Besturingssystemen I - Architectuur

Samenvatting cursus 2008-2009, door Tim Besard.

Overzicht van computersystemen en besturingssystemen

Basisbegrippen uit de computerarchitectuur

Hoofdelementen

Computers bestaan uit 4 hoofdelementen:

1. Processor: verwerking.
2. Hoofdgeheugen: enige rechtstreeks adresseerbaar geheugen, toegang gesynchroniseerd via geheugencontroller.
3. Apparaatcontrollers: verplaatsen van gegevens tussen computer en externe omgeving.
4. Systeemverbindingen/bussen: communicatie tussen alle elementen.

Instructiecyclus

Dit is de verwerking vereist voor het uitvoeren van een programmainstructie.

1. Opvraagcyclus: instructie uit hoofdgeheugen naar instructieregister.
2. Uitvoeringscyclus: verwerken van de instructie. Kan zorgen voor extra opvraagcycli.

Meeste processors: meerdere instructies per klokpuls verwerken (omdat interne registers zo snel zijn).

Processormodus

De processormodus bepaalt welke instructies een applicatie mag uitvoeren. Minimaal twee niveaus (kernelmodus en gebruikersmodus), om potentieel schadelijke instructies (vb. I/O) af te schermen. Modus bits in het programma'statuswoord bepalen de huidige modus.
Zo wordt ook de toegang tot bepaalde registers (die interne toestand van de processor beschrijven, zoals de programmateller, instructieregister, timers, programma'statuswoord, ...) beperkt tot de kernelmodus.

Geprogrammeerde I/O

Meest eenvoudige vorm van I/O, waarbij de processor via een wachtlus zal wachten tot de I/O operatie afgehandeld is (= synchrone behandeling). Tijdens elke wachtcyclus van 3 instructies leest de processor bepaalde geheugenadressen van de I/O controller in (die fungeren als signaleringsbits), bepaalt het resultaat van van dat register, en voert een voorwaardelijke spronginstructie uit. Vergelijkbare gereserveerde adressen worden gebruikt om de controller details over de gewenste I/O operatie door te geven.
Dergelijke speciale delen van het geheugen noemt men I/O-poorten, in tegenstelling tot de buffer (het gewone data-geheugen).

Interrupts

Methodiek om de processor iets asynchroon te melden, vb. wanneer een I/O operatie afgehandeld is. Hardware kan een interrupt genereren door een signaal op de IRQ-lijn te zetten, dewelke periodiek (extra stap in de instructiecyclus) door de processor gecontroleerd wordt. Is er een signaal aanwezig, start de processor een routine voor interruptafhandeling in kernelmodus, mits eerst de processorregisters op de stack geplaatst te hebben. Achteraf worden die teruggezet, samen met de programmateller zodat het onderbroken programma opnieuw hervat wordt. Als er een interrupt gegenereerd wordt tijdens de interruptafhandelingsroutine, wordt die ofwel tijdelijk genegeerd (interrupts uitgeschakeld), of selectief doorgestuurd (via een hardwarematige prioriteitenanalyse).

De hardware kan zichzelf identificeren bij een interrupt (hetzij via verschillende interruptlijnen per type hardware, hetzij via softwarematige oplossingen), en het besturingssysteem roept zo een gespecialiseerde interruptafhandelingsroutine aan (soms reeds hardwarematig bepaald via een interruptvector).

Sommige I/O-controllers ondersteunen ook DMA, waarbij de controller zelfstandig hele geheugenblokken naar het hoofdgeheugen overzet en slechts een enkele interrupt genereert.

De meeste besturingssystemen gebruiken interrupts voor meer dan enkel I/O-operaties (vb. foutmeldingen, timers, ...), moderne besturingssystemen zijn zelfs grotendeels interruptgestuurd (de kernel zelf wordt uigevoerd tijdens interruptafhandeling).

I/O-operaties kunnen dankzij interrupts asynchroon uitgevoerd worden: de processor kan andere programma's uitvoeren en die onderbreken als de I/O operatie voltooid is. Actief wachten kan echter nog voorkomen als een programma I/O request wanneer een andere I/O operatie nog actief is.

Geheugenhiërarchie

Geheugen varieert in doorvoersnelheid en capaciteit. Zo onderscheidt men verschillende niveaus:

1. Registers: zeer snel en duur processorgeheugen
2. Primaire (L1) en secundaire (L2) caches: cache tussen processor en het primaire geheugen, in de hardware geïmplementeerd in SRAM. L1 zit in de processor, met latenties van slechts enkele cycli.
3. Hoofdgeheugen: direct adresseerbaar geheugen, geïmplementeerd in DRAM (trager).
4. Externe media

Om globale toegangstijd de beperken wordt de toegangsfrequentie op elk niveau lager gehouden dan op het niveau erboven (= dynamische verdeling, met continu migraties).

Principe van de lokaliteit: verwijzingen naar code en gegevens hopen zich op in clusters, daarom altijd in blokken migreren en niet byte per byte. Blokgrootte gekozen in functie van beschikbaar geheugen. Vervangingsalgoritme bepaalt welk blok door welk moet vervangen worden. Schrijfstrategie bepaalt welke blokken naar lager gelegen niveaus moeten weggeschreven worden.

Migraties tussen de processorregisters en de caches zijn onzichtbaar voor programma's of het besturingssysteem, en worden dus hardwarematig aangepakt. Lagere lagen worden gehandhaafd door het OS, via een gespecialiseerde memory manager (zie voorbeeld pagina 10).

Doel en functie van besturingssystemen

Interface

Voor de **eindgebruiker** wordt de hardware als een verzameling toepassingen voorgesteld

Voor de **programmeur** worden er abstractielagen voorzien, om de hardware makkelijker te besturen. Dit wordt gerealiseerd met systeemprogramma's in gebruikersmodus (vb. compiler, debugger, text editor, ...) die veelgebruikte diensten aanbieden, maar ook door het aanbieden van een hardwareonafhankelijke API (vb. POSIX) om via systeemaanroepen met de kernel te communiceren.
Om het beheer van bronnen voor zijn rekening te nemen, biedt het OS nog een API aan: de *virtual machine*. Deze zorgt niet alleen voor een efficiënt gebruik van bronnen, maar zorgt er ook voor dat software overdraagbaar is op verschillende hardwareplatformen (met vb. andere hoeveelheden RAM geheugen).

De taken van het besturingssysteem bestaan algemeen uit:
- Uitvoeren van programma's
- Foutafhandeling
- Hulpprogramma's aanbieden
- API voor I/O-apparaten
- API voor toegang tot bestanden
- Beheer en delen van bronnen
- Beveiliging
- Statistieken

Beheer van bronnen

Het OS moet volgende bronnen beheren:
- Processor: beslissen welk programma uitgevoerd wordt (scheduling)
- Geheugen: ruimte verdelen onder applicaties (geheugenbeheer)
- I/O-apparaten: gelijktijdige toegang van processen tot bronnen toelaten (gelijktijdigheid)

Het OS gebruikt hiervoor beschikbare hardwarefaciliteiten.
Hoewel het besturingssysteem delegeert, verbruikt het zelf ook bronnen. Het moet daarom veel de controle over de processor uit handen geven, maar kan die via interrupts terug krijgen wanneer nodig.

Kenmerken van moderne besturingssystemen

Multitasking en multithreading

Met behulp van interrupts kan men voorkomen dat de processor niets te doen heeft omdat het actieve programma wacht op een I/O-bewerking. Als er echter een tweede proces I/O vereist maar het apparaat reeds bezig is kunnen interrupts niet helpen.

Multitasking of multiprogramming zorgt er voor dat er regelmatig van actief proces gewisseld wordt, waarbij de volgorde afhangt van de prioriteit en het feit of een proces wacht op I/O. Hierdoor blokkeren opeenvolgende applicaties die dezelfde I/O vereisen de processor niet meer. Er wordt enkel geschakeld tussen programma's in geval van een interrupt, waarbij de scheduler eventueel kan overschakelen naar vb. een programma met een hoge prioriteit.

Bij **multithreading** kan een proces meerdere instructiesporen (*threads*) bevatten.

Multiprocessing

Hierbij heeft het besturingssysteem meerdere processoren tot zijn beschikking, waardoor er meerdere programma's tegelijk kunnen uitgevoerd worden.

Bij **asymmetrische multiprocessing** zal de kernel van het besturingssysteem steeds uitgevoerd worden om een bepaalde master processor. Gebruikersprogramma's worden dan enkel op de overige processors uitgevoerd. Deze opzet vereist weinig aanpassingen aan een multitasking OS, maar is niet efficiënt en de master processor kan een bottleneck voor het systeem vormen.

Bij **symmetrische multiprocessing** zal de kernel op beide processoren draaien. Ofwel wordt het besturingssysteem daarbij opgedeeld in meerdere processen, ofwel draait elke processor een kopie van het besturingssysteem. Dit introduceert problemen wat betreft onderlinge berichtuitwisseling en synchronisatie van beheer van bronnen.
Ook het geheugenbeheer wordt veel complexer indien beide processoren over een eigen cache bezitten.
De performantie stijgt echter niet lineair, wegens overhead, en het feit dat er niet altijd genoeg processen zijn om het hele systeem efficiënt te belasten.

Modulair ontwerp

Historisch ontwerp berust op een enkele monolitische kernel die alle functionaliteit ondersteund. Deze systemen zijn heel efficiënt, maar moeilijk te handhaven. Daarom streeft men naar een modulair ontwerp.

Een mogelijke opzet daarbij is een **hiërarchisch gelaagd systeem**, met eenvoudige interfaces tussen de gescheiden lagen. De laagste laag staat in verbinding met de hardware, terwijl de hoogste laag de gebruikersinterface voorziet. Dit model, waarbij in theorie elke laag de implementatiedetails verbergt, heeft echter zijn beperkingen wanneer toegepast op OS-niveau.

- Onveilig en onstabiel: het hele OS draait in kernelmodus, en elk deel heeft (mogelijk onrechtstreeks) toegang tot de hardware en het geheugen
- Geen strikte gelaagdheid: vaak heeft een laag interactie met een niet-aanliggende laag. Proforma doorgeven van de berichten langs de tussenliggende lagen maakt alles echter veel minder efficiënt.

Hierdoor vindt men deze gelaagdheid niet terug op OS-niveau, maar wel op subsysteem-niveau (vb. I/O-subsysteem).

Een alternatief is de **microkernelarchitectuur**, waarbij enkel de essentiële kernelfunctionaliteit in kernelmodus werkt (= de microkernel). Subsystemen draaien als een server-applicatie in usermodus, en kunnen enkel onrechtstreeks (via de microkernel) de hardware benaderen. Interactie tussen de microkernel en serverprocessen, en tussen serverprocessen en gebruikersprocessen gebeurt via een uniform communicatieprotocol.
Deze opzet heeft vele voordelen:

- Overdraagbaar: alle hardwareafhankelijkheid binnen de microkernel.
- Stabiel: een crash in een subsysteem haalt de kernel niet neer.
- Makkelijk te distribueren: processen zijn er zich niet bewust van dat de subsystemen zich mogelijk op verschillende fysieke toestellen bevindt.
- Uitbreidbaar/reduceerbaar: modules zijn eenvoudig toe te voegen en te verwijderen.
- Te weinig performant: de noodzakelijke berichtenuitwisseling vertraagt het systeem sterk.

Aangezien de superieure microkernelarchitectuur nog te belastend is voor huidige hardware, grijpt men vaak naar een **hybride concept**. Zo kent men wel het modulaire

principe van invoegbare en verwijderbare deelcomponenten, maar door die monolitisch te implementeren vermijdt men de problematiek van de communicatie ertussen.

Overzicht van enkele implementaties

Windows NT

De Windows NT kernel is een hybride microkernel: ze bestaat uit een microkernel en verschillende onafhankelijke modules met een gestandaardiseerde interface, waarbij enkele modules wegens performantieredenen (directe hardwaretoegang) in kernelmodus worden uitgevoerd.

Het deel van de kernel dat in kernelmodus wordt uitgevoerd heet de Executive, en is opgebouwd uit de volgende lagen:

- *Hardware abstraction layer* (HAL): vertaling van algemene opdrachten naar hardwarespecifieke instructies (processorinstructiesets, DMA, SMP, geheugenmapping, ...). Bovenliggende lagen gebruiken de uniforme API van de HAL
- Microkernel: beheert scheduling en synchronisatie, en verwerkt interrupts. Altijd in het geheugen geladen en niet te onderbreken
- Executieve diensten: NT modules in kernelmodus
    - *I/O-manager*
    - *Cache manager* (beheert schrijfcaches)
    - *Security reference monitor* (controleren van toegang)
    - *Object manager*
    - *Process manager* (beheer proces- en threadobjecten)
    - *Virtual memory manager*
    - Grafische module

De belangrijkste modules in usermode zijn:

- Afschermingssubsystemen: leveren beveiligingsfuncties (identificatie en beveiligingsbeleid).
- Omgevingssubsystemen: interactie met gebruiker, zorgen voor een systeem-API.

Aangezien de systeem-API door modules verzorgd wordt, kan de kernel meerdere API sets ondersteunen (Dos, Win16, Win32, Win64, POSIX, OS/2). Alle niet-Win32 API aanroepen worden vertaald naar Win32 formaat om zo de Executive te benaderen. Indien de kernel een 64-bits versie is wordt hieraan een thunking-laag toegevoegd die 32-bits aanroepen in 64-bit formaat omzet.
Elk programma heeft wel maar toegang tot 1 omgevingsubsysteem, dat dynamisch geladen wordt.

Alle subsystemen (zowel usermode als executieve) zijn serverprocessen, en wachten op aanvragen om hun diensten aan te bieden. Die aanvragen kunnen komen van gebruikerstoepassingen of andere servers, en worden onder de vorm van een *local procedure call* (LPC) via de executieve naar de juiste server geleid. Het LPC mechanisme is geïmplementeerd in een serie DLL's, dat gebruikerstoepassingen met functies voorziet die API aanroepen als bericht verpakken en naar de juiste module sturen.

De kernel is objectgeoriënteerd ontworpen, maar niet geschreven in een objectgeoriënteerde taal. Objecten worden gebruikt wanneer de Executive gegevens openstelt (tussen modules, of naar userland toe). Binnen een module is dit niet noodzakelijk het geval. Objecten zijn genoemd volgens een padstructuur, met een enkele root. Security descriptors, verbonden met een object, beperken indien nodig de toegang tot het object. Usermode heeft op een indirecte manier toegang tot de objecten, gebruik makende van objecthandles en functies voorzien in de Executive. Alle objecten worden beheerd door de Object Manager.

Unix

Unix bestaat uit een monolitische kernel, en vele userland tools die samen het OS vormen. Naast 2 interfaces (hardware & gebruiker) bestaat de kernel uit twee subsystemen:

- Procesbeheer: monolitisch blok dat zorgt voor geheugenbeheer, scheduling en interprocescommunicatie.
- Bestandbeheer en I/O: interruptafhandeling, stuurprogramma's.

Aangezien alle machineafhankelijke code in 1 subsysteem zit, is Unix makkelijk te porten naar alternatieve hardware.

Recentere Unix-varianten

Modernere Unix kernels kennen een meer modulaire architectuur, waarbij een kleine kern instaat voor elementaire functionaliteit voor de modules.

De Linux kernel staat los van de Unix-kernel, maar lijkt met zijn monolitische ontwerp sterk op Unix (wegens performantieredenen). Toch zijn er dynamisch laadbare modules (*loadable kernel module*), die bij uitvoering aan de kernel gelinkt worden waardoor het ook in kernelmodus draait.

Solaris kent een meer doorgedreven microkernelarchitectuur, maar is daardoor ook minder performant.

Procesbeheer

Procestoestanden

Doel: verweven van processen (multitasking), ten behoeve van maximaal processorgebruik mits aanvaardbare responstijd.

Een programma is de code die uitgevoerd moet worden (passief), terwijl een proces de actieve entiteit is die een programma uitvoert. Elk proces heeft de illusie dat het alleen uitgevoerd wordt, met een eigen programmateller die indiceert wat er moet uitgevoerd worden.

