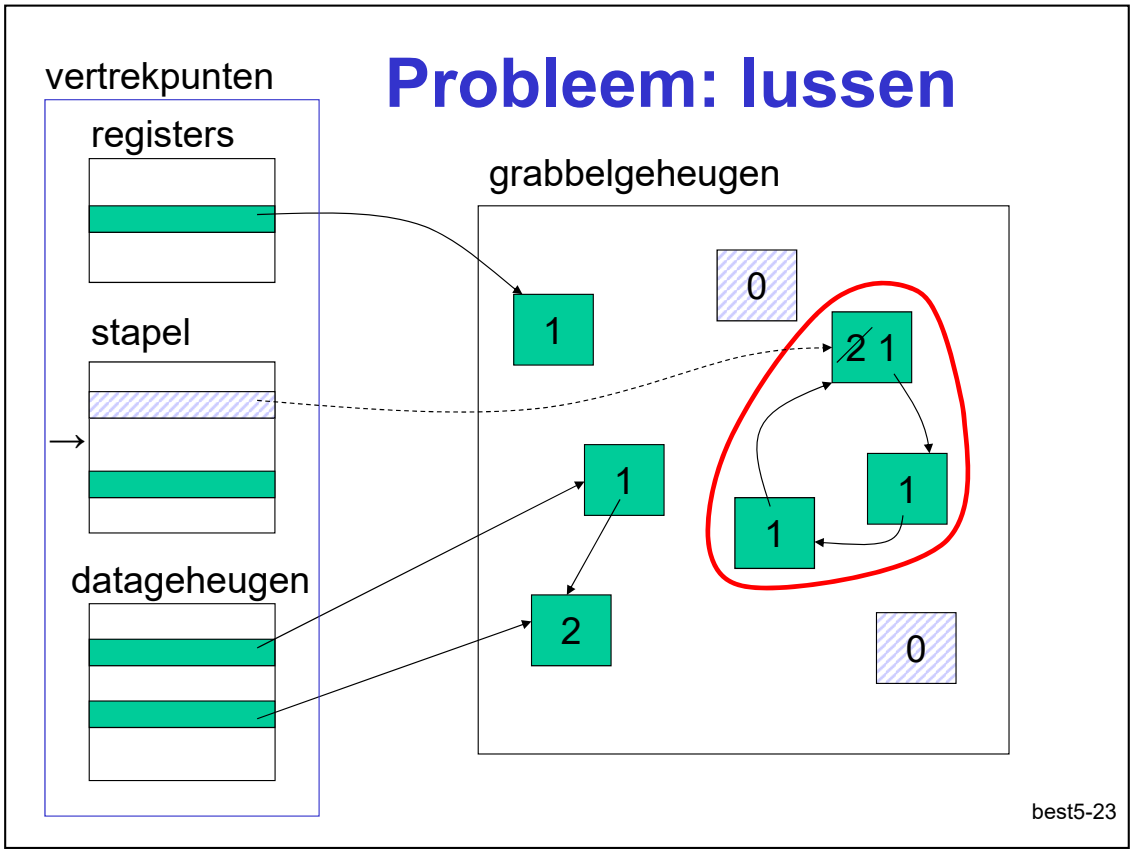


Bij directe geheugensanering wordt er per object een tellertje (reference count) bijgehouden. Dit tellertje houdt bij door hoeveel andere objecten dit object gerefereerd wordt. Het object wordt als onbereikbaar beschouwd van zodra zijn tellertje op 0 komt te staan.

Het tellertje wordt bijgewerkt telkens wanneer een wijzer weggeschreven wordt. Indien een wijzer overschreven wordt, dan moet het tellertje van het object waarvan de wijzer overschreven wordt, verlaagd worden, en het tellertje van het nieuwe object verhoogd worden. Indien een tellertje op 0 komt te staan kan het object meteen toegevoegd worden aan de lijst met vrije objecten. De tellertjes van alle objecten waarnaar dat vrijgekomen object verwijst, moeten ook allemaal verlaagd worden, hetgeen eventueel kan leiden tot nog meer onbereikbare objecten.

Het bijhouden van al deze tellertjes zorgt uiteraard voor een niet-verwaarloosbare vertraging van de uitvoering van het programma. Het overschrijven van de beginwijzer van een graaf zal er toe leiden dat heel de graaf kan vrijgegeven worden.

Daar staat dan wel tegenover dat het vinden van een vrij blok zeer eenvoudig is. Het volstaat om het gaan zoeken in de objecten met tellertje = 0.

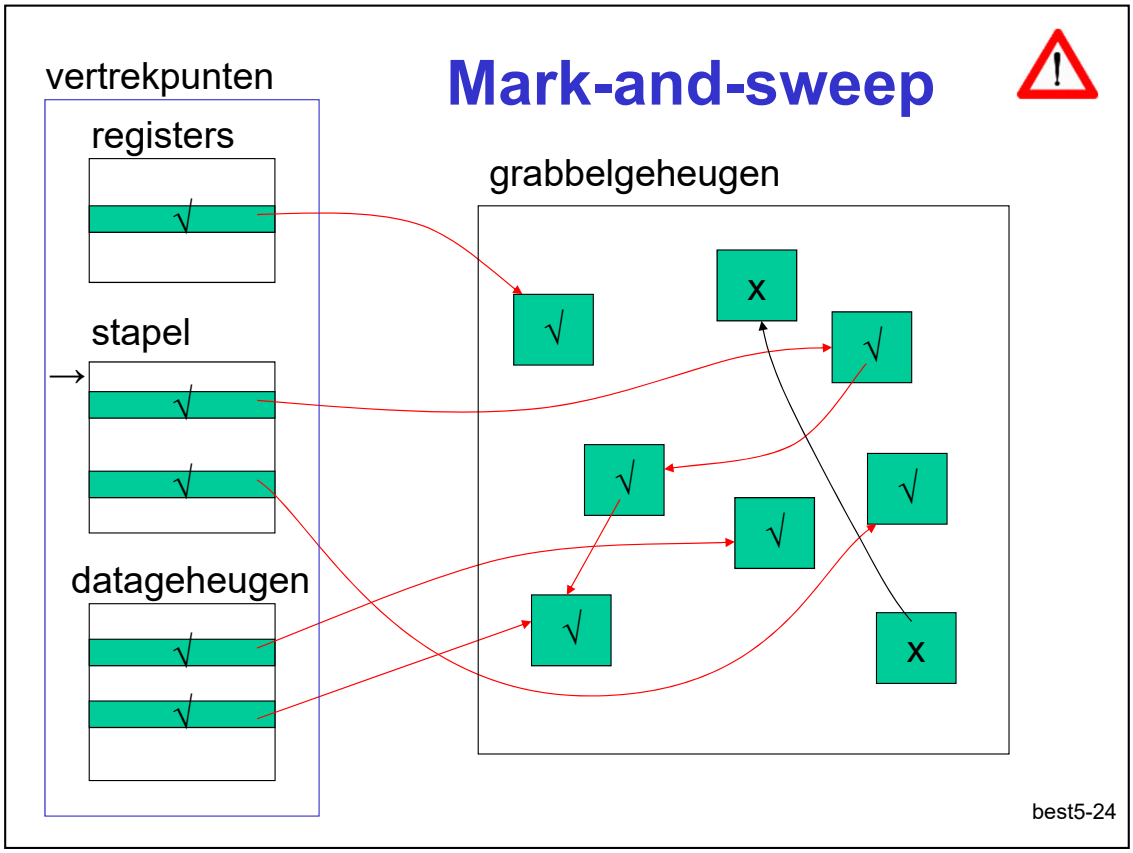


Directe geheugensanering (reference counting) faalt indien er zich in de gegevensstructuren lussen bevinden zoals getoond in de bovenstaande figuur. Bij het wegvallen van de verwijzing vanuit de stapel wordt het tellertje van het betrokken object verlaagd, maar het komt door de aanwezigheid van de circulaire referenties niet op 0 te staan.

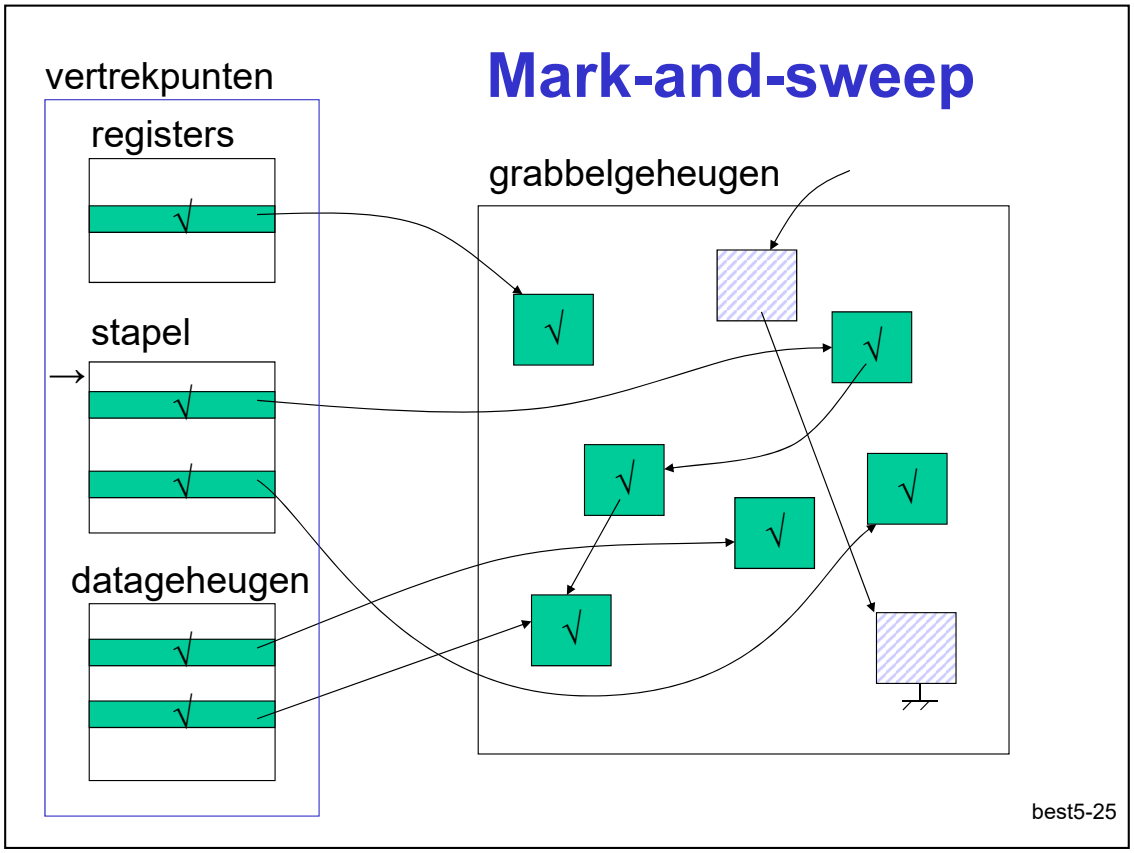
De gemakkelijkste manier om dit probleem te omzeilen is om het te negeren en nu en dan eens alle gegevensstructuren af te lopen met een afscannende geheugensanering (zie verder).

Samengevat voor directe geheugensanering

- Voordelen
  - Kost van de geheugensanering wordt gespreid over de gehele uitvoering van het programma
  - Onbereikbare object worden meteen toegevoegd aan de lijst met vrije blokken
- Nadelen
  - Aanzienlijke overhead per wijzermanipulatie
  - Bijkomende ruimte nodig om de tellertjes op te slaan
  - Niet 100% juist (lussen)



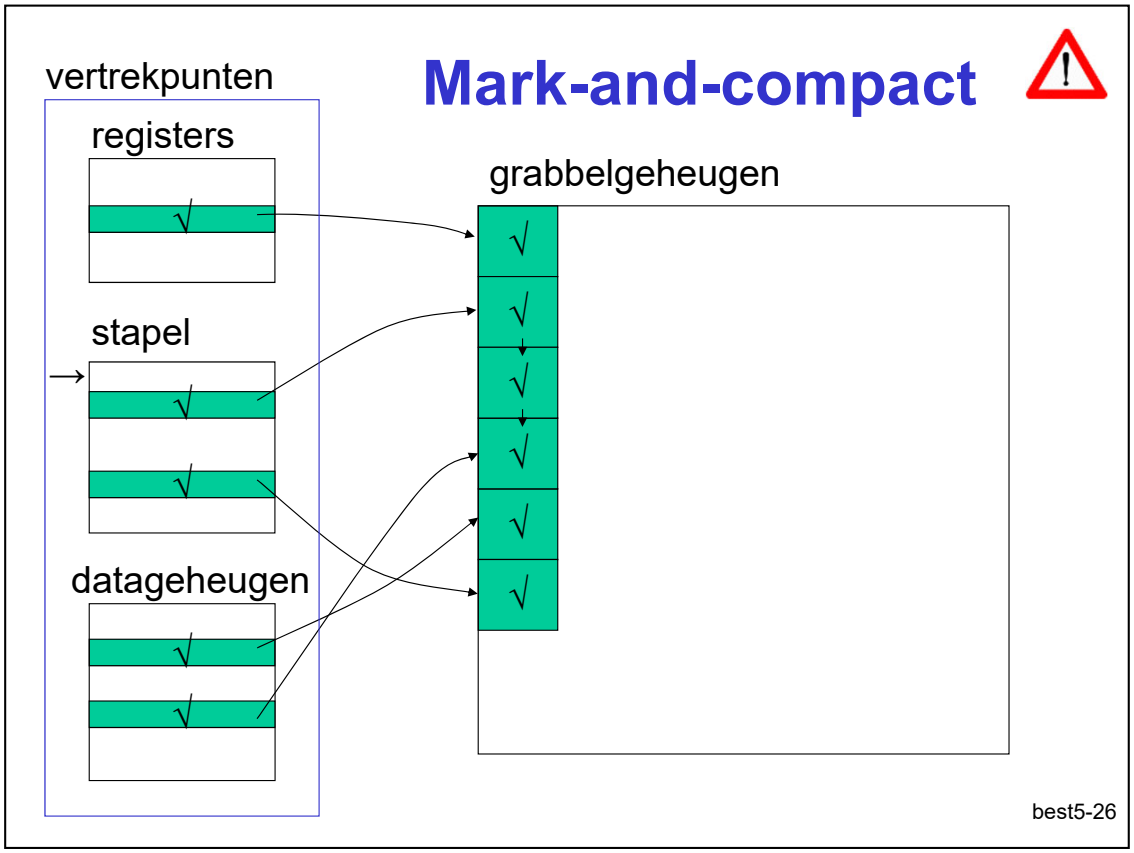
In de praktijk wordt directe geheugensanering niet zo vaak gebruikt, maar geeft men de voorkeur aan een afscannende geheugensanering. Het principe van de afscannende geheugensanering is dat – vertrekkende van de vertrekpunten – alle wijzers afgelopen worden en dat op die manier alle bereikbare objecten gemarkeerd worden. Objecten die via 2 verschillende paden bereikbaar zijn hoeven uiteraard maar eenmaal gemarkeerd te worden. De uitvoeringstijd van de markeerfase is evenredig met het aantal bereikbare objecten in het grabbelgeheugen – niet met de grootte van het geheugen. Onbereikbare objecten worden immers niet bezocht.



Na de markeerfase kan men de onbereikbare objecten opnieuw verzamelen in een lijst met vrije blokken. Deze fase noemt men de sweep-fase (vandaar de naam van dit algoritme: mark-and-sweep). Na de sweep-fase hebben we dus inhoudelijke dezelfde lijst van vrije blokken als deze verzameld door de directe geheugensanering (met uitzondering van de lussen uiteraard). Tijdens de sweep fase worden alle objecten van het grabbelgeheugen afgelopen.

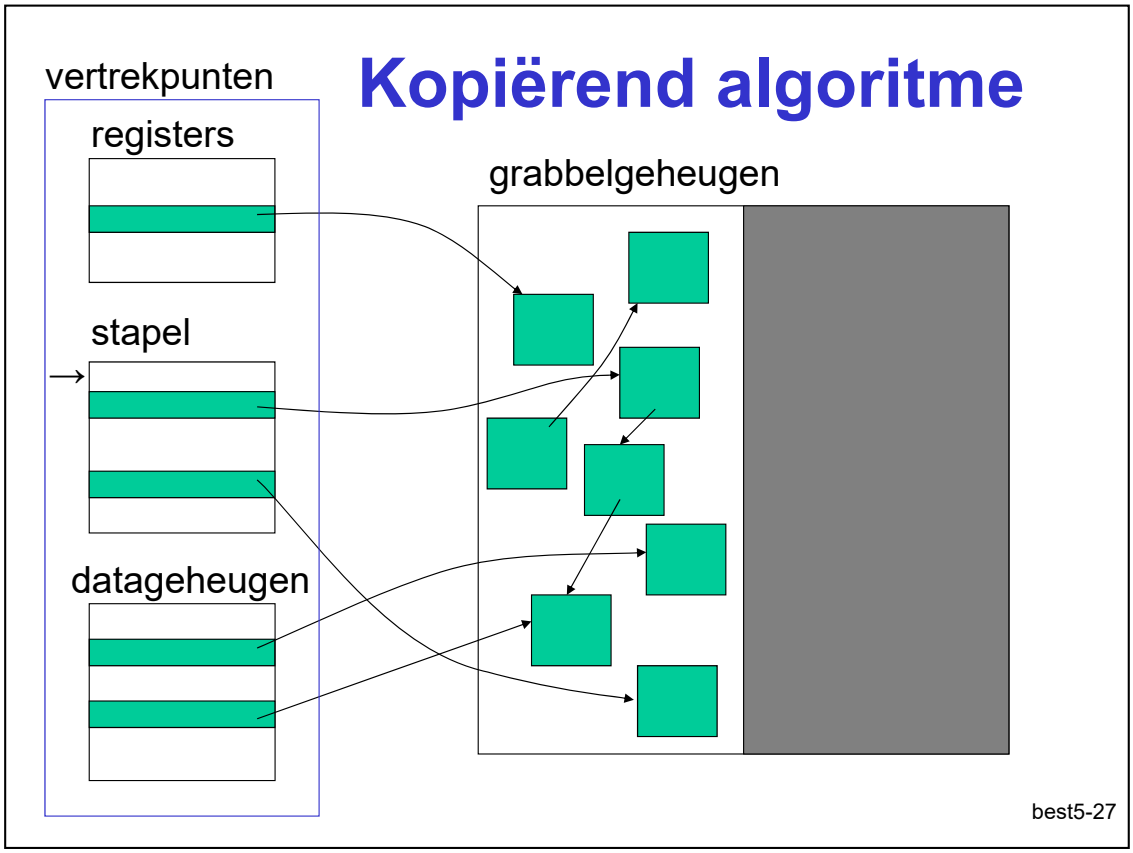
Mark-and-sweep was het eerste afscannende geheugensaneringsalgoritme. Een afscannende geheugensanering wordt niet continu uitgevoerd, maar wordt opgeroepen op het moment dat het grabbelgeheugen vol dreigt te geraken. Op dat moment kan het nodig zijn om verschillende honderden MiB aan geheugen af te scannen op zoek naar bereikbare objecten. Dit kan redelijk wat tijd in beslag nemen (seconde of meer). Verder vereist het algoritme dat het programma ondertussen het grabbelgeheugen onaangeroerd laat omdat tijdens de uitvoering de wijzers stabiel moeten blijven. In de praktijk zal men de uitvoering van het programma dan ook moeten stilleggen (zgn. ‘stop the world’). Dit is uiteraard onaanvaardbaar voor systemen die in ware tijd moeten functioneren.

