Embedded Systems Engineering

Operating Systems

Jos Onokiewicz, Marc van der Sluys

Klassen: ES2, ES2D
5 februari 2017
versie 2.7

Hogeschool van Arnhem en Nijmegen

Embedded Systems Engineering

Documenthistorie

	Datum	Wie	Wijzigingen
...			…
2.0		Jos Rouland	Uitgevoerd door Jos Onokiewicz
2.1	19-08-2013	Jos Onokiewicz	Lay-out verbeterd, geschikter voor schermpresentatie. Aandacht voor OS-9 verwijderd. Race-conditie begrip toegevoegd. Index toegevoegd
2.2	24-01-2014	Jos Onokiewicz	Enkele tekstcorrecties uitgevoerd. Reentrant en threadsafe functies onderwerp toegevoegd.
2.3	10-07-2014	Jos Onokiewicz	Lay-out aangepast, meer lege regels. SCHED_DEADLINE omschrijving toegevoegd.
2.4	29-08-2015	Jos Onokiewicz	Meer aandacht voor data races. SHR conform POSIX toegevoegd. Condition variables toegevoegd. Kleine verbeteringen in teksten.
2.5			
2.6	13-08-2016 28-11-2016	Marc van der Sluys Jos Onokiewicz	Diverse aanpassingen en toevoegingen, duidelijker indeling en tekst. H11 verwijderd, H14 naar dictaat real-time systemen. Enkele layoutaanpassingen.
2.7	05-02-2017	Marc van der Sluys	§5.4 opgesplitst in §5.4 en §5.5. §6.2 en 6.3 verwisseld.

Inhoud

1 Inleiding

De eerste versies van dit dictaat zijn gemaakt door Jos Rouland.

Dit dictaat bevat het eerste gedeelte van het studiemateriaal voor de onderwijseenheid ‘Operating Systems’ (OPS). Aan bod komt het **GNU/Linux** operating system of korter aangeduid **Linux**. Linux past men toe voor desktopsystemen en servers. Embedded Linux wordt zeer veel gebruikt voor embedded applicaties. Na dit eerste dictaat worden achtereenvolgens de dictaten 'Linux device drivers' en 'Real-time scheduling' behandeld. Bij de voorbeelden en het practicum is uitgegaan van Linux v2.6, maar deze werken ook in de huidige versies van Linux (>= v4).

Voor het practicum wordt de practicumhandleiding ‘Operating systems’ gebruikt. Bij het practicum worden de command line en man pages gebruikt. Het is aan te raden voor het practicum het artikel *Efficient use of the Linux command line in the Bash shell* te bestuderen.

Hoofdstuk 2 geeft een overzicht en de kenmerken van een aantal operating systems. Hoofdstuk 3 gaat in op het begrip real-time. In hoofdstuk 4 wordt de lagenstructuur van een operating system besproken. Hoofdstuk 5 gaat in op de toestanden waarin een taak zich kan bevinden en wordt aandacht besteed aan multitasking, multithreading en diverse methoden van scheduling. Hoofdstuk 6 gaat dieper in op intertaakcommunicatie met behulp van command line parameters, multitasking en multithreading. De hoofdstukken 7, 8 en 9 behandelen intertaakcommunicatie met behulp van pipes, shared memory en signals. In hoofdstuk 10 worden semaforen besproken. Hoofdstuk 11 gaat in op het al dan niet het optreden van deadlocks. Tot slot komen in hoofdstuk 12 de diverse methoden van geheugenbeheer aan de orde

Bijlage 1 laat een simulatie van een scheduler zien. Bijlage 2 geeft een voorbeeld om een zombie proces te maken. Bijlage 3 gaat in op de exit status.

In dit dictaat wordt de Hongaarse notatie voor de namen van C-variabelen gebruikt. Nieuwe toegevoegde codevoorbeelden zijn niet meer met Hongaarse notatie uitgevoerd.

2 Overzicht en kenmerken van operating systems

Dit hoofdstuk bevat een omschrijving van de belangrijkste functies van een operating system. Er wordt ingegaan op essentiële aspecten van real-time systemen, waarin (real-time) multitasking operating systems worden toegepast. De begrippen multitasking en multiprocessing komen beide aan de orde, maar de aandacht gaat uit naar multitasking operating systems.

2.1 Functie van een besturingssysteem

Indien we door een computersysteem programma's willen laten uitvoeren, is een **operating system** (besturingssysteem) een onmisbaar stuk software. Een operating system realiseert globaal de volgende twee functies:

- het beheren van het gemeenschappelijke gebruik van een beperkt aantal **resources** (hulpbronnen)
- het realiseren van een **virtuele machine** die de onderliggende hardware 'verbergt'

2.1.1 Gemeenschappelijk gebruik van resources

Een computer wordt in zeer veel gevallen door meer dan één gebruiker of proces gelijktijdig gebruikt. Een programma dat in het werkgeheugen van de computer is geladen en wordt uitgevoerd door de processor, wordt aangeduid met de naam **taak**, **task**, **proces** of **process**. De hardware moet dit samen met het operating system mogelijk maken.

Een operating system verricht een groot aantal 'huishoudelijke taken'. Het is een **resource manager**. Het operating system heeft de taak zo efficiënt mogelijk de resources aan de gebruikers en processen ter beschikking te stellen. Denk hierbij onder andere aan een hoeveelheid intern of extern geheugen, een randapparaat (bijvoorbeeld een printer of een harde schijf) of een bepaalde hoeveelheid CPU-tijd. Een belangrijk probleem vormt de beperkte beschikbaarheid van resources, zodat het operating system de resources moet toekennen op basis van bepaalde criteria.

Een operating system voert onder andere de volgende taken uit:

- **processor management**: het toekennen van processortijd aan de verschillende processen (programma's) die gelijktijdig in het interne geheugen aanwezig zijn. Dit komt bij het onderwerp multitasking uitgebreid aan de orde.
- **memory management**: het beheer van het interne geheugen, het bepalen waar programma's (en data) worden geladen en uitgevoerd
- **file management**: het beheer van het externe geheugen, het mogelijk maken dat allerlei gegevens in bestanden, op b.v. harde schijf en memory sticks, kunnen worden weggeschreven en uitgelezen
- **I/O management**: het laten uitvoeren van I/O door programma's, het toekennen van allerlei randapparatuur (printers en plotters)

- **protectie management**: meerdere gebruikers de mogelijkheid bieden om gebruik te maken van dezelfde computer zonder dat de verschillende processen elkaar nadelig kunnen beïnvloeden. Indien bijvoorbeeld een proces crasht, moeten andere processen gewoon kunnen doorgaan.
- **error management**: de foutstatus van de computer aangeven en eventueel adequate maatregelen uitvoeren

Computersystemen benaderen we als gebruiker meestal via een **terminal** (beeldscherm met toetsenbord). Een terminal is een zelfstandig systeem dat meestal via een seriële verbinding gekoppeld is aan ons computersysteem. Vanaf een terminal kunnen we de computer opdrachten geven met behulp van commando's en allerlei symbolische aanduidingen van resources. Een operating system stelt een gebruiker in staat programma's te laten uitvoeren.

2.1.2 Realisatie van een virtuele machine

Het operating system transformeert de 'kale' hardware naar een “machine” die veel gemakkelijker te hanteren is dan de eigenlijke machine (= de hardware) zelf. Een operating system pakt de hardware eigenlijk in en vormt een **softwarelaag** rond de hardware. Dit betekent dat we bij het schrijven van een programma voor een computer niet rechtstreeks de hardware benaderen, maar softwarefuncties van het aanwezige operating system aanroepen. Deze aanroepen van softwarefuncties van een operating system worden **system calls** genoemd.

Een operating systeem creëert hiermee een abstractere machine die makkelijker te programmeren of te gebruiken is. We hoeven immers niet stil te staan bij de interne hardware-details, waardoor allerlei mogelijke programmeerfouten worden voorkomen. We noemen een dergelijke abstractere machine een **virtuele machine**. Indien de interne hardware zou veranderen, kunnen we het operating system aan de buitenkant hetzelfde laten. Dit betekent dat de namen van system calls en hun bijbehorende parameterlijsten hetzelfde blijven. Uiteraard moeten de implementaties van de system calls aangepast worden. Dit wordt bijna altijd door de leverancier van het operating system gedaan. Applicatiesoftware kan **hardware-onafhankelijk** ontwikkeld worden, omdat een operating system een standaardbenadering van de interne hardware van computersystemen kan realiseren.

2.1.3 Verschillende soorten operating systems

De aard van de virtuele machine wordt bepaald door de toepassing. Het ontwerp van een operating system wordt sterk bepaald door het soort gebruik waarvoor de machine bestemd is. De meest eenvoudige operating systems zijn in staat om één gebruiker één programma tegelijkertijd te laten uitvoeren. In dit geval noemen we een dergelijk systeem wel **single-user/single-tasking** (bv. Palm OS). Is een systeem geschikt voor één gebruiker die meerdere taken gelijktijdig kan laten uitvoeren, dan spreekt met van **single-user/multitasking** (bv. Windows 7 en Mac OS X). Multitasking wordt ook wel **multiprogramming** genoemd.

Bij een **multi-user** operating system (bv. Unix) kunnen meerdere gebruikers tegelijkertijd taken laten uitvoeren.

Een **multiprocessing** operating system ondersteunt het gebruik van meerdere processoren binnen één computer. Hierbij worden de taken over de verschillende processoren verdeeld. Voorbeelden van zulke operating systems zijn Unix-varianten, Mac OS X en Windows Embedded.

Voor zakelijke doeleinden zijn operating systems ontworpen die gespecialiseerd zijn op **transaction processing** (bv. AIX). Hier is sprake van het veelvuldig raadplegen en muteren van (omvangrijke) databases. Voorbeelden zijn: banksystemen (bv. PayPal) en vliegtuigreserveringssystemen (bv. Amadeus, Galileo, Sabre, Worldspan).