Basismodel

Een proces wordt al dan niet uitgevoerd, en kent dus twee toestanden: actief en niet-actief. Nieuwe processen worden gecreëerd op vraag van bestaande processen, en komen binnen in de toestand niet-actief.

Regelmatig onderbreekt het OS een programma zonder het daar om gevraagd heeft (= preëmptief), en kiest de CPU scheduler een niet-actieve taak om uitgevoerd te worden. Niet-actieve taken worden in een wachtrij geplaatst, meestal geïmplementeerd als gekoppelde lijst met verwijzingen naar de processen (alternatieven zijn mogelijk in geval van prioriteiten).

Meestal bestaat er ook een idle-proces met laagste prioriteit, om de situatie op te vangen wanneer geen proces te processor nodig heeft. Dit proces voert dan meestal een energiebesparende instructie uit (HLT).

Blokkeren

In de niet-actieve toestand kan er echter nog een onderscheid gemaakt worden:
- Gereed / *ready*: voor processen die klaar zijn om uitgevoerd te worden
- Geblokkeerd / *blocked*: voor processen die wachten op een bepaalde gebeurtenis (vb. I/O interrupt)

Onderbreken (*swappen*)

Wegens de beperkte hoeveelheid hoofdgeheugen kan het echter voorkomen dat er geen ruimte meer is voor nieuwe processen. Daarom voert men de suspended toestand in, waarbij een proces uitgeswapt wordt naar secundair geheugen (vb. schijfgeheugen). Het toestandsdiagram kent daardoor twee nieuwe toestanden:
- Gereed suspended
- Geblokeerd suspended

Vooral als veel processen die wachten op I/O de processor blokkeren, kan uitswappen de performantie verbeteren. Het risico bestaat echter dat de situatie nog erger wordt, aangezien het uitswappen zelf ook een I/O-operatie inhoudt.
Zo kan het OS in noodsituaties ook processen uit de toestand gereed uitswappen, of een actief proces direct uit de *running* toestand migreren naar *ready-suspended*. Eveneens kunnen processen pro-actief uit de toestand geblokkeerd suspended terug in het hoofdgeheugen geladen worden.

Beschrijving van processen

Een proces wordt getypeerd door zijn procesbeeld, dat bestaat uit:
- *Proces control block* (PCB)
- Private adresruimte
    - Programmainstructies
    - Programmagegevens
- Usermode stack
- Kernelmode stack

De stack dient hierbij om lokale variabelen en parameters bij functieaanroepen op te slaan.
Bij het uitvoeren moet het procesbeeld in het hoofdgeheugen geladen zijn, enkel bij de onderbroken toestand zal het procesbeeld zich op een schijf bevinden.

Het besturingssysteem houdt tabellen bij van de informatie die het maakt. Meestal vindt men de volgende tabellen terug:
- Geheugentabellen: beheren hoofdgeheugen en virtueel geheugen
- I/O-tabellen: beheren I/O-controllers (statusinformatie, procestoewijzingen, adressen van DMA, ...)
- Bestandstabellen: bestandsinformatie (meestal uitbesteed aan een bestandsbeheersysteem)
- Procestabellen: beheren van processen

De procestabellen zijn meestal opgedeeld in een primaire tabel, die een ingang bevat voor elke proces en zo verwijst naar de secundaire tabellen die men terug vind in het *process control block* (PCB). Het PCB is een verzameling van attributen, dat opgeslagen wordt in de geheugenruimte van het proces zelf (niet centraal beheerd). Om de werking van het OS te garanderen moet de PCB echter altijd in het hoofdgeheugen geladen zijn, ook als het proces uitgeswapt is.

Het procesbesturingsblok

Elk PCB blok bevat alle informatie over een proces die het OS nodig heeft. Ze zijn enkel manipuleerbaar in kernelmodus. Men onderscheidt de volgende categorieën:

- Procesidentificatie (PID): unieke numerieke identificatie, meestal index in primaire procestabel. Is soms gebaseerd op PID van ouder.
- Processortoestandsinformatie: inhoud van alle processorregisters.
- Procesbesturingsinformatie: aanvullende informatie.
    - Scheduling- en toestandsinformatie (processorstaat, prioriteit, gebeurtenissen waarop het proces wacht, hoelang het proces wacht, uitvoeringstijd, ...)
    - Structurele informatie (koppelt PCB's aan elkaar, verwijzing naar vb. parent/child)
    - Locatie van elk deel procesbeeld
    - Locatie geheugentabellen (vb. ter allocatie van dynamisch geheugen)
    - Bronnen (aangevraagd, aangewezen)
    - Privileges
    - Limieten & quota's
    - Vlaggen, signalen

Stappen bij creatie proces

Het maken van een nieuw proces bestaat uit een aantal stappen:

- Genereren PID en entry in primaire procestabel
- Toewijzing ruimte procesbeeld
- Initialisatie PCB
- Koppelingen instellen (oa. wachtrijen gereed/gereed-suspended, PCB verwijzingen van parent/child)
- Aanmaken/uitbreiden van andere gegevensstructuren)

Wisselen van processen

Opportuniteiten

Proceswisselingen zijn enkel mogelijk wanneer het OS de besturing in handen heeft. Dit is mogelijk bij:

1. Interrupts: veroorzaakt door een gebeurtenis buiten het huidige actieve proces.
    - Klokinterrupts: om vast te stellen dat proces zijn quota overschreden heeft. Gegenereerd via een klok & timer met vaste frequentie.
    - I/O-interrupts: melden dat een I/O-aanvraag afgehandeld is.
    - Paginafouten: verwijzingen naar virtueel geheugen dat zich niet in het hoofdgeheugen bevindt.

Paginafouten moeten synchroon afgehandeld worden met het programma, dwz. dat het programma pas kan hervat worden als de interrupt verwerkt is en het geheugen ingeladen is. Klokinterrupts en I/O-interrupts zijn asynchroon: ze kunnen onmiddellijk uitgevoerd worden, of kunnen uitgesteld worden.

2. Trap: een exception, uitzonderingsconditie binnen het actieve proces (vb. geweigerde toegang, of deling door 0). Worden synchroon afgehandeld.
3. Systeemaanroep: vormt de communicatie tussen het besturingssysteem en gebruikersprogramma's. Alle I/O-operaties worden zo gerealiseerd. Veroorzaakt op expliciet verzoek van het actieve proces.
    - Berichtgestuurd: in geval van een microkernel. Processen versturen een aanvraag op een communicatiekanaal, en laten zich blokkeren tot er een antwoord beschikbaar is. Zo ook laten serverprocessen de controle over aan de scheduler als alle aanvragen beantwoord zijn.

    - Proceduregestuurd: klassiekere benadering, waarbij elke systeemaanroep een identificatie van de aanroep in een bepaald register plaatst en vervolgens een trap genereert. In geval van meer parameters dan registers, wordt een verwijzing naar een tabel in de registers geplaatst. Zo wordt ook het resultaat van de aanroep in een register geplaatst. Deze mechanismen zitten verpakt in bibliotheekfuncties om makkelijk in gebruik te zijn. Minder prioritair dan I/O-interrupts.

Wisselen van processen

Twee types wisselingen:

1. Contextswitch
    - Context van de processor opslaan. Hierbij wordt de processortoestandsinformatie hardwarematig opgeslaan (1 instructie), en het adres naar die informatie opgeslaan in een register.
    - Programmateller instellen op adres van nieuw proces of routine (vb. interruptafhandelingsroutine).
2. Processwitch (acties bovenop de contextswitch)
    - Procesbesturingsinformatie van het oude en nieuwe proces bijwerken.
    - PCB oude proces in gepaste wachtrij plaatsen.
    - Gegevensstructuren geheugenbeheer bijwerken.
    - Context van het nieuwe proces laden.

Een contextswitch is vrijwel volledig hardwarematig geïmplementeerd, waarbij een enkele instructie meerde registers opslaat. Een processwitch daarentegen is veel trager, aangezien het OS vele datastructuren moet overlopen en aanpassen.

Handelingen indien de processor een interrupt detecteert:

- Contextswitch naar de interruptafhandelingsroutine.
- Modusswitch naar kernelmode.
- Instructiecyclus voortzetten, interruptafhandeling starten.

Na afhandeling van de interrupt wordt de scheduler-code uitgevoerd, die beslist welk proces de besturing over de processor zal krijgen:

- Het oude proces: hierbij wordt enkel de context opnieuw geladen en de programmateller hersteld (eveneens meestal hardwarematig).
- Een ander proces: hierbij zal het OS een processwitch uitvoeren naar het nieuwe proces.

Uitvoering van besturingssystemen

Besturingssystemen, in wezen een normaal programma, wordt in relatie tot processen op verschillende manier geïmplementeerd:

1. Oudere besturingssystemen: kernel zonder processen, met eigen geheugengebied & stack. De kernel wordt in geprivilegeerde mode uitgevoerd bij elke interrupt, waardoor er veel contextswitches optreden.
2. Unix & Linux: het besturingssysteem wordt beschouwd als een verzameling systeemaanroepen, waardoor het besturingssysteem steeds in de context van het proces wordt uitgevoerd. Alle processen delen een gemeenschappelijke adresruimte met de functies van het besturingssysteem, maar voeren die uit in een private kernelstack. Een interrupt veroorzaakt zo enkel een modusswitch, waarna eventueel een contextswitch plaatsvindt naar een routine voor proceswisseling (die wél buiten de processen om wordt uitgevoerd).
3. Microkernel benadering: besturingssysteemfuncties als aparte processen, uitgevoerd in kernelmodus. Zorgt voor veel processwitches, maar heeft als voordeel dat de functies van het OS via prioriteiten met andere processen kunnen verweven worden.

Multithreading

Een thread is een eenheid voor de verdeling van procesorinstructies (= uitvoeringspad). Een proces zien we als een eenheid voor de eigendom van bronnen, en kan dus meerdere threads bevatten.

- Traditionele besturingssystemen (DOS, Unix): elk proces heeft 1 thread. Elk proces kent een traditionele layout, met een PCB, geheugen voor het programma & gegevens (= *user address space*), en een usermode & kernelmode stack.
- Moderne besturingssystemen (NT, Solaris): een proces kan meerdere threads bevatten, waarbij er dan meerdere programmatellers aanwezig zijn. Alle threads delen dezelfde bronnen. De layout van een proces verschilt: het PCB en user address space zijn gedeeld (dus ook de programmacode!), maar elke thread kent een eigen stack plus een *thread control block* (vereenvoudigde versie van het PCB, zonder de suspended toestanden).
  Scheduling gebeurt op niveau van threads, waardoor sommige processwitches gereduceerd kunnen worden tot een contextswitch (indien in hetzelfde proces).

Voordelen

Een programma gebaseerd op threads efficiënter dan de versie gebaseerd op een aantal processen:

- Toestand en bronnen gedeeld. Ook de code is gedeeld, waardoor er vele threads kunnen gecreëerd worden zonder veel overhead.
- Communicatie tussen threads buiten de kernel om.
- Creëren en wisselen van threads verloopt veel sneller.
- Een wachtende thread blokkeert het programma niet, andere threads kunnen ander werk verrichten.

Implementaties

Threads kunnen op verschillende manier geïmplementeerd worden::

1. User-level threads: het proces gebruikt een threadlibrary (Green Threads, Fiber, Pthreads), dat zelf zorgt voor het threadbeheer. De kernel heeft geen weet van de threads, en schedulet enkel het volledige proces.
   - Overdraagbaar naar andere besturingssystemen.
   - Efficiënte threadswitches, geen nood aan modusswitch en uitsparen van contextswitches (functies voor threadbeheer bevinden zich in de user address space van het proces).
   - Scheduling kan aangepast worden naar behoeven van de applicatie.
   - Een blocking systeemaanroept blokkeert het hele proces.
   - Het proces kan maar 1 thread tegelijk verwerken (geen voordeel aan multiprocessing).
2. Kernel-level threads: beheerd door het besturingssysteem zelf.
   - Verwisselen van thread vereist modus- en contextswitches.
   - Worden beïnvloed door de scheduler van het OS.
   - Bij een blocking systeemaanroep kan een andere thread voortgezet worden.
   - Het proces kan profiteren van multiprocessing.

Userland serverapplicaties (zoals de Apache webserver) maken veel gebruik van *threadpools*. Hierbij worden tijdens het starten vele threads aangemaakt, die in een centrale pool wachten op aanvragen. Dit maakt het gemakkelijk om dynamisch de grootte van de pool aan te passen.

3. Hybride vorm: combinatie van beide type threads, waarbij de userlevel threadbibliotheek communiceert met de kernel via lichtgewicht processen (*lightweight processes*). De kernel gebruikt een kernellevel-thread per

lichtgewicht proces, terwijl de threadbibliotheek deze gebruikt om verschillende userlevel-threads met uit te voeren. De specifieke koppeling van userlevel threads aan kernellevel threads wordt gekozen in functie van efficiënt programmeren.
    - Een blocking systeemaanroep van een userlevel thread blokkeert het hele lichtgewicht proces, maar andere threads verbonden met een ander lichtgewicht proces kunnen wel nog werken.

Thread API's

De POSIX standaard voor threads, **Pthreads**, definiëert enkel een specificatie voor userlevel threads. Elk OS kan dit anders implementeren. Er is geen strikte relatie tussen een thread van de Pthreads API, en eventuele kernellevel threads. Er zijn 2 vormen van scheduling gespecificeerd:

- *System-contention scope*: om te beïnvloeden welke kernellevel threads uitgevoerd worden op welke processor.
- *Process-contention scope*: op te beïnvloeden hoe userlevel threads gekoppeld worden aan lichtgewicht processen.

Scheduling vindt plaats gebaseerd op instelbare dynamische prioriteit, waarbij timeslicing voor processen met gelijke en hogere prioriteit optioneel is.

Java-threads zijn de threads die Java implementeert. Ze worden volledig door de JVM beheerd, en zijn dus moeilijk te classificeren.

- Koppeling tussen userlevel en kernellevel threads kan niet gewijzigd worden, en kan onderling verschillen.
- Scheduling is niet gedetailleerd beschreven. Prioriteiten zijn beschikbaar, maar of ze gerespecteerd worden kan afwijken.

Procesbeheer in reële besturingssystemen

Unix (klassiek)

Grootste deel van de taken uitgevoerd in de context van applicaties (mits eventuele modusswitches). Er bestaan echter ook systeemprocessen die in kernelmode werken, en gecreëerd worden bij het opstarten van het systeem (swapper, pagedaemon, init). Multithreading wordt niet ondersteund.

Kindprocessen worden aangemaakt met *fork()*, waarbij de volgende code in kernelmode van het ouderproces uitgevoerd wordt:

- Nieuwe entry in de procestabel
- Exacte kopie van procesbeeld ouder naar procesbeeld kind (met als uitzondering: gedeeld geheugen)
- PID kind aan ouder, 0 aan kind

Aan de hand van de code die *fork()* teruggaf, kunnen ouder en kind andere code gaan uitvoeren. Met de *execve()* call kan het kindproces zijn procesbeeld vervangen door de inhoud van een bestand. Om zo duplicaat werk te vermijden gebruikt men *vfork()* waarbij het procesbeeld van de ouder niet gedupliceerd wordt.

Een proces kan zichzelf beëindigen met de *exit()* aanroep, waarbij het procesbeeld (met uitzondering op het PCB) uit het geheugen verwijderd wordt (= *zombie proces*). Met de *wait()* aanroep kan een ouderproces wachten tot een kindproces gedaan heeft met uitvoeren, waarna het zombie-kindproces volledig verwijderd wordt.

Unix differentiëert de *actief/running* toestand naargelang het proces zich in user- of kernelmode bevindt. Als een proces van toestand wil veranderen, moet het altijd eerst

naar kernelmode switchen (zie figuur 2.33).
De toestand *gereed/blocked* wordt ook gesplitst*:*

- *Preëmptief beëindigd/preempted*: voor kernelprocessen die onderbroken werden door een interrupt. Na interruptafhandeling wordt dit proces onmiddellijk hervat.
- *Ready to Run*: voor userprocessen die onderbroken werden door een interrupt. Dit proces wordt echter niet met zekerheid hervat, daar beslist de scheduler over.