Niettegenstaande al deze nadelen is mark-and-sweep globaal toch sneller dan reference counting en heeft het betere eigenschappen (het detecteert ook lussen). De meeste systemen voor geheugensanering zijn gebaseerd op mark-and-sweep.

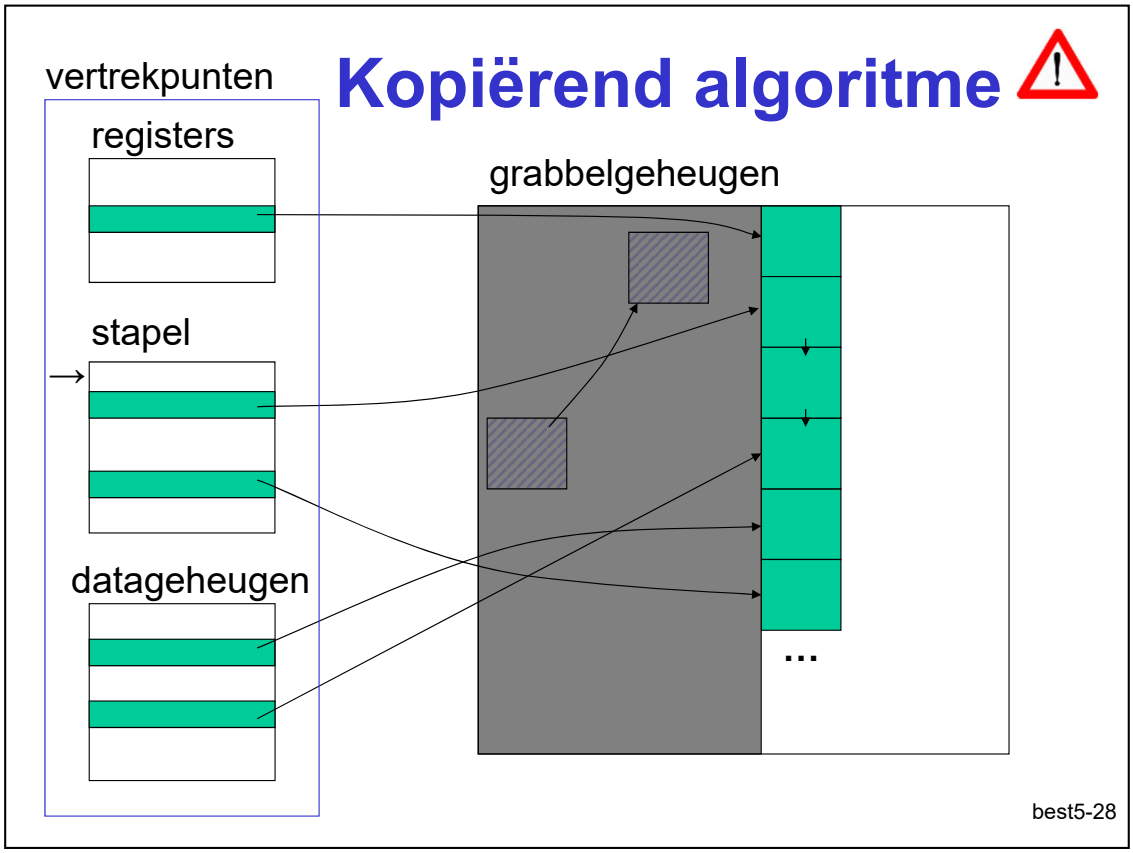


De oorspronkelijke mark-and-sweep heeft als nadeel dat het vrije geheugen in gefragmenteerde vorm achterlaat. Daarom bestaat er naast een mark-and-sweep ook een mark-and-compact waarbij het geheugen na de markeerfase gecompacteerd wordt. Hierbij worden alle bereikbare objecten naar een contigu gebied van het grabbelgeheugen gekopieerd. Dit vereist natuurlijk dat de programmeertaal die toelaat dat objecten in het geheugen verplaatst worden tijdens de geheugensanering (alle wijzers naar het object zullen uiteraard moeten aangepast worden). Mark-and-compact is snel indien er maar een klein aantal bereikbare objecten zijn (en veel onbereikbare). Het compacteren van de objecten in dezelfde geheugenruimte is echter niet eenvoudig en vereist dat objecten soms verschillende keren bezocht moeten worden om tot een goed resultaat te leiden.

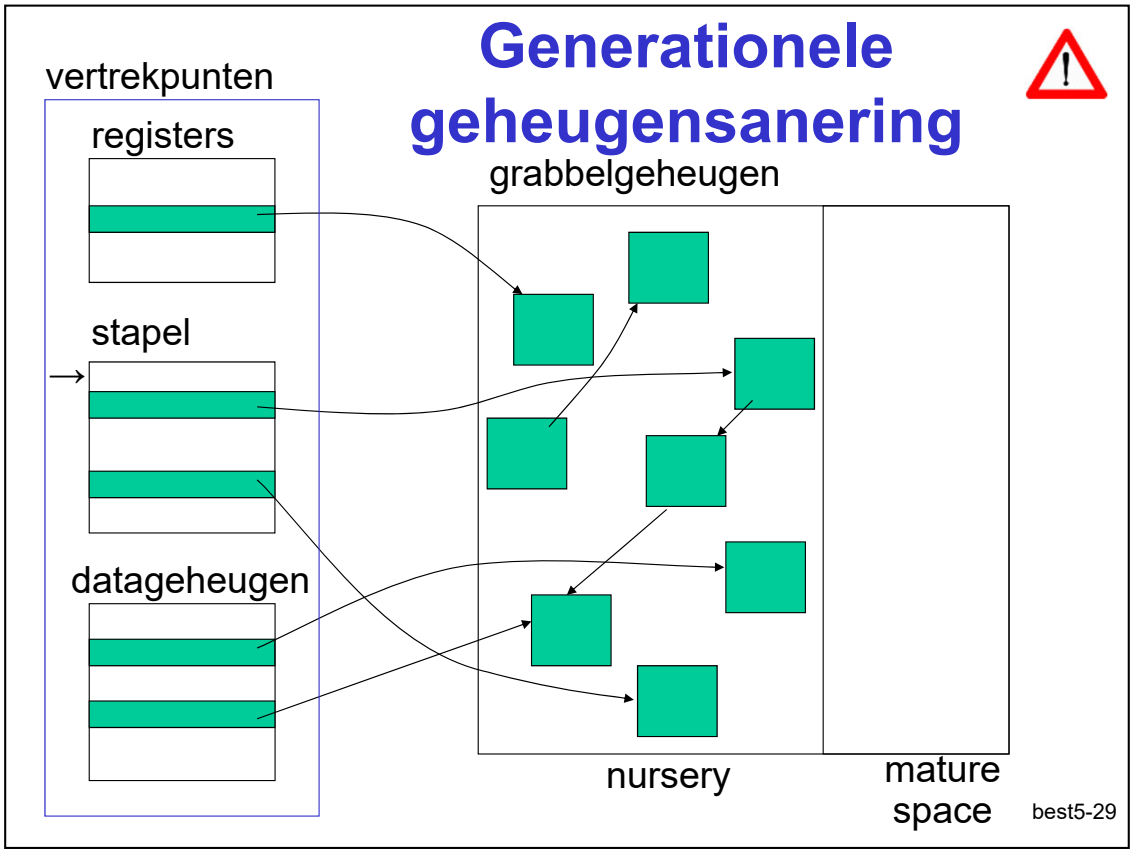
Dit compacteren neemt veel tijd in beslag, maar het grote voordeel van compacteren is dan wel dat het vrije geheugen als één groot blok ter beschikking komt, en dat de allocatie veel gemakkelijker wordt (men kan gewoon sequentieel de objecten alloceren in het geheugen – het beheer van de vrije lijst valt compleet weg).



Een mogelijke implementatie van kopiëren is om het grabbelgeheugen te onderverdelen in twee helften, en de bereikbare objecten meteen te kopiëren naar de vrije helft.



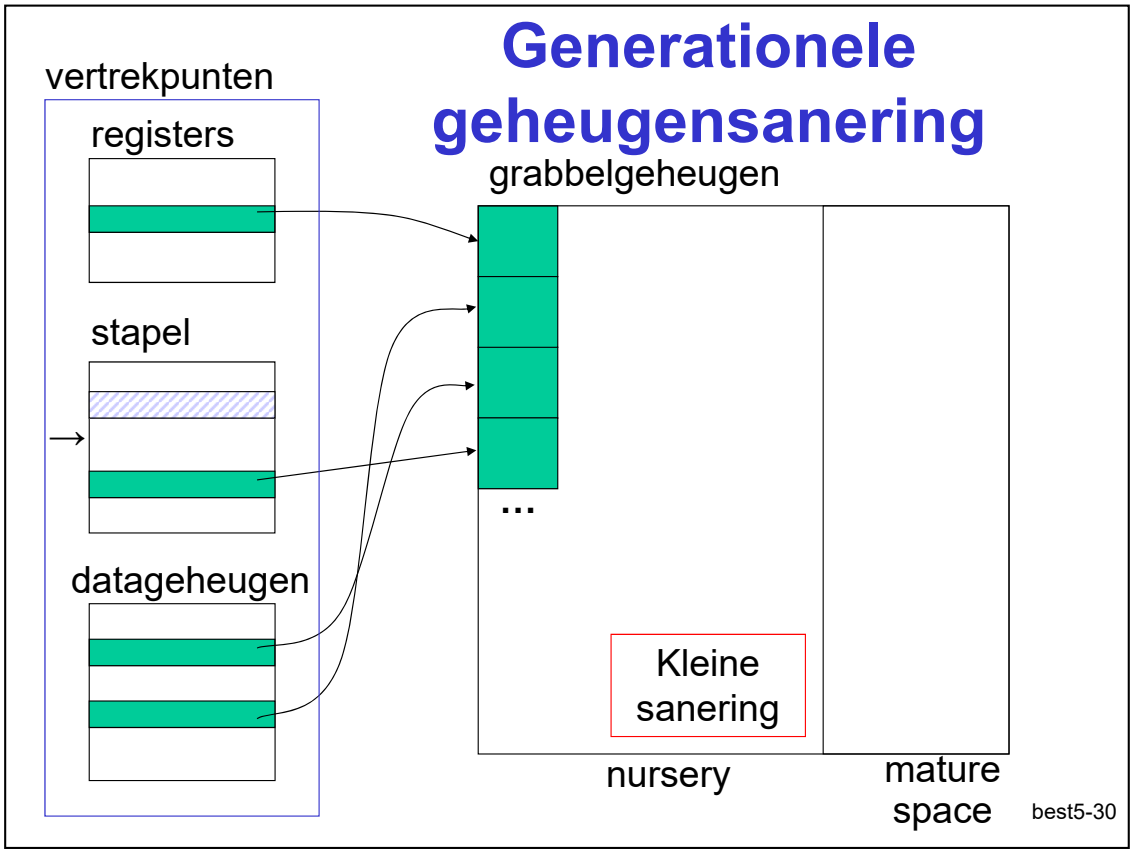
Nadat alle bereikbare objecten gekopieerd werden kan de uitvoering van het programma gewoon verder gezet worden in de tweede helft van het grabbelgeheugen. De oorspronkelijk helft blijft dan beschikbaar om bij een volgende geheugensanering de blokken terug naar de eerste helft te kopiëren. Het voordeel van dit algoritme t.o.v. van mark-and-compact is dat het kopiëren eenvoudiger is omdat er kan gekopieerd worden naar een totaal vrije ruimte. Het grootste nadeel is dat slechts de helft van het grabbelgeheugen effectief gebruikt kan worden. Voor kleine grabbelgeheugens kan dit in absolute termen misschien nog meevallen, maar voor grote geheugens is dit niet aanvaardbaar. Het grote voordeel van dit algoritme is echter opnieuw dat de allocatie achteraf bijzonder eenvoudig wordt.



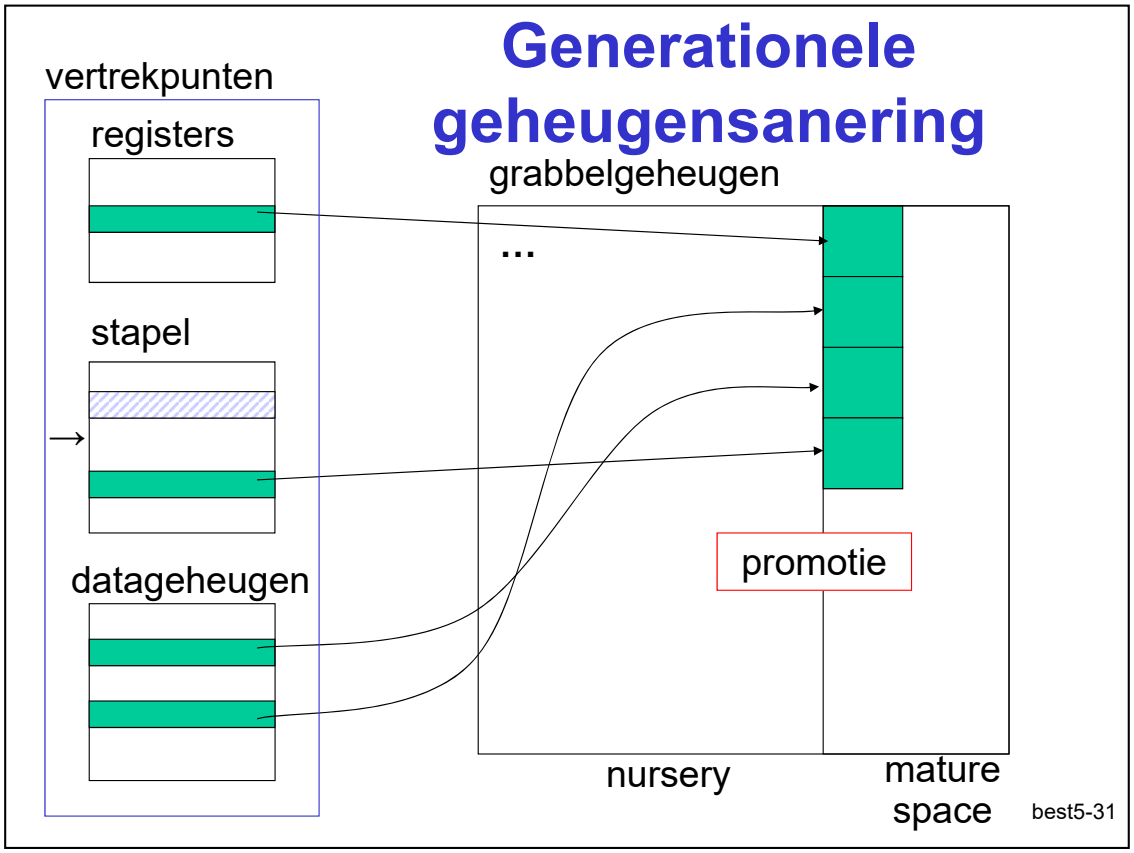
De afscannende algoritmen overlopen steeds alle bereikbare objecten. Als men dan nagaat wat er in werkelijkheid gebeurt is het vaak zo dat de geheugensanering veel onnodig werk doet door telkens opnieuw oude objecten af te scannen, vast te stellen dat ze nog steeds bereikbaar zijn en ze al dan niet te compacteren of te kopiëren. De afscannende algoritmen presteren het best bij de meest recent gealloceerde objecten. Dit komt omdat heel wat objecten kortlevend zijn (b.v. strings) en er onder dat type van objecten veel te saneren valt.

Generationele geheugensanering is op deze vaststelling gebaseerd. Het grabbelgeheugen wordt gepartitioneerd in generaties en objecten worden gealloceerd in de jongste generatie en hoe langer ze leven hoe verder ze opschuiven naar oudere generaties.

In het geval er slechts 2 generaties zijn, worden deze vaak de nursery en de mature space genoemd.



De nursery is kleiner dan het volledige grabbelgeheugen. Hierdoor zal deze dus sneller vol zitten met objecten, maar de sanering ervan zal ook minder tijd in beslag nemen. Een dergelijke sanering noemt men een kleine sanering (minor collection).



Objecten die een kleine sanering enkele keren overleefd hebben, kunnen overgebracht worden naar de volgende generatie (de zgn. promotie). De mature space zal na verloop van tijd ook vol zitten met objecten. Dan wordt er een geheugensanering op de mature space uitgevoerd die men een grote sanering (major collection) noemt.