AIX	International Business Machines
Mac OS X	Apple
Windows 8	Microsoft Corporation
UNIX	AT&T Bell Laboratories
GNU/Linux	The Linux Foundation

Fig. 2.1 Voorbeelden van operating systems voor zakelijke toepassingen

In het volgende figuur zien we een aantal voorbeelden die gebruikt worden voor (real-time) embedded systemen en voor mobiele applicaties in smartphones.

Windows Embedded CE	Microsoft Corporation
Keil RTX	Keil, ARM and Cortex-M devices
VxWorks	Windriver Systems Sytems
µC/OS-II	Micrium
QNX	QNX Software Systems
FreeBSD	University of California, Berkeley
FreeRTOS	Real Time Engineers Ltd.
Embedded Linux	The Linux Foundation
RTAI	Hard real-time Linux extension
Xenomai	Hard real-time Linux extension
LynxOS RTOS	LynuxWorks
Android	Google (smart phones)
iOS	Apple (smart phones)

Fig. 2.2 Voorbeelden van operating systems voor embedded toepassingen

Een **embedded operating system** is bedoeld voor embedded systems. Ze zijn zeer compact en werken uiterst efficiënt. Voor het besturen van processen moet het operating system de hardware in staat stellen binnen een gegarandeerde tijdsduur op externe signalen te reageren. Deze externe signalen kunnen op willekeurige momenten en in willekeurige volgorde optreden. De computer verwerkt signalen van sensoren en bepaalt de signalen naar de actuatoren. Operating systems die dit real-time

gedrag mogelijk maken, noemt men **real-time operating system**s (bv. QNX, VxWorks, RTLinux, Windows CE).

2.2 Ontbreken van een standaard voor begrippen

In de wereld van de operating systems wordt een groot aantal begrippen gebruikt. Er is echter geen sprake van uitgebreide standaardisatie. Dit betekent dat er verschillende aanduidingen kunnen bestaan voor eenzelfde begrip en dat eenzelfde aanduiding per systeem verschillende betekenissen kan hebben.

2.3 Communicerende processen

Voor de toepassing van computers ten behoeve van de besturing van industriële processen kunnen we twee subsystemen onderscheiden:

- het **te besturen systeem**
- het **besturende systeem**

We gaan er nu verder vanuit dat het besturende systeem een programmeerbaar computersysteem is, dat via interfaces gekoppeld met sensoren en actuatoren het te besturen systeem zinvol beïnvloedt. In het te besturen industriële proces gebeuren allerlei zaken op willekeurige momenten en de volgorde van gebeurtenissen ligt in veel gevallen niet vast. Tevens kan men stellen dat allerlei (deel)processen zich parallel min of meer onafhankelijk van elkaar voltrekken. Dit laatste geeft aanleiding tot de behoefte dat ook in het besturende systeem de verschillende besturingsprocessen of taken parallel moeten kunnen worden uitgevoerd.

De hoofdbesturingstaak die het besturende systeem moet uitvoeren is op te splitsen in een aantal deelbesturingstaken die sequentieel en/of parallel moeten worden uitgevoerd. De verschillende taken of processen hebben een min of meer zelfstandige functie. Er bestaat echter wel een bepaalde relatie tussen de verschillende taken zodat er sprake moet zijn van uitwisseling van data en besturingssignalen tussen deze taken. Er moet dus gecommuniceerd en gesynchroniseerd worden. We hebben te maken met **communicerende processen**.

De realisatie van een deelbesturingstaak kan worden uitgevoerd door een hardwarecombinatie, bestaande uit een processor, geheugen en I/O-componenten. Een dergelijke combinatie duiden we kortweg aan als **processing module**. Het in het geheugen aanwezige computerprogramma bepaalt de relatie tussen input en output van een processing module.

Afhankelijk van de gestelde eisen kunnen we een hoofdbesturingstaak dus uitvoeren met verschillende processing modules die met elkaar moeten kunnen communiceren. Indien we kiezen voor een oplossing met verschillende processoren die met elkaar communiceren, dan spreken we van **multiprocessing**. Het is echter voor een groot aantal toepassingen niet noodzakelijk om te kiezen voor een oplossing met meerdere processoren.

Het alternatief voor deze parallelle oplossing is een nabootsing van dit parallellisme door een processing module die in het geheugen meerdere programma's (processen of taken) heeft staan. De processor voert om de beurt een deel van een van de programma's uit. Bijvoorbeeld na elke 0,01 sec komt het volgende programma gedurende 0,01 sec aan de beurt; het volgende programma gaat verder waar het bij de vorige keer is gebleven. Dit geeft de suggestie van parallellisme; men noemt dit ook wel **quasi-parallellisme**.

Een processor die zijn tijd cyclisch verdeelt over verschillende programma's wordt een **multitasking** computersysteem genoemd. In dit laatste geval hebben we te maken met een aantal **virtuele processing modules**. In de volgende paragraaf gaat hier dieper op in. Om dit te realiseren kunnen we gebruik maken van een operating system.

2.4 Multitasking versus multiprocessing

In de voorafgaande paragraaf hebben we gezien dat een groot aantal taken moet worden uitgevoerd. Dit moet in veel gevallen voor een belangrijk deel parallel geschieden. Onderlinge communicatie is essentieel. Een volgende stap is gericht op de realisatie van een systeem. We moeten de verschillende taken gaan afbeelden op hardware en software. In het algemeen is dit een computersysteem met één of meer processoren (of cores) en eventueel een operating system.

We kunnen groepen van deeltaken bij elkaar groeperen. Een dergelijke groep van deeltaken of activiteiten wordt door bepaalde hardware en/of software uitgevoerd. Deze verschillende taken kunnen we op twee manieren laten uitvoeren en wel op basis van:

- **multiprocessing**
  door fysiek verschillende processoren met elk eigen (lokaal) geheugen en/of gemeenschappelijk (globaal) geheugen. De processoren kunnen met elkaar communiceren.

  De verschillende processoren kunnen bijvoorbeeld met elkaar communiceren via gemeenschappelijk geheugen (shared memory) of via een computernetwerk. In het eerste geval spreekt men van **tightly-coupled systemen**, in het tweede geval van **loosely-coupled systemen**. Gemeenschappelijk geheugen kan men bijvoorbeeld realiseren door gebruik te maken van dual ported RAM dat door twee processoren benaderd kan worden.

- **multitasking**
  door één processor die zijn tijd over de verschillende taken verdeelt. Ook hier moeten de taken met elkaar kunnen communiceren (data uitwisseling). Men spreekt dan van intertaakcommunicatie (inter-process communication, **IPC**).

Bij multitasking spreekt men ook wel over **concurrency in software** en bij multiprocessing over **concurrency in hardware**. Het zal duidelijk zijn dat een operating system en de bijbehorende programmeertalen uitgebreid dienen te zijn met functies die

deze concurrency (parallellisme) en communicatie tussen taken mogelijk maken. De compiler voor de gebruikte programmeertaal moet voorzien zijn van een bibliotheek van system calls voor het gebruikte operating system.

2.4.1 Scheduling in multiprocessing systems

In een multitaskingsysteem zal er software nodig zijn die bepaalt welke runnable taak processortijd krijgt. Deze software wordt de **scheduler** genoemd. Een **runnable** taak is een taak die direct kan worden uitgevoerd, met andere woorden hij hoeft niet te wachten op een of andere event, bv. een melding dat een I/O-operatie is uitgevoerd. Een **running** taak is een taak die bezig is uitgevoerd te worden en die dus processortijd heeft. In het ideale geval is elke runnable taak altijd running. Als er in een systeem meer runnable taken zijn dan processoren, kunnen niet alle runnable taken running zijn. Een aantal taken zal moeten wachten en de scheduler moet bepalen welke processen running zijn.

We beperken onze aandacht tot multitasking systemen met één processor. Alle taken krijgen om beurten gedurende een bepaalde korte tijd de processor toegewezen. De tijd die een taak krijgt toegewezen wordt een **time slice** genoemd. Deze duurt meestal 10 tot 100 ms. Dit herhaalt zich cyclisch. Het lijkt net of de taken parallel worden uitgevoerd: **quasi-parallel processing**. Het onderbreken van de uitvoering van een taak gebeurt niet door de taak zelf, maar wordt veroorzaakt door een **periodieke timer interrupt**. Na deze interrupt wordt er een interrupt service routine gestart die de overgang naar de volgende taak regelt. Deze routine wordt de **scheduler** genoemd. De scheduler wordt ook wel aangeduid als de **dispatcher**.

Het volgende voorbeeld geeft het voordeel aan van een multitasking systeem. Stel dat we een aantal dingen willen laten uitvoeren, zoals het maken van een financiële berekening, het sorteren van gegevens en het laten printen van een tekstfile. Op een single-tasking systeem moeten we de verschillende opdrachten aan de computer één voor één achter elkaar laten uitvoeren. Als alle opdrachten zijn uitgevoerd, kunnen we pas starten met bijvoorbeeld het editen van een tekst.

In een multitasking omgeving laten we de editor op de voorgrond (interactief met de gebruiker) uitvoeren. De andere taken worden quasi-parallel op de achtergrond uitgevoerd op het moment dat we een tekst aan het editen zijn.

De werking van een multitasking systeem waarin drie taken quasi-parallel worden uitgevoerd, kan vereenvoudigd worden weergegeven als aangegeven in fig. 2.3. De hardware die uitgevoerd is met één processor, gedraagt zich schijnbaar als een systeem met drie gelijke, zelfstandige processoren. Het operating system zorgt ervoor dat de hardware zich gedraagt als drie virtuele processing modules die parallel functioneren. De uitvoering van de scheduler voor elke taakovergang kost uiteraard enige tijd. Dit is niet aangegeven in fig. 2.3. Schedulingtijd vormt dus overhead in een multitasking systeem. Als we de time slice tijd kleiner maken dan neemt de overhead van de schedulingtijd dus toe.