Door deze differentiatie weet men zeker dat er bij kernelprocessen nooit een proceswissel zal plaatsvinden. Dit vereenvoudigt de interne datastructuren, maar maakt de kernel ongeschikt voor realtime gebruik en multiprocessing.

Linux

De PCB informatie wordt niet helemaal in het procesbeeld van het proces bewaard, maar deels in aparte substructuren waarnaar het PCB een verwijzing bevat.
fork() wordt vervangen door *clone()*, waarbij echter ook vlaggen doorgegeven kunnen worden (specificeren wat er gedeeld wordt door ouder en kind). Zonder vlaggen werkt *clone()* dus zoals *fork()*.
Processen en threads worden gelijkgesteld onder de noemer *task*, waarbij een thread gewoon een task is waarbij het adresgeheugen gedeeld wordt met de ouder (instellen via *clone()* vlaggen).

Solaris

Solaris is sterk gericht op gebruik van threads, waarbij het verschillende structuren ondersteund:

- Userlevel threads: interface voor het ontwerpen van multithreaded applicaties.
- Kernellevel threads: entiteiten die gescheduled kunnen worden op een bepaalde processor. Het OS zelf gebruikt deze threads (zonder de link naar lichtgewicht processen, om proceswisselingen te voorkomen).

Userlevel thread worden bij uitvoering dynamisch aan een lichtgewicht processen gebonden, zodat er minder van nodig zijn om alle threads te kunnen uitvoeren.

Userlevel threads verlaten de status *actief* om verschillende redenen:

- Synchronisatie: de thread zal slapen tot aan de voorwaarde voor synchronisatie voltooid is.
- Gestopt worden: door zichzelf of door een andere userlevel thread.
- Preëmptief onderbroken worden: als een thread met hogere prioriteit actief wordt, of als de thread de controle overlaat aan een andere thread met gelijke prioriteit.

Als de thread actief is kan het corresponderende lichtgewicht proces verschillend statussen aannemen, waardoor een actieve thread niet altijd praktisch in uitvoering is.

Windows NT

Geïmplementeerd als objecten die overerven van dezelfde superklasse, waarbij elk subsysteem specifiekere functionaliteit implementeert. Hierdoor kunnen sommige omgevingssubsystemen threads ondersteunen (Win32, OS/2), en andere niet (Win16, POSIX). Elk proces heeft:

- Primair token: een access token voor beveiliging, bevat kopie beveiligingsidentificatiecode gebruiker, gebruikt om te controleren of de gebruiker toegang heeft tot beveiligde objecten.
  Threads verkrijgen enkel een token op aanvraag (*imitatie*), vb. als een serverproces iets uitvoert in naam van een gebruiker.
- Virtuele adresruimte, beheerd door de Virtual Memory Manager (module v/d Executive).

- Objecttabel, bevat handles (gestandaardiseerde interface voor alle types objecten) naar andere objecten (vb. threads v/e proces)

Usermode modules worden als afzonderlijke processen geïmplementeerd, die een aantal kernellevel threads omvatten. Sommige modules binnen de Executive worden geïmplementeerd als threads (= systeemthreads), die administratief in 1 proces gebundeld worden (het *System* proces = idle proces). Enkel de microkernel is niet in processen of threads uitgevoerd, en kan dus niet preëmptief onderbroken worden.

NT-threads kennen 6 verschillende toestanden:
- *Gereed/Ready*
- *Actief/Running*
- *Beëindigd/terminated*
- *Standby*: bevat de thread uit *gereed/ready* met hoogste prioriteit, wordt als eerste uitgevoerd. Indien de prioriteit groot genoeg is wordt actieve thread preëmptief onderbroken, zoniet zal de thread moeten wachten tot de actieve thread ophoudt of onderbroken wordt. Er is 1 dergelijke *standby* status per processor.
- *Wachtend/waiting*: als een thread wacht op iets (= I/O), of zichzelf vrijwillig onderbreekt. Na het wachten gaat de thread naar *ready* indien de bron beschikbaar is, of naar *transition* indien er langer moet gewacht worden.
- *Transition*: hier wacht een thread tot een bepaalde bron beschikbaar is (vb. I/O aanvraag afgehandeld).
- *Terminated*: als een thread zichzelf beëindigd heeft.

Aangezien kernellevel threads nog veel overhead kennen biedt NT de *fiber*-bibliotheek aan voor userlevel threads. Een fiber wordt gevormd door een thread die zich omzet naar een fiber, en vervolgens eventueel nieuwe fibers maakt. Een fiber heeft toegang tot de adresruimte van het proces, tot de TLS van de thread en tot een Fiber Local Storage eigen aan de fiber.
NT kent ook specifieke API's voor kernellevel poolthreads. Het aantal worker-threads per pool wordt door NT zelf dynamisch aangepast.
Dankzij deze twee voorzieningen moet er veel minder gebruik gemaakt worden van threads, en kan men threads hergebruiken. De combinatie van fibers en pools is functioneel nagenoeg identiek aan de Solaris implementatie van kernellevel-threads en lichtgewicht processen.

Gelijktijdigheid

Problematiek

Bij multitasking komen verschillende problemen naar boven:
- communicatie tussen processen
- delen van bronnen
- synchronisatie van processoren
- toewijzen van processortijd (scheduling, apart hoofdstuk)

Algoritmen classificeren op basis van de mate waarin processen zich bewust zijn van elkaar.

Wederzijdse uitsluiting

Als processen bronnen delen zonder ze het weten (= geen uitwisseling van informatie).
Conflict: twee processen die niet van elkaar weten maken tegelijkertijd gebruik van dezelfde bron.
Oplossing: ben beschouwt de bron als een *kritieke bron* en de sectie waarin een proces gebruikt maakt van die bron een *kritieke sectie*. Het stellen dat slechts 1 proces tegelijk

zich in zijn kritieke sectie mag bevinden heet men *wederzijdse uitsluiting / mutual exclusion*.

Mogelijke problemen bij wederzijdse uitsluiting:
- *Patstelling/deadlock*: twee processen hebben twee bronnen tegelijkertijd nodig, en elk proces bevindt zich in de kritieke sectie van 1 van de 2 bronnen.
- *Uithongering/starvation*: drie processen hebben toegang nodig tot een bron, maar de toegang wordt enkel toegewezen aan de eerste twee processen waardoor het derde proces nooit toegang krijgt tot de bron.

Het besturingssysteem moet voorzieningen aanbieden voor wederzijdse uitsluiting (API's voor vergrendelen en ontgrendelen van bronnen), met de volgende kenmerken:
- Dwingend: slechts 1 proces mag zich in de kritieke sectie van een welbepaalde bron bevinden.
- Geen deadlocks.
- Geen uithongering.
- Eerlijke verdeling van tijd in kritieke sectie.
- Gevoelloos voor onderbrekingen.
- Geen veronderstellingen over aantal processoren, of over relatieve snelheid van de processen (zoniet: raceproblemen)

Synchronisatie

Soms delen processen bepaalde globale variabelen doelbewust om te communiceren met elkaar. Hierbij moet synchronisatie verzekeren dat de cruciale operaties (meestal schrijfoperaties) in de juiste volgorde uitgevoerd worden. Het besturingssysteem moet hiervoor mechanismen aanbieden om te regelen in welke volgorde processen toegang hebben tot de gedeelde data.

De gemakkelijke oplossing hierbij is om de cruciale instructies als een kritieke sectie te beschouwen. Dit is echter zo restrictief dat er maar slecht gebruik kan gemaakt worden van multiprocessing.

Een veelgebruikt probleem is dat van de producent-consument, waarbij een circulaire buffer gebruikt wordt om de tijdelijke gegevens in op te slaan. Men moet voorkomen dat de consument een leeg item inleest, en dat de producent een bestaand item overschrijft. Wederzijdse uitsluiting is hier te restrictief, en doet het voordeel van de circulaire buffer (die de verschillen in tempo tussen producent en consument opvangt) teniet. Synchronisatie is noodzakelijk, zodat de consument nooit uit een lege buffer leest en de producent nooit in een volle buffer schrijft.

Softwarebenaderingen voor wederzijdse uitsluiting

Softwarematige oplossingen zijn algoritmes die opgelegd worden aan elk proces dat de bron nodig heeft. Ze bestaan meestal uit twee delen:
- *Entry protocol*: om te controleren of een proces de kritieke sectie mag binnen gaan.
- *Exit protocol*: wat het proces moet doen na de kritieke sectie afgehandeld is.

Er wordt gebruik gemaakt van geheugenarbiters: gedeelde elementen die het algoritme gebruikt om te bepalen wie aan beurt is. Om dit te werken moet toegang tot dit geheugen atomair zijn.

Algoritme van Dekker

Dit is het oudste algoritme voor wederzijdse uitsluiting.

Twee concurrerende processen

Hierbij wordt een gedeelde variabele gebruikt, die indiceert welk van beide processen de kritieke sectie mag uitvoeren. Als exit protocol wordt deze variabele geinverteerd. Beide processen wachten actief tot de waarde gelijk is aan (1|0).

Er zijn echter problemen met deze implementatie: moest 1 proces crashen zal het tweede permanent geblokkeerd worden. Ook wordt het tempo van uitvoering bepaald door het langzaamste proces.

Twee concurrerende processen - globale array

Bij de eerste uitbreiding maakt men nu gebruik van een globale variabele vlag[2]. Hierbij zal elk proces tijdens het actief wachten de vlag van het andere proces in het oog houden. Is die vlag gunstig, zal het proces als entry protocol zijn eigen vlag op "busy" zetten, de kritieke sectie uitvoeren, en als exit protocol zijn vlag op "free" zetten.

Wederzijdse uitsluiting is hierbij echter niet verzekerd. Moesten beide processen gelijktijdig vaststellen dat het andere proces zich niet in de kritieke sectie bevindt, treedt een race-conditie op en gaan beide processen hun kritieke sectie binnen.

Twee concurrerende processen - uitbreiding globale array

Om dit euvel te voorkomen stelt men de eigen vlag op "busy" in voor het actief wachten. Men kan dus stellen dat de *vlag* array indiceert of een proces geïnteresseerd is om zijn kritieke sectie uit te voeren.
Wederzijdse uitsluiting is nu wel verzekerd, maar er als beide processen nu tegelijk vaststellen dat het andere proces op "busy" staat zullen beide processen eeuwig actief wachten (= *livelock situatie*, variant op de deadlock).

Twee concurrerende processen - correctie livelock

Om deze livelock te voorkomen voert men tijdens het actief wachten wat extra code toe: eerst zet men de eigen vlag op "free", wacht men een willekeurige tijd, om daarna de vlag opnieuw op "busy" te zetten en eventueel de kritieke sectie binnen te treden. Hierdoor wordt de kans op een deadlock geminimaliseerd, maar niet uitgesloten (moest de random() waarde altijd identiek zijn levert dit een livelock op).

Twee concurrerende processen - wederzijdse beleefdheid

Hierbij introduceert men opnieuw de globale variabele beurt, die nu indiceert in welke volgorde processen het recht hebben om hun kritieke sectie uit te voeren. Hierdoor is het onmogelijk dat een proces de kritieke sectie monopoliseert, aangezien de *beurt* variabele omgewisseld wordt na het uitvoeren van de kritieke sectie.

Algoritme van Peterson

Deze vereenvoudiging van het algoritme van Dekker voegt de twee controles (of het ander proces niet geïnteresseerd is in de kritieke sectie en of de beurt wel aan ons is) samen. De vlag van het proces wordt nu ook enkel gezet voor het actief wachten, en indiceert dus enkel of het proces geïnteresseerd en/of actief is in de kritieke sectie.
Het algoritme is dus herleid tot de derde implementatie van het algoritme van Dekker, waarbij de "beurt" variabele gebruikt wordt om livelocks te voorkomen, aangezien in geval van gelijktijdige start het onmogelijk is dat aan beide condities van het actief wachten voldaan is.

Meerdere concurrerende processen

Voor deze uitbreiding zal men eerst het algoritme herschrijven zodat beide processen dezelfde code kunnen gebruiken. Hiertoe wordt een wachtrij gebruikt.

Algoritme van Eisenberg en McGuire

Stelt een bovengrens aan het aantal wachtcycli van elk proces.

Algoritme van Lamport

Dit algoritme implementeert een ticket systeem, naar analogie van een winkel waarbij klanten een nummer krijgen en de klant met het kleinste nummer bediend wordt. Na bediening, gooit de klant zijn ticket weg.

Praktisch vraagt elk proces een ticket op aan de hand van een globale variabele, door die te verhogen en het resultaat ervan te nemen. Dit is geen atomaire instructie, waardoor twee processen een identieke ticket kunnen krijgen. Een vlag indiceert of proces bezig is met het opvragen van een ticket.

Na een proces een ticket gekregen heeft, zal het alle processen buiten zichzelf overlopen en een aantal controles doen. Eerst en vooral wordt er actief gewacht indien een proces bezig is met het opvragen van een ticket, want als een proces onderbroken werd bij het genereren van zijn ticket kan verkeerde informatie in de globale array opgeslaan zijn, waardoor een proces met hogere ticket de kritieke sectie kan binnengaan om vervolgens te conflicteren met het originele proces als dat weer hervat wordt en de laagste ticket-waarde bevat.
Daarna wordt actief gewacht indien de waarde van het ticket van een proces lager is dan de eigen ticket waarde. Tenslotte wordt er gewacht indien beide ticket waardes identiek zijn, maar het andere proces een lagere prioriteit heeft.

Indien aan al deze voorwaarden voldaan is, en er uit de lus gegaan wordt, kan het proces zijn kritieke sectie binnengaan om vervolgens zijn eigen ticket te resetten naar 0.

Hardwarebenaderingen voor wederzijdse uitsluiting

Uitschakelen van interrupts

Als een proces tijdens het uitvoeren van de kritieke sectie proceswisselingen elimineert door interrupts uit te schakelen, is wederzijdse uitsluiting verzekerd. Het geeft gebruikersprocessen echter de macht om het systeem te bevriezen, en om de processor of een andere bron de monopoliseren. Het is echter wel aanvaardbaar binnen de kernel van het besturingssysteem, maar niet voor gebruikersprocessen. Bovendien werkt het niet in het geval van meerdere processoren (geen garantie op wederzijdse uitsluiting).

Atomaire instructies

Om dit probleem te vereenvoudigen, zijn er instructies ontworpen die minimaal twee handelingen kunnen uitvoeren zonder onderbroken te worden. Dit zijn atomaire instructies. *testset* is zo een instructie, waarbij een gegeven geheugenadres ingelezen wordt. Indien dat geheugenadres een 0 bevat, wordt die op 1 gezet en wordt een 1 teruggegeven. Is de waarde niet 0, wordt geen actie ondernemen en geeft de functie een 0 terug. Dit vereenvoudigd het protocol voor wederzijdse uitsluiting sterk (Dekker versie 1).

Een andere instructie is *exchange*, waarbij de inhoud van twee geheugenadressen verwisseld wordt. Hiermee is opnieuw een eenvoudiger protocol voor wederzijdse uitsluiting (met een globale array, Dekker versie 3) te maken.

In geval van meerdere processoren met eigen cache introduceren atomaire instructies echter een grote overhead: de overeenkomstige cache entries van de lokale geheugenmanipulatie moet als een *cache invalidation* gemarkeerd worden in de cache van de andere processor.

Primitieven in het besturingssysteem

De besproken software- en hardwarebenaderingen kennen echter enkele problemen:

- De verantwoordelijkheid ligt bij de processen, die zich exact aan de regels van het protocol zou moeten houden. Valsspelen is mogelijk, het protocol is niet dwingend.
- Veel overhead, door o.a. het actief wachten.
- Deadlocks kunnen voorkomen als het proces in de kritieke sectie preëmptief beëindigd wordt omdat het actief wachtende proces een hogere prioriteit heeft.
- Uithongering is mogelijk, aangezien de selectie van het wachtende proces dat geactiveerd wordt meestal willekeurig is.