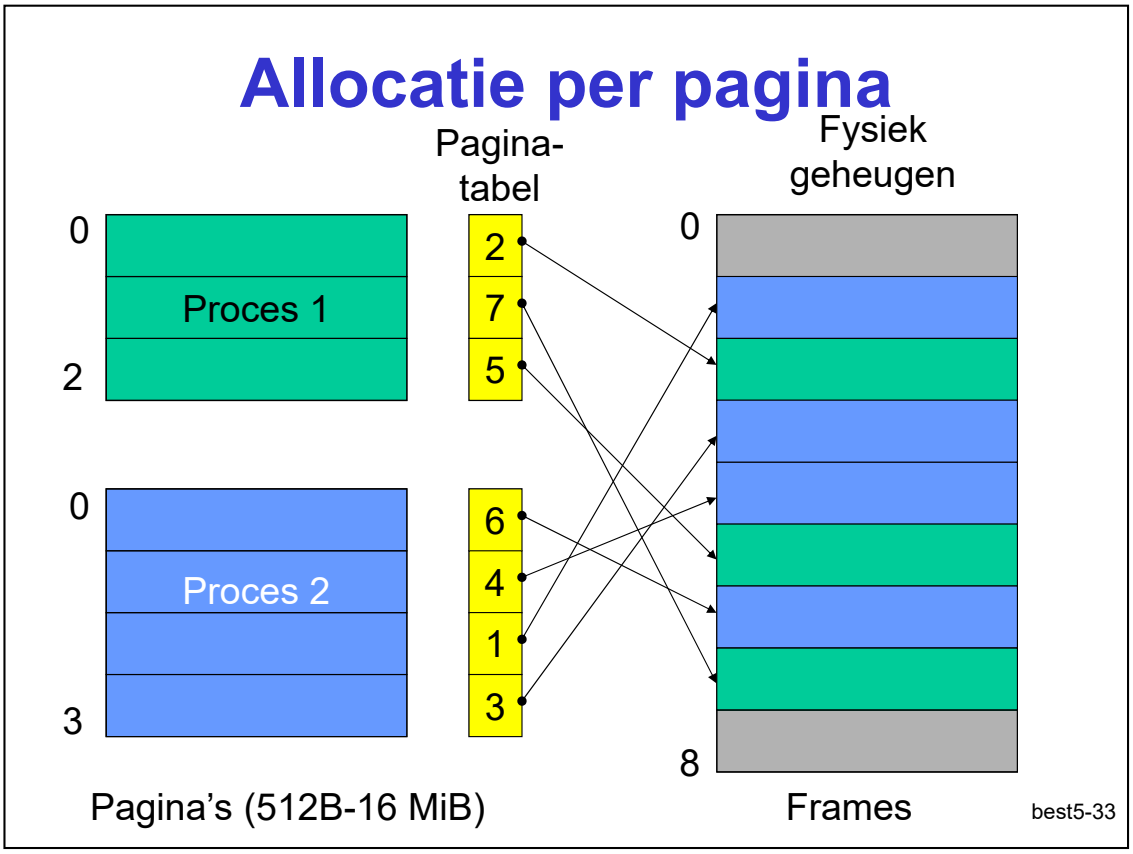
Om onnodig kopiëren en scannen te voorkomen kan men ook beslissen om een aantal objecten een definitieve plaats te geven in het grabbelgeheugen waardoor ze niet langer zullen betrokken worden bij het saneringsproces. Ze worden dan als het ware onsterfelijk.

Generationele geheugensanering heeft zeer veel goede eigenschappen, maar er kleeft ook een nadeel aan vast. Er moet bij de geheugensanering toegezien worden op intergenerationele wijzers. Eigenlijk zijn alle wijzers die van de mature space naar de nursery vertrekken ook vertrekpunten voor een kleine sanering. Een lijst van dergelijke wijzers moet dus afzonderlijk bijgehouden worden en dit vergt een bijkomende inspanning.

Overzicht

- Logische vs. fysieke adressen
- Contigue allocatie van geheugen
- Niet-contigue allocatie van geheugen
  - Paginering
  - Segmentering
  - Segmentering + paginering

best5-32



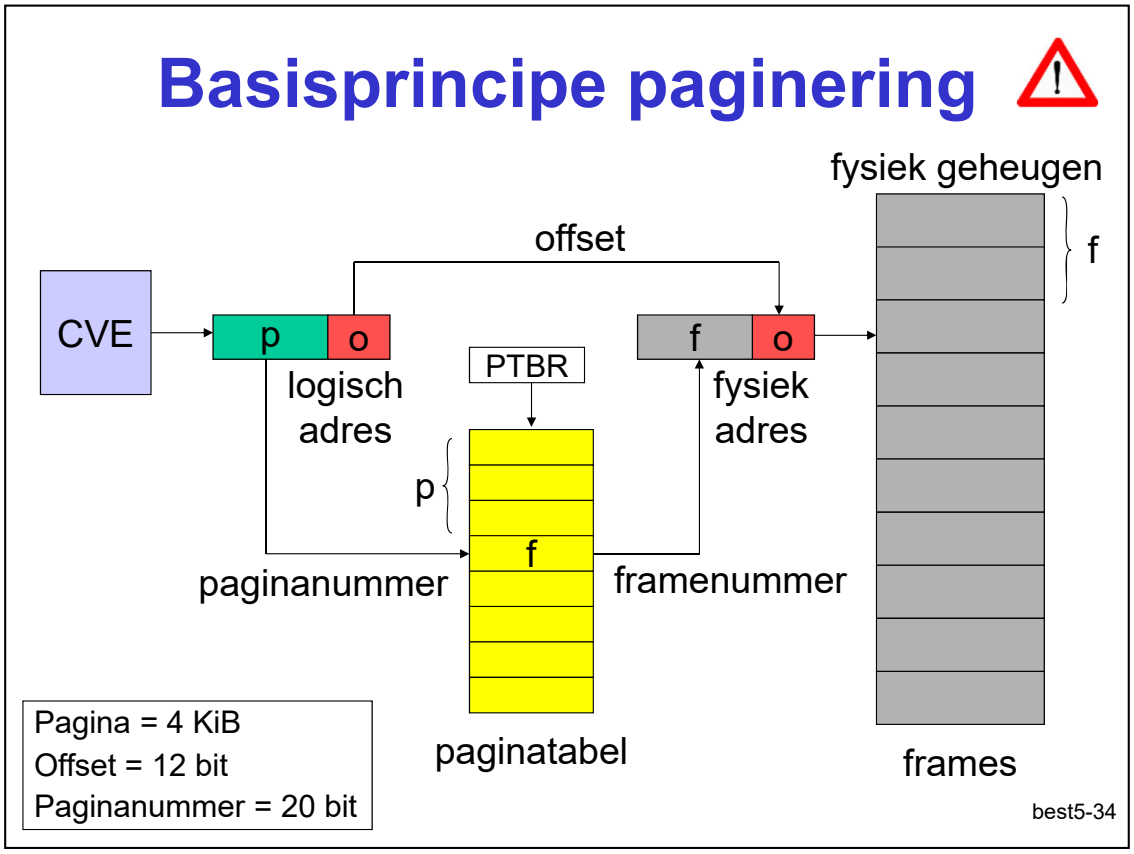
Allocatie van aaneengesloten blokken veroorzaakt heel wat problemen in het beheer van het hoofdgeheugen (en ook in het beheer van de swapruimte – zie later). Daarom maakt men tegenwoordig bijna altijd gebruik van niet-contigue allocatie van het fysiek geheugen.

Daarvoor deelt men het proces op in zgn. **pagina's**, en het fysiek geheugen in **frames**. Pagina's en frames hebben een vaste grootte die varieert tussen 512 byte tot 16 MiB. In de praktijk zijn groottes van 4 KiB en 8 KiB meest courant.

Indien een proces uit 3 pagina's bestaat, volstaat het nu om 3 vrije frames te vinden om deze pagina's op af te beelden. Zolang er ergens in het systeem 3 vrije frames beschikbaar zijn, zal het proces kunnen ingeladen worden. Het probleem van de externe fragmentatie wordt hiermee effectief opgelost. Doordat een proces niet noodzakelijk even groot is als een veelvoud van de paginagrootte, zal er gemiddeld een halve pagina interne fragmentatie ontstaan. Behoudens in die gevallen waarbij men kiest voor extreem grote pagina's, stelt dit geen probleem.

De informatie over de relatie tussen de logische adressen en de fysieke adressen wordt bijhouden in de zgn. paginatabel. De afbeeldingsfunctie ξ zal van deze tabel gebruik maken om de logische adressen op de fysieke adressen af te beelden. We hebben hier te maken met een afbeeldingsfunctie die aanzienlijk complexer is dan bij het gebruik van een relocatieregister.

Men kan paginering zien als een uitbreiding van het relocatieregister waarbij elke pagina van het proces over zijn eigen relocatieregister beschikt en waarbij het limietregister een vaste waarde heeft (en in een aantal gevallen zelfs vastgelegd werd door de hardware).



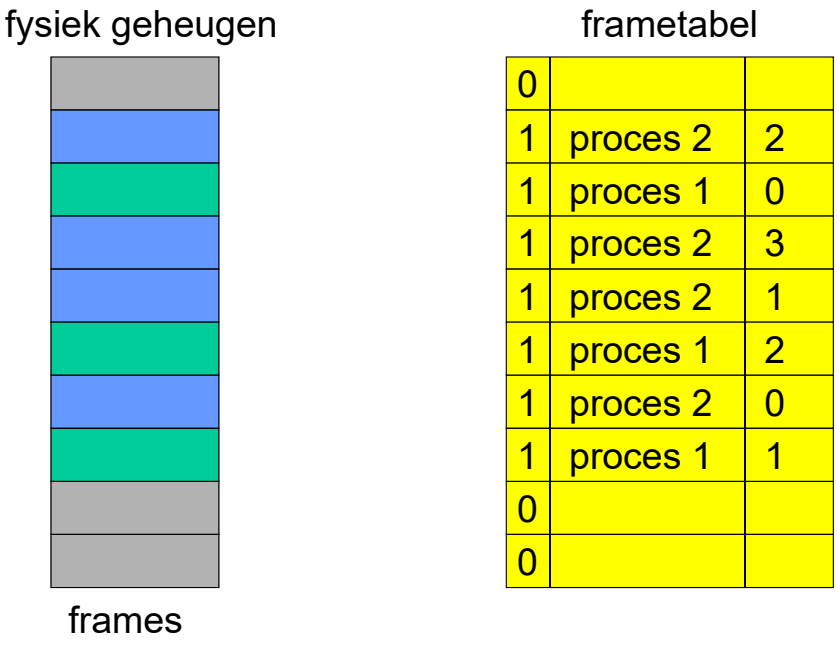
De geheugenbeheerseenheid werkt zoals hierboven schematisch weergegeven. Het logisch adres zoals dit door de CVE gegenereerd wordt, wordt opgedeeld in een paginanummer en een offset (b.v. 32 bit = 20 bit + 12 bit). Het paginanummer wordt gebruikt om de paginatabel te indexeren. Op het p-de element in de paginatabel ligt het framenummer opgeslagen. Het paginanummer wordt vervangen door het framenummer en de offset blijft behouden. Dit is enkel mogelijk indien de frames gealigneerd zijn in het geheugen waardoor ze steeds eindigen op (in dit geval) 12 nulbits.

Een alternatieve implementatie kan bestaat uit het opslaan van het basisadres van het frame i.p.v. het framenummer. In dat geval moet de offset natuurlijk bij het basisadres opgeteld worden om het fysiek adres te berekenen. Het nadeel van deze methode is dat ze meer rekenwerk vergt. Het voordeel is dat de frames niet gealigneerd hoeven te zijn in het geheugen.

Per proces hebben we nu een paginatabel bij te houden. De paginatabel wordt aangewezen door middel van een speciaal register in de geheugenbeheerseenheid: het page table base register (PTBR). Bij elke contextwisseling moet – samen met de registers voor algemeen gebruik – ook de PTBR mee bewaard en hersteld worden.

Paginering ondersteunt automatisch ook protectie. De paginatabel is niet manipuleerbaar door een gebruikersproces en bevat enkel verwijzingen naar die frames die toegewezen werden aan het gebruikersproces. Daardoor is een gebruikersproces dus niet in staat om het fysiek geheugen te adresseren dat het niet bezit. Om aan te geven of een bepaalde pagina ook een corresponderend frame bezit wordt er per pagina in de paginatabel een extra bit bijgehouden (valid). Indien het valid bit op vals staat, zal de adresvertalingshardware een adresseringsfout genereren.

Frametabel



best5-35

Het besturingssysteem houdt intern een frametabel bij, dit is een tabel die per geheugenframe bijhoudt of het al dat niet in gebruik is, en indien het in gebruik is, door welk proces en eventueel het corresponderende paginanummer uit dat proces. Het bijhouden van deze datastructuur is essentieel om snel vrije frames te kunnen vinden indien een proces daarom vraagt.

Additionele informatie

Paginatabel (4 KiB)

21221000
54F54000
545E5000
5751A000
25487000

12 ongebruikte bits

- Valid
- Read-only
- Dirty
- Reference
- Lock
- Cached
- …

best5-36

Als de frames gealigneerd zijn in het geheugen, zullen de laagste adresbits altijd op 0 staan. Bij een framegrootte van 4 KiB gaat het om 12 bits die eigenlijk geen nuttige informatie bevatten. Deze bits kunnen gebruikt worden om extra informatie over een pagina bij te houden. Deze informatie zal ons toelaten om later virtueel geheugen te implementeren (zie later).

Vaak voorkomende informatiebits zijn:

-Valid bit (of present bit): houdt bij of de pagina een geassocieerd frame bezit of niet

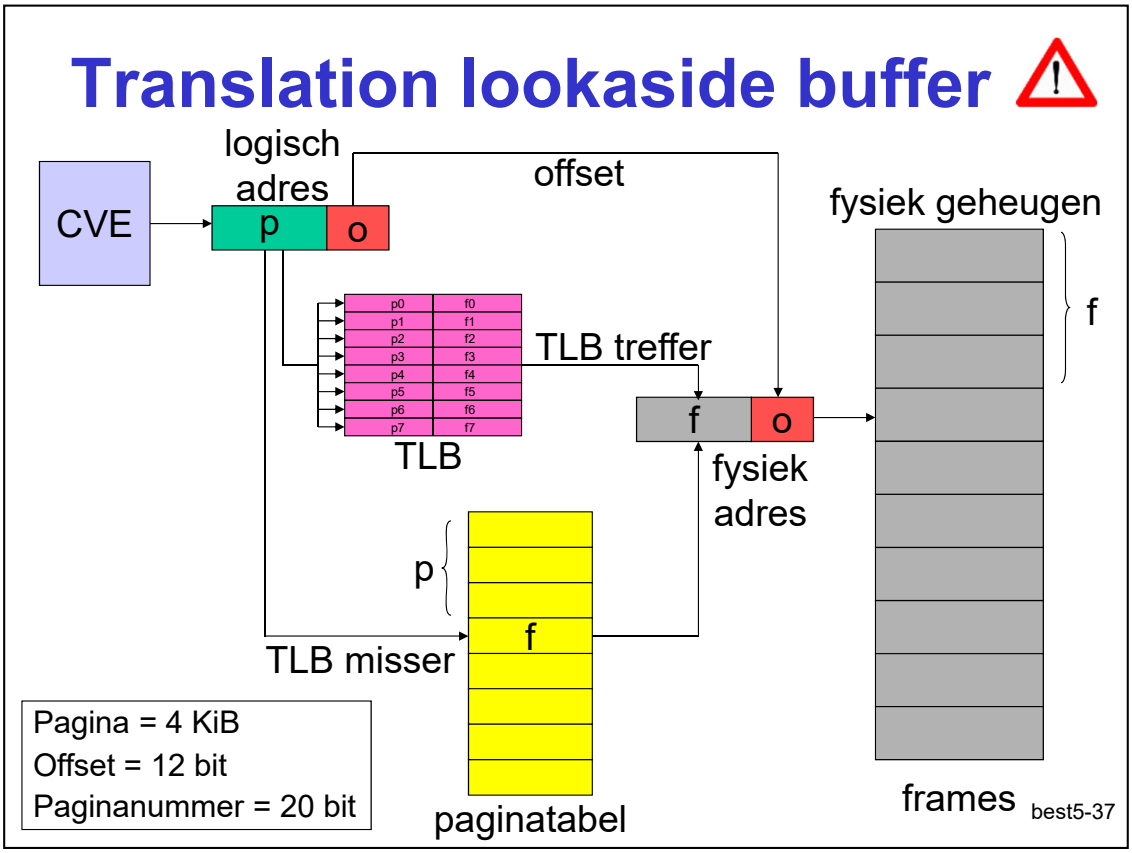
-Read-only bit: houdt bij of de pagina al dan niet mag veranderd worden

-Dirty bit (of modified bit): houdt bij of deze pagina beschreven werd

-Reference bit: houdt bij of de pagina gebruikt werd (om te lezen of te schrijven)

-Lock bit: geeft aan dat de pagina in het geheugen moet blijven (zie later)

-Cached bit: geeft aan of gegevens uit deze pagina al dan niet mogen opgenomen worden in de cache