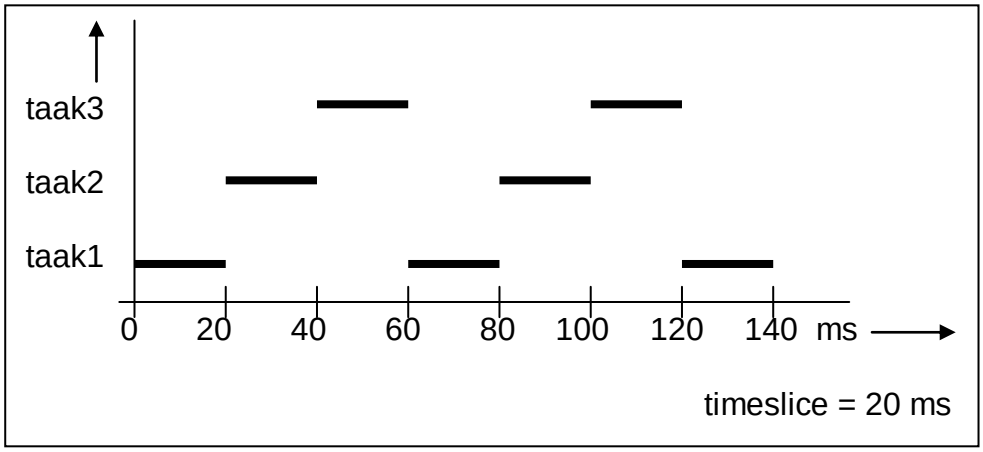


Fig. 2.3 Verdeling van de processortijd over drie taken

Multitasking kan het ook mogelijk maken dat meerdere gebruikers met elk een eigen terminal gelijktijdig **interactief gebruik** kunnen maken van een computersysteem met één processor. Men spreekt dan van een **timesharing system**. Bij interactief gebruik beoogt men een zo kort mogelijke responstijd te verkrijgen. Een minimale responstijd is echter niet gegarandeerd.

2.5 Programma versus taak

Met **programma** wordt bedoeld de binaire code en bijbehorende (initialisatie)data voor een microcontroller die in een file op disk staat. Een programma is dus een statisch begrip.

Een programma dat in het werkgeheugen van de computer is geladen en wordt uitgevoerd door de processor, wordt aangeduid als **taak**, **task**, **proces** of **process**. Een proces is dynamisch. Het voert activiteiten uit met zijn omgeving, waardoor het voortdurend data wijzigt en er acties worden uitgevoerd door randapparatuur (zie ook §5.2). Een zelfde programma kan dus meerdere keren naast elkaar opgestart worden. Dit betekent dat bij één programma dus meerdere taken kunnen horen.

3 Real-time systemen en real-time operating systems

Computersystemen spelen een belangrijke rol in het besturen van allerlei processen. In veel gevallen worden dergelijke systemen ingebouwd bij het te besturen proces. Besturingssignalen moeten op tijd beschikbaar zijn, wil het te besturen proces in goede orde verlopen. We noemen dergelijke applicaties **embedded real-time besturingen**. De microprocessor of microcontroller met de bijbehorende software en interfaces naar sensoren en actuatoren is ingebed in een groter systeem en heeft daarin een specialistische functie. Enkele voorbeelden zijn inbraakalarmsysteem, kopieermachine, automanagementsysteem en een zelfstandig werkende onderzoekrobot op de planeet Mars.

Real-time besturingen spelen ook een belangrijke rol in toepassingen op het terrein van bijvoorbeeld de **mechatronica**. Mechatronica houdt zich bezig met technische systemen waarin toepassingen van software, sensoren en actuatoren, elektronica, mechanica en meet- en regeltechniek een rol spelen. Dit hoofdstuk gaat verder in op real-time operating systems.

3.1 Het begrip real-time

Een **real-time operating system** (**RTOS**) moet ervoor zorgdragen dat een computersysteem **binnen een gegarandeerde maximale tijd** kan reageren op **events** die op willekeurige momenten en in een willekeurige volgorde kunnen plaatsvinden in de buitenwereld (asynchronous events). Ook signalen met een periodiek karakter (sampling van sensorsignalen) kunnen een rol spelen. Er is dus niet alleen meer sprake van logische condities die de werking van het systeem bepalen, maar er zijn nu ook beperkende timingseisen (temporele eisen) **time constraints**.

Een systeem waarvoor het niet halen van een **deadline** fataal is, wordt een **hard real-time system** genoemd. De kosten die optreden bij het niet halen van een deadline, zijn onacceptabel hoog. Denk hierbij aan bijvoorbeeld besturingen van chemische procesinstallaties of nucleaire en militaire systemen. Indien er niet sprake is van een fatale fout bij het overschrijden van de deadline, dan spreekt men van een **soft real-time system**. We kunnen hier dus stellen dat de kosten bij het niet halen van een deadline acceptabel zijn.

Het halen van deadlines stelt bijzondere eisen aan een operating systeem als we real-time eisen willen garanderen. Een real-time operating system moet ervoor zorgen dat een programma zich voor real-time besturingen voorspelbaar (**predictable**) kan gedragen. Bij gegeven inputcondities moet een outputconditie altijd vóór de geeiste deadline bereikt zijn. Het omgaan met hardware interrupts, veroorzaakt door events, speelt hierbij een zeer belangrijke rol. De exacte **voorspelbaarheid** van het gedrag is dus een belangrijk kenmerk van real-time systemen.

Bij het ontwerp van real-time systemen gaan we uit van kennis van de karakteristieken van de omgeving waarin het systeem moet worden toegepast (domeinkennis).

Hiervoor dienen **events** nauwkeurig omschreven te worden in een zogenaamde **event list**.

Een event voldoet aan de volgende eisen:

- een event treedt **instantaan** op en niet gedurende een periode
- het computersysteem dient op een van tevoren nauw omschreven manier te reageren: een **eenduidige** (beschrijving van de) **respons**

De events die niet door het besturende systeem maar alleen door het te besturen systeem kunnen worden gegenereerd, worden **external events** genoemd. Deze external events komen meestal overeen met hardware interrupts die aan het besturende systeem via sensoren worden aangeboden.

De literatuur geeft een mogelijke omschrijving van een **time constraint**. Een time constraint bestaat uit een vijftal elementen:

R de naam van de uit te voeren **r**espons die behoort bij de genoemde event
b de event en het bijbehorende tijdstip die bepalen waarna de uitvoering van de genoemde respons mag beginnen (**b**egin-time constraint)
c de vereiste beperking aan de (maximale) executietijd van de genoemde respons (**c**omputation time)
f de (maximale) **f**requentie waarmee de respons herhaald moet kunnen worden
d de **d**eadline (tijdstip) waarvóór de executie van de respons moet zijn beëindigd

Bij het aandachtspunt 'de maximale frequentie' moet men niet alleen denken aan strikt periodieke events en bijbehorende responses. Het is een meer algemene aanduiding hoe snel een zelfde event achter elkaar kan optreden zonder dat er sprake hoeft te zijn van een periodiek verschijnsel, maar eerder van een onregelmatig verschijnsel. We zouden het beter als volgt kunnen aangeven: de minimale tijdsduur die kan liggen tussen het optreden van twee dezelfde events.

Voor een uitvoerbare (= werkende) oplossing moet het volgende gelden voor alle responses R_i in het betreffende systeem:

$$\mathbf{c_i < d_i - b_i < 1/f_i}$$

Een **gegarandeerde maximale responstijd** op events is dus kenmerkend voor een real-time computersysteem. Na het optreden van een event moet de respons op de event vóór een bepaalde **deadline** afgerond zijn. De duur van de respons speelt dus ook een belangrijke rol (zie ook fig. 3.1).

Real-time systemen zijn dus niet per definitie snel! De gegarandeerde maximale responstijd kan best in de ordegrootte liggen van minuten tot zelfs uren. Voor industriële besturingen ligt de maximale responstijd doorgaans tussen de 1-1000 ms. In fig. 3.2 wordt een aantal voorbeelden van real-timesystemen genoemd.

e: plaatsvinden van event
b: begin-time constraint
d: deadline

responses R1, R3 en R4 voldoen niet aan real-time eisen
response R2 voldoet aan real-time eisen

Fig. 3.1 Real-time respons

In veel gevallen zal b gelijk zijn aan het moment van optreden van e. Maar bij het besturen van bepaalde processen kan b (veel) dichter bij d liggen.

Het ontwikkelen van real-time systemen vereist een goede domeinkennis (kennis van het toepassingsgebied) zodat het vaak noodzakelijk is dit in een multidisciplinair team uit te voeren.

- process monitoring en besturing:
  - chemische procesinstallaties
- communicatie:
  - monitoring en besturing van satellieten
  - telefooncentrales
- transactiegeoriënteerde verwerking:
  - on-line banking
  - on-line vliegtuigreserveringssystemen
- vluchtsimulatie en besturing:
  - automatische piloot
  - shuttle missie simulator
- productieautomatisering:
  - industriële robot (assemblage)
- transport:
  - verkeerssystemen
  - luchtverkeersbegeleidingssystemen
- interactieve grafische toepassingen
  - beeldverwerking
  - video games
- detectiesystemen:
  - radarsystemen
  - inbraakalarmsystemen
- 'virtual reality' systemen:
  - simulatie virtuele werelden

Fig. 3.2 Voorbeelden van real-time systemen

De definitie van de IEEE van een **real-time operating system** is als volgt:

> ***"The IEEE defines a real-time operating system as one that performs its functions and responds to external, asynchronous events within a deterministic (predictable) amount of time."***

De fabrikant Ready Systems past de volgende boodschap toe bij verkoop van zijn real-time operating system VRTX32:

> ***"Real time is different.....***
> ***The right answer at the wrong time is wrong."***

Real-time systemen verschillen duidelijk met **interactieve timesharing systemen**, waar een minimale responstijd noodzakelijk is voor maximale gebruikersvriendelijkheid. Zij garanderen geen maximale responstijd (halen van deadline).