Men lost dit op door de implementatie te verhuizen naar het besturingssysteem, en primitieven onder de vorm van systeemaanroepen te voorzien. Achter de schermen gebruikt men dan meestal wel hardwarematige oplossingen.

- Uniprocessor systemen: tijdelijk uitschakelen van bepaalde interruptniveaus.
- Multiprocessor systemen: atomaire instructies. Soms vervangt men de wachtlus door blokkerende threads, maar aangezien threads overhead opleveren kiest het OS in geval van korte wachttijden voor een spinlock (actief wachten). Duurt de spinlock te lang, gaat men over op blokkerende threads (= delayed blocking).

Wat betreft de systeemaanroepen, zijn er verschillende mogelijke implementaties, waarbij semaforen de te verkiezen variant is.

Signalen

Hierbij voorziet het besturingssysteem een *kill(proces, code)* systeemaanroep waarbij een signaal naar een bepaald proces kan gestuurd worden. Via de *signal(code, functie)* kan een proces een signaalcode binden aan een functie, die uitgevoerd zal worden bij ontvangst van dat signaal.
Alle threads van een proces ontvangen ook de ontvangen signalen, tenzij een thread een bepaald signaal gemaskeerd heeft. Zo kan de afhandeling van signalen aan specifieke threads uitgespendeerd worden.

Signalen worden gebruikt om twee systeemaanroepen te implementeren: *slaap()* en *wek(proces)*. Met de *slaap()* aanroep blokeert een proces zichzelf tot een ander proces het wekt met de *wek(proces)* systeemaanroep. Zo kan het producent-consument probleem vereenvoudigd worden.
Er is echter een risico op een race aangezien de toegang tot de globale variabele aantal niet afgeschermd is. Als de buffer leeg is (aantal = 0) maar de consument preëmptief beëindigt is vooraleer hij kon slapen (maar na de if-conditie), zal de wek-aanroep van de producent zijn effect missen en na verloop van tijd beide processen slapen.

Er bestaat ook het *pause()* signaal, waarbij een proces blokeert tot het een willekeurig signaal ontvangt. POSIX specifiëert varianten hierop die wachten op een specifiek signaal (*sigsuspend*, *sigwait*).

Sequencers en eventellers

Deze implementatie vereenvoudigd een ticket-systeem zoals het algoritme van Lamport. Hiervoor biedt het besturingssysteem twee structuren aan:

- Sequencer: een incrementele teller. Dit elimineert het randgeval waar twee tickets gelijk waren.
- Eventellers: een structuur dat een geheel getal bijhoudt, en ook een wachtrij van processen.

Deze systeemaanroepen moeten echter atomair werken (en dus ongevoelig zijn voor procesveranderingen).

Om met deze datastructuren te werken zijn volgende systeemaanroepen geïmplementeerd:

- *ticket(sequencer)*: deze systeemaanroep geeft de waarde van de sequencer terug, en verhoogt de waarde ervan achteraf. Dit is ongevoelig voor proceswisselingen.
- *await(eventeller, int)*: deze aanroep plaatst het proces in de wachtrij van de eventeller, en wacht tot de waarde van de eventeller even groot is als doorgegeven integer. Dit doet de processen niet actief wachten, maar laat ze in de wachtrij blokkeren en wachten op een bepaalde gebeurtenis.
- *advance(eventeller)*: hierbij wordt de waarde van de eventeller verhoogd.

Het ticket-algoritme van Lamport werkt dan als volgt:

- Proces vraagt ticket aan via *ticket(sequencer)* systeemaanroep.
- Proces wacht tot hij aan de beurt is via de *await(eventeller, proces_ticket)* aanroep.
- Vervolgens mag de kritische sectie uitgevoerd worden met garantie op wederzijdse uitsluiting.
- Tenslotte verhoogt het proces de waarde van de eventeller via *advance(eventeller)*, zodat eventuele volgende processen in de wachtrij aan de beurt gelaten worden.

Het aantal processen in de wachtrij is altijd te berekenen via *sequencer-eventeller-1*, en de volgorde van uitvoering wordt bepaald door de waarden die de sequencer teruggeeft (puur incrementeel = processen worden in volgorde van aanvraag uitgevoerd).

Op vergelijkbare manier kan men ook het probleem van producent en constument oplossen.

Semaforen

Een semafoor is een datastructuur die een geheel getal bijhoudt, en een wachtrij van (geblokeerde) processen. Het geheel getal kan naargelang zijn teken twee betekenissen hebben

- Positief: er staan N weksignalen te wachten op verwerking.
- Negatief: er staan N processen geblokkeerd in de wachtrij.

Naast deze datastructuur biedt het besturingssysteem enkele systeemaanroepen aan:

- *neer()*: dit verlaagt de waarde van de semafoor met 1. Als de waarde daardoor negatief wordt, blokkeert het OS het proces en neemt het op in de wachtrij van processen. Is dit niet het geval, kan het proces de gewone werking voortzetten.
- *op()*: hierbij wordt de waarde van de semafoor verhoogd met 1. Als de semafoor negatief was, zal er 1 proces uit de wachtrij verwijderd worden en hervat worden in zijn geblokkeerde *neer()* instructie.

Bij deze systeemaanroepen is het imperatief dat het inspecteren van de waarde, het veranderen ervan en het eventuele (de)blokkeren van het proces 1 enkele atomaire instructie vormt.

Het concept van semaforen heeft veel gemeen met sequencers/eventellers, maar er zijn verschillen:

- *advance()* brengt een deelverzameling processen in gerede toestand, *op()* deblokkeert er slechts 1.
- Semaforen leggen niet vast in welke volgorde processen deblokkeren.
- Er bestaat (algemeen) geen methode om de waarde van de semafoor op te halen zonder daarvoor een aparte systeemoproep te gebruiken.

Met semaforen kan wederzijdse uitsluiting heel eenvoudig gerealiseerd worden:

- Per bron wordt een globale semafoor geïnitialiseerd, met als startwaarde het maximale aantal instanties dat de bron aanbiedt.
- Entry-protocol: het proces roept *neer(semafoor)* aan. Hierbij zal het proces geblokkeerd worden als het maximale aantal instanties overschreden is (semafoor negatief).
- Exit-protocol: het proces roept *op(semafoor)* aan. Hierbij worden eventuele processen geblokkeerd in de *neer()* aanroep gedeblokkeerd.

Deze constructie, die garandeert dat slechts 1 thread zich in een bepaalde sectie bevindt, wordt soms een *turnstile* constructie genoemd (turnstile is Engels voor *draaihek*).

Alle moderne besturingssystemen implementeren semaforen volgens een gelijkaardig principe, met soms speciale features. Zo biedt Linux bijvoorbeeld specifieke aanroepen voor binaire semaforen (*spinlock()*, *spin_unlock()*, *spin_trylock()*), als tellende semaforen (kernel semaforen, *down()* en *up()*). Ook UNIX System V semaforen worden ondersteund, die als belangrijkste feature atomaire bewerkingen op *semaphore sets* toelaten. De POSIX instructies tenslotte bevatten ondersteuning voor *unnamed* semaforen (gebruik icm. pthreads, proces- of threadwide) en *named* semaforen (voor willekeurige processen, zijn dus persistent en system-wide).

Synchronisatie van processen kan ook gemakkelijk opgelost worden met behulp van semaforen.

Producent vs consument

Bij dit synchronisatieprobleem zijn er drie semaforen nodig om correcte verwerking te bekomen:

- Leeg: deze semafoor zorgt ervoor dat de producent enkel gegevens zal toevoegen als er nog plaats is. Ze wordt verlaagd telkens de producent een geproduceerd item in de buffer wil plaatsen, en verhoogd als de consument een item uit de buffer gehaald heeft.
- Gevuld: deze semafoor zorgt ervoor dat de consument nooit gegevens zal lezen als er geen meer aanwezig is. Ze wordt verlaagd als de consument een item uit de buffer wil halen, en verhoogd als de producent een item toegevoegd heeft.
- Mutex_buffer: deze binaire semafoor garandeert wederzijdse uitsluiting bij toegang tot de variabelen *in* en *uit*. Deze variabelen, eigen aan respectievelijk producenten en concumenten, zijn gedeeld en moeten dus beschermd worden indien er meerdere producenten en/of consumenten zijn.

Slapende kapper

Bij deze probleemstelling moet de situatie opgelost worden waarbij processen requests doen aan een servertoepassing die slechts 1 aanvraag tegelijk kan verwerken. De analogie beschrijft een kapperszaak, waarbij de kapper een enkele klant kan bedienen, en er een aantal stoelen beschikbaar zijn om te wachten. Indien de kapper slaapt moet die ook gewekt worden.

Er worden drie semaforen gebruikt om dit probleem te behandelen:

- Kapper: een binaire semafoor die de status van de kapper (slaapt of knipt) beschrijft. De kapper verhoogt ze als die een klant knipt, en de klanten verlagen de semafoor als ze geknipt willen worden. Daardoor zullen de klanten automatisch wachten tot de kapper vrij is.
- Klanten: een semafoor die indiceert hoeveel klanten er wachten. Ze wordt verhoogd als een klant geïnteresseerd is in een knipbeurt, en verlaagd door de kapper in het begin van zijn lus. Hierdoor zal de kapper wachten als er geen klanten aanwezig zijn, en gewekt worden als een klant geknipt wil worden.
- Mutex_aantal: een binaire semafoor die wederzijdse uitsluiting bij toegang tot de variabele "aantal" garandeert. Deze variabele indiceert het aantal wachtende klanten, omdat het niet eenvoudig is om de waarde van de semafoor Klanten uit te lezen.

Lezers en schrijvers

Dit probleem is vooral van toepassing op databases, waar simultane leesprocessen toegelaten zijn maar slechts 1 schrijfproces, tijdens dewelke er geen andere activiteit mag zijn. Daarvoor worden de volgende semaforen gebruikt:

- Db: deze semafoor bewaakt de toegang tot de database, hetzij door een enkel schrijfproces, hetzij door meerdere leesprocessen. Hiertoe verlaagt het schrijfproces deze semafoor en verlaagt het leesproces ze indien het aantal leesprocessen gelijk is aan 1 (eenmalige verhoging). Als het aantal leesprocessen op het einde van de leesactie nul is, verhoogt het leesproces de semafoor.
- Mutex_aantal: deze binaire semafoor wordt gebruikt als een mutex, en garandeert wederzijdse uitsluiting bij toegang tot de globale variabele aantal (die het aantal leesprocessenbijhoudt).

Door deze opzet hebben leesprocessen echter voorrang op schrijfprocessen. De optimalisatie die schrijvers voorrang geven, is echter ruim complexer.

Etende filisofen

Dit probleem beschrijft een analogie voor de situatie waar een aantal processen toegang willen tot ene beperkt aantal verschillende bronnen. Aangezien de bronnen verschillend zijn kan niet gewerkt worden met een semafoor met startwaarde groter dan nul.
Het probleem beschrijft de situatie waarin een aantal filisofen met evenveel vorken rond de tafel zitten, en elk twee vorken nodig heeft om te eten.

Indien elke filosoof de vork links van hem neemt, vervolgens die rechts van hem en start te eten, zal er een deadlock optreden. Deze lock kan vermeden worden door de linkervork terug te leggen indien de rechter niet beschikbaar is, maar de kans op een deadlock blijft indien de acties van de filosofen gesynchroniseerd optreden.

Door de sequentie waarbij een filosoof vorken controleert, opneemt, eet, en dan de vorken teruglegt te beschermen via een turnstile constructie wordt elk probleem verholpen. Het nadeel is echter dat slechts 1 filosoof tegelijkertijd deze sequentie kan uitvoeren, en dus ruim inefficiënt is.

Een derde oplossing stelt voor om alle filosofen zonder semafoor links een vork te laten nemen, maar 1 filosoof dwingen om eerst rechts te nemen. Hierdoor wordt de deadlock verholpen, maar er is een alternatief.

Een vierde oplossing werkt volledig symmetrisch (identieke code voor alle filosofen), wat te verkiezen is boven oplossing 3. Hierbij neemt een filosoof geen vorken op tenzij beide

beschikbaar zijn, en gebruikt daarvoor 1 globale mutex en 2 tabellen. Het algoritme werkt als volgt:

- Een globale mutex beschermt een deel van het algoritme:
    - In de globale array *toestand* geeft de filosoof te kennen dat hij wil eten.
    - * Vervolgens zal de filosoof zijn persoonlijke mutex in de globale array *vlag* verhogen indien geen van beide buren aan het eten is (en hij zelf geïnteresseerd is in te eten).
- Nu zal de filosoof zijn persoonlijke mutex verlagen. Dit zal enkel non-blocking zijn indien zijn buren niet eten, en de vorken beschikbaar zijn.
  Hierna gaat de filosoof eten, en veranderd zijn toestand daarbij ook van "geïnteresseerd" naar "bezig met eten".
- Het exit-protocol vindt ook plaats in door de mutex beschermde code:
    - De filosoof herstelt zijn toestand van "aan het eten" naar "idle".
    - Gebruik makende van de code uit * zal de filosoof beide buren controleren: als ze geïnteresseerd zijn om te eten en hun respectievelijke buren zijn niet aan het eten, zal hij hun persoonlijke mutex verhogen waardoor ze kunnen eten. Hierbij mag het voorkomen dat daarbij beide buren aan het eten gaan, daarvoor zijn er vorken genoeg.

Dit algoritme lost alle problematiek op, en dat met een volledig symmetrische code.

Monitors

Omdat het programmeren met semaforen te vereenvoudigen, biedt men soms monitors aan. Dit zijn high-level synchronisatiebouwstenen die een aantal procedures, datastructuren en variabelen groeperen tot een object. In objectgeoriënteerde programmeertalen kan dit bijvoorbeeld een speciaal type *class* zijn.

Een monitor lijkt sterk op een object uit objectgeoriënteerde talen: er kunnen procedures aan toegevoegd worden, maar alle datastructuren en variabelen binnen de monitor zijn enkel rechtstreeks benaderbaar van binnenuit de monitor (*private*). De hoofdeigenschap van een monitor is dat er slechts 1 thread tegelijk kan actief zijn binnen de monitor. Als een thread een procedure van de monitor aanroept, maar er blijkt al een andere thread actief te zijn, wordt de thread geblokkeerd en aan een wachtrij (de *entryqueue*) toegevoegd. Wanneer de monitor weer beschikbaar wordt, zulllen alle threads sequentieel in de wachtrij gedeblokkeerd worden.
Via dit mechanisme kan men **wederzijdse uitsluiting** garanderen, waarbij de kritieke sectie als procedure binnen de monitor geïmplementeerd worden.

Om **synchronisatie** te bekomen zijn er echter nieuwe structuren nodig, aangezien een thread niet gewoon actief kan wachten tot aan een bepaalde voorwaarde voldaan is omdat het daarbij alle andere threads die van de monitor gebruik maken ook zou blokkeren. Daarom voorziet men een aanroep waarbij de thread die een functie van de monitor aanroept zijn toestand kan controleren aan de hand van een conditievariabele (waarbij een enkele conditievariabele effect kan hebben op meerdere threads). Zo kan een proces zichzelf blokkeren indien er aan een bepaalde voorwaarde niet voldaan is door *wait()* uit te voeren op de bijhorende conditievariabele. Eveneens kan een bepaalde thread een andere groep threads wekken, door *signal()* of *notify()* op hun conditievariabele uit te voeren. Conditievariabelen zijn enkel toegankelijk van binnenuit de monitor.

Als een thread een geblokkeerde thread wekt door een signaal te sturen naar de desbetreffende conditievariabelen, kunnen er echter meerdere threads tegelijk in de monitor actief worden. Daartoe zijn er twee conventies:

- *Signal-and-exit* conventie van *Brinch Hansen*: hierbij wordt door de compiler afgedwongen dat de *signal()* aanroep de laatste is in de thread.