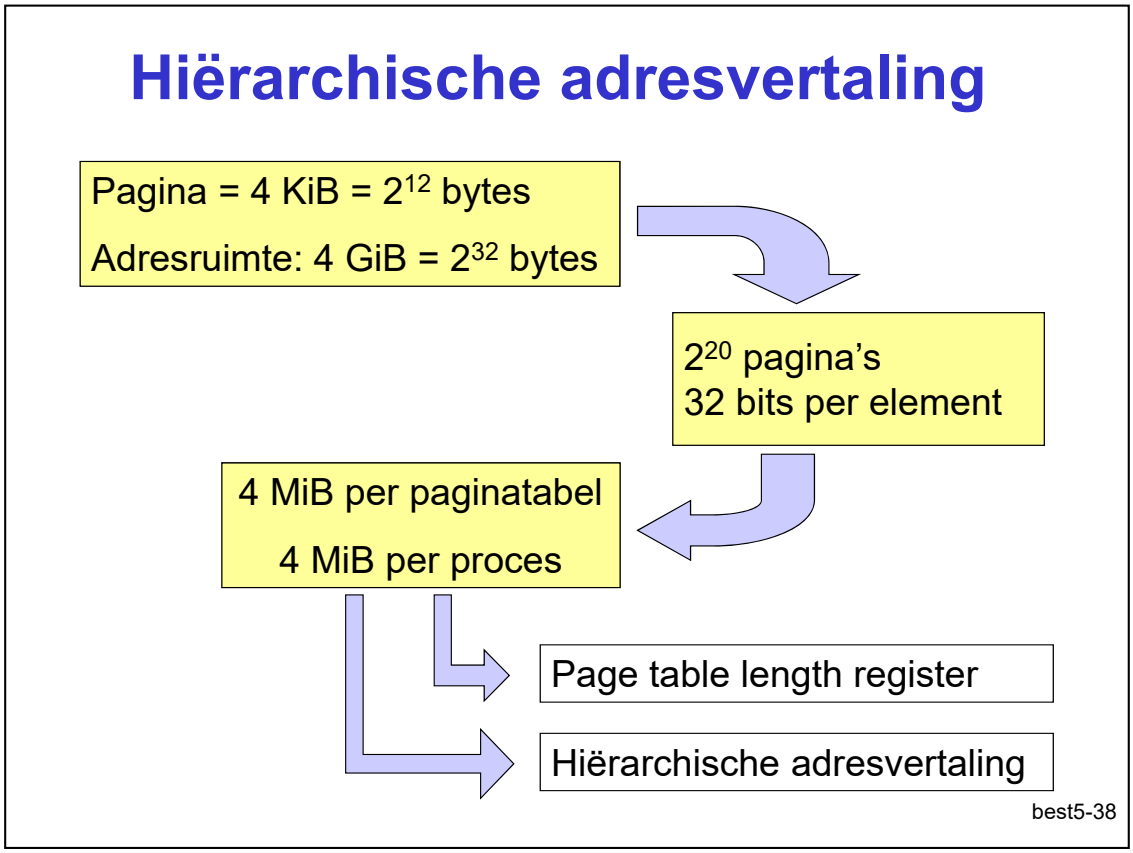
Een probleem met de basisadresvertaling is dat we per geheugentoegang een adresvertaling moeten uitvoeren en dat deze adresvertaling op haar beurt ook een geheugentoegang vereist. Concreet wil dit zeggen dat we voor het ophalen van een instructie nu 2 geheugentoegangen nodig hebben, en voor het ophalen en uitvoeren van een load/store instructie zelfs 4 geheugentoegangen – twee om de instructie op te halen en twee om de gegevens te lezen of te schrijven. Wetende dat het geheugen veel trager is dan de processor, veroorzaakt dit uiteraard een onaanvaardbare vertraging.

Een eenvoudige oplossing bestaat erin om de vertalingen te cachen. Door de meest recente gebruikte vertalingen bij te houden in een vertalingscache (translation lookaside buffer of TLB) kan een belangrijk deel van deze overhead weggewerkt worden. Deze cache hoeft niet groot te zijn (64-1024 elementen volstaan – dit laat toe om van 256 KiB tot 4 MiB aan fysiek geheugen te adresseren). Bij het vertalen van een adres wordt eerst de TLB geraadpleegd. Indien de vertaling in de TLB gevonden wordt, spreekt men van een TLB treffer en wordt het gevonden framenummer meteen gebruikt om de vertaling door te voeren. Indien de vertaling niet gevonden wordt, dan spreekt men van een TLB misser en wordt de paginatabel gebruikt om de vertaling door te voeren. Een TLB treffer zal wel 10 x sneller zijn dan een TLB-misser. De TLB bevat naast de paginanummer-framenummer paren ook alle extra bits die in een paginatabelelement voorkomen.

Men kan beslissen om in de TLB al dan niet de identificatie van het proces bij te houden. Indien men dat niet doet, moet de TLB gewist worden bij elke contextwisseling. Indien men dat wel doet, kan de inhoud van de TLB behouden blijven bij een contextwisseling.

Indien alle elementen van de TLB in gebruik zijn, en er treedt een TLB-misser op, dan moet er een bestaand element vervangen worden. LRU en random zijn vaak gebruikte vervangingsstrategieën. Indien de extra informatiebits tijdens hun aanwezigheid in de TLB veranderd werden, moeten deze teruggeschreven worden naar de paginatabel. Sommige elementen in de TLB kunnen ook verankerd

worden zodat ze nooit zullen vervangen worden. Dat kan nuttig zijn om de snelle adresvertaling van belangrijke blokken geheugen veilig te stellen.



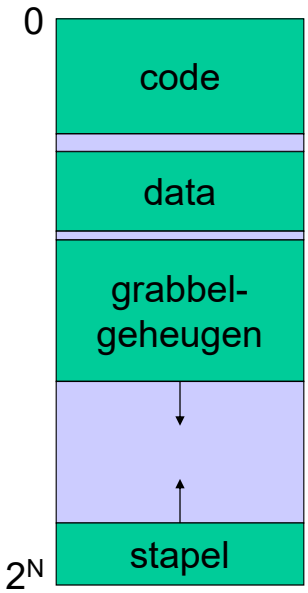
Voor een realistische adresruimte (4 GiB) en voor een courante paginagrootte (4 KiB) wordt de paginatabel 4 MiB groot. Deze tabel moet aaneengesloten in het geheugen opgeslagen worden want de paginatabel wordt gebruikt als een array van framenummers, en kan om die reden dus niet in stukken opgeslagen worden.

Verder is er het probleem van de omvang. Als men voor elk proces een dergelijke paginatabel moet bijhouden, zal de overhead aan extra benodigde geheugenruimte wel erg groot zijn (ook voor een proces van 1 MiB heeft men een tabel van 4 MiB nodig).

De oplossingen die voorgesteld worden zijn.

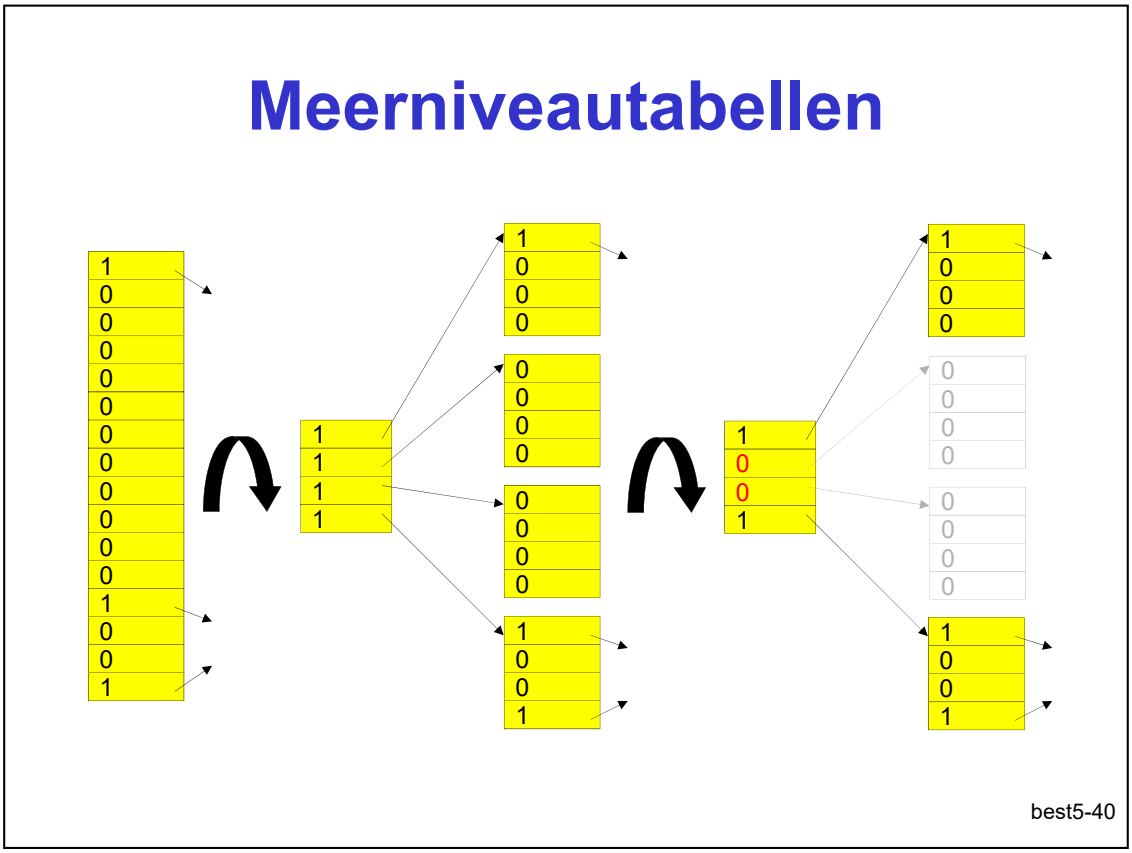
1. Het bijhouden van de nuttige lengte van de paginatabel. Een proces met een omvang van 1 MiB heeft slechts 256 frames van 4 KiB nodig. De paginatabel kan dan ook beperkt worden tot de 256 eerste elementen wat neer komt op een paginatabel van 1 KiB i.p.v. 4 MiB. Het nadeel van deze methode is wel dat het gebruikte deel van de adresruimte steeds in het begin moet liggen. Als een proces zou bestaan uit vier stukken van elk 256 KiB verdeeld over de volledige adresruimte van 4GiB, dan helpt de aanwezigheid van een PTLR nauwelijks
2. Het opsplitsen van de paginatabel in verschillende niveaus. Dit is de meest courant gebruikt techniek. Hij wordt verder besproken in de volgende slides.

Typische geheugenlayout



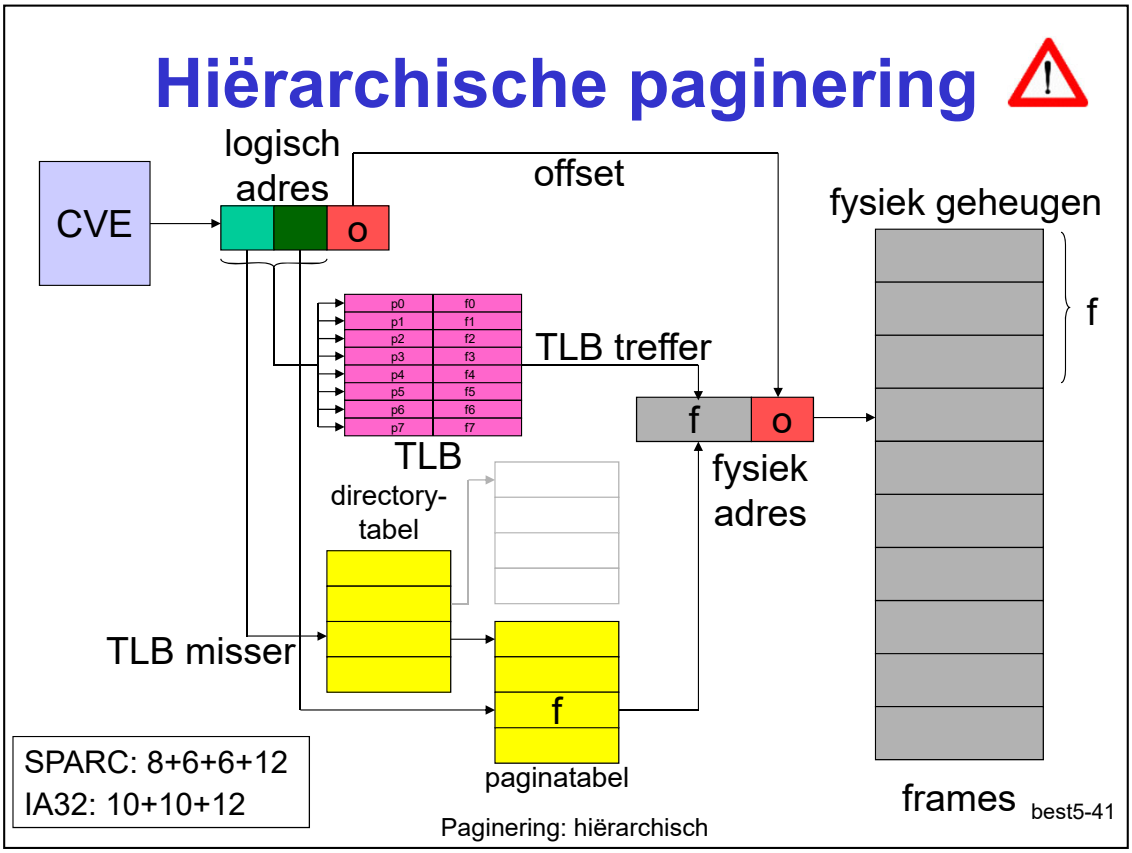
best5-39

Als een programma geladen wordt, zullen de voornaamste secties afzonderlijk in het geheugen opgeslagen worden: de code, de statisch gealloceerde data, het grabbelgeheugen (initieel leeg), en de stapel (initieel leeg). Elk van deze secties worden doorgaans op de een of andere manier gealigneerd (b.v. op 64 kiB) waardoor er tussen de verschillende secties wat externe fragmentatie kan ontstaan. Verder zorgt men er meestal voor dat het grabbelgeheugen (dat naar grotere adressen groeit), en de stapel (die naar kleinere adressen groeit) ver genoeg van elkaar verwijderd zijn zodat er voldoende groeiruimte overblijft. Aangezien de beide gebruikt worden om dynamisch gealloceerde gegevens in op te slaan, blijft de keuze tussen meer opslag in het grabbelgeheugen of meer op de stapel vrij.



Door meer niveaus van tabellen te gebruiken kan men ervoor zorgen dat grote stukken van een paginatabel met ongeldige framenummers niet meer hoeven gealloceerd te worden. In het bovenstaande voorbeeld staat een paginatabel met 16 elementen waarvan er maar 3 gebruikt worden. Desalniettemin moet de volledige tabel opgeslagen worden omdat zowel het eerst als het laatste element in gebruik is, en het gebruik van het PTLR de tabel niet kan inkorten.

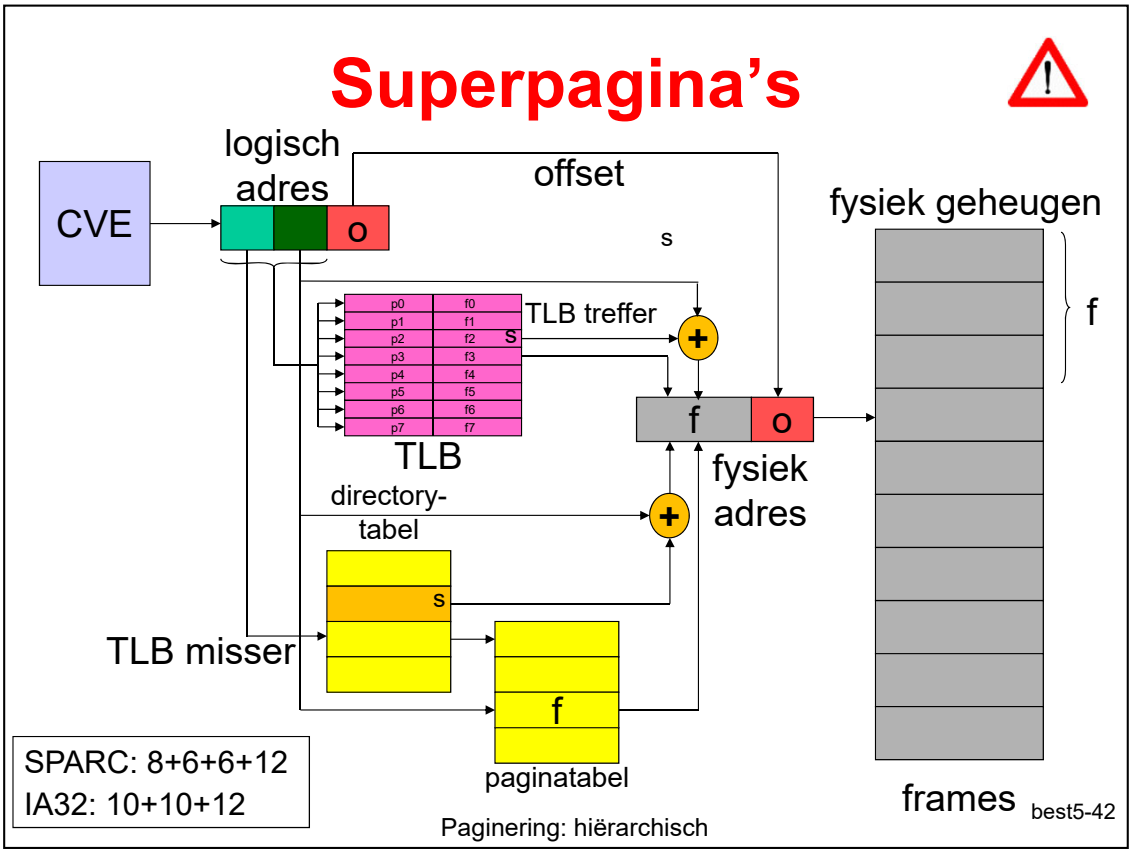
Door de tabel op te splitsen in 4 subtabellen van 4 elementen en deze 4 tabellen samen te voegen via een zgn. directorytabel (of buitenste paginatabel) kunnen we twee subtabellen achterwege laten doordat de betrokken subtabellen volledig uit ongeldige adressen bestaan. Dit kan aangegeven worden door de overeenkomstige elementen in de directorytabel op invalid te zetten. Een andere mogelijkheid is dat er 1 subtabel met alleen maar ongeldige framenummers is, en dat we alle ongeldige directoryelementen naar dezelfde subtabel laten wijzen.