In de praktijk kan het voorkomen dat zowel real-time als timesharing applicaties op dezelfde computer worden uitgevoerd. Beide typen applicaties dienen op verschillende manieren door het operating system behandeld te worden. Het operating system moet ervoor zorg dragen dat de real-time applicaties aan de timingseisen voldoen (halen van deadlines) en dat met behulp van de overgebleven resources de responstijd voor de interactieve applicaties geoptimaliseerd wordt.

3.2 Real-time operating systems maken systemen niet zo maar real time

Het is niet zo dat een computersysteem vanzelf in staat is een proces real-time te besturen als we dit voorzien van een **real-time operating system**! Afhankelijk van de gestelde eisen aan de responstijden op events moeten we een groot aantal beslissingen nemen om deze eisen te behalen.

Een real-time operating system vormt een laag tussen de real-time applicatiecode en de hardware. Dit betekent dat er nogal wat tijd 'verloren' kan gaan door de vele functies die een real-time operating system zelf verricht naast de code van de applicatie. Met andere woorden, real-time operating systems hebben altijd enige **overhead**. Real-time operating systemen moeten zodanig geïmplementeerd zijn dat deze overhead zo klein mogelijk is. Tevens moet een ontwikkelaar uitstekende kennis hebben van de toegepaste programmeertaal en mogelijk toe te passen codebibliotheken om de vereiste performance te kunnen garanderen. Ook moet kennis aanwezig zijn om bij het gebruik van C en C++ de juiste compileroptimalisaties te gebruiken. Let erop dat de meest compacte code (size optimization) niet de snelste oplossing hoeft te bieden (speed optimization). Het weglaten van toegevoegde debug code door de juiste compilerinstelling is een vereiste (geen debug versie maar een release versie compileren). Door optimalisatie is de directe relatie met de programma coderegels niet meer aanwezig.

Een ontwikkelrichtlijn uit de praktijk zegt dat een programma eerst logisch correct moet functioneren en dat pas na vaststelling van de run-time bottlenecks door ade-

quate tools (code profiler) de code in een programma moet worden aangepast. Onnodige programmacode-optimalisatie kan leiden tot slechter te onderhouden en te testen software omdat de overzichtelijke opbouw is verdwenen.

3.3 Opgaven

1. Welke events treden op bij Analoog Digitaal Converters (ADCs)?

2. Een real-time systeem dient zich volledig voorspelbaar te gedragen. Waarom is dit zo belangrijk voor industriële besturingen?

3. Geef aan waarom de volgende beschrijvingen wel of niet events zijn:

   - de auto staat voor stoplicht nr1.6
   - de lawine dondert naar beneden
   - de steen treft de bergbeklimmer
   - de ingestelde wachttijd is aan het verlopen
   - de synchronisatiepuls duurt 10msec
   - de robotarm bereikt zijn eindpositie

4 Structuur van operating systems

Operating systems zijn meestal vrij complex van opbouw. Voor de ontwikkeling en het onderhoud van dergelijke software kiest men doorgaans voor een modulaire opbouw. Het operating system is dan te beschouwen als een blokkendoos van functies waaruit de ontwikkelaar naar eigen behoefte kan kiezen of (gedeeltelijk) kan aanvullen.

4.1 De lagenstructuur

Kenmerkend aan moderne operating systems is de toepassing van een **lagenstructuur** (zie fig. 4.1). Elke laag creëert een andere virtuele machine die de details van de onderliggende software en hardware verder verbergt (abstractie). Dit maakt het systeem flexibel, doorzichtig (duidelijk herkenbare structuur) en door de gebruiker/ontwikkelaar relatief eenvoudig 'op maat' te maken.

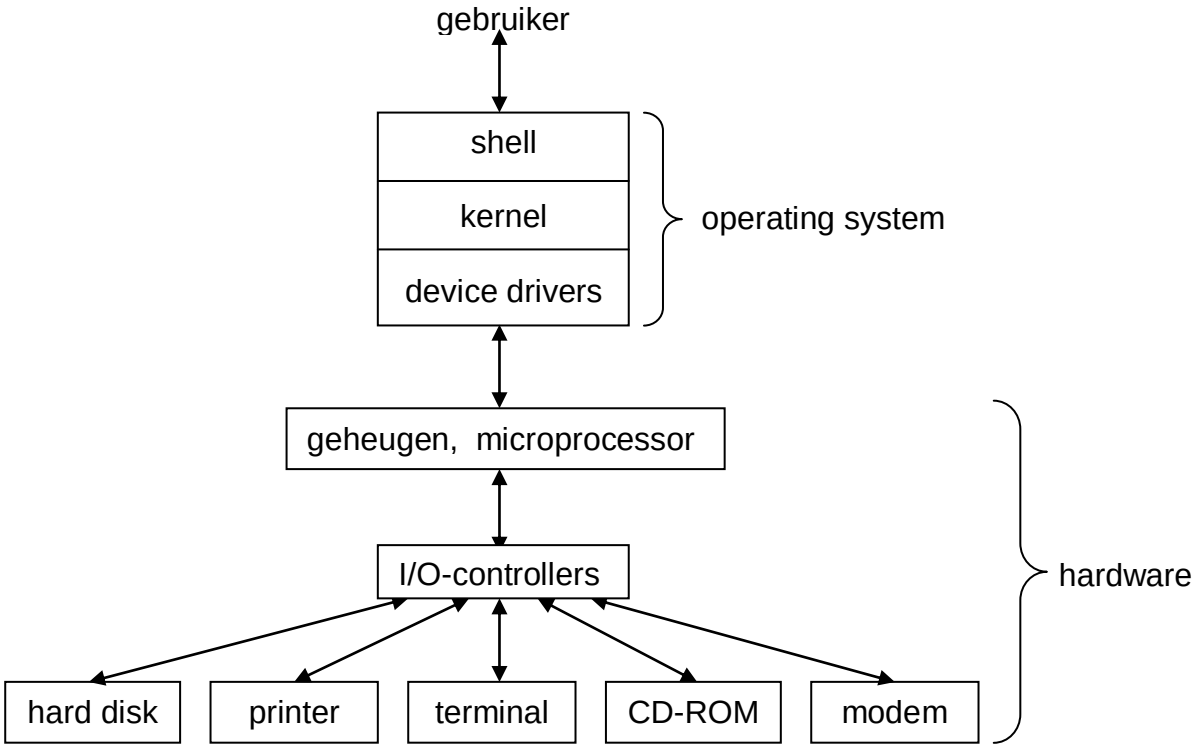


Fig. 4.1 De gelaagde structuur van een operating system

Tussen elke laag is een **interface** te definiëren. Uiteraard gelden strikte regels voor deze software interfaces. Als de lagen geheel uit software bestaan, dan bestaat de interface uit een aantal afspraken betreffende de betekenis van de toegepaste registers en de aan te roepen functies.

Voor ontwikkeldoeleinden heeft een gelaagde structuur het voordeel dat we voor het ontwikkelen van een laag alleen maar naar de interface van de aangrenzende lagen hoeven te kijken. Kennis van de overdrachtsparameters en de bijbehorende functies van de aangrenzende lagen is voldoende. Hoe de functies door de aangrenzende lagen worden uitgevoerd, is niet van belang (black box benadering).

Laag N kan bij een 'net' ontwerp alleen maar communiceren met laag N-1 en met laag N+1. Het is dus niet toegestaan dat laag N-1 direct met laag N+1 communiceert. Voor real-time doeleinden waar hoge snelheid centraal staat, kan deze gelaagdheid ten nadele van de uitvoeringssnelheid werken.

Een ander aspect dat invloed heeft op de uitvoeringssnelheid is **het aantal parameters** die aan een bepaalde system call of in het algemeen aan een interface wordt meegegeven. Hoe groter het aantal, hoe meer parameterwaarden door de betreffende laag geëvalueerd dienen te worden. Zonder twijfel kan dit in conflict komen met de timingeisen die gesteld zijn aan tijd kritisch (real-time) gedrag.

De lagen zorgen ervoor dat de benadering van de computer in hogere lagen steeds meer hardware-onafhankelijk wordt. De gebruiker en de ontwikkelaar hoeven in de hogere lagen steeds minder op de details (bits en bytes) van de hardware te letten.

4.2 De shell

Een **shell** stelt ons in staat via een terminal met het systeem te communiceren. De shell interpreteert de ingetypte commando's op de terminal. Hij voert deze meestal niet zelf uit, maar zorgt ervoor dat hiertoe de juiste acties worden ondernomen door de opdracht door te spelen aan de kernel, een hulpprogramma (utility) te starten of de gewenste applicatie op te starten. De shell wordt daarom ook wel aangeduid als **command line interpreter** (CLI). De shell wordt voor interactieve applicaties bij het opkomen van het computersysteem gestart. Voorbeelden van CLIs zijn **`tcsh`**, **`bash`** en **`zsh`**. Hoewel de command line vaak als Spartaans wordt gezien, kan deze zeer krachtig en efficiënt zijn, onder meer door het gebruik van slimme toetsencombinaties (bv. **`Ctrl-`**R voor het hergebruik van commando's) en sneltoetsen (zoals *tab completion*). Het document *"**Effcient use of the Linux command line in the Bash shell**"* geeft tips om het gebruik van de command line te vergemakkelijken en wordt aangeraden voor het practicum.