- *Signal-and-continue* conventie van *Hoare*: hierbij mag het proces dat een signaal uitvoert blijven doorwerken, en zal het gesignaleerde proces niet onmiddellijk geactiveerd worden maar met verhoogde prioriteit in de entry-queue geplaatst worden.

De *wait()*, *notify()* en *signal()* aanroepen zijn geen primitieven van het besturingssysteem, en worden meestal geimplementeerd met behulp van semaforen. Maar omdat de effectieve gelijktijdigheid afgehandeld wordt door de compiler, is de kans op fouten veel kleiner. De ondersteuning van monitors door programmeertalen en compilers is echter niet zo wijdverspreid, terwijl semaforen wel door de meeste besturingssystemen ondersteund worden.

Java

Java associeert een monitor (*lock*) per objecttype, die enkel beschouwd wordt indien de aangeroepen functie als *synchronised* gedeclareerd is. Indien een thread een synchronised functie aanroept moet de thread in bezit zijn van de lock, of de thread blokkeert en komt in de ongesorteerde *entryset* van het object terecht.

Een object geeft de lock vrij indien een thread een gesynchroniseerde procedure verlaat. Java kent echter ook conditievariabelen, maar in een gereduceerde vorm waar er 1 conditievariabele per thread toegewezen wordt. Een thread kan zo zichzelf blokkeren door *wait()* aan te roepen, waarbij hij in de *waitset* van het object geplaatst wordt en er 1 geblokkeerd proces uit de *waitset* naar de *entryset* verplaatst wordt.

De aanroepen *notify()* en *notifyall()* verplaatsen respectievelijk 1 of alle threads uit de *waitset* naar de *entryset*. De aanroepende thread wordt echter niet geblokkeerd, waardoor de thread eerst een *wait()* moet aanroepen of de gesynchroniseerde procedure verlaten vooraleer er effectief een procedure uit de *entryset* uitgevoerd wordt. Aangezien de gewekte threads geen informatie meekrijgen bij het wekken, moeten ze zelf controleren of aan de synchronisatievoorwaarde voldaan is. Meestal gebruikt men *notifyall()*, zodat de geblokkeerde threads zelf kunnen beslissen wie de beste kandidaat voor uitvoering is. Dit zorgt echter voor veel overhead, wat men soms het *denderende kudde / thundering herd* fenomeen noemt.

Liftalgoritme

Indien er meerdere processen wachten op een bepaalde conditievariabele, zal een systeemaanroep meestal het langst wachtende proces activeren. Dit geeft echter niet altijd het beste resultaat, en daartoe voorziet men de *wait()* waanroep soms met een extra prioriteitsparameter die gerespecteerd wordt bij het activeren van een thread. Daarbij wordt meestal de laagste prioriteiten als de belangrijkste geïnterpreteerd.

De introductie van prioriteiten vereenvoudigd tal van problemen. Een voorbeeld daarvan is het liftprobleem, waarbij men zo optimaal mogelijk een lift laat bewegen, daarbij rekening houdende met interne en externe oproepen. Daarbij moet aan de volgende voorwaarden voldaan worden:

- Zolang mogelijk in dezelfde richting bewegen.
- Voldoen aan alle voorwaarden, ongeacht type of volgorde van aanvraag.
- Bedieningszin pas omdraaien als er geen aanvragen meer zijn in een bepaalde richting.
- De beslissing om naar een verdiep te bewegen niet onderbreken.

Dit algoritme wordt best geïmplementeerd met behulp van een monitor. Deze monitor kent twee wachtrijen (interne conditievariabelen): *stijgend* en *dalend*. Het huidige doelniveau wordt intern opgeslaan in de variabele *niveau*. Externe en interne verzoeken

worden geïmplementeerd als een functie binnen de monitor (*aanvraag(int)*), en aan de hand van de huidige positie en richting wordt de aanvraag geblokkeerd in 1 van de twee wachtrijen:

- Indien de lift vrij is, wordt de variabele *niveau* ingesteld op het aangevraagde niveau.
- Indien de lift bezet is (en dus aan het bewegen is), zal de procedure vooraleer het niveau in te stellen een cyclus doorlopen waarbij het proces geblokkeerd kan worden:
    - Indien het gevraagde niveau zich onder het doelniveau bevindt, wordt het proces in de wachtrij *stijgend* geplaatst.
    - Als het niveau boven het doelniveau ligt, komt het proces in de wachtrij *dalend*.
    - Indien het niveau gelijk is aan het doelniveau, wordt het proces in de wachtrij geplaatst die de huidige beweegrichting indiceert.

Een tweede monitorprocedure, *activeer()*, wordt aangeroepen telkens de liftdeur sluit. Hierbij wordt afhankelijk van de beweegrichting een signaal gestuurt naar 1 van de twee wachtrijen, waardoor de lift zal bewegen naar het meest prioritair niveau in de wachtrij. Als de wachtrij leeg blijkt te zijn, word de beweegrichting omgedraaid en een doel uit de andere wachtrij geselecteerd.

Dit algoritme wordt teruggevonden bij schijfscheduling, waarbij het imperatief is dat men de zoektijd minimaliseert. Dit wordt bekomen door de bewegingen van de schijfkop zo efficiënt mogelijk te laten verlopen, waarvoor het liftalgoritme gebruikt wordt.

Doorgeven van berichten

In het geval van gedistribueerde systemen (of microkernel implementaties) voldoen semaforen niet bij gebrek aan gedeeld geheugen. Men kan dan wederzijdse uitsluiting bekomen door berichtuitwisseling tussen de processen, waartoe functies van het besturingssysteem gebruikt worden. Dit systeem is echter altijd trager dan semafooroperaties, aangezien het bericht vaak via vele intermediaire buffers en contextswitches naar een andere adresruimte overgebracht wordt.

Er bestaan verschillende strategieën bij het versturen van berichten:

- Een proces dat een bericht stuurt kan onmiddellijk verder werken. Een proces dat een bericht wil ontvangen, wordt geblokkeerd tot dit bericht aangekomen is.
- De zender wordt geblokkeerd na het sturen van een bericht, totdat de eveneens geblokkeerde ontvanger het bericht effectief ontvangen heeft.
  Met *rendez-vous* mechanisme kan men sterke synchronisatie bekomen, zoals het RPC mechanisme.
- Zender noch ontvanger worden geblokkeerd bij het sturen of ontvangen. Deze asynchrone aanpak maakt het voor de ontvanger mogelijk om snel weer verder te werken na het ontvangen van enkele berichten, zelf als er nog verzonden berichten te wachten staan om ontvangen te worden.
  Deze aanpak vergt echter dat het besturingssysteem een infrastructuur voor intermediaire buffers voorziet.

Ook de manier waarop de primitieven *send()* en *receive()* geïmplementeerd worden, kunnen verschillen:

- Directe adressering: *send()* primitieve vraagt de identificatiecode van het proces. Asymmetrische implementatie: *receive()* heeft geen argumenten nodig en ontvangt alle berichten. Symmetrische implementatie: ook *receive()* heeft als argument een identificatiecode nodig.
- Indirecte adressering: een mailbox wordt gebruikt als buffer waar meerdere processen berichten kunnen in plaatsen en uit halen. Eenvoudigere varianten (poorten) zijn mogelijk bij 1-op-1 relaties (met 1 ontvangend proces).

De meeste besturingssystemen gebruiken mailboxen, zo bijvoorbeeld is een Unix *pipe* een mailbox zonder begrenzing tussen de berichten. *Named pipes* en *message queues* kunnen gebruikt worden voor communicatie tussen processen zonder onderlinge verwantschap.

Doorgeven van berichten is volledig equivalent aan het gebruik van semaforen of monitoren. Het producent-consument probleem kan eenvoudig geïmplementeerd worden door geen buffer meer te gebruiken, maar de geproduceerde items te verzenden naar de consument. Door gebruik te maken van blocking *receive()* calls, zullen producent en consument wachten indien de wachtrij respectievelijk vol of leeg is. Om de producent te laten wachten bij een volle wachtrij, stuurt de consument bij opstarten N berichten waarvan de producent er 1 leest bij elke productie, en stuurt de consument er opnieuw 1 bij elke consumptie.

Deadlocks

Een deadlock is een situatie waarbij een verzameling processen geblokkeerd is en wacht op een bepaalde gebeurtenis. Die gebeurtenis kan echter enkel veroorzaakt worden door een bepaald proces in die verzameling, waardoor de situatie geblokkeerd is.
Meestal is de gebeurtenis in kwestie het vrijgeven van een bepaalde (vaak herbruikbare) bron. De bron is ook meestal niet-preemtabel, waardoor het niet van de momentele eigenaar kan afgenomen worden zonder het proces te doen mislukken.

Deadlocks treden enkel op als er aan 4 voorwaarden tegelijk voldaan is:

1. Wederzijdse uitsluiting: de bron kan maar door 1 proces tegelijk gebruikt worden.
2. Vasthouden en wachten: het proces houdt 1 voorziening vast terwijl het wacht op de toewijzing van een andere.
3. Geen preëmptieve verwijdering: enkel het proces zelf kan de toegang tot bron afstaan. Indien preëmptieve verwijdering wel ondersteund wordt, moet het proces een *rollback mechanisme* voorzien, waarbij in geval van verwijdering de toestand teruggebracht wordt naar het moment van aanvraag van de eerste bron.
4. Cyclisch wachten: er bestaat een cyclische keten van processen en bronnen (P1 heeft A nodig, terwijl P2 die vasthoudt. Tegelijkertijd heeft P2 B nodig, terwijl A die vasthoudt).

Er zijn enkele strategieën om met deadlocks om te gaan:

- Detectie en opheffing.
- Vermijden door zorgvuldige toewijzing (en weigering in geval van risico op deadlock).
- Onmogelijk maken door 1 van de 4 voorwaarden uit te sluiten.

Aangezien het uitsluiten van deadlocks veel overhead met zich meebrengt of te strikte beperkingen oplegt, oppert men meestal voor een andere mogelijkheid

- De kans op deadlocks minimaliseren

Dit is echter onaanvaardbaar voor missiekritieke of realtime besturingssystemen.

Toestandsdiagrammen en resourcegrafen

In een **toestandsdiagram** beschrijft elke knoop een toestand van het gehele systeem, waarin bronnen aan processen toegekend zijn en andere bronnen nog beschikbaar zijn. Een systeem wisselt van toestand door drie mogelijke acties van threads op bronnen:

- *request()*
- *acquire()* / *allocate()*
- *release()* / *deallocate()*

- *withdrawal()*, waarbij een thread afziet van de *request()* vooraleer deze toegekend is.
  Sommige besturingssystemen blokkeren een proces automatisch als een bron niet onmiddellijk beschikbaar is, waardoor de *withdrawal()* aanroep niet beschikbaar is.

Een toestandsdiagram in zijn geheel bevat elke mogelijke toestand van een systeem, en het pad dat een systeem volgt door een toestandsdiagram is afhankelijk van de processen.

Een deadlocksituatie kan men herkennen in het toestandsdiagram als een toestand waar enkel pijlen naartoe gaan, maar geen uit vertrekken. Ze kunnen meestal opgelost worden door een *withdrawal()* pijl terug te trekken naar de vorige toestand, maar het proces moet hiertoe ondersteuning bieden.

Als men een enkele toestand in detail wil beschrijven, stelt men een **resourcegraaf** op. Hierin worden alle processen en bronnen als knopen voorgesteld, respectievelijk cirkelvormig en vierkant. Een bron heeft 1 of meerdere contactpunten, die het aantal mogelijke processen indiceert die de bron aankan. Een pijl van bron naar proces wil zeggen dat een proces controle heeft over die bron. Een omgekeerde pijl indiceert dat een proces geblokkeerd is op een bron.
In geval van een deadlock zal de resourcegraaf van de knoop in kwestie altijd een circulaire set pijlen tonen.

Men kan aantonen dat deadlocks kunnen verholpen worden. Als N processen 2 uit N bronnen nodig hebben, bestaat er risico op een deadlock. Afhankelijk van hoe het besturingssysteem de threads schedulet, kan de deadlock echter verholpen worden. Aangezien het besturingssysteem dus een hand heeft in het optreden van deadlocks, zal men verschillende strategieën gebruiken om ze tegen te gaan.

Detectie

Hierbij worden applicaties nooit beperkt, en zal het probleem slechts aangepakt worden als het optreedt. Het is echter niet eenvoudig om te bepalen wanneer een deadlock optreedt. Aangezien het enkel kan optreden als er niet kan voldaan worden aan een toegangsrequest, zal er op die momenten een detectiemechanisme uitgevoerd worden.

Indien er van elke bron maar 1 element beschikbaar is, zal men uit de resourcegraaf een **wacht-op graaf** afleiden. Dit is de resourcegraaf met enkel processen, waarbij twee processen verbonden zijn indien 1 wacht op de andere (dwz. als er in de resourcegraaf een bron tussen ligt en de pijlen in 1 richting gaan). Indien er een cyclus voorkomt in een dergelijke graaf, komt er een deadlock voor en is elk proces in die cyclus geblokkeerd. Een algoritme om een cyclus in een graaf te detecteren staat beschreven op pagina 95.

Indien er meerdere elementen beschikbaar zijn van bepaalde bronnen kan de resourcegraaf niet omgezet worden naar een wacht-op graaf. Men gaat proberen de graaf te reduceren, waarbij herhalend al de pijlen van een bepaald proces verwijderd worden indien de uitgaande pijlen (die indiceren dat een proces wacht op een bron) om te draaien zijn (dwz. als de bron ook effectief beschikbaar is). Als de hele graaf zo te reduceren valt, treedt er geen deadlock op.
Numeriek lost men dit probleem als volgt op:

- Stel voor de bronnen twee vectoren op (elk element vertegenwoordigt een bron):
  - Een **aanwezigheidsvector** met het totale aantal instanties per bron.
  - Een **beschikbaarheidsvector** met het aantal beschikbare instanties.
- Stel voor de processen twee matrices op (elke rij vertegenwoordigt een proces, elk element daarvan met een bron):

    - Een **toewijzingsmatrix**, die het aantal toegewezen instanties van elke bron per proces indiceert.
    - Een **aanvraagmatrix**, die het aantal aangevraagde maar niet toegewezen instanties aanduidt.
- Herhaal vervolgens cyclisch:
    - Zoek een ongemarkeerd proces waarvoor de waarden van de aanvraagmatrix kleiner zijn dan die in de beschikbaarheidsmatrix. Dit indiceert dat het proces een aantal bronnen vereist, maar die eis in te willigen is en het proces dus uit te voeren is.
    - Markeer het proces als "uitvoerbaar"
    - Tel de toegewezen bronnen van het proces op bij de beschikbaarheidsvector (aangezien dit proces rechtstreeks uitvoerbaar is, waardoor de bronnen toegewezen aan dit proces onrechtstreeks bruikbaar zijn voor andere processen)
- Als na het uitvoeren van dit algoritme er nog ongemarkeerde processen aanwezig zijn, bevindt het systeem zich in een deadlock.

Opheffen

Repetitief controleren van het systeem veroorzaak veel overhead. In geval van een deadlock, moet men het systeem herstellen waarbij verliezen meestal niet te vermijden zijn. Er zijn een aantal opties:

- Ruwe maar veelgebruikte methode: processen in een deadlock 1 voor 1 abrupt afbreken, en de detectie steeds opnieuw uitvoeren. Soms wordt ook een proces buiten de deadlock afgesloten, die een bron bezit uit de deadlockcyclus.
- Een applicatie terugrollen (*rollback*) naar een eerder gemaakt checkpoint (bevat het volledige procesbeeld). Het risico bestaat dat de deadlock vervolgens opnieuw optreedt.
- Bronnen preëmptief beëindigen. Hierbij wordt een proces de toegang tot een bron ontzegd, en aan een ander proces toegekend. Of dit effectief mogelijk is hangt af van het type bron.