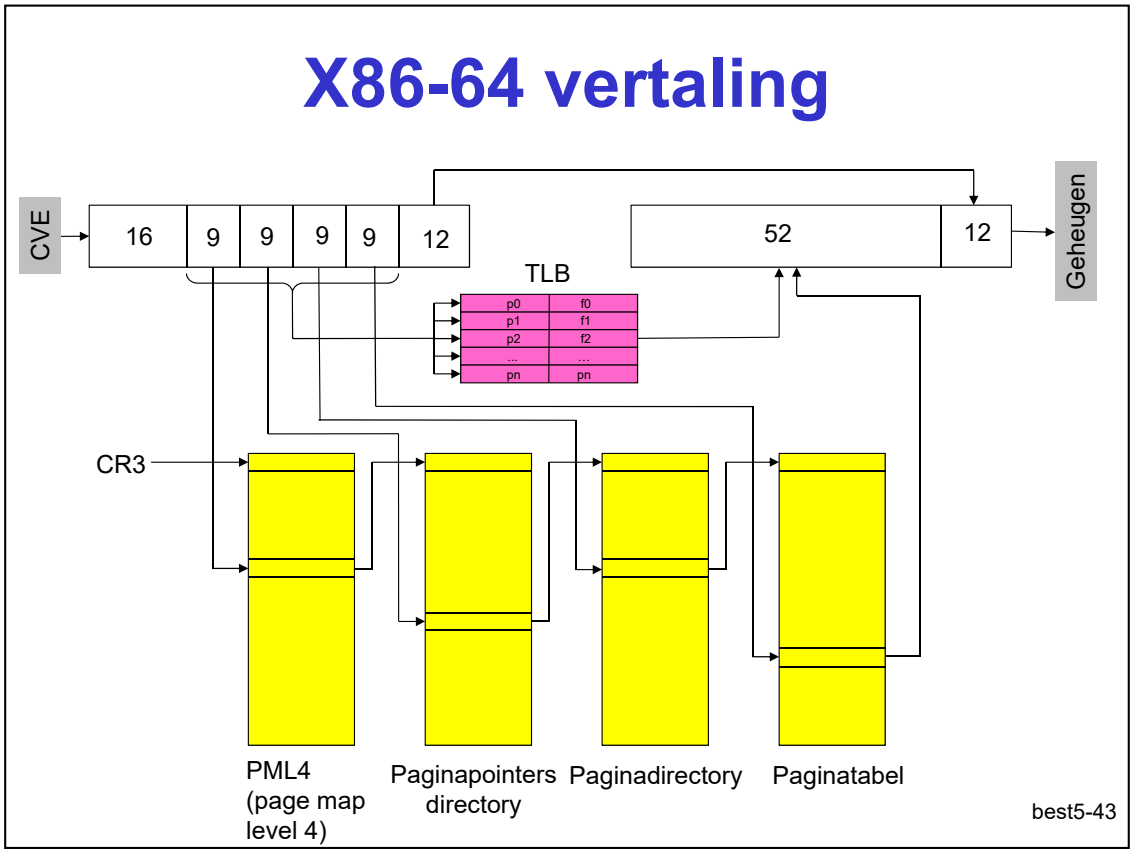
In het geval van tweeniveaupaginering wordt het paginanummer opgedeeld in twee delen. Het eerste deel zal de directorytabel indexeren. De directorytabel bevat op haar beurt een wijzer naar het begin van een paginatabel. Het tweede deel van het paginanummer kan dan gebruikt worden om de paginatabel te indexeren en het framenummer op te halen.

In een systeem waarbij de paginagrootte 4 KiB is, en het paginanummer 20 bit, is het het eenvoudigst om het paginanummer op te delen in twee gelijk delen van 10 bit. De eerste 10 bit moet dan gebruikt worden om de directorytabel te indexeren (die 4 KiB groot is). De tweede helft wordt dan gebruikt om de paginatabel te indexeren die eveneens 4 KiB groot is. Het feit dat zowel de directorytabel als de paginatabellen precies even groot zijn als een frame vereenvoudigt de allocatie ervan. Een paginatabel is goed voor de adressering van 4 MiB fysiek geheugen. Als een proces 4 afzonderlijke geheugengebieden alloceert (code, data, grabbelgeheugen, stapel) en deze zijn niet groter dan 4 MiB en gepast gealigneerd, dan is de overhead voor de paginatabellen beperkt tot 20 KiB wat aanzienlijk kleiner is dan 4 MiB.

In deze constellatie is de TLB uiteraard nog belangrijker dan is bij de basispaginering. Elke geheugentoegang wordt bij tweeniveaupaginering immers vervangen door 3 geheugentoegangen. De TLB moet uiteraard wel blijvend aangestuurd worden door het oorspronkelijke paginanummer.

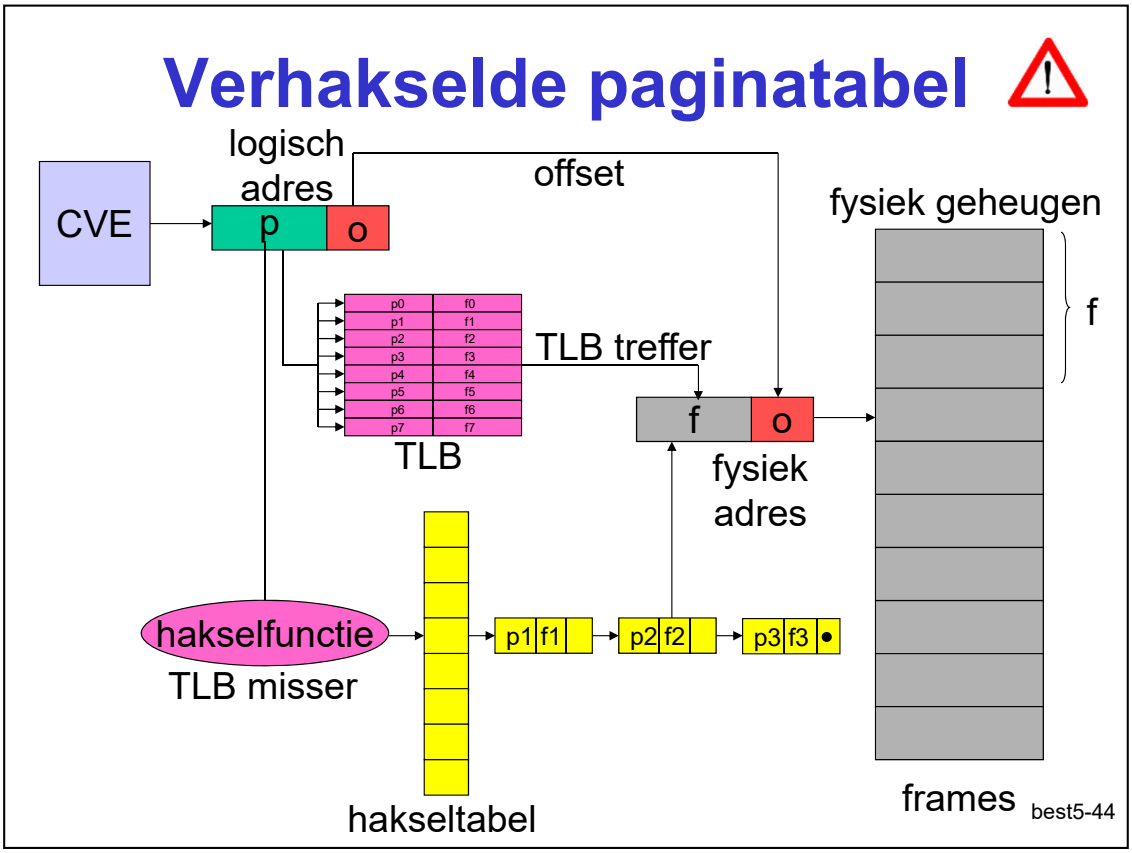


In het geval dat alle elementen van een paginatabel ingevuld zijn, en dat ze verwijzen naar een contigu blok fysiek geheugen (dat is in dit geval dan een blok van 4 MiB), kan men de vertaling eigenlijk rechtstreeks vanuit de directorytabel doen. Het volstaat om het framenummer van het beginframe van het contigu blok in de directorytabel op te nemen, en aan te geven dat het over een zgn. superpagina gaat. Door de index in de paginatabel op te tellen bij het framenummer uit de directorytabel krijgt men het correcte framenummer voor het fysiek adres. Deze optimalisatie zorgt ervoor dat er minder geheugentoegangen nodig zijn voor een vertaling (één ipv twee), maar misschien nog belangrijker verlicht het ook de druk op de TLB omdat er maar één element is voor een blok van 4 MiB (i.p.v. 1024).



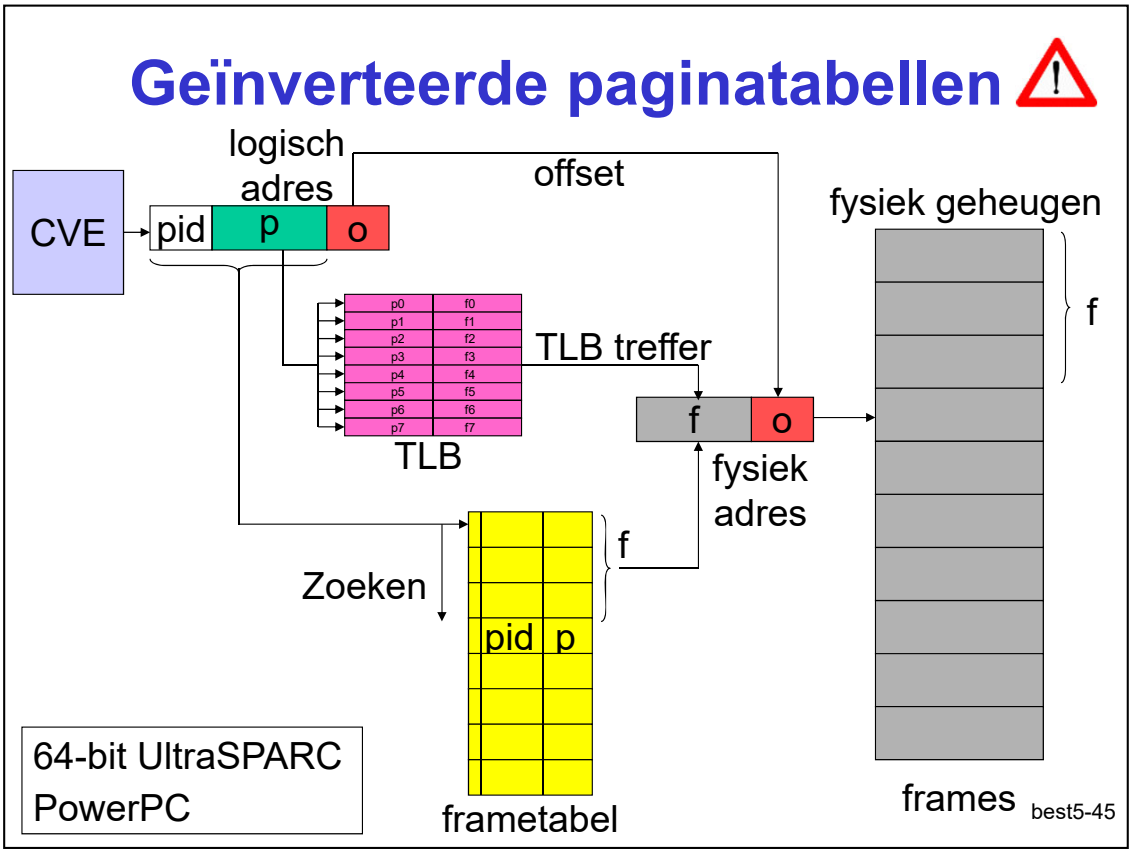
Adresvertaling voor een 64-bit processor is praktisch niet mogelijk in twee niveaus omdat de tabellen dan gewoon te groot worden. Bij de intelprocessor splitst men het adres op in vijf stukken waarvan er vier gebruikt worden bij de adresvertaling. De hoogste 16 bit zijn een tekenuitbreiding en worden niet gebruikt bij de adresvertaling. De resterende 48 bit zijn genoeg om 256 TiB te adresseren – dat volstaat voorlopig. De tabellen zijn nog steeds 4 KiB groot, maar de elementen van de tabel zijn nu 64 bit (beginadres van de tabel van het volgende niveau). Hierdoor kunnen er maar 512 elementen in een tabel opgenomen worden (9 bit index). Elke geheugentoegang zal nu vervangen worden door 5 geheugentoegangen (vier tabellen + geheugentoegang). Dit is uiteraard zeer traag. Daarom wordt er voorzien in een krachtige TLB als adresvertalingscache.

Naast de gewone pagina's zijn er ook superpagina's van 2 MiB en van 1 GiB.



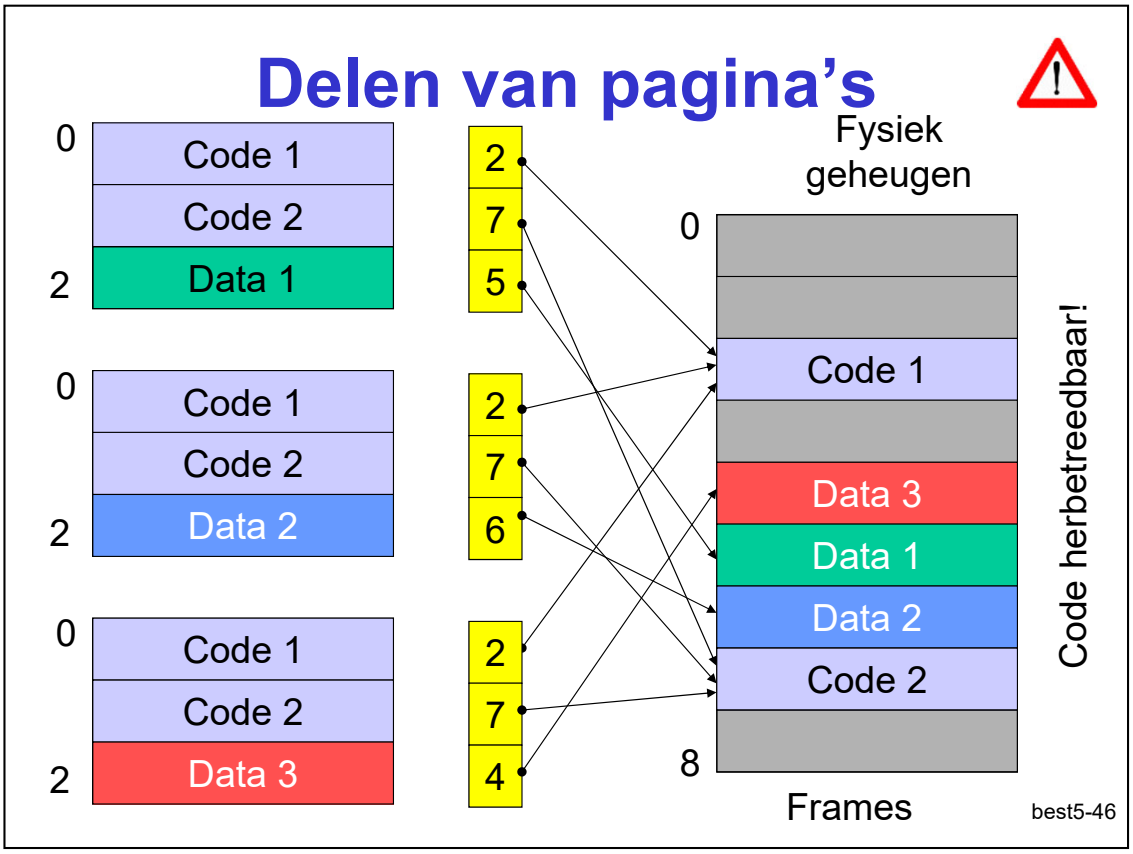
Voor adresruimten van groter dan 4 GiB is het gebruik van standaard paginatabellen niet echt handig omdat het aantal elementen in de tabel extreem groot wordt, of het aantal niveaus onhandig groot wordt. Een mogelijke oplossing is om de adresvertalingsinformatie op te slaan in een kleinere tabel die geïndexeerd wordt met een hakselfunctie (hash functie). Het grote paginanummer (b.v. 52 bits) wordt door de hakselfunctie eerst gereduceerd naar een kleiner aantal bits (b.v. 16 bits) die dan gebruikt worden om een tabel van b.v. 64 Ki elementen te indexeren. Indien verschillende paginanummers op hetzelfde element afgebeeld worden, worden er bijkomende elementen met die index geassocieerd.

Het berekenen van de hakselfunctie neemt niet veel tijd in beslag (doorgaans de xor van een aantal bits, kan in hardware zeer snel gebeuren). Het aflopen van de korte lijst hoeft ook niet lang te duren (in vergelijking met de tijd nodig om b.v. een vijfniveau hiërarchische adresvertaling te doorlopen). Verder is er ook nog de TLB om de belangrijkste eerder gevonden vertalingen bij het houden en snel terug ter beschikking te stellen.