Shells kunnen meer of minder geavanceerd zijn uitgevoerd. De professionele ontwikkelaar heeft vaak behoefte aan zeer krachtige commando's. In de praktijk kan men vaak een keuze maken uit verschillende shells. Een shell die uitgebreidere faciliteiten biedt, wordt wel **extended shell** genoemd.

Een grafische user interface (**GUI**) is onmisbaar om bijvoorbeeld de status van industriële processen weer te geven. Er worden dan allerlei grafische objecten gebruikt die de verschillende componenten uit het bestuurde proces symbolisch weergeven. De grafische symbolische weergave is veel gemakkelijker te interpreteren dan een procesweergave in de vorm van tekst en rijen getallen op een niet-grafische terminal. Bij allerlei symbolen die verschillende componenten weergeven uit het te besturen proces, kunnen bepaalde meetgegevens real-time en/of de trend (het verloop) van deze data in de tijd in de vorm van een grafiek weergegeven worden. Het weergeven van de status van een te besturen proces wordt **monitoring** genoemd.

4.3 Utilities

De **utilities** bestaan uit compilers, editors en allerlei andere hulpprogramma's om het systeem te kunnen gebruiken en onderhouden en om met het systeem nieuwe applicaties te kunnen ontwikkelen en testen. Denk hierbij aan interne commando's als **`ps`**, **`nice`**, **`fg`**, **`bg`** en **`kill`** om processen te beheren, **`gcc`**, **`gdb`**, **`gprof`** en *Valgrind* om code te compileren, debuggen en profilen, teksteditors als **`nano`**, **`emacs`** of **`vi`** en source-control systems als **`git`**, **`bzr`** en **`svn`**. Van bijzonder belang is het commando **`man`**.

4.3.1 Man pages

De man pages geven informatie over zowel commando's op de command line, als functies uit de standaard C-library en de Linux system calls. De man pages worden opgeroepen via het commando **`man`**. Zo kan informatie over het commando **`kill`** worden opgevraagd met het commando **`man kill`**.

De man pages kennen verschillende genummerde hoofdstukken (sections). De meest belangrijke sections zijn:

1. **user commands** – informatie over commando's op de command line
2. **system calls** – informatie over Linux system calls
3. **C-library-functions** – informatie over de standaard C-library

Om informatie uit een specifiek hoofdstuk op te vragen wordt het nummer van het hoofdstuk opgegeven direct na het man-commando. Zo is **`kill`** zowel een commando op de command line als een Linux system call. Informatie over het gebruik van het commando wordt opgevraagd met **`man 1 kill`**, de syntax van de system call, en de benodigde header file, kan worden geraadpleegd met **`man 2 kill`**. Meer informatie over het gebruik van de man pages en andere utilities is te vinden in het document ***"Effcient use of the Linux command line in the Bash shell"***. Zie ook het commando **`man man`**.

4.4 De kernel

De **kernel** (bij andere operating systems ook wel aangeduid als **core**, **nucleus** of **executive**) vormt het hart van een operating system. De belangrijkste kernel functies zijn:

- systeeminitialisatie na een reset
- afhandeling van service requests (system calls)
- geheugenbeheer (memory management)
- CPU-management (multitasking, scheduling)
- Input/Output en file management
- afhandeling van exceptions en interrupts

In het geval van Linux worden vaak zowel de **Linux kernel** als het **GNU/Linux** operating system "Linux" genoemd.

4.5 System calls

De operating systeem functies die in een applicatie aangeroepen kunnen worden, worden **system calls** genoemd. In een applicatieprogramma kan men via in het operating system vastgelegde system calls van een hoger naar een lager niveau (bv. device drivers) doordringen. Dit helpt mede om het computersysteem te beschermen tegen mogelijke ontwerpfouten. Met andere woorden, dit helpt om een betrouwbaarder systeem te bouwen. De softwareontwikkelaar is 'gedwongen' system calls te gebruiken. Informatie over system calls is beschikbaar in sectie 2 van de man pages.

Vooral bij een multi-user systeem en in mindere mate bij een multitasking systeem is protectie van fundamenteel belang. Een fout van een gebruiker of in een gebruikersprogramma mag immers geen fout(en) voor andere gebruikers ten gevolgen hebben, laat staan dat er een catastrofale fout mag ontstaan waardoor het hele computersysteem down gaat. Iedere fout die een mogelijk catastrofaal gevolg kan hebben, dient te worden opgevangen door een **exception routine**.

Moderne microprocessoren ondersteunen de beveiliging van systemen tegen gebruikersfouten. Zij kennen de supervisor mode en de user mode. In operating systems termen spreekt men over **system state** en **user state**. Device drivers en exception routines en alle functies van de kernel worden in system state uitgevoerd. Als in een applicatieprogramma een system call wordt gedaan, wordt even overgeschakeld van user state naar system state en omgekeerd. In user state is het aantal activiteiten dat uitgevoerd kan worden beperkt. Zo zijn bv. alle acties die betrekking hebben op de hardware voor de gebruiker afgeschermd en kunnen deze alleen in system state worden uitgevoerd.

4.6 Het file system

Het file system heeft niet alleen betrekking op files, maar ook op alle andere hardware. Het file system is dus in feite een uniforme 'hardware-onafhankelijke' I/O-structuur. Bij Linux spreekt men daarom van een **Virtual File System** (VFS). Het Virtual File System is een softwarelaag in de kernel die alle system calls afhandelt die betrekking hebben op het filesysteem.

Voor de gebruiker maakt het niet uit of hij toegang wil krijgen tot een file of tot een ander device. Hierdoor kan het aantal verschillende system calls sterk gereduceerd worden en het gebruik ervan vereenvoudigd. Er kan toegang tot een file of andere input of output (bv. een printer of terminal) worden gekregen door een pad naar die file, resp. I/O device te openen en daar vervolgens lees- of schrijfacties uit te voeren. De gebruiker kan m.b.v. dezelfde system calls lezen en schrijven. Het maakt niet uit of het om files (op disk drives) of data van andere devices gaat.

Het is ook mogelijk om op eenvoudige wijze de paden voor I/O om te buigen van het ene naar het andere device, het zogenaamde **redirectioneren van I/O**. Zo kan bv. output in plaats van naar de terminal naar een printer worden omgeleid.

Als de kernel een system call ontvangt om data uit te wisselen met een I/O device, dan roept hij hierbij de hulp in van een device driver.

4.7 Device drivers

Per type device is er een device driver voor de fysieke I/O-functies. De device driver bevat functies voor het initialiseren, lezen, schrijven, status ophalen, status veranderen, beëindigen van de I/O en de error handling voor het betreffende device. Zo heeft bijvoorbeeld een disk-drivermodule de basisfuncties lees en schrijf een fysieke sector. De driver houdt zich niet bezig met file directories en dergelijke. Deze worden op een hoger niveau afgehandeld door de kernel (in het Virtual File System).

Ruwweg zijn er drie typen device drivers:

- **character device drivers**
  Hierbij worden de data karakter voor karakter met een device uitgewisseld. Denk hierbij aan terminals en printers.

- **block device drivers**
  In dit geval gaat de data uitwisseling met het device in blokken tegelijk. Een dergelijk blok data wordt opgeslagen in een geheugenbuffer en van daaruit door de system calls verder verwerkt. Hierbij valt te denken aan disk drives. Ook als de applicatie slechts één byte uit een file nodig heeft, wordt toch de hele sector die deze byte bevat gelezen en in een geheugenbuffer opgeslagen. De benodigde byte wordt vervolgens uit deze buffer gehaald. Is later een andere byte uit de file nodig en staat deze al in de buffer, dan wordt de driver niet opnieuw aangesproken.

- **network device drivers**
  Network device drivers regelen het versturen en ontvangen van data packets naar en van hardware interfaces die verbonden zijn met externe systemen. Het doel van de drivers is het aanbieden van een uniforme interface dat kan worden gebruikt door netwerkprotocollen.

Character device drivers worden uitgebreid behandeld in het dictaat 'Linux device drivers'.

4.8 Host en target systems

In een ontwikkelomgeving zijn in veel gevallen meerdere terminals op een systeem aangesloten. Dit maakt het mogelijk dat meerdere ontwikkelaars gelijktijdig kunnen werken aan de realisatie van real-time applicaties. De real-time applicaties kunnen niet getest worden in de ontwikkelomgeving, maar worden bijvoorbeeld via een netwerk overgebracht naar de hardware waar de real-time applicatie moet lopen. Het systeem waarop ontwikkeld wordt, wordt het **host system** genoemd. De hardware waarop de applicatie moet draaien, wordt het **target system** genoemd. Een target system bevat in het algemeen allerlei interfaces (onder andere GPIO, ADCs, UARTs) met het te besturen systeem.

4.9 Opgaven

1. Geef aan of en zo ja hoe een gelaagde indeling van een te ontwerpen systeem het samenwerken van verschillende ontwikkelaars vergemakkelijkt of bemoeilijkt.

2. Er zijn operating systems die kunnen voldoen aan zeer hoge snelheidseisen voor real-time toepassingen. Geef globaal aan of deze zeer snelle operating systems in het algemeen
   - veel code bevatten (omvangrijke object code)
   - uit veel lagen bestaan
   - functies bevatten met veel parameters
   - uitgebreid *error checking* uitvoeren

3. Met welk commando kun je informatie over de indeling van de man pages oproepen op de command line?

5 Multitasking en scheduling

Bijna alle operating systems bieden multitasking faciliteiten. Het is daarom belangrijk om nader kennis te maken met de principes en de implementatie. In het bijzonder komen ook de real-time eisen aan bod.