Resourcetrajecten en claimgrafen

Een **resourcetraject** is een manier om de evolutie van een systeem voor te stellen, waarin processen wedijveren voor een beperkt aantal bronnen. Men illustreert de dynamiek tussen twee processen door op een tweedimensionale figuur de ene as toe te wijzen aan de instructies van proces A, en de andere as voor proces B. Elke lijn uit de oorsprong indiceert dan een mogelijk pad dat het systeem uitvoert.
In de figuur arceert men gebieden waarin de processen beide bronnen tegelijk gebruiken. Dit zijn ontoegankelijke gebieden waarin het systeem zich niet kan bevinden. Een gebied dat zowel rechts van als boven zich een gearceerd gebied heeft, is dus een deadlocksituatie. Indien de scheduler deze gebieden markeert en het uitvoeringspad zodanig kiest, kunnen *deadlocks vermeden worden*.

Ter ontwikkeling van een algoritme wordt nog een type resourcegraaf ontwikkeld: de **claimgraaf**. Een claimgraaf bevat naast de standaard informatie uit een resourcegraaf, ook een verzameling potentiële aanvragen (streepjeslijnen). Naarmate het proces verder uitvoert worden potentiële aanvragen omgezet in werkelijke aanvragen, waarbij echter het totaal aantal connecties tussen proces en bron in een claimgraaf altijd constant blijft.

Vermijden

Bij het vermijden van deadlocks definieert men in de toestandsdiagrammen twee soorten toestanden:

- Onveilige toestanden: waarbij tenminste 1 van de mogelijke overgangen leidt tot een deadlocksituatie.
- Veilige toestanden: waarbij geen enkele overgang leidt tot een deadlocksituatie.

Onveilige toestanden leiden niet noodzakelijk tot een deadlocksituatie, maar ze maken het systeem onderhevig aan de willekeur van toevallige aanvragen. In een veilige situatie kan gegarandeerd worden dat alle processen kunnen afgewerkt worden.
Men partitioneert de toestanden ook in drie regio's, naargelang het veilige, onveilige of deadlocksituaties zijn (waarbij de regio met deadlocksituaties een deelverzameling is van de onveilige regio).

Een toestand is een veilige toestand indien de bijhorende claimgraaf volledig gereduceerd kan worden. Een veilige situatie kan, na toewijzing van een bron aan een proces, niet meer volledig reduceerbaar zijn waardoor duidelijk wordt dat men naar een onveilige situatie geëvolueerd is. Zie voorbeeld 3.44 pagina 101.

Dankzij deze principes kan men het systeem steeds in de veilige regio houden, door bij elke aanvraag fictief de aanvraag te volbrengen en de bijhorende claimgraaf te analyseren. Migreert men door de aanvraag naar de onveilige regio, dan wordt de aanvraag geweigerd, zelf al is de bron beschikbaar.
Dit conservatief algoritme heet het **bankiersalgoritme**, ontwikkeld door Dijkstra. Het wordt echter nooit globaal toegepast omdat het onmogelijk is om op voorhand te weten welke bronnen een proces zal nodig hebben. Processen kunnen ook lange tijd geblokkeerd worden, omdat een ander proces dezelfde bron ooit nodig zal hebben, al kan dit moment nog lang op zich laten wachten.

Om dit algoritme numeriek op te lossen kan men dezelfde datastructuren gebruiken al bij detectie van deadlocks. De aanvraagmatrix bevat nu echter ook de bronnen die een proces slechts later zal nodig hebben. Komt het algoritme tot een einde en zijn alle processen gemarkeerd, dan bevindt het systeem zich in een veilige situatie. Indien niet, bevindt het systeem zich in een onveilige situatie en zal het bij maximaal gebruik in een deadlock terecht komen.

Als een bron slechts 1 instantie kan toekennen wordt de situatie weer vereenvoudigd: men kan het vroeger aangehaalde detectiealgoritme voor cycli gebruiken om de claimgraaf te analyseren (waarbij een onveilige situatie bereikt wordt indien een toewijzing resulteert in een directionele cyclus in de graaf).
Als men echter in de niet-gerichte claimgraaf (de claimgraaf waarbij alle pijlen vervangen zijn door gewone lijnen) geen enkele cyclus vindt, zijn deadlocks in dit systeem volledig onmogelijk!

Onmogelijk maken

Door 1 van de vier voorwaarden onmogelijk te maken, kunnen deadlocks uitgesloten worden:

1. Wederzijdse uitsluiting: meestal is deze voorwaarde noodzakelijk om chaos te vermijden. Via intermediaire buffers (*spoolen* van de uitvoer) kan men echter de indruk opwekken dat dit niet mogelijk is: zo kunnen verschillende toepassingen schrijven naar een virtuele printer, terwijl enkel de printdaemon deze output correct doorsluist naar de fysieke printer.
2. Vasthouden en wachten: deadlocks kunnen uitgesloten worden door te vermijden dat processen die een bron in hun bezit hebben, wachten op andere bronnen. Dit kan men bekomen door te eisen dat een proces alle bronnen tegelijk aanvraagt, voor men aan uitvoering begint. Hierbij monopoliseren processen echter bronnen zonder ze te gebruiken, en moet elk proces op voorhand weten wat het zal gebruiken.
   Een alternatief stelt dat bij het aanvragen van een bron, alle toegewezen bronnen tijdelijk beschikbaar moeten gemaakt worden. Een variant hierop

(*tweefasen-lock-protocol*) wordt vaak gebruikt bij databases, waarbij het proces bij een transactie twee fasen doorloopt. Bij de *groeifase* overloopt het alle records die het nodig heeft, en zet ze 1 voor 1 op slot. Blijkt er een record reeds op slot te zijn, verwijdert het al zijn sloten en begint het opnieuw. Indien dit eens lukt, betreedt het proces de *krimpfase*, waarbij het veranderingen toepast en de sloten vervolgens verwijderd worden. Dit werkt enkel als het proces overal tijdens de eerste fase kan gestopt en herstart worden, wat niet van toepassing is op de meeste processen.

3. Preëmptieve verwijdering van een bron vastgehouden door een proces: dit is nog moeilijker te realiseren, aangezien het enkel toepasbaar is als de toestand van de bron makkelijk op te slaan is en later te herstellen valt. Dit is vb. niet het geval bij printers of tapedrives.
   Indien de bron bovendien toegewezen is aan een niet-geblokkeerd proces, zijn aanvullende procedures voor checkpoints en rollbacks vereist.
4. Cyclisch wachten: door bronnen globaal te nummeren en van processen te eisen dat hun aanvragen de volgorde van de nummering volgen, kan het cyclisch wachten doorbroken worden. Bovendien moeten meerdere instanties van dezelfde bron in 1 keer aangevraagd worden.
   extra info nodig Hierdoor kunnen de resourcegraaf en de wacht-op-graaf nooit cycli bevatten.
   Hoewel deze methode inefficiënt is (processen worden vertraagd en onnodig toegang tot bronnen ontzegd) is de ze enige die in de praktijk enigsinds haalbaar is en wordt daarom ook het meest gebruikt. De verantwoordelijkheid wordt echter bij het programma wordt gelegd, maar sommige besturingssystemen controleren of de aanvragen correct gebeuren aan de hand van een slotdiagnose module (*witness*).

Geheugenbeheer

Vereisten

Het geheugenbeheer staat oa. in voor:

- Verplaatsen van informatie tussen hoofdgeheugen en secundaire geheugen.
- Programma's inladen zodat ze kunnen uitgevoerd worden.

Het is 1 van de meest complexe taken, waarvoor hardwareondersteuning absoluut vereist is.

Elk geheugenbeheersysteem moet voldoen aan een aantal randvoorwaarden:

1. Zoveel mogelijk processen in het hoofdgeheugen geladen hebben, dit verhoogt de efficiëntie van de processor en van de I/O-voorzieningen. Omdat de hoeveelheid hoofdgeheugen beperkt is, zullen er altijd processen uitgeswapt worden naar het secundaire geheugen. Omdat dit een langzame bewerking is en proceswisselingen danig vertraagt, moet het geheugenbeheersysteem swapping zodanig organiseren dat de processor zoveel mogelijk bezig gehouden wordt.
2. Adresverwijzingen binnen programmacode moeten vertaald worden naar fysiek geheugen. Daarbij zal bij het terugswappen niet alleen de ingangen van het PCB, maar ook verwijzingen binnen het programma (hetzij naar eigen data, hetzij spronginstructies) moeten aangepast worden.
3. Beveiliging van het geheugen, zodat processen niet zonder toestemming geheugen van een ander proces kunnen lezen of schrijven. Ook het besturingssysteem en zijn datastructuren moet afgeschermd worden van gebruikersprocessen. Toch moet er flexibiliteit zijn om vb. processen die dezelfde code uitvoeren gegevensstructuren in het hoofdgeheugen te laten delen.
4. Het eendimensionale geheugen segmenteren zodat het geheugen efficiënt toegewezen wordt aan verschillende modules van programma's.

Partitionering

Het besturingssysteem heeft zelf ook geheugen nodig, meestal wordt hiervoor het uiterst lage of uiterst hoge gebied gebruikt aangezien de processor daar de interruptvector verwacht. Het overige geheugen moet verdeeld worden, waarvoor verschillende partitioneringsmethodes bestaan.

De eenvoudigste manier is **vaste partitionering**, waarbij het geheugen verdeeld wordt in gebieden met vaste begrenzing. Het geheugen wordt gepartitioneerd bij initialisatie (waarbij het aantal partities het aantal actieve processen limiteert), en het geheugenbeeld verandert achteraf niet meer. Deze implementatie is heel eenvoudig: het besturingssysteem moet enkel een tabel bijhouden die indiceert welke partities nog beschikbaar zijn.

Het geheugen wordt echter weinig efficiënt gebruikt. Aangezien elk proces een volledige partitie bezet en die zelden overeenkomt met zijn effectieve geheugenbehoefte, treedt er *interne fragmentatie* op (toegekend geheugen wordt niet gebruikt), vooral als er veel kleine processen actief zijn. Qua partitiegrootte zijn er twee mogelijkheden:

- Gelijke grootte: hierbij zijn alle partities even groot, en maakt het niet uit welke partitie aan een proces toegewezen wordt.
- Verschillende grootte: deze implementatie tracht de fragmentatie te verminderen door een partitie toe te wijzen die nauwer aansluit bij de grootte van het procesbeeld.
    - Wachtrij per partitie: hierbij wordt een proces aan een partitie toegewezen naargelang de grootte van het procesbeeld, en zal het proces vervolgens wachten tot de partitie beschikbaar wordt. Deze implementatie introduceert latenties, en ongebruikt geheugen.
    - Enkele wachtrij: hierbij zal de scheduler processen laten wachten in een enkele wachtrij, en telkens de kleinst beschikbare partitie selecteren die een proces kan bevatten. Deze implementatie is beter, hoewel de fragmentatie er door kan vergroten.

Soms is het procesbeeld zo groot dat het binnen geen enkele partitie past. Dan moet het proces zelf voor *overlay* mechanismen zorgen, waarbij enkel de instructies en gegevens die op dat moment nodig zijn zich binnen het hoofdgeheugen bevinden. Daartoe voert het proces zijn code uit binnen een *overlaydriver*, dewelke partiële geheugenbeelden naar behoefte inleest. Dit gebeurt volledig buiten het besturingssysteem om.

Dynamische partitionering lost deze beperkingen op, door te zorgen voor een variabel aantal partities met variabele grootte. Hierbij krijgt elk proces een dynamisch gealloceerde partitie, dat precies het procesbeeld kan omvatten. Processen worden dynamisch uitgeswapt om plaats te maken voor nieuwe processen. Aangrenzende vrije partities worden hierbij samengevoegd (*coalescing*). Op den duur treedt er echter *externe fragmentatie* op, waardoor er soms wel genoeg vrij geheugen kan aanwezig zijn, maar niet in een aaneengesloten blok.

Externe fragmentatie kan verholpen worden door *compactie*, waarbij het besturingssysteem alle processen verschuift zodat het vrije geheugen een aaneengesloten entiteit vormt. Samen met coalescing wordt dit soms *garbage collection* genoemd. Dit zijn echter processorintensieve taken.

Plaatsingsalgoritmen

Om frequente compactie te vermijden probeert het besturingssysteem de processen op een efficiënte manier in het geheugen te plaatsen. Hierbij bestaan verschillende algoritmen:

- Best-fit: hierbij wordt het blok gekozen dat zo nauw mogelijk aansluit bij de grootte van de gewenste partitie.
  Traag, en blijkt de slechtste resultaten op te leveren (sterke fragmentatie).
- First-fit: hierbij wordt het eerste blok gekozen dat de partitie kan omsluiten.
  Snelste methode, en blijkt ook best te werken. Zorgt uiteindelijk voor 33% onbruikbaar geheugen.
- Next-fit / Rotating-first-fit: hierbij wordt het eerste passende blok gekozen, sequentieel afgezocht vanaf het punt van de laatste plaatsing.
  Blijkt grote geheugenblokken op het einde van het geheugen te fragmenteren, waardoor compactie nodig wordt.
- Worst-fit: hierbij wordt het grootste blok gekozen, in de hoop grote gaten over te houden.
- Optimal-fit: complexere strategieën die dynamisch tussen de vier basisalgoritmen wisselen.

Onafhankelijk van de plaatsing bestaat het risico dat de processen in het hoofdgeheugen geblokkeerd zijn, waardoor er te weinig geheugen beschikbaar is om nieuwe processen te laden. Dit wordt verholpen met vervangingsalgoritmen.

Adresvertaling

Aangezien procesbeelden na verloop van tijd verschillende partities kunnen bezetten, of door compactie verplaatste worden, zijn de adressen waarnaar het programma verwijst niet vast. Daarom wordt er onderscheid gemaakt tussen:

- Logische adressen: dit zijn verwijzingen naar geheugenlocaties onafhankelijk van de huidige plaatsing van het proces, in het perspectief van het proces zelf. In systemen met partitionering kan men gebruik maken van *relatieve adressen*, adressen ten opzichte van een bepaald punt (meestal het begin van het programma). Vb. %esp adres, dat naar de stack van het programma verwijst.
- Fysieke of absolute adressen: verwijzen naar de werkelijke locaties in het hoofdgeheugen, in het perspectief van de processor en het besturingssysteem.

Vooraleer geheugentoegang mogelijk is, worden logische adressen vertaald naar fysieke adressen. Deze dynamische adresvertaling wordt uitgevoerd door een RISC processor op de processorkaart, de *memory management unit* (MMU), op het moment dat de instructie uitgevoerd wordt. Hiertoe wordt bij het inladen een *base register* en *bounds register* ingesteld, en zal bij het uitvoeren elk logisch adres vermeerderd worden met dat basisregister en vergeleken worden met het begrenzingsregister (om ev. een interrupt te genereren).

Administratie van geheugengebruik

Er zijn drie methodes om bij te houden welke geheugenpartities in gebruik zijn:

1. Bitmaps: een vector waarbij elk element een deel van het geheugen representeert (allocatie-eenheid). Een te grote allocatie-eenheid verhindert nauwkeurige allocaties, terwijl een te kleine allocatie-eenheid resulteert in een grote bitmap. Deze techniek wordt niet vaak gebruikt aangezien het opzoeken van een block vereist dat de hele bitmap doorlopen wordt voor een rij met N nullen te zoeken
2. Gekoppelde lijsten (Donald Knuth): hierbij zijn de geheugenblokken onderdeel van een lijst, en staat er op het einde van elk blok wat informatie over het block (status, grootte, ...). De vrije blokken worden zelf dubbel gelinkt, zodat het gemakkelijk is om de buren te controleren of ze ook vrij zijn.
   Deze implementatie blijkt de meest efficiënte te zijn.
3. Buddysysteem (Donald Knuth): dit algoritme teert op het binair verdelen van het geheugen. Als een proces een bepaalde hoeveelheid geheugen nodig heeft, zal

er een blok toegekend worden met als grootte de dichtstbijzijnde macht van 2. Bestaat er zo geen vrij blok, wordt een groter blok verdeeld in twee gelijke *buddy's* met die grootte (eventueel recursief, mits bovenlimiet, als het verschil in machten tussen het gevraagde en het beschikbare blok groter dan 1 is).
Het algoritme houdt een gekoppelde lijst bij per niveau van onderverdeling. Bij splitsing van blok wordt de ingang verwijderd, en een enkele van de budddy's aan een onderliggende lijst toegevoegd. Zo ook bij het vrijgeven wordt gekeken of de buddy van het vrijgekomen blok vrij is, en eventueel samengevoegd in de bovenliggende lijst.
Het mechanisme kent voor- en nadelen:

- Het versnelt het samenvoegen van blokken, aangezien de geheugenmanager slechts 1 lijst moet overlopen om te zien of het blok samenvoegbaar is (itt. alle gaten overlopen bij andere algoritmes). Daarom wordt dit mechanisme oa. gebruikt bij kernelstructuren van Unix.
- Het zorgt voor veel interne fragmentatie, aangezien blokgroottes moeten afgerond worden naar een macht van 2.