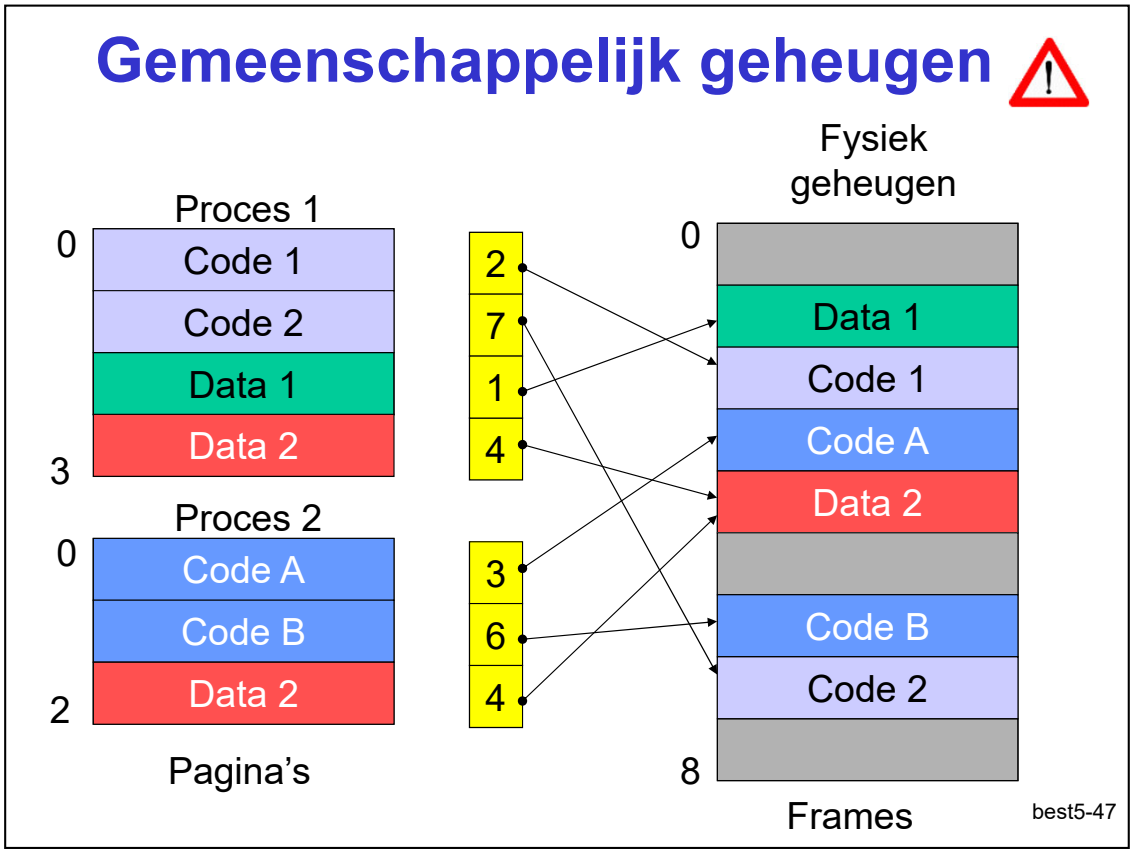
Een alternatieve mogelijkheid om de adresvertalingsinformatie in minder geheugen op te slaan is om gebruik te maken van de frametabel die toch al aanwezig is in het systeem. Aangezien de frametabel geïndexeerd wordt met het framenummer moet – gegeven het paginanummer – de frametabel afgelopen worden op zoek naar het voorkomen van een paginanummer (met bijhorende procesidentificatie). Deze manier van werken vereist veel minder geheugen (slechts 1 tabel voor het hele systeem, i.pv. 1 tabel per proces) maar het doorzoeken van de tabel neemt te veel tijd in beslag, zeker indien de hoeveelheid fysiek geheugen groot is (wat te verwachten is in een 64-bit machine).

Een oplossing voor het langen zoeken is gebruik te maken van een hakselfunctie om op die manier het doorzoeken van de volledige frametabel te beperken.



Het pagineringssysteem kan gebruikt worden om bepaalde frames in de adresruimte van meer dan één proces op te nemen. Veronderstellen we even dat er drie processen gecreëerd werden van hetzelfde programma (b.v. een tekstverwerker of een compiler). Indien de gegenereerde code herbetreedbaar is (re-entrant), dan kan de code – eenmaal deze aanwezig is in het fysiek geheugen – gebruikt worden door de andere processen. Het volstaat om de framenummers op te nemen in de respectievelijke paginatabellen.

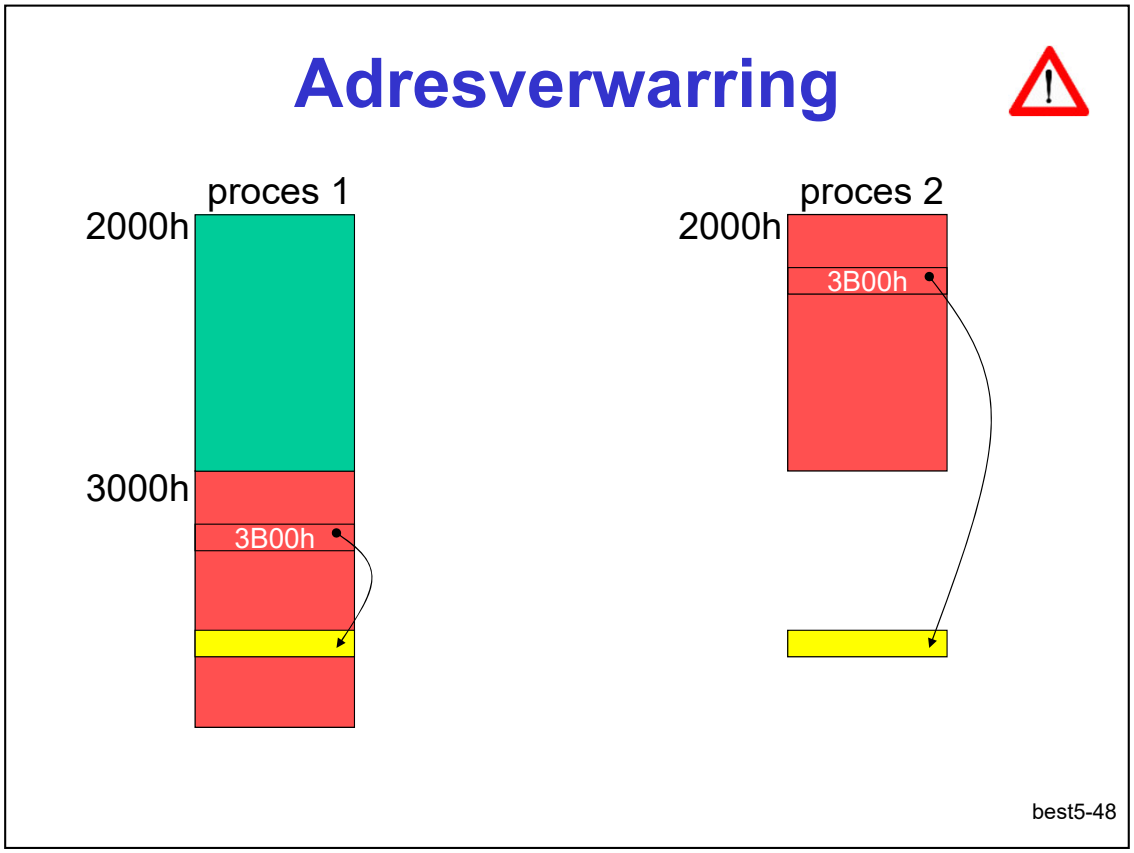
Doordat de processen aangemaakt werden vertrekkende van hetzelfde programma zullen de adressen van de code dezelfde zijn in alle processen. Ongeacht het proces zal een sprong naar een bepaald adres dus altijd op hetzelfde logisch adres uitkomen, en na vertaling dus ook op hetzelfde fysiek adres, en dus op dezelfde instructie. Ofschoon het geen absolute verplichting is dat frames die gedeeld worden door verschillende processen ook hetzelfde logisch adres hebben, maakt dit de uitwisseling van informatie via de gemeenschappelijke frames wel veel gemakkelijker.



Op precies dezelfde manier kunnen twee processen ook datapagina's delen. Deze pagina's kunnen dan gebruikt worden om gegevens uit te wisselen tussen de twee processen. Deze manier van communicatie is zeer gewoon tussen draden die per definitie de adresruimte van hun proces delen, maar dit is niet evident tussen processen. Om een stuk geheugen gemeenschappelijk te kunnen gebruiken zal dit moeten aangevraagd worden aan het besturingssysteem.

Bij de aanvraag moet men melden over hoeveel geheugen het gaat, en waar men dit geheugen wil afbeelden in de logische adresruimte. Indien de opgegeven adressen in de logische adresruimte reeds in gebruik zijn door het programma zal de kern automatisch een ander gebied voorstellen.

In Unix dient men gebruik te maken van de volgende systeemoproepen om gedeeld geheugen tussen processen te beheren: shmget, shmat, shmdt, shmctl.

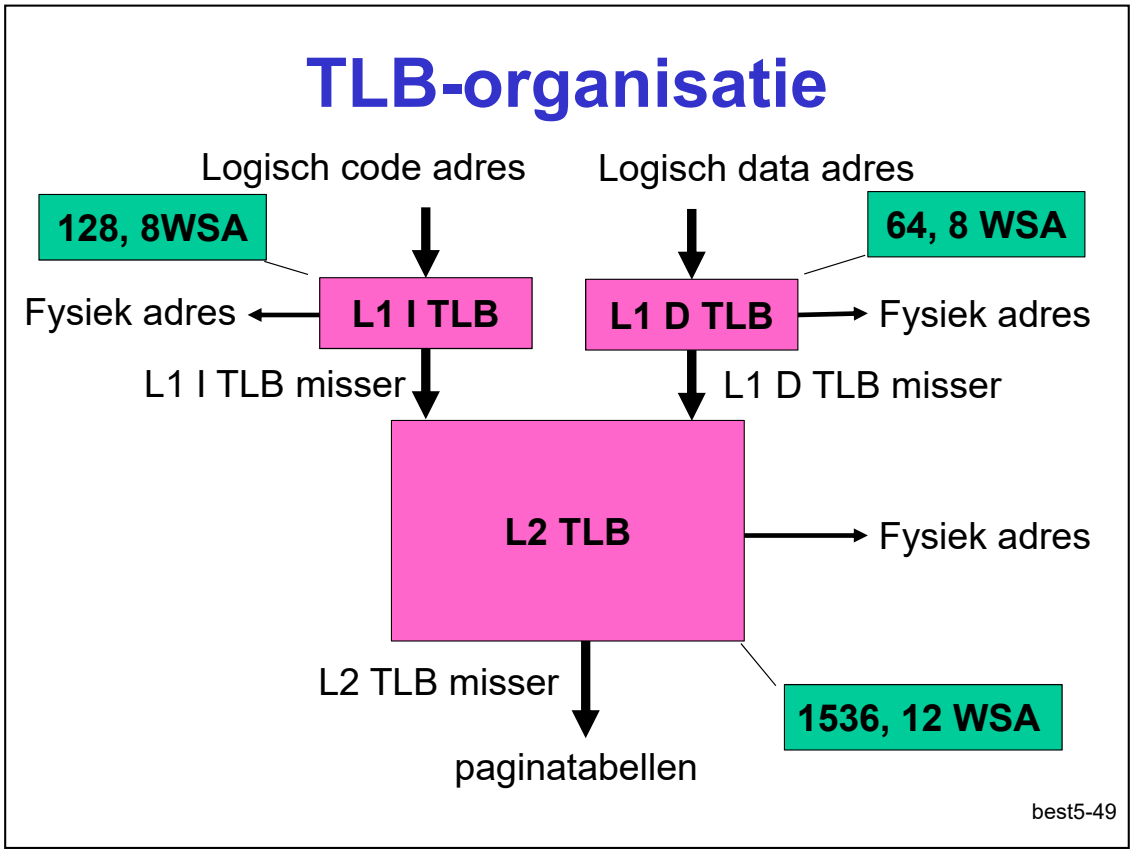


Indien het gemeenschappelijke fysiek geheugen niet in alle adresruimten op dezelfde logische adressen zit kan er zich adresverwarring gaan voordoen omdat de logische adressen van 1 proces dan een andere fysieke betekenis krijgen in een ander proces. Hierboven wordt een illustratie gegeven van adresverwarring.

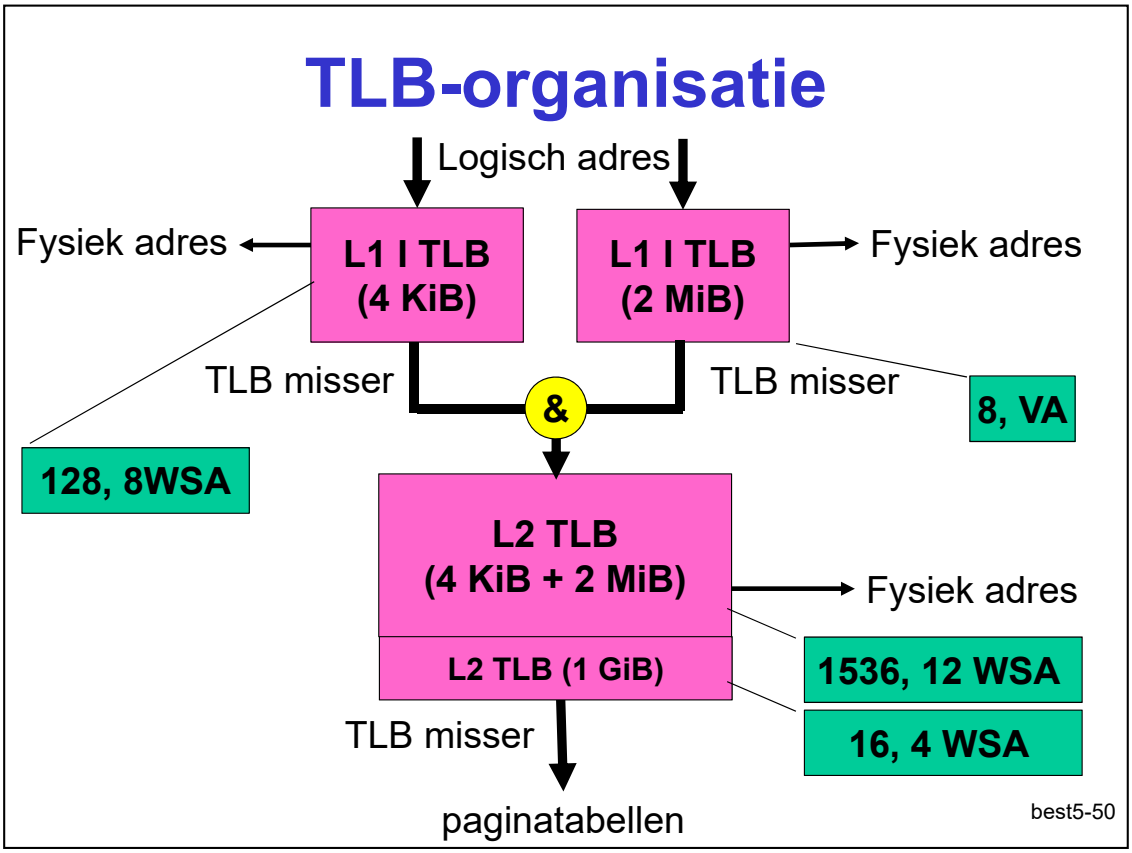
Stel dat er in proces 1 een gelinkte lijst opgebouwd wordt die nadien door proces 2 zal moeten doorlopen worden. Proces 1 slaat in een bepaald lijstelement een wijzer aan naar een ander lijstelement (3B00h). Beide lijstelementen liggen opgeslagen in het gedeelde geheugen.

Als proces2 nu die lijst wil gaan overlopen zal het op een bepaald moment adres 3B00h willen adresseren. Doordat de adresvertaling van 3B00h in het tweede proces anders verloopt zullen we uiteindelijk bij een ander fysiek adres uitkomen of zal de adresvertaling mislukken (zoals hierboven geïllustreerd, adres 3B00h valt buiten het adresseerbereik van proces 2). Dit kan vermeden worden door ervoor te zorgen dat het gedeelde geheugen dezelfde logische adressen krijgt in de beide processen. Indien dat niet mogelijk is moet er rekening gehouden worden met de verschillende adressen, en zal men bij de adressering de nodige aanpassingen aan de adressen moeten doorvoeren (voor proces 2 volstaat het b.v. om alle adressen met 1000h te verminderen om opnieuw correct te kunnen adresseren).

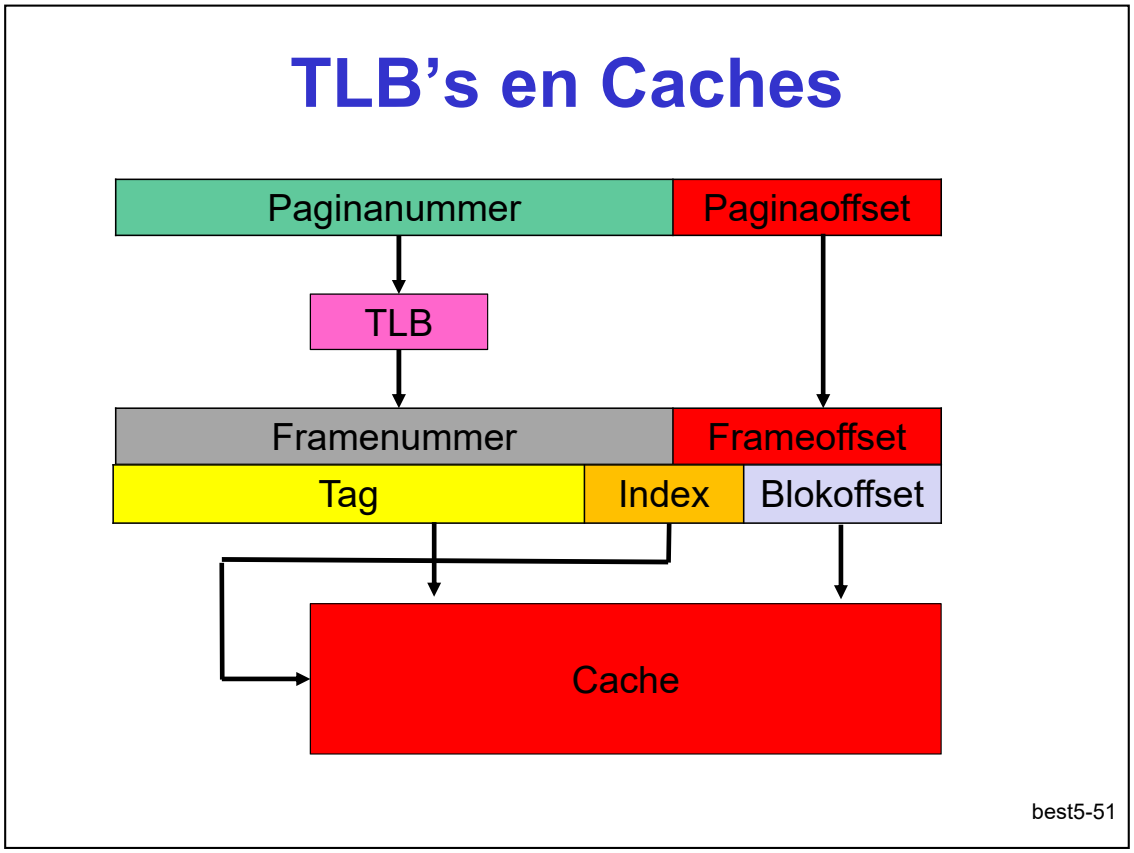
Een gevaar dat altijd om de hoek loert bij het gedeelde geheugen tussen processen is dat men een adres dat niet behoort tot het gedeelde geheugen toch in het gedeelde geheugen opslaat. Proces 2 zal in dit geval het gegeven niet kunnen vinden omdat de gegevens waarnaar de wijzer wijst niet toegankelijk zijn vanuit proces 2.