Er wordt onderscheid gemaakt tussen pre-emptive en cooperative multitasking systems. Bij **pre-empted multitasking** wordt de running taak onvrijwillig afgebroken. In dit geval beslist de scheduler wanneer de running taak stopt en een ander proces (weer) running wordt. Dit kan zijn doordat er een interrupt optreedt ten behoeve van een proces met een hogere prioriteit dan het running proces of doordat de timeslice van het running proces afloopt.

Bij **cooperative multitasking** bepaalt de running taak zelf wanneer hij stopt. Dit is bv. het geval als de taak wacht op het voltooien van een I/O-operatie. Het grote nadeel van deze methode is dat een taak alle andere taken volledig en langdurig kan blokkeren. Er is niet te voorspellen wat de responstijd op een event zal zijn. Cooperative multitasking is dus niet geschikt voor real-time systems. Vrijwel alle moderne operating systems werken volgens het principe van pre-emption.

5.1 Multitasking volgens het timeslice principe

Een **taak** (process, task) is een verzameling instructies (object code) die sequentieel wordt uitgevoerd en die een min of meer zelfstandige functie uitvoert. Object code wordt in de vorm van een module in een extern geheugen bewaard en voor gebruik in het interne geheugen geladen. Het operating system bepaalt waar de modules in het werkgeheugen geplaatst worden.

Een verzameling taken heeft in het algemeen het doel om quasi-parallel uitgevoerd te worden en onderling te communiceren. Alle taken binnen één processing module krijgen om beurten de CPU toegewezen gedurende een bepaalde tijdsduur (**timeslice**), meestal 10 tot 100 ms.

Fig. 2.3 geeft de verdeling van de processortijd over drie taken. Hierbij is aangenomen dat deze taken niet op elkaar of op I/O moeten wachten, dat er geen interrupts ontvangen worden en dat de prioriteit van de taken gelijk is. Het lijkt voor iedere taak alsof deze over een eigen processor beschikt, maar dan wel een processor die op 1/3 van de snelheid van de fysieke processor werkt. We hebben in fig. 2.3 de overhead ten gevolge van het omschakelen van de ene naar de andere taak buiten beschouwing gelaten.

5.2 Task states en task switches

5.2.1 Task control blocks of process descriptors

Iedere taak heeft binnen het geheugen van de computer een eigen codegebied, een eigen datagebied en een eigen stack. Tevens houdt het operating system van iedere taak een **task control block** (TCB) of **process control block** (PCB) bij. Een task control block is een datastructuur die relevante gegevens over de toestand van een taak bevat. Zodra een taak de processor weer terugkrijgt, kan deze taak weer verder gaan alsof hij niet onderbroken was geweest. Binnen Linux wordt een TCB of PCB een **process descriptor** of **task descriptor** genoemd. TCBs vormen een (vaak doubly) linked list (queue), de **process list** of **task list** (zie §5.3 en fig. 5.3-5.5).

5.2.2 Text segments, user segments en data segments

Fig. 5.1 toont de drie segmenten van een taak of proces. Het **text segment** bevat de uit te voeren instructies in machinecode. Dit segment is **read-only** en kan dus niet door het proces worden gewijzigd. Hierdoor is het mogelijk dat meerdere processen tegelijk gebruik maken van hetzelfde text segment, wat geheugenruimte bespaart. Als een programma meerdere malen wordt opgestart, hoeft het text segment slechts één keer in het werkgeheugen te worden gezet.

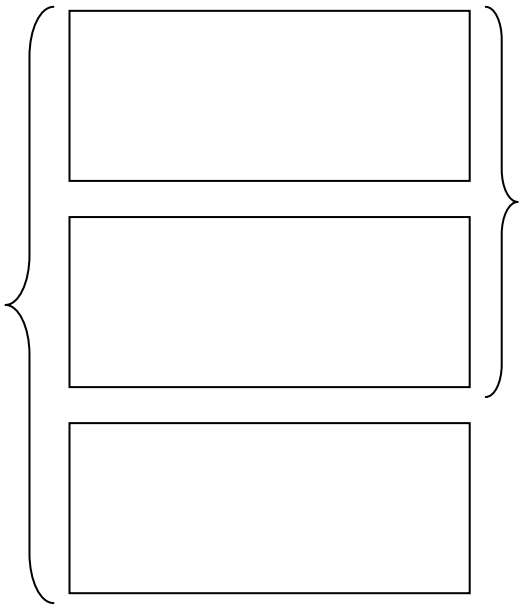


Fig. 5.1 De onderdelen van een proces en programma

Het **user data segment** bevat alle data waarmee het proces werkt tijdens de uitvoering. Indien een programma meerdere keren (ongeveer gelijktijdig) wordt opgestart, krijgt elk proces zijn eigen user data segment toegewezen. Zo wordt voorkomen dat de verschillende processen elkaars data vernielen. Ook krijgt elk proces zijn eigen **system data segment** toegewezen. Hierin staan de gegevens die de kernel over het

proces bijhoudt, zoals het **process ID** (procesidentificatie) en het TCB/PCB. Elk proces krijgt van de kernel een eigen process ID toegewezen bij opstarten. Hiermee kan de kernel de verschillende processen van elkaar onderscheiden. In het system data segment wordt ook bijgehouden wat de status van het betreffende proces is, zoals de inhoud van de processorregisters tijdens een task switch (of context switch) aan het einde van een timeslice.

In fig. 5.1 is ook te zien dat een programma een statisch ding is op een disk, terwijl een proces of taak een dynamisch ding is in het werkgeheugen van de processor waarbij interactie plaatsvindt tussen instructiecode, user data en system data.

5.2.3 Task states en task switches

In de eenvoudigste vorm (zie fig. 5.2) kan een taak in de volgende toestanden (**task states**) verkeren:

- **running**
  de taak **heeft processortijd** en wordt uitgevoerd door de processor. In een systeem met één processor kan er maar één taak running zijn.

- **active**
  de taak **wacht op processortijd**; de active toestand wordt ook vaak aangeduid als de **ready** toestand. Vaak zijn meerdere taken active, die in een **queue** (zie ook fig. 5.4 en 5.5) afwachten tot ze processortijd krijgen van de **scheduler**.

- **blocked (waiting)**
  de taak **kan** om één of andere reden **niet verder**; de taak wacht bv. op input of output van een device of op een signal; men spreekt hier ook wel van **sleeping**.

Het toelaten van wachtlussen waarin continu op een bepaalde conditie wordt getest voordat het programma door kan gaan, **busy waiting**, moet in multitasking applicaties vermeden worden. Het wachten en testen is een verspilling van processortijd. Een taak moet zichzelf not active (blocked, waiting, sleeping) kunnen maken als er reden is om niets te doen dan alleen maar wachten.

Na het verstrijken van een timeslice zal een timer een interrupt genereren. Hierdoor worden de program counter en het statusregister van de onderbroken taak op zijn stack gezet. Daarna krijgt een speciale operating system taak, de **scheduler** of **dispatcher** genaamd, processortijd.

De scheduler zal allereerst de registerinhoud van de onderbroken taak redden in een register save area binnen de process descriptor van die taak. Vervolgens wordt deze process descriptor zodanig bijgewerkt dat die taak achter aan de active queue aansluit. De voorste taak in de active queue wordt vervolgens running gemaakt door de process descriptor van die taak overeenkomstig bij te werken. Daarna worden de registerinhouden van deze 'nieuwe' taak uit de process descriptor van deze taak ge-

haald. Ten slotte zal deze nieuwe taak verdergaan alsof hij niet onderbroken is geweest.

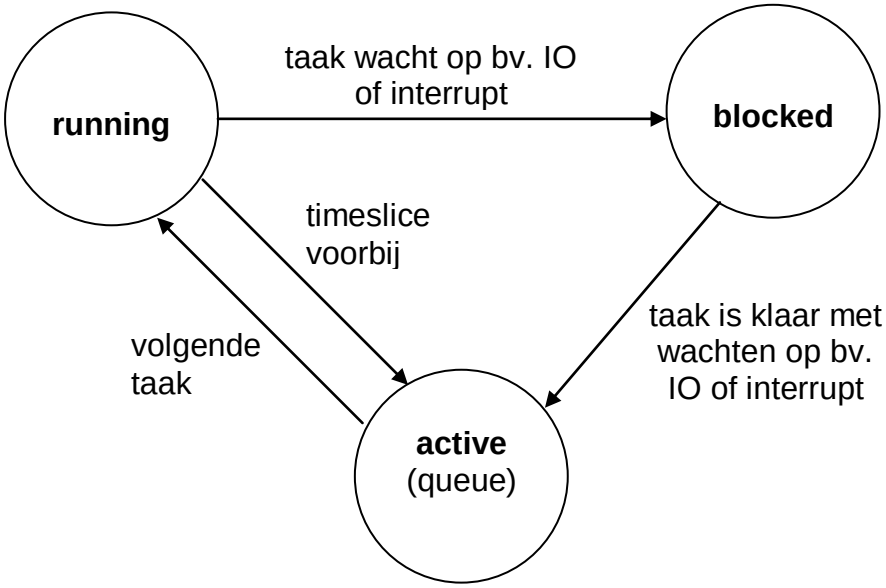


Fig. 5.2 Task states binnen een eenvoudig operating system

Bij **round-robin scheduling** krijgt elk active process om de beurt een timeslice toegewezen.

Het hiervoor geschetste mechanisme dat ervoor zorgdraagt dat een nieuwe taak processortijd krijgt, wordt een **task switch** of **context switch** genoemd. Omdat een task switch enige tijd in beslag neemt, is het niet mogelijk een timeslice steeds maar kleiner te maken: de task switch gaat op een gegeven moment veel meer tijd kosten dan dat een taak running kan zijn. Tevens moet er een redelijk aantal instructies uitgevoerd kunnen worden.