Paginering en segmentatie

Vaste partitionering (interne fragmentatie) en dynamische partitionering (externe fragmentatie) hebben gemeen dat het geheugen als aaneengesloten blokken wordt toegewezen. Daarom worden alternatieven aangeboden die willekeurige stukken vrij geheugen kan benutten.

Paginering

Paginering is equivalent aan vaste partitionering, waarbij men het hoofdgeheugen in *frames* verdeelt (stukken met gelijke grootte) en alle procesbeelden in *pagina's* indeelt ter grootte van frames. Het besturingssysteem houdt per proces een paginatabel bij, die de framelocatie van elke pagina bevat.
In tegenstelling tot vaste partitionering zijn frames echter veel kleiner, en hoeven de frames niet aaneengesloten te zijn. Externe fragmentatie is hierdoor afwezig, en de interne fragmentatie wordt beperkt tot een fractie van de laatste pagina.

Bij paginering bestaat een logisch adres uit een paginanummer, en een relatieve positie binnen die pagina. Aangezien men voor de grootte van pagina's steeds een macht van twee neemt (N), kan men een logisch adres blijven opvatten als een relatief adres ten opzichte van het begin van het programma waarbij de laatste N bits de positie binnen de pagina indiceren en de overige bits het paginanummer betekenen.
Door de grootte als een macht van 2 te nemen vereenvoudigt de dynamische adresvertaling ook: de MMU moet gewoon het paginanummer substitueren door het framenummer.

Segmentatie

Hierbij streeft men de doelstellingen van dynamische partitionering na, waarbij de grootte van de geheugenverdelingen variabel is. Daartoe wordt de adresruimte van het programma in blokken verdeeld met dynamische grootte, die via de segmenttabel gekoppeld zijn met een overeenkomstig segment van het hoofdgeheugen.
In tegenstelling tot dynamische partitionering moeten die segmenten niet aaneengesloten zijn, waardoor interne fragmentatie afwezig is en externe fragmentatie (*checkerboarding*) geminimaliseerd wordt.

Door de variabele segmentgrootte is de vertaling van logische adressen complexer. Een logisch adres bevat nog altijd een segmentnummer, en de relatieve locatie binnen dat

segment. Tijdens de adresvertaling haalt de MMU de locatie en lengte van het segment uit de processegmenttabel, en berekent het fysieke adres door de relatieve locatie op te tellen bij de fysieke locatie van het segment. Indien de relatieve locatie de lengte van het segment overschrijdt, wordt een interrupt gegenereerd.

Terwijl paginering verborgen blijft voor de programmeur, wordt segmentatie bewust zichtbaar gemaakt. Hiermee kan de modulariteit verhoogd worden, door individuele modules van het programma als individuele gegevensstructuren te implementeren. Die modules kunnen dan onafhankelijk van elkaar worden gewijzigd en gecompileerd. De geheugenbehoefte van een proces kan zo logisch behandeld worden.

Segmenten kunnen dynamisch aangemaakt worden, vb. een heap & stack per object. Ook kan men een segment tijdens uitvoering vergroten, wat de flexibiliteit enorm verhoogt. Indien men vb. voor een tabel een apart segment reserveert, kan die eenvoudig vergroot of verkleind worden. Dit in tegenstelling tot vb. paginering, waarbij de tabel een vaste grootte had en omringd werd door andere datastructuren, mogelijk verspreid over verschillende pagina's.

Virtueel geheugen

Paginering en segmentatie kennen twee grote verschillen tegenover partitionering:

- Geheugenverwijzingen zijn logische adressen. Na vb. uitgeswapt te zijn kan eenzelfde logisch adres dus verschillende gebieden bezet hebben.
- Procesbeeld wordt opgebroken en niet-aaneengesloten gebieden.

Men eiste echter altijd dat het hele procesbeeld in het hoofdgeheugen geladen was. Dit kan versoepeld worden via paginering of segmentatie, door slechts enkele pagina's of segmenten in het hoofdgeheugen (*reëel geheugen*) te laden (de *residente set* van het proces). De rest van het programma bevindt zich dan in het *virtueel geheugen*, en wordt dynamisch ingeladen.

Indien bij het uitvoeren van een instructie de processor een geheugenadres tegenkomt dat zich niet in de residente set van het proces bevindt, wordt een interrupt gegenereerd die een geheugentoegangsfout aangeeft (*paginafout* of *page fault*). Het besturingssysteem reageert hierop door het correcte segment uit het virtuele geheugen in het hoofdgeheugen te laden:

- Het besturingssysteem plaatst een I/O-request die het geheugen moet inlezen
- Terwijl de I/O afgehandeld wordt, krijgt een ander proces rekentijd
- Als het geheugen ingelezen is, wordt een I/O-interrupt gegenereerd die de controle weer aan het besturingssysteem overlaat.
- Het besturingssysteem werkt de paginatabellen bij, en plaatst het proces weer in de *gereed* wachtrij.

Het gebruik van virtueel geheugen biedt voor- en nadelen:

- Ruimte voor meer processen in het geheugen. De kans is daardoor groter dat tenminste 1 proces zich in de toestand *gereed* bevindt.
- Geheugenhoeveelheid per proces wordt nu enkel beperkt door de schijfgrootte, en niet door de hoeveelheid hoofdgeheugen. Technieken zoals *overlaying* worden nu ook overbodig.
- Inladen van processen is efficiënter, omdat soms maar enkele stukken effectief gebruikt worden vooraleer het proces weer geblokkeerd wordt.
- Het inladen van virtueel geheugen creëert veel overhead: het besturingssysteem moet tweemaal de controle overnemen (pagefault, I/O-interrupt) en een ander stuk uitswappen.

Indien de frequentie van pagefaults te hoog is, treedt *trashing* op: de processor besteedt nu meer tijd aan het swappen van stukken dan aan het uitvoeren van instructies. Om de situatie te analyseren worden twee parameters beschouwd:

- L: de gemiddelde tijd tussen 2 paginafouten
- S: de gemiddelde tijd nodig om een pagina te vervangen

Indien L>>S wordt de schijf waarop geswapt wordt onderbenut. Indien L<<S zijn er meer paginafouten dan het besturingssysteem en I/O-systeem aankan.
Daarom is het belangrijk dat het besturingssysteem bij het inladen van een proces goed kan inschatten welke stukken wel en niet gebruikt zullen worden. Gelukkig hoopt geheugen zich op in clusters (beginsel van lokaliteit), waardoor de veronderstelling dat slechts enkele stukken van een proces nodig zijn in elke tijdsperiode in de praktijk bevestigd wordt. Recente evoluties promoten echter het gebruik van vele kleine programma's (objectgeoriënteerde technieken) waardoor de lokaliteit van verwijzingen binnen een proces vermindert.

Virtueel geheugen met paginering

De paginatabel - steeds in het hoofdgeheugen, tenzij heel het proces uitgeswapt is - ziet er als volgt uit:

- Framenummer/framelocatie (beide zijn direct gerelateerd).
- Present-flag: duidt aan of de pagina aanwezig is in het hoofdgeheugen.
- Modified-flag: duidt aan of de pagina gewijzigd is sinds het laatst in het hoofdgeheugen geladen is. Dit om te controleren of de pagina weggeschreven moet worden bij het uitswappen.
- Andere besturingsbits (ter bescherming, of voor sharing op paginaniveau). Vb. welke processormodus benodigd is om de pagina te lezen.

Het schijfadres waar de pagina bewaard wordt, staat in tabellen van het besturingssysteem. De paginatabel bevat enkel data voor het vertalen van een logisch naar een fysiek adres.

Voor virtueel geheugen met paginering is er hardwareondersteuning vereist, in de vorm van twee processorregistes:

- Page Table Base Register (PTBR): beginadres van de paginatabel voor het proces.
- Page Table Length Register (PTLR): lengte van de paginatabel.

Het besturingssysteem vult deze registers in uit de processortoestandsinformatie in het PCB blok. De MMU gebruikt het paginanummer dan als offset voor de PTBR om het framenummer op te zoeken. Indien de P-bit een 0 bevat wordt een page-fault gegenereerd.

De keuze van de paginagrootte is belangrijk. Kleine pagina's verminderen interne fragmentatie maar zorgen voor grote paginatabellen, terwijl grote pagina's zorgen voor een efficiënte blokoverdracht. Ook het aantal paginafouten wordt beïnvloed door de paginagrootte:

- Bij kleine pagina's kan er een groot aantal pagina's per proces geladen zijn en verlaagt het aantal paginafouten, bij grote pagina's daalt het aantal toegewezen pagina's en vergroot de paginafoutfrequentie.
- Gemiddeld blijkt dat er 200 instructies uit een pagina uitgevoerd worden vooraleer een nieuwe pagina opgehaald wordt.

Deze twee opmerkingen pleiten voor relatief kleine pagina's.

Structuur van paginatabellen

Meestal is de aanspreekbare virtuele adresruimte ruim groter dan het reëel geheugen (vb. Windows NT: de totale 32-bit adresruimte, mits *nonpaged pool* gereserveerd voor het besturingssysteem). Hierdoor worden echter de paginatabellen per proces te groot om in het hoofdgeheugen te laten. Hiervoor bestaan enkele oplossingen.

Paginatabellen opslaan in virtueel geheugen. Hiervoor gebruikt het systeem het

forward-mapped page table mechanisme, waarbij twee niveaus aan paginatabellen gebruikt worden. Het paginanummer uit een logisch adres wordt nu gesplitst in een deel als index voor de buitenste (steeds residente) paginatabel, en een deel als index voor de binnenste tabel die verwijst naar een fysiek frame. Zo kan een request twee paginafouten opleveren.
Sommige systemen bieden meer dan twee niveaus, waarbij het aantal bits per niveau soms instelbaar is. Voor de 64-bit adresruimte moet men echter op zoek naar andere oplossingen.

Geïnverteerde paginatabel: een globale paginatabel die de per-proces paginatabel vervangt. De tabel bevat een ingang per geheugenframe, waardoor de grootte van de tabel vast is en afhankelijk van de hoeveelheid geheugen. Daardoor is de tabel gesorteerd op framenummer, en zou het opzoeken van een paginanummer door sequentieel aflopen van de tabel lang duren. Het alternatief:

- Een hashfunctie zet de *Address Space Itendifier* (ASID, combinatie van het paginanummer en PID) om naar een kleinere waarde. Dit reduceert het aantal items dat sequentieel zal moeten worden afgelopen.
- Die waarde wordt gebruikt als index in een hashtabel, en levert een verwijzing op naar een item van de paginatabel. Hoe groter deze hashtabel, hoe kleiner de gekoppelde lijsten uit de volgende stap. Dikwijls heeft deze hashtabel evenveel ingangen als de paginatabel.
- Elk item in de paginatabel bevat het paginanummer, de index in de paginatabel en een verwijzing naar een volgend element (circulaire gekoppelde lijst).
- Eenmaal het paginanummer gevonden is, kan de index van het element gecombineerd worden met het beginadres van de paginatabel om zo het frameadres te berekenen. Hierdoor moet het frameadres niet expliciet worden opgeslaan.

Dit alternatief biedt een snellere oplossing, maar heeft als nadeel dat gedeeld geheugen niet eenvoudig te implementeren is aangezien elk paginanummer overeenkomt met een enkel frame.

De vorige implementaties zorgen echter steeds voor twee geheugentoegangen bij verwijzingen naar een virtueel adres. Daarom implementeren de meeste MMU's een ***translation lookaside buffer*** (TLB) die als cache dient voor de recent gebruikte paginatabelingangen. Hoewel de tabel meestal klein is (max 1024 ingangen) ken het dankzij het lokaliteitsprincipe meer dan 90% van de aanvragen opvangen.
Wegens het klein aantal elementen kan men in een TLB moeilijk via *direct mapping* opslaan (indexering, het paginanummer wordt gebruikt als index), en zal men *associative mapping* gebruiken (elke ingang in de tabel bevat het paginanummer) met hardware die parallel kan zoeken.

Een vertaling van een virtueel adres verloopt nu als volgt:

- De TLB wordt overlopen om te zien of er geen entry is voor het paginanummer. Is die aanwezig, wordt die onmiddellijk teruggegeven aan de MMU.
- Indien niet, wordt het paginanummer gebruikt als index in de procespaginatabel die zich bevindt op het adres gespecificeerd door de PTBR.
- Als de pagina niet residentieel is (P bit is 0), wordt een paginafout gegenereerd, die de pagina laadt en de paginatabel bijwerkt.
- Tenslotte vormt de MMU het reëel adres aan de hand van het frameadres en de offset waarde.

Bij een opzoeking in geval dat de TLB een *cache miss* gaf, wordt een waarde in de TLB vervangen door het resultaat. Sommige TLB's ondersteunen persistente ingangen voor kernelcode. Andere TLB's kunnen ingangen voor verschillende processen bevatten door elke ingang te identificeren met de ASID. Is dit niet geval moet de TLB geleegd worden bij elke proceswisseling. Door de kernel op te nemen in de adresruimte van elk proces wordt zo vermeden dat de TLB moet gewist worden bij elke contextswitch.

Virtueel geheugen met segmentatie

Net zoals bij paginering bevat elk proces een paginatabel, die flags P en M bevat. Bijkomend is er nu een veld die de lengte van het segment indiceert. De processor bevat een *Segment Table Base Register* (STBR) die het begin van de segmenttabel indiceert. De adresvertaling van een virtueel adres ziet er als volgt uit:

- Het segmentnummer (linker bits van het virtueel adres) wijst na optelling met de STBR naar een item in de paginatabel waaruit het frameadres gehaald wordt.
- Indien het segment niet residentieel is of de offset groter is dan de framelengte, wordt een interrupt gegenereerd3
- Het reëel adres wordt berekend door de offset op te tellen bij het frameadres.

Illegale geheugentoegang wordt voorkomen door de beschermingsbits in de segmenttabel te controleren.

Virtueel geheugen met paginering en segmentatie

Segmentatie vereenvoudigt afhandeling van groeiende structuren, terwijl paginering handig is door zijn transparantie voor de programmeur en eliminatie van externe fragmentatie. Om deze voordelen te combineren, ondersteunt moderne hardware en software virtueel geheugen met zowel paginering als segmentatie. Hierbij wordt het proces onderverdeeld in een aantal segmenten (gekozen door de programmeur) die zelf verdeeld worden in pagina's met vaste grootte.
Zo wordt het geheugenbeeld anders ervaren naargelang het standpunt:

- Volgens het proces: gesegmenteerd geheugenbeeld.
- Volgend het systeem: gepagineerd geheugen met frames van vaste grootte.

Een logisch adres bestaat nu uit:

- Segmentnummer (*selector*)
- Segmentpositie
    - Paginanummer
    - Paginapositie

Elk proces heeft een segmenttabel (*descriptor table*):

- *Segment base pointer*
- Segmentlengte
- Controlebits (*sharing*, *protection*)

Maar elk segment heeft nu ook een paginatabel:

- Framenummer
- P & M bit

Een adresvertaling gebeurt nu als volgt:

- Het segmentnummer wordt gebruikt als index (opgeteld bij het *segment table base register*) om in de segmenttabel van het proces de *page table pointer* op te halen.
- Het framenummer gebruikt als index (opgeteld bij de *page table pointer*) om in de paginatabel van het segment het framenummer op te halen.
- Aan de hand van het framenummer wordt een frameadres opgehaald, die na optelling bij de offset uit het logische adres het reële adres vormt.