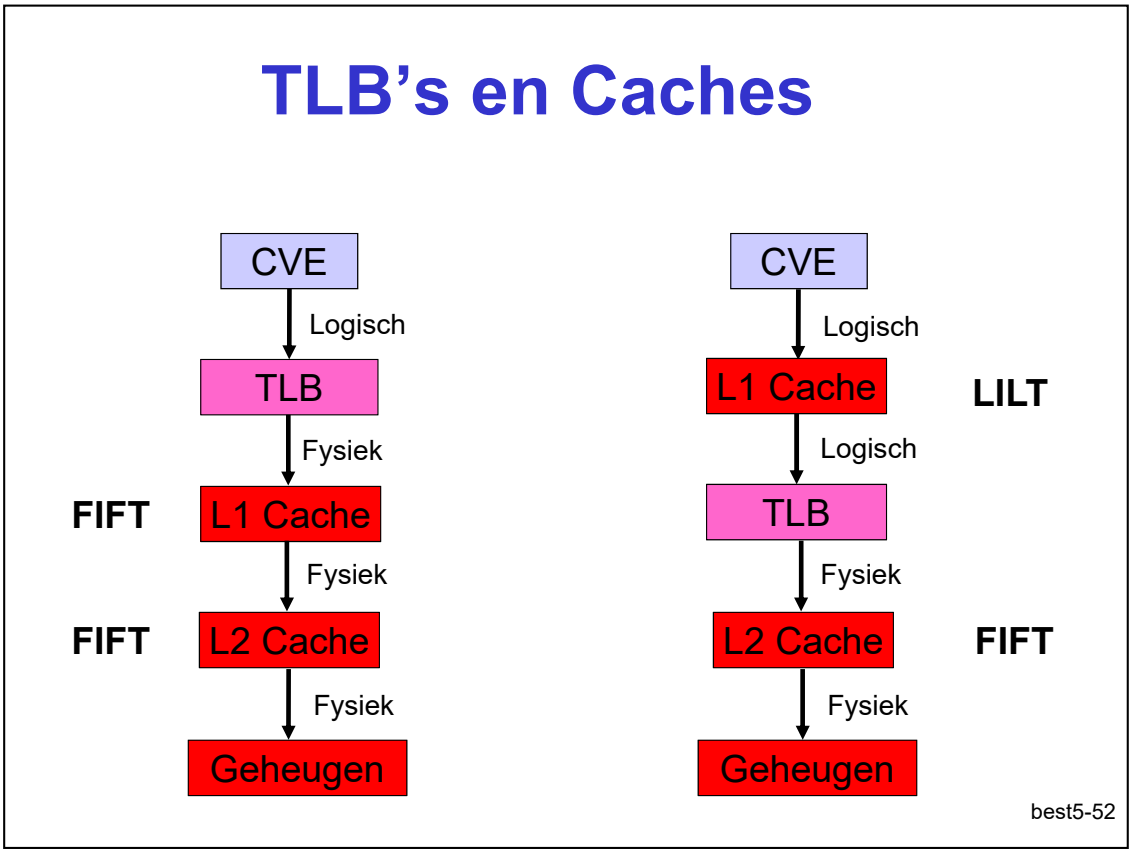
Aangezien TLB's snel moeten zijn, en liefst ook zo groot mogelijk om TLB-missers te voorkomen, worden TLB's soms hiërarchisch georganiseerd. In de figuur staat de organisatie van de TLB-architectuur per core in de 64-bit Kaby Lake Intel processor (gelanceerd in september 2016). Om de adresvertaling zo snel mogelijk te maken wordt er gewerkt met twee niveaus. Het eerste niveau is bovendien opgesplitst per cachetype en bevat 128 elementen voor de instructiecache (8-wegs set-associatief) en 64 elementen voor de datacache (8-wegs set-associatief). Het tweede niveau bevat 1536 elementen en is 12-wegs set-associatief. De L1 TLB's worden gescheiden voor de klassieke redenen: (i) ze zullen minder missers veroorzaken omdat instructies en data elkaar niet verdringen in de TLB (ii) ze kunnen fysiek dichter bij de caches gebouwd worden. Als een vertaling niet gevonden wordt in een L1 TLB, dan wordt er gezocht in de L2 TLB. Als er een vertaling in de L2 TLB gevonden wordt, dan wordt deze opgenomen in de L1 TLB, en wordt de geheugenoperatie opnieuw gestart (waarbij de vertaling nu gegarandeerd wel gevonden wordt in L1 TLB). Als de vertaling niet gevonden wordt in de L2 TLB, dan worden de paginatabellen geconsulteerd en indien de vertaling geldig is, wordt ze opgenomen in de L2 TLB, en vandaar ook in een L1 TLB, waarna de geheugenoperatie opnieuw uitgevoerd wordt.



Om op efficiënte manier om te kunnen gaan met superpagina's, zal men de L1 TLB's nog eens opsplitsen per paginagrootte.
Voor de LI1 I TLB, krijgen we een deel-TLB van 128 elementen (8 wegs set-associatief) voor pagina's van 4 KiB, gecombineerd met een LI1 I TLB van 8 elementen (volledige associatief) voor pagina's van 2 MiB (in de vorige figuur zijn enkel de TLB's voor pagina's van 4 KiB opgenomen). In de L2 TLB worden alle elementen gedeeld door de twee paginagroottes. Daarnaast is er nog een afzonderlijke TLB van 16 elementen (4 WSA) voor extra grote superpagina's van 1 GiB.

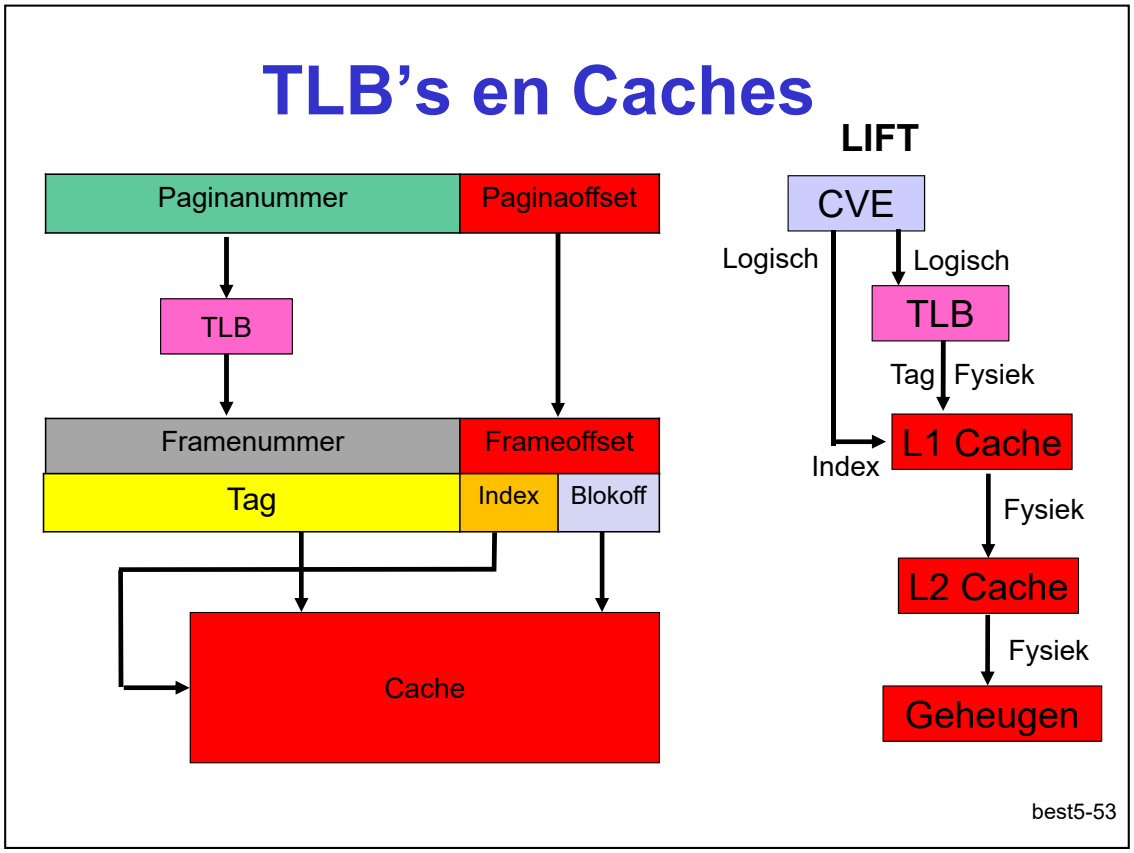


De interactie tussen TLB's caches verdient enige aandacht. De TLB zet een logisch adres om in een fysiek adres. Dat fysiek adres wordt dan gebruikt om een cache te bevragen. Daartoe wordt het fysiek adres opgedeeld in een tag, een index en een offset. De index bepaalt in welke set er gezocht moet worden, en via een vergelijking met de tag wordt nagegaan of het gevraagde blok zich in de cache bevindt. Via de blokoffset kan de gevraagde byte dan opgevraagd worden. Het feit dat de adresvertaling moet gebeuren voordat de cache kan bevraagd worden, zorgt uiteraard voor een extra vertraging (zelfs bij een TLB-treffer).



De situatie van de vorige dia staat links afgebeeld. Eerst wordt de adresvertaling uitgevoerd, en pas dan worden de caches bevraagd. Men geeft dit aan met FIFT (fysieke index + fysieke tag). Om sneller de L1 cache te kunnen bevragen zou men de cache ook gebruik kunnen laten maken van logische adressen. De L1 cache wordt dan omschreven als LILT (logische index + logische tag). Op die manier wordt het gebruik van de L1 cache bij een TLB-treffer niet vertraagd door de adresvertaling, en wordt de kost van de vertaling enkel maar betaald in het geval met een beroep moet doen op de L2 cache.

De oplossing aan de rechterzijde lijkt dus beter. Er kleeft echter een nadeel aan vast. Logische adressen zijn enkel gelding binnen één proces. Dat wil zeggen dat de informatie in de cache na een contextwisseling onbruikbaar is en dat een contextwisseling dus vereist dat de L1 caches gewist worden. Bij een korte onderbreking (bv het even laten uitvoeren van een proces dat in een actieve synchronisatielus zit – en meteen terug een yield() uitvoert) zal de cache opnieuw moeten opgewarmd worden. Dat is niet efficiënt.



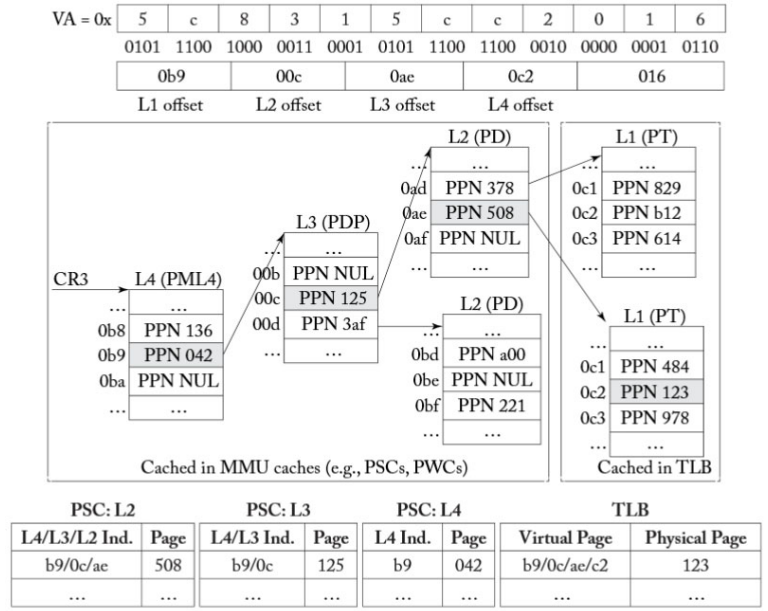
Mochten we erin slagen om de adresvertaling parallel met de cacheoperatie uit te voeren, dan zou de cache misschien toch gebaseerd kunnen blijven op fysieke adressen. Dit blijkt mogelijk te zijn als we de cache voldoende klein maken zodat het aantal bits voor de index + blokoffset passen in de offset van een pagina. In dat geval zullen die indexbits niet vertaald moeten worden door de TLB (omdat ze onderdeel zijn van de offset binnen een pagina), en worden de tag-bits gelijk aan het framenummer.

Dit wil zeggen dat we voor de L1-cache meteen kunnen beginnen met het selecteren van de juiste set, en dat we de tag-bits maar nodig hebben tegen het ogenblik dat de verschillende tags binnen een set moeten vergeleken worden. Die extra tijd kan gebruikt worden om de adresvertaling uit te voeren. Men noemt dit type van cache LIFT (logische index + fysieke tag) om voor de hand liggende redenen, maar aangezien de logische en de fysieke index bij dit soort van cache identiek zijn, kan je dit type van cache eigenlijk evengoed als FIFT omschrijven.

Een mogelijke organisatie voor een dergelijke cache is 32 KiB met blokken van 64 bytes en 8 WSA. Een dergelijke cache heeft sets van 8 blokken van 64 bytes of 512 bytes. Zo gaan er 64 in een cache van 32 KiB. De offset kan gecodeerd worden in 6 bit, en index voor de selectie van één van de 64 sets is ook 6 bit. Samen is dit 12 bit of het aantal bits van de offset van een pagina van 4 KiB. Binnen deze ontwerpruimte is het aantal mogelijke cacheorganisaties uiteraard beperkt.

TLB-organisatie

Paging structure caches



PSC: L2

L4/L3/L2 Ind.	Page
b9/0c/ae	508
…	…

PSC: L3

L4/L3 Ind.	Page
b9/0c	125
…	…

PSC: L4

L4 Ind.	Page
b9	042
…	…

TLB

Virtual Page	Physical Page
b9/0c/ae/c2	123
…	…

Figure 4.7: x86-64 page table walk for virtual address 0x5c8315cc2016. TLBs cache L1 PTEs and MMU caches store L2-L4 PTEs. Conventional caches can store all entries [18].

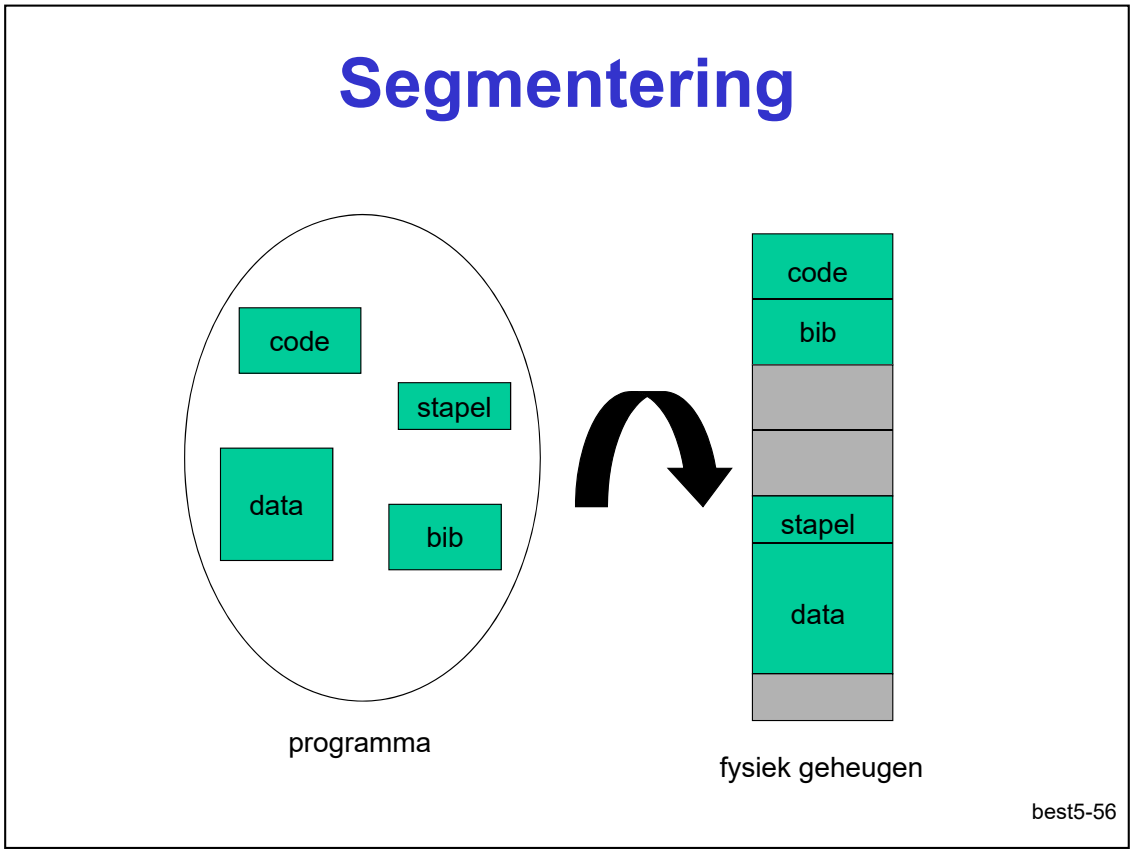
best5-54

Al het voorgaande betrof de situatie waarbij de adresvertaling kan teruggevonden worden in één van de vele TLB's die men op een moderne architectuur terugvindt. Als dat niet het geval is moeten de paginatabellen afgelopen worden, en bij een 64-bit architectuur vereist dit vier geheugentoegangen. Zelfs als die in de datacache kunnen gevonden worden, neemt dit veel meer tijd in beslag dan wanneer er een TLB-treffer is. Om het aflopen van de tabellen te versnellen kan men in *paging structure caches* voorzien. Deze worden soms ook MMU-caches genoemd. Het idee is eenvoudig: als op basis van het paginanummer (dat in de praktijk uit 4x9=36 bit bestaat), het framenummer niet kan gevonden worden, is het misschien wel mogelijk om via de MMU-cache het adres van de L1-paginatabel te vinden. De tag die men daarvoor nodig heeft zijn de 3x9 meest beduidende bits van het paginanummer. Met de minst beduidende 9 bit kan men dan de paginatabel indexeren. Als dat niet lukt, kan het adres van de directorytabel (L2) misschien gevonden worden op basis van de 2x9 meest beduidende bits. De resterende bits kunnen dan gebruikt worden om de juiste paginatabel en het framenummer te vinden. Als dat niet lukt kan op basis van de meest beduidende 9 bits de L3 tabel gevonden worden. Op die manier kan de adresvertaling ook in het geval van een TLB-misser versneld worden: men gaat op zoek naar de treffer in de MMU-cache die correspondeert met het langste prefix van een paginanummer, en men loopt de paginatabellen van dat punt verder af.