Indien een taak om één of andere reden niet verder kan (bijvoorbeeld omdat deze op invoer van buitenaf of op data van een andere taak moet wachten), wordt deze taak in de **blocked state** geplaatst. Zodra aan bepaalde voorwaarden is voldaan (bijvoorbeeld totdat een interrupt aangeeft dat er invoer beschikbaar is) wordt de taak weer active gemaakt. De process descriptor van deze taak zal dan weer in de active queue worden gezet.

Indien een taak is beëindigd, wordt de bijbehorende process descriptor niet meer opgenomen in de active queue en wordt de door de taak en de bijbehorende process descriptor gebruikte geheugenruimte teruggegeven aan het systeem.

5.3 Datastructuren voor de administratie van process descriptors

Het in een bepaalde state verkeren, betekent voor de computer dat de betreffende process descriptor (PCB/TCB) in de bij die state behorende **gelinkte lijst** (toegepast

als queue) voorkomt. Fig. 5.3 geeft een voorbeeld van de gelinkte lijststructuren zoals deze door de scheduler worden bijgehouden.

Elk onderdeel (node = TBC/process descriptor) in een gelinkte lijst is in de programmeertaal C beschreven door een **struct** met verschillende data-elementen. In deze struct zit ook een pointer naar de volgende node. Gelinkte lijsten hebben een dynamisch karakter omdat tijdens de uitvoering nodes toegevoegd en verwijderd moeten worden. De C functies **malloc()** en **free()** verzorgen de dynamische allocatie en deallocatie van nodes.

We geven hier een vereenvoudigd task-switch algoritme in C. We kiezen de volgende datastructuur voor de process descriptor:

```
typedef struct Descr {
    struct Descr *pNextDescr;
    ...     /* task status data */
    ...
} ProcDescr;
```

We gaan ervan uit dat het operating system de ruimte voor een nieuwe process descriptor dynamisch alloceert bij het creëren van een taak (zie de C-functie **malloc()**; **`man 3 malloc`**). Als een taak eindigt, wordt deze ruimte weer vrijgegeven (zie de C-functie **free()**; **`man 3 free`**).

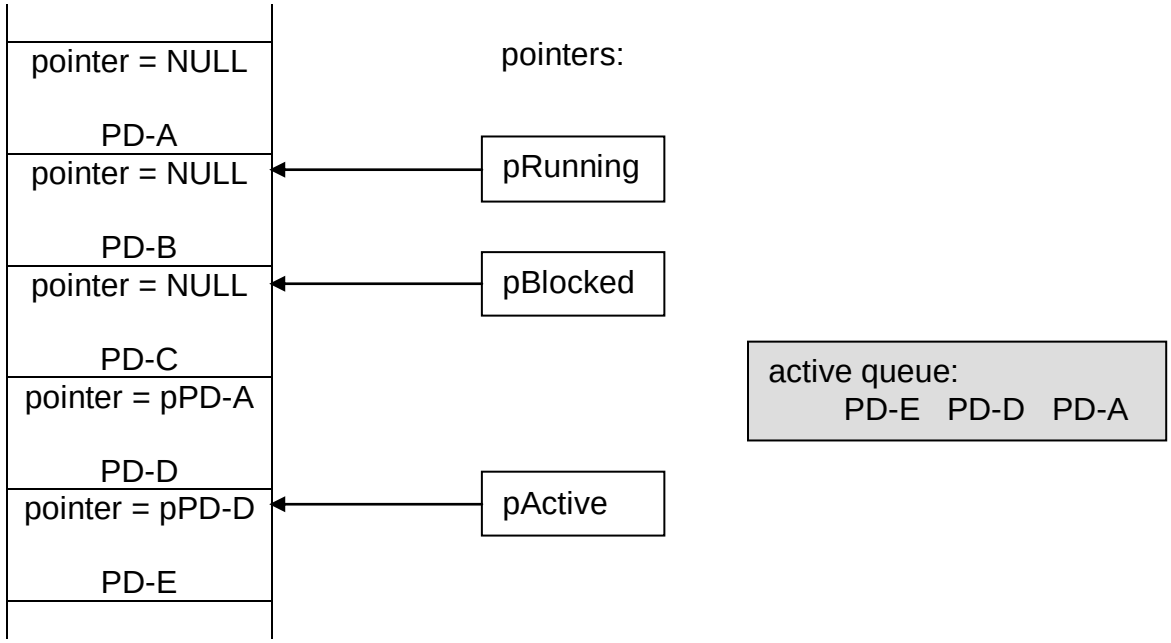


Fig. 5.3 Fysieke plaatsing van process descriptors

Op een bepaald moment is een taak running en enkele taken zijn in de active toestand. De functie **TaskSwitch()** voegt de running task aan het einde van de active list toe en maakt de voorste taak in de active list running. We laten de blocked toestand buiten beschouwing.

De pointer pRunning wijst naar de process descriptor van de running task en de pointer pActive wijst naar de eerste taak in de active list. De declaratie van de genoemde globale variabelen ziet er als volgt uit:

```
ProcDescr *pRunning, *pActive;
```

De volgende functie verricht de task-switch administratie:

```
void TaskSwitch (void)
{
    ProcDescr *pTemp;

    pTemp = pActive;

    while (pTemp->pNextDescr)
    {
        pTemp = pTemp->pNextDescr;
    }
                /* pTemp wijst nu naar de laatste process
                   descriptor in de active queue       */

    pTemp->pNextDescr = pRunning;
    pRunning = pActive;
    pActive = pActive->pNextDescr;
    pRunning->pNextDescr = NULL;
}
```

5.4 Process scheduling bij Linux

Zie Hoofdstuk 7 "Process scheduling" van het boek "Understanding the Linux kernel" voor meer achtergrond.

5.4.1 Scheduling en prioriteit

Voor real-time applicaties kunnen we niet volstaan met een eenvoudig scheduling algoritme waar elke taak netjes op zijn beurt wacht en evenveel tijd krijgt als alle andere taken (**round-robin scheduling**). Real-time systemen stellen meer eisen: het scheduling algoritme moet gebaseerd zijn op **priority-based pre-emptive scheduling**.

Bepaalde taken moeten een hogere **prioriteit** kunnen krijgen dan andere. Dit heeft tot gevolg dat taken met een hogere prioriteit vaker aan de beurt komen dan taken met een lagere prioriteit. Tevens kan een taak met een hogere prioriteit een langere timeslice krijgen dan een taak met een lagere prioriteit.

Na een timeslice interrupt of na het opgeven van de processor door de running taak zal een volgende taak uit de active queue running worden. De scheduler beslist volgens een bepaald algoritme aan de hand van de prioriteiten welke taak dat zal zijn. In §5.5 gaan we verder in op de formele definitie van prioriteit.

5.4.2 Process states

De Linux scheduler maakt onderscheid in vijf toestanden waarin een proces kan verkeren:

TASK_RUNNING
Het proces is bezig uitgevoerd te worden of staat te wachten om uitgevoerd te worden. In feite is dit een samenvoeging van de toestanden **running** en **active** uit fig. 5.2. Processen in deze toestand worden **runnable processen** genoemd.

TASK_INTERRUPTIBLE
Het proces slaapt en wacht op een speciale event zoals een interrupt van een I/O device of een weksignaal. Het proces wordt dan runnable.

TASK_UNINTERRUPTIBLE
Het proces slaapt en kan alleen gewekt worden door een specifiek event (bijvoorbeeld een serieel IO event, dat langzaam is in vergelijking met de processor). Signals zijn geblokkeerd.

TASK_STOPPED
Het proces is gestopt na het ontvangen van een stopsignaal. Met een continue signal kan een stopped proces weer runnable worden gemaakt.

TASK_ZOMBIE
Het proces is geëindigd, maar het parent proces heeft nog geen wait() call uitgevoerd en dus de exit status van het (child) proces nog niet gelezen (zie §6.3.1).

5.4.3 Run queues en doubly-linked lists

Runnable processen staan in zogenaamde **run queues** of **active queues**. Dit zijn **dubbel gelinkte circulaire lijsten** (**doubly-linked circular lists**) van process descriptors (TCBs/PCBs; zie fig. 5.4). Hierbij bezit elke process descriptor van de lijst een pointer naar de volgende en een pointer naar de vorige process descriptor. Het eerste element van een dubbel gelinkte circulaire lijst wijst dus naar het volgende en naar het laatste element. Het laatste element wijst naar het vorige en naar het eerste element.

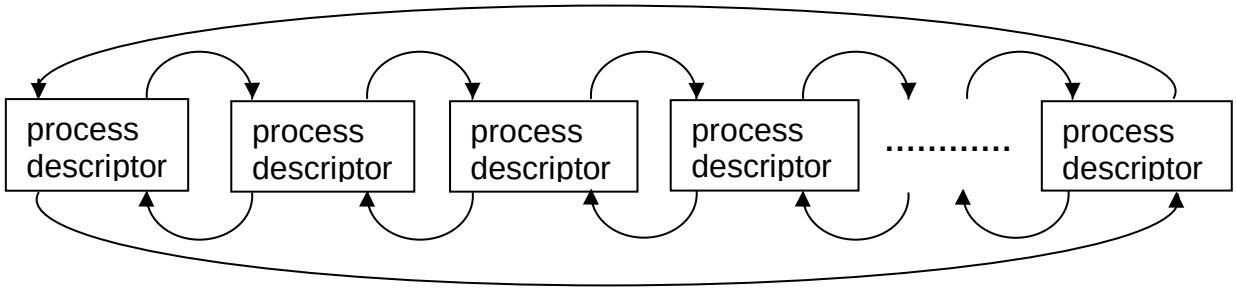


Fig. 5.4 Run queue als **dubbel gelinkte circulaire lijst** van process descriptors

Een **process descriptor** bevat verder alle gegevens van een proces, zoals de inhouden van de processorregisters op het moment van de task switch, de prioriteit van het proces, zijn procesidentificatie, enz.