Om de combinatie van segment- en paginanummers efficiënt te vertalen in een framenummer, worden soms TLB's geïmplementeerd.

Niet alle pagina's van een segment moeten zich in het hoofdgeheugen bevinden, waardoor het mogelijk blijft om geheugen per frame toe te wijzen.
Er zijn echter veel varianten mogelijk op dit mechanisme. Zo zet de Pentium architectuur een 48-bit logisch adres eerst via een segmenttabel om naar een 32-bits lineair adres, dat vervolgens geïnterpreteerd wordt in een tweelaags pagineersysteem van elk 1024 elementen. Zo kan elk proces tot 2^13 segmenten aanspreken, terwijl er nog 2^13 gedeelde segmenten mogelijk zijn. Om compatibiliteit met de 80286 te onderhouden ondersteunt de Pentium eveneens een systeem met zuivere segmentatie (waarbij het

lineair adres onmiddellijk wordt geïnterpreteerd als een fysiek adres).

Besturingssystemen zoals Linux maken zo weinig mogelijk gebruik van segmentatie om zoveel mogelijk hardware te ondersteunen:

- 1 segment voor kernelcode.
- 1 segment voor kernelgegevens.
- 1 segment voor gebruikerscode: gedeeld tussen processen.
- 1 segment voor gebruikersdata: gedeeld tussen processen.
- Een *Task State Segment* (TSS): individueel per proces, gebruikt om tijdens proceswisselingen hardwarecontext op te slaan.
- Een *Local Descriptor Table* (LDT): standaard 1 globaal segment, maar elk proces kan een LDT aanmaken.

Door minimaal gebruik van segmenten kan de segmenttabel steeds in processorregisters opgeslaan worden. Linux gebruikt tot 3 niveaus voor de paginatabel van elk segment. Op platformen die maar 2 niveaus ondersteunen (vb. Pentium) wordt het middelste niveau uitgeschakeld.

Strategieën voor geheugenbeheer

Omdat de meeste systemen hetzij zuivere paginering implementeren (Windows NT), hetzij segmentatie combineren met paginering (Linux) focust men zich om de efficiëntie van pagineringmechanismen te verbeteren door het aantal paginafouten te minimaliseren.

Ophaalstrategieën

Er zijn twee mogelijke strategieën om te bepalen wanneer een pagina moet worden overgebracht naar het hoofdgeheugen:

- Vraagpaginering (*paging on demand*): hierbij wordt een pagina pas overgebracht als er om gevraagd wordt. Dit leidt meestal tot een groot aantal paginafouten bij het opstarten van een proces, totdat alle benodigde pagina's in het geheugen geladen zijn.
  Unix implementeert deze strategie.
- Prepaginering (*prepaging* of *prefetching*): hierbij wordt bij het opstarten van een proces meerdere aaneengesloten pagina's binnengehaald zoals ze opgeslagen zijn op het secundaire medium, ook al is er maar 1 vereist. Dit kan een voordeel opleveren, maar soms worden veel van de pagina's niet gebruikt.
  Windows NT implementeert deze strategie (*clustering*), en laat het aantal pagina's afhangen van de hoeveelheid geheugen en de aard van de pagina.

Plaatsingsalgoritmen

Men moet ook bepalen waar in het hoofdgeheugen de pagina's geladen worden, en volgt hierbij de reeds besproken strategieën (*best-fit*, *next-fit*, *first-fit*, *worst-fit*).

Vaste of variabele toewijzing

Dit bepaald hoeveel paginaframes er worden toegewezen aan elk actief proces. Als een proces een pagina opvraagt maar boven deze limiet gaat, wordt een andere pagina uitgeswapt.
Kiest men een kleine hoeveelheid frames:

- Grote hoeveelheid processen in het hoofdgeheugen.
- Veel kans dat er tenminste 1 proces in de *gereed* wachtrij staat.
- Meer kans op paginafouten indien té klein.

Kiest men een grote hoeveelheid frames:

- Minder paginafouten.
- Geen merkbare verbetering indien té groot (overtoewijzing).

Daarom zijn er twee mogelijke strategieën voor paginatoewijzingen:
- Vaste toewijzing: een proces krijgt een vast aantal pagina's toegekend bij creatie.
- Variabele toewijzing: het aantal toegewezen paginaframes kan veranderen om zo het aantal paginafouten te verminderen. Dit monitoren zorgt echter voor overhead, en kan er voor zorgen dat een proces heel verschillend kan presteren door externe omstandigheden. Toch leidt het tot een betere performantie en is dus gebruikelijker.

Vooral in systemen met vaste toewijzing (maar ook in systemen met variabele toewijzing) maakt men gebruik van de *copy-on-write* techniek om vb. bij de *fork()* call het kindproces de pagina's van het ouderproces te laten gebruiken, wat het aanmaken van een kind sterk versnelt. Pas als 1 van de processen probeert te schrijven naar die gedeelde pagina, zal een individuele kopie gemaakt worden.

Lokale of globale vervanging

De vervangingsstrategie bepaalt of op het moment dat een pagina moet geladen worden, enkel lokale pagina's (die toehoren tot het proces die aanvraagt) mogen vervangen worden, of globaal pagina's mogen vervangen worden.
Lokale benaderingen hebben als voordeel dat een *trashing* proces (meer tijd in paginafouten dan instructies) geen frames kan afnemen van andere processen waardoor die ook zouden kunnen trashen.
Bij vaste toewijzing wordt enkel lokale strategieën gebruikt, bij variabele toewijzing zijn beide mogelijk.

Variabele toewijzing + globale vervanging: eenvoudigst te implementeren, omdat zo tegelijkertijd de variabele toewijzing kan geregeld worden.
Hierbovenop wordt nog gebruik gemaakt van een paginabuffering-techniek:
- Het besturingssysteem houdt extra lijsten van frames bij:
  - Lijst met vrije frames: bevat frames die vrijgegeven zijn.
  - Lijst met vrije frames en gewijzigde pagina's: bevat frames die vrijgegeven zijn, maar wijzigingen bevatten die nog naar het secundaire geheugen moeten weggeschreven worden (*dirty-bit*).
- Als een pagina wordt uitgeswapt, wordt die nog niet onmiddellijk weggeschreven maar in 1 van de twee intermediaire lijsten toegevoegd.
- Wanneer een pagina wordt ingelezen, wordt het frame aan het begin van 1 van beide lijsten genomen.
  Pas dan wordt dat frame effectief vernietigd. Dit heeft als voordeel dat als er naar een pas verwijderde pagina gerefereerd wordt, die kan hersteld worden zonder extra I/O.

Windows NT past deze paginabuffering ook toe, hoewel het variabele toewijzing combineert met een lokale vervangingsstrategie.

Variabele toewijzing + lokale vervanging: de pagina moet nu gekozen worden uit de residente set van het aanvragende proces. Het is daarvoor belangrijk om zowel de toekomstige behoeften als de ideale grootte van de residente set te kunnen inschatten. Hiervoor wordt soms (vb. Windows NT) een werksetbenadering toegepast:
- De werkset op het moment t is de set unieke pagina's waar een proces naar verwezen heeft in het laatste tijdsinterval d (venstergrootte).
- De werksetgrootte varieert met de tijd, en kent meestal stabiele periodes afgewisseld door overgangsperiodes.

- Treedt er een paginafout op, dan is deze set aan het veranderen. Periodiek worden pagina's die niet meer in de werkset voorkomen uit de residente set verwijderd.
- Een proces werkt slechts efficiënt als de residente set de werkset kan bevatten, zoniet treden er veel paginafouten op.
  Indien de som van werksetgroottes van de processen het totaal aantal frames overschrijdt, treedt er trashing op. Het systeem verwijdert dan best enkele processen volledig uit het geheugen.

De werksetbenadering kan zo het aantal actieve programma's en hun geheugengebruik beheren, en trashing voorkomen.
Het meten van de werkset vraagt echter veel overhead, en de werksetgrootte wordt daarom vaak vervangen door de paginafoutfrequentie. Daalt deze onder een bepaalde drempel, kunnen er frames uit de residente set verwijderd worden. Wordt ze echter te groot, heeft het proces meer frames nodig.
Windows NT past een andere heuristiek toe om de werksetgrootte te bepalen: in geval van een paginafout zal de werksetgrootte met 1 toenemen, tot een bepaalde bovengrens. Frames zullen enkel vervangen worden indien er dan nog paginafouten optreden. Indien de lijst met vrije frames te klein wordt, zal de *Working Set Manager* processen de opdracht geven pagina's vrij te geven en hun werksetgrootte daarbij aan te passen.

Wegschrijfstrategieën

Deze strategie bepaalt wanner het nodig is een gewijzigde pagina weg te schrijven naar het secundaire geheugen.

- Vraagopschoning (*cleaning-on-demand*): hierbij wordt een gewijzigde pagina slechts weggeschreven als die geselecteerd is voor vervanging.
- Opschoning vooral (*precleaning*): hierbij worden gewijzigde pagina's in batches weggeschreven, zelf al zijn ze nog niet aan vervanging toe. De kans bestaat echter dat de pagina's later opnieuw gaan moeten weggeschreven worden.
- Compromis gebaseerd op paginabuffering: enkel de pagina's in de intermediaire lijst van frames met gewijzigde pagina's wordt periodiek weggeschreven. De *Working Set Manager* van Windows NT voert dit vb. uit van zodra er meer dan 300 pagina's in die lijst zit.

Vervangingsstrategieën

Hierbij wordt bepaald welke pagina's er moeten vervangen worden. Het doel hierbij is dat de kans dat de pagina in de nabije toekomst weer nodig zal zijn, minimaal is. Het toekomstig gedrag wordt voorspeld aan de hand van het recente verleden.

Optimale strategie: de pagina vervangen waarvoor de tijdsduur tot volgende vervanging het langst is. Dit is een louter theoretisch model, dat dient ter vergelijking van andere modellen.

De ***last recently used*** (LRU) strategie vervangt de pagina waarnaar het langst niet verwezen is, en behoudt recent gebruikte pagina's omdat die volgens het lokaliteitsprincipe wellicht opnieuw zullen gebruikt worden.
Deze strategie is een vrij goede benadering voor de optimale, maar presteert extreem slecht in cyclische verwijzingspatronen waarbij het paginabereik groter is dan de hoeveelheid geheugen (en produceert daarbij een vaste stroom aan paginafouten).
Het is ook moeilijk te implementeren, aangezien er een speciale datastructuur zou voor nodig zijn die bij elke paginatoegang gewijzigd zou moeten worden, wat enorme vertragingen oplevert. Het kan versneld worden door hardwarematig voor een systeem met N paginaframes een NxN matrix bij te houden.

- Initieel zijn alle elementen 0.

- Bij frametoegang wordt in rij X alles op 1 ingesteld, en in kolom X alles op 0.
- Op elk moment indiceert de binaire waarde van een rij hoe recent de pagina gebruikt is, met de laagste waarde als minst recent gebruikte pagina.

Dit valt echter niet toe te passen tussen het secundaire geheugen en het hoofdgeheugen, wegens te grote hardware-eisen. Op hoger gelegen lagen zou het echter wel realistisch zijn.

Bij gebrek aan hardwarematige LRU implementatie moet men het algoritme softwarematig simuleren. Licht gewijzigd doet men dat met de ***not frequently used*** (NFU) strategie. Hierbij wordt aan elke pagina een softwareteller toegewezen, waarvan bij elke geheugenverwijzing de meest linkse bit op 1 wordt gezet. Bij elke kloktik (meestal een veelvoud ervan) worden de bits van de tellers naar rechts opgeschoven, wat een deling door 2 inhoudt (geheugenverwijzingen in een vorig interval krijgen exponentieel dalend gewicht). Bij elke paginafout wordt de pagina verwijderd waarvan de teller het kleinst is.

Bij de strategie ***first in, first out*** (FIFO) worden de toegewezen frames voorgesteld als een circulaire buffer, waarbij de frames die zich het langst in de buffer bevinden verwijderd worden. Dit is de meest eenvoudige strategie, die veronderstelt dat nieuwe pagina's frequenter gebruikt worden. De tijd die een pagina in het geheugen zit is echter geen goede indicatie, en faalt vb. bij frequent aangeroepen routines. FIFO presteert globaal dus zeer slecht, zelf slechter dan LRU die wel detecteert dat sommige pagina's hoewel lang geleden gealloceerd, frequent gebruikt worden.

De **klokstrategie** (of **tweede-kans-algoritme**) koppelt een *use*-bit aan elk frame, opgeslaan in de paginatabel. Dit valt te interpreteren als een NFU-algoritme met een 1-bit teller. De *use*-but wordt op 1 ingesteld als een pagina geladen wordt, of als er naar verwezen wordt (liefst hardwarematig).
Er wordt een circulaire buffer bijgehouden, geïmplementeerd zoals het FIFO-model. Als er een frame moet vervangen worden, zal het besturingssysteem de lijst doorlopen (met een pointer naar een frame als wijzer), maar frames negeren met de *use*-bit op 1. Het systeem verandert de waarde van de bit echter naar 0, zodat in het geval dat geen enkel frame geschikt was een tweede *pass* wel een geschikte kandidaat zal opleveren.
Deze strategie werkt dus zoals de FIFO strategie, met verschil dat het recent geadresseerde frames overslaat. Zo worden de prestaties van LRU/NFU benaderd, en dit met weinig overhead.
Om de prestatie te verbeteren worden soms meerdere *use*-bits gebruikt (Linux). Soms betrekt men ook de *dirty*-bit bij het algoritme (Mac OS), om frames te vinden die niet recent gebruikt zijn en ook niet gewijzigd zijn sinds paginering (om zo I/O te vermijden). Een tweede *pass* negeert de dirty-bit, en verlaagt de *use*-bit zoals normaal (waardoor dit algoritme het derde-kans-algoritme heet).

Het **klokalgoritme in Unix** is verfijnd, door enerzijds geheugentoewijzingen voor de kernel te implementeren via een niet-virtueel buddysysteem (aangezien de kernel veelvuldig datastructuren<<paginagrootte gebruikt), en anderzijds de paginering voor gebruikersprocessen te laten uitvoeren door een specifiek proces (*pagedaemon*, proces 2). Die houdt een dubbelgekoppelde lijst met vrije paginaframes bij, waaruit er in geval van een paginafout 1 verwijderd wordt. Periodiek onderzoekt het proces of er genoeg vrije pagina's zijn (> systeemparameter *lotsfree*), en begint indien nodig pagina's uit het geheugen naar de schijf over te brengen. De vervangingsstrategie daarbij is globaal, en het aantal frames per proces is variabel.
Aangezien *passes* van het klokalgoritme lang blijken te duren, worden twee wijzers gebruikt. De voorste wijzer zet de *use*-bit op 0, terwijl de achterste wijzer pagina's met *use*-bit op 0 voor verwijdering selecteert. De hoek tussen de wijzers is constant maar configureerbaar:

- 360°: het algoritme werkt zoals het oorspronkelijke klokalgoritme
- ~0°: enkel héél frequent gebruikte pagina's zullen worden behouden

Als er te vaak gepagineerd wordt, activeert het systeem de *swapperdaemon* (proces 0) die 1 of meer processenvolledig zal uitswappen.

Windows NT gebruikt dit ook het klokalgoritme, maar enkel op uniprocessor systemen. Op multiprocessorsystemen met TLB wordt gewoon FIFO toegepast. Alle geheugenbeheer wordt uitgevoerd door een zestal systeemthreads (verschillende prioriteit) binnen de *virtual memory manager* module van de executive, die door verschillende gebeurtenissen kunnen geactiveerd worden:

- Op tijdsbasis.
- Door de scheduler (bij hoge paginafoutfrequentie).
- Indien de lijst met vrije frames te klein wordt.

Ook volledige processen kunnen uitgeswapt worden. Van elke thread die een aantal seconden (3 tot 7) inactief is, wordt de stack uitgepagineerd. Geldt dit voor alle threads van het proces, wordt het volledige procesbeeld geswapt.