Overzicht

- Logische vs. fysieke adressen
- Contigue allocatie van geheugen
- Niet-contigue allocatie van geheugen
  - Paginering
  - Segmentering
  - Segmentering + paginering

best5-55



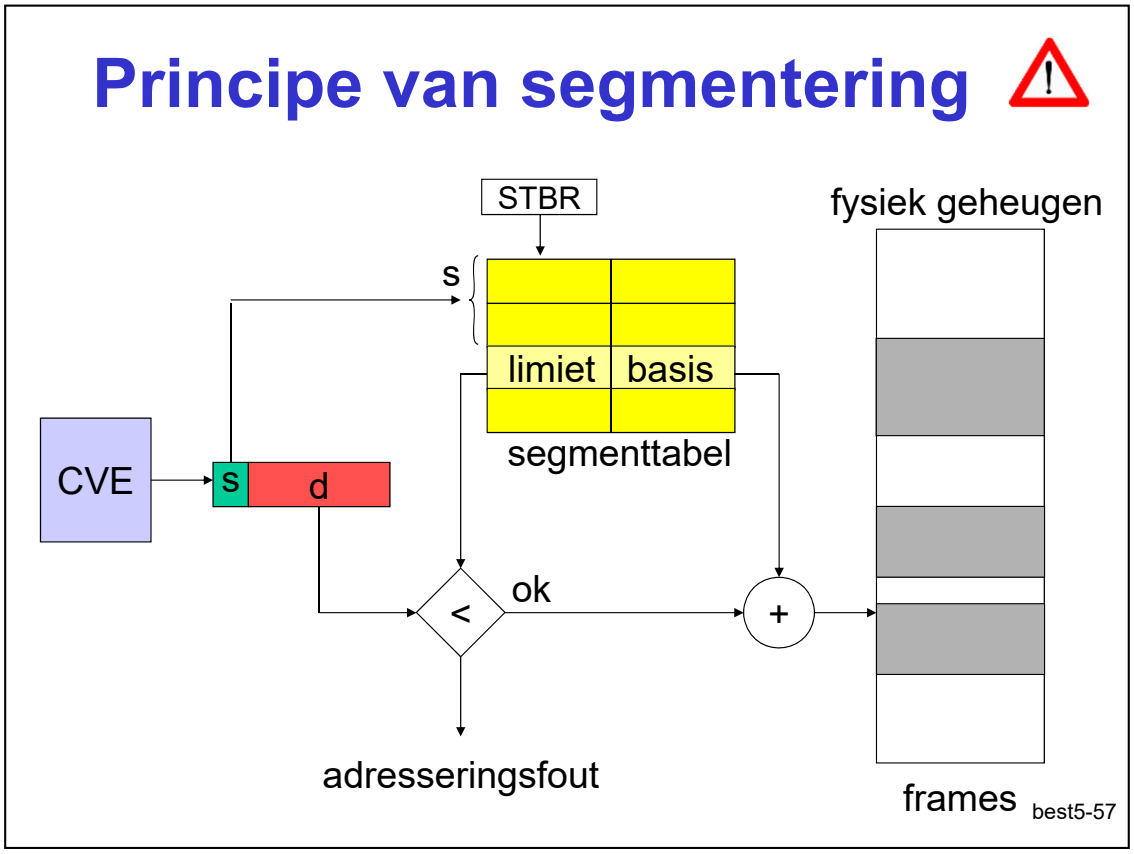
Paginering is een schitterende techniek om externe fragmentatie te vermijden en maakt gebruik van dynamische relocatie. De techniek garandeert een waterdichte afscherming van de verschillende processen en laat toe dat processen blokken geheugen met elkaar delen. In deze opzichten voldoet hij aan alle criteria voor een goed geheugenbeheer.

Hij heeft echter als nadeel dat hij niet zo gemakkelijk toelaat om de verschillende onderdelen van het programma tegen elkaar te beschermen. Programma's kunnen per ongeluk in hun eigen code schrijven, read-only gegevens kunnen vrij overschreven worden. Het is wel zo dat we per pagina een aantal beveiligingsbits ter beschikking hebben, maar deze zijn niet altijd bruikbaar. Wat doen we b.v. met een pagina die voor de helft uit instructies en voor de helft uit data bestaat?

Verder is het gebruik van een lineaire adresruimte ook niet zo handig in de aanwezigheid van datastructuren die moeten kunnen groeien zoals de stapel of het grabbelgeheugen.

Een betere techniek voor de bescherming van de verschillende secties waaruit een programma bestaat en die ook toelaat dat ze groeien, is segmentering. Zo kunnen we een segment definiëren voor de code, voor de data, de stapel, een gedeelde bibliotheek, en zelfs voor een individuele datastructuur. Het volstaat om per segment de beschermingsbits goed te definiëren en het hele segment is beschermd.

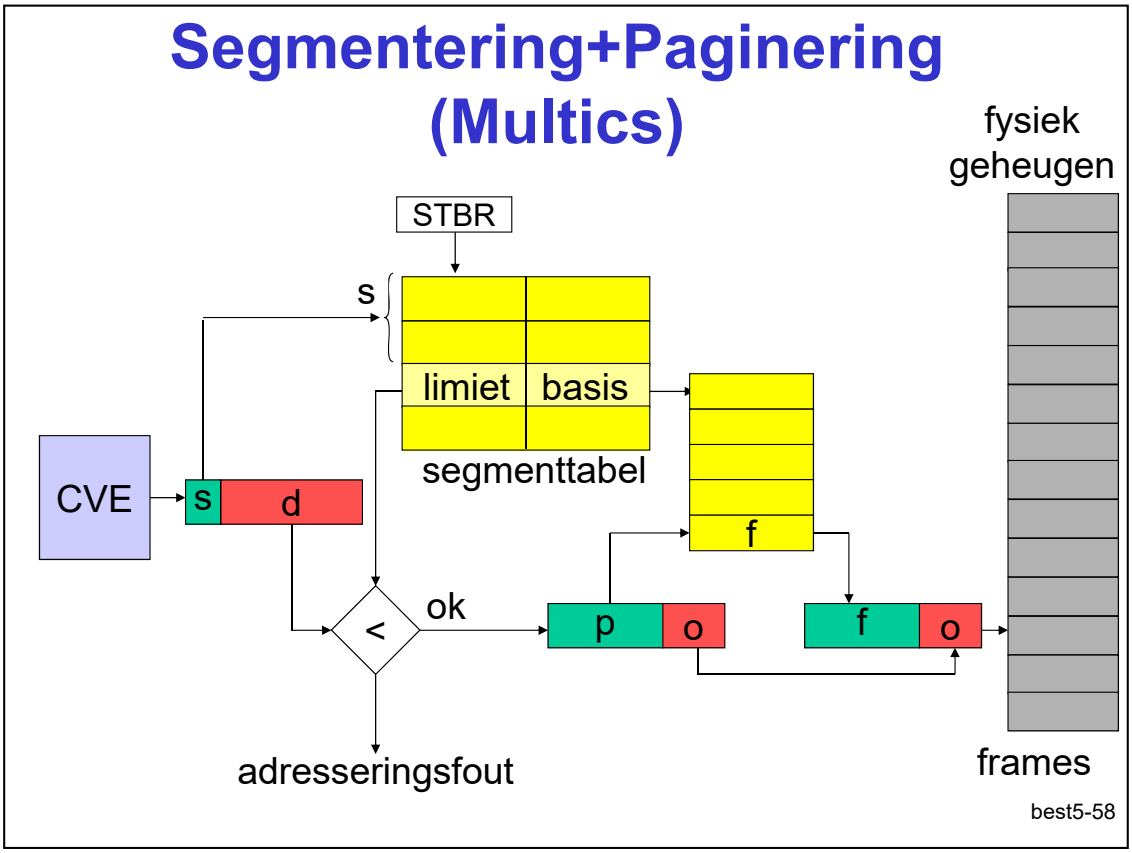
In een gesegmenteerde adresruimte zal een adres nu bestaan uit de identificatie van het segment en een offset binnen dat segment. Voor het gemak gaan we er hiervan uit dat we kunnen werken met een segmentnummer. In de IA32 architectuur heten de segmenten: CS, DS, SS, ES, FS, GS.



Het principe van de segmentering is vrij eenvoudig. Per segment is er een element in de segmenttabel die het beginadres en de lengte van dat segment in het fysiek geheugen bevat. Bij de adresvertaling wordt eerst gecontroleerd of de offset wel degelijk binnen het segment valt, en vervolgens wordt de offset opgeteld bij het basisadres van het segment. Op die manier krijgen we het fysiek adres. De segmenttabel wordt aangewezen door het segment table base register (STBR). Indien nodig kan er ook een segment table length register (STLR) ingevoerd worden.

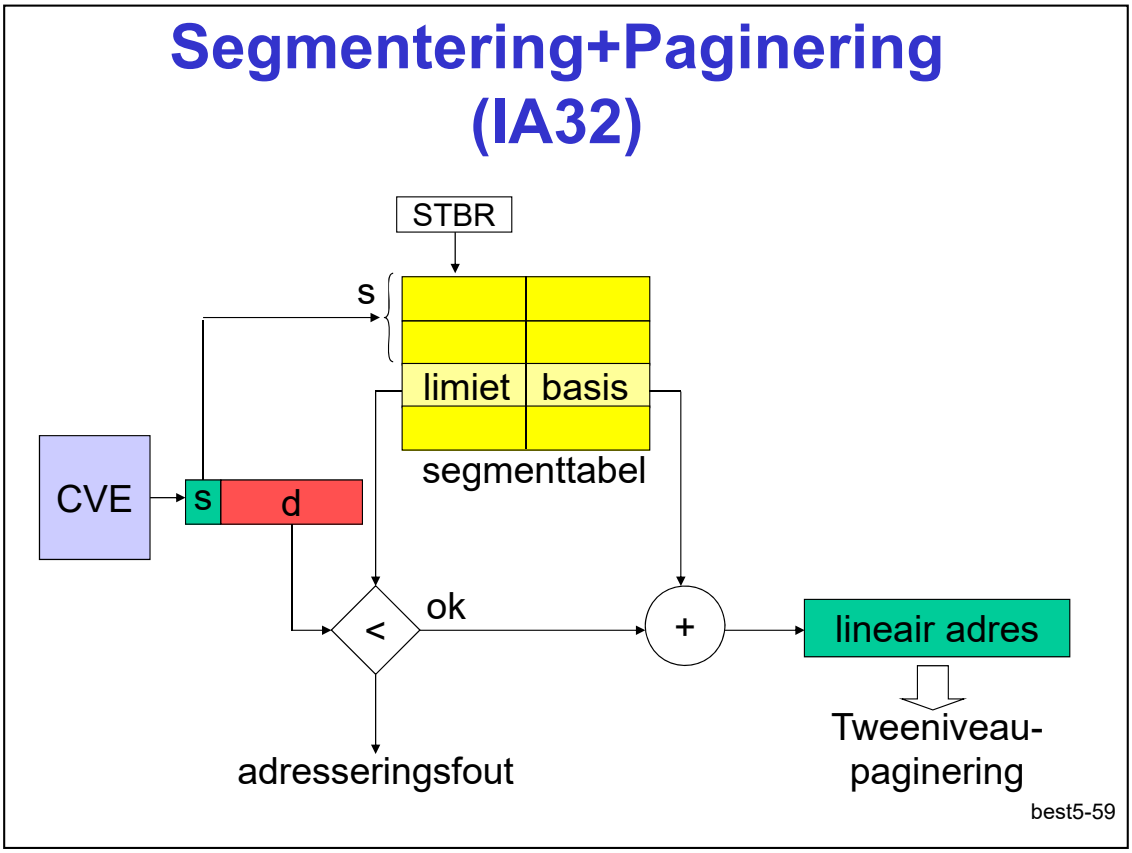
Segmentering laat toe om processen van elkaar af te schermen, het volstaat dat de verschillende segmenten in het fysiek geheugen disjunct zijn. Segmentering laat ook toe om geheugen te delen door een bepaald segment uit het fysiek geheugen op te nemen in de segmenttabel van twee verschillende processen (let wel: ook hier kan adresverwarring ontstaan indien de segmentnummers niet dezelfde zijn; dit kan in de code opgelost worden door enkel maar relatieve adressen binnen het huidige codesegment te gebruiken voor controletransfers – dus zonder expliciet het segmentnummer op te geven). Verder is relocatie mogelijk door een segment te kopiëren naar een andere plaats in het geheugen, en het basisadres uit het bijhorende element in de segmentabel op gepaste wijze te veranderen.

Het probleem van de allocatie van segmenten ligt wat moeilijker. Aangezien segmenten geen vaste lengte hebben moet er teruggevallen worden op de algoritmen voor het dynamisch alloceren van partities die eerder deze les besproken werden. Een andere denkpiste is om een segment op zijn beurt ook nog eens te gaan pagineren en op die manier het allocatieprobleem op te lossen.



Een mogelijke manier om een segment te pagineren is om het basisadres niet langer naar het begin van het fysiek segment te doen wijzen, maar naar een paginatabel. Het invoeren van een hiërarchische paginatabel heeft hier niet veel zin omdat een segment per definitie een contigu stuk geheugen is, en er in de segmenttabel een maximale grootte van de offset voorkomt. We weten m.a.w. hoeveel elementen er in de paginatabel moeten voorkomen. En daarbuiten kan er in principe niet geadresseerd worden. Dit systeem werd jaren geleden in Multics gebruikt.

De segmentabel in Multics bevat elementen die 36 bits lang zijn en die naast het beginadres van het segment en de lengte ook nog informatie bijhouden over of het segment al dan niet gepagineerd wordt, en hoe groot de pagina's zijn. Standaard zijn de pagina's 1024 bytes lang en de paginatabel per segment kan tot 64 elementen lang worden.



De IA32 architectuur gebruikt een ander systeem. Daar wordt er eerst een zgn. lineair adres berekend, uitgaande van de informatie opgeslagen in de segmenttabel en de offset. Het lineaire adres wordt dan nadien verder gepagineerd aan de hand van een tweeniveaupagineringssysteem (10+10+12) zoals voordien besproken (de IA32 kent ook een 1-niveaupaginering met enkel de directorytabel + pagina's van 4 MiB).

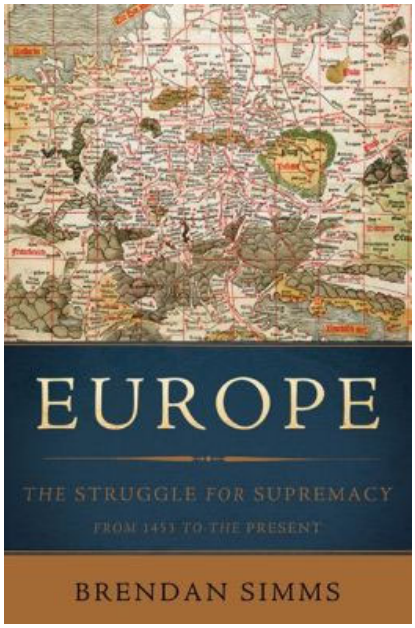
Er kunnen 16Ki segmenten zijn die elke tot 4GiB groot kunnen worden. Er zijn 6 segmentregisters die 6 elementen van de segmenttabel kunnen cachen voor snellere toegang.

Ondanks het feit dat de IA32 segmentering aanbiedt, is er behalve OS/2 geen besturingssysteem dat er ook gebruik van maakt. Zowel Windows NT als Linux zorgen ervoor dat alle segmenten in de processor wijzen naar de standaard 4GiB adresruimte (segmenten beginnen op adres 0 en zijn 4 GiB lang). Hun programma's werken a.h.w. rechtstreeks met lineaire adressen. Deze worden naderhand wel nog gepagineerd.

Overzicht

- Logische vs. fysieke adressen
- Contigue allocatie van geheugen
- Niet-contigue allocatie van geheugen
  - Paginering
  - Segmentering
  - Segmentering + paginering

best5-60


EUROPE
THE STRUGGLE FOR SUPREMACY
FROM 1453 TO THE PRESENT
BRENDAN SIMMS


GEERT MAK
IN EUROPA
REIZEN DOOR DE TWINTIGSTE EEUW
ATLAS

best5-61