Een proces dat running is geweest, wordt aan het einde van de run queue geplaatst. Dit is een erg eenvoudig proces. Het enige dat hoeft te gebeuren is de run pointer gelijk te maken aan de next pointer van het proces dat zojuist door de processor is uitgevoerd. Hierdoor krijgt het volgende proces in de run queue processortijd en komt het huidige proces automatisch achteraan in de run queue te staan. Dit houdt in dat de task switch een vaste tijd duurt en niet afhankelijk is van de lengte van de run queue. Dit is dus een ideale scheduler. Men spreekt hierbij van een **O(1) type scheduler**. Lees O(1) als 'Order 1'. De 1 staat dus niet voor de executietijd!

Bepaalde schedulers, zoals die met een queue die bestaat uit een sinlgy-linked list (zie fig. 5.5), kunnen ook **O(n)** gedrag vertonen. De executietijd voor het toevoegen of verwijderen van een taak is hier evenredig met het aantal taken dat de scheduler beheert (**`n`**); om de pointer van het vorige naar het nieuwe element in de lijst te maken moet eerst de hele circulaire lijst (dus **`n`** elementen) worden doorlopen om het vorige element te vinden. Singly-linked lists zijn gevoeliger voor memory leaks en worden minder vaak gebruikt voor kritische systemen.

5.4.4 Wait queues en singly-linked lists

De **interruptible** en **uninterruptible processen** staan in **wait queues**. Er is een aparte wait queue voor elk specifiek event waarop een proces wacht. Het gaat hierbij om **enkelvoudig gelinkte circulaire lijsten** (**singly-linked circular lists**) die alleen een pointer naar het volgende element bevatten en een pointer naar de bijbehorende process descriptor van het wachtende proces (zie fig. 5.5). Als zo'n lijst leeg is, is de next pointer van het eerste element de NULL pointer.

De stopped en zombie processen zijn niet opgenomen in een wachtrij. Zij worden rechtstreeks door hun parent bereikt via hun process ID.

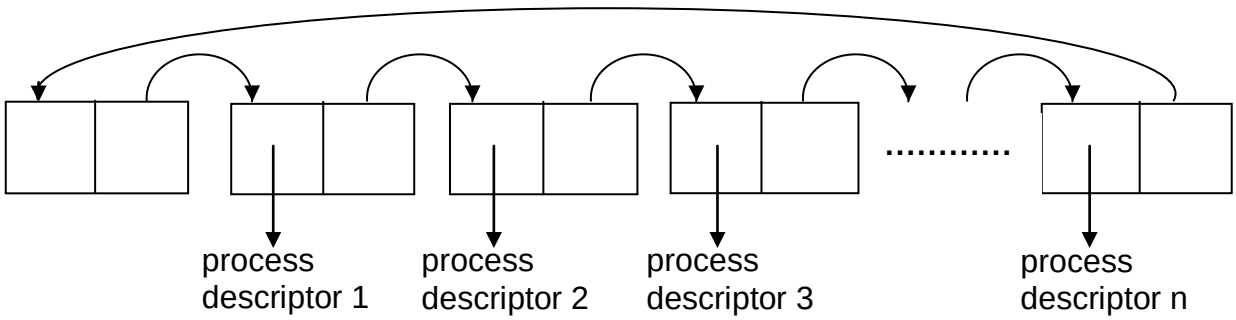


Fig. 5.5 Wait queue als **enkelvoudig gelinkte circulaire lijst**

5.4.5 Prioriteiten, de run queue en classificaties van processen

De Linux kernel kent een aparte run queue voor elke procesprioriteit (zie §5.5). Bovendien wordt er onderscheid gemaakt tussen een **active run queue** met taken die nog niet hun complete timeslice hebben verbruikt en de **expired run queue** met taken die wel hun hele timeslice hebben verbruikt. Zodra de timeslice van een taak is verbruikt, wordt die taak overgebracht van de active naar de expired run queue. Als alle taken uit de active run queue hun timeslices hebben verbruikt (dus als deze queue leeg is), worden de pointers van de expired en active run queue verwisseld; de expired run queue wordt de active run queue en omgekeerd. Nu krijgen alle taken opnieuw een timeslice toegewezen. Dit voorkomt dat taken met lage prioriteit nooit processortijd krijgen (**process starvation**).

Als een taak van de active run queue naar de expired run queue wordt overgebracht (wanneer zijn timeslice is verbruikt), wordt de grootte van de timeslice opnieuw berekend. Deze is afhankelijk van de prioriteit van de taak. Hoe groter de prioriteit van een proces (dus hoe lager de prioriteitswaarde), des te groter is zijn timeslice. De prioriteit van processen is dynamisch. De kernel houdt bij wat de processen doen en verhoogt de prioriteit van een proces dat langdurig geen gebruik heeft gemaakt van de processor. Processen die de processor langdurig gebruiken, krijgen een lagere prioriteit. Processen worden wat dit betreft verdeeld in:

- **I/O-gebonden processen**
  Deze processen maken veel gebruik van I/O-devices en besteden veel tijd aan het wachten op het voltooien van I/O-operaties. Ze zijn vaak runnable, maar telkens voor slechts korte tijd. Ze gebruiken hun timeslice vrijwel nooit tot het einde. Het gaat hier doorgaans om **interactieve processen**.

- **CPU-gebonden processen**
  Processor-gebonden processen gebruiken vrijwel continu processortijd en gebruiken vrijwel altijd hun volledige timeslice. Ze zijn vrijwel uitsluitend bezig met het uitvoeren van code en maken weinig gebruik van I/O.

De scheduler zorgt ervoor dat de I/O-gebonden processen een **hogere prioriteit** krijgen dan de processor-gebonden processen. Zo wordt bereikt dat I/O-gebonden processen een korte responsetijd hebben. De processor-gebonden taken krijgen minder, maar wel **langere timeslices**. Zodoende is de voortgangssnelheid van deze processen hoog.

Een andere classificatie van processen is de volgende:

- **interactieve processen**
  Deze processen hebben voortdurend interactie met de gebruiker en zijn hier vaak aan het wachten op een toetsaanslag of het bewegen van de muis. Als er input wordt ontvangen, moet het systeem hierop snel reageren anders denkt de gebruiker dat het systeem niet actief is of dat er een fout is opgetreden. De reactietijd moet liggen in de orde van grootte van 0,1 s.

- **batch-processen**
  Deze processen behoeven geen interactie met een gebruiker en draaien veelal in de background. Deze hebben geen groot belang bij korte reactietijden of snelle voortgang.

- **real-time processen**
  Aan deze processen worden strenge timingeisen gesteld. Ze vereisen een korte responstijd. Ze moeten dus een hoge prioriteit hebben om niet geblokkeerd te worden door processen met een lagere prioriteit.

Linuxprocessen zijn **pre-emptive**:

- Een proces kan worden onderbroken aan het einde van zijn timeslice (en in de **expired run queue** geplaatst).
- Als een proces runnable wordt, krijgt dit proces processortijd als zijn prioriteit groter is dan die van het huidige running process. Het huidige proces wordt dan voortijdig afgebroken (vóór het einde van zijn timeslice) en teruggeplaatst in de **active run queue**.

5.4.6 Epochs en clock ticks

De Linux scheduler verdeelt de processortijd in zogenaamde **epochs**. Een epoch eindigt als alle runnable processen hun volledige timeslice hebben verbruikt (alle active run queues zijn dan leeg). Aan het begin van de nieuwe epoch wordt de grootte van de timeslice van elk user proces opnieuw berekend. Een proces kan gedurende één epoch meerdere malen processortijd krijgen. Dit is het geval als het proces door een proces met hogere prioriteit wordt onderbroken of als het zichzelf blokkeert om te wachten op het einde van een I/O-operatie. De grootte van de timeslice is een veelvoud van het aantal clock ticks en afhankelijk van de prioriteit. Een timer genereert elke 10 ms een interrupt, een **clock tick**. Tijdens een epoch wordt telkens de **average sleep time** van de processen bepaald. Dit is de tijd gedurende welke een proces zich in de wait queue bevindt, verminderd met de tijd dat het proces running is. De maximale waarde hiervan is 1 s.

Een **counter** in de process descriptor houdt bij hoeveel clock ticks van de timeslice nog resteren. De counter van het running proces wordt bij elke tick met 1 verlaagd. Als deze counter 0 wordt, wordt het proces gedescheduled en in de expired run queue geplaatst.

5.4.7 Schedulers voor real-time en user processes

De Linux scheduler maakt onderscheid tussen:

- **real-time processen** met een **statische prioriteit** tussen 1 en 99 (SCHED_FIFO en SCHED_RR)

- **user processen** (niet-real-time processen) met een **dynamische prioriteit** tussen 100 en 139 (SCHED_OTHER en SCHED_BATCH die een statische prioriteit van **0** hebben).

De scheduler kiest telkens real-time taken uit de active run queue met de hoogste prioriteit (het laagste nummer) om te laten uitvoeren. Dit gebeurt aan de hand van 5 32-bits woorden (voor de 139 prioriteiten). De bits die overeenkomen met lege active run queues zijn hierin 0, de andere 1. Als er geen real-time taken meer zijn, krijgen de user taken processortijd. Dit gaat om beurten, ongeacht hun prioriteit. Wel zijn de timeslices groter naarmate de prioriteit groter is (dus een lagere waarde heeft).

Linux processen kunnen gebruik maken van de volgende methoden van scheduling voor real-time systemen:

- **SCHED_FIFO: first-in first-out scheduling** (real-time)
  Als een shed_fifo proces runnable wordt, onderbreekt dit altijd elk running sched_other, sched_batch of sched_idle proces en wordt aan het eind van de active run queue van zijn prioriteit gezet.

  Bij fifo scheduling treedt geen timeslicing op! Dit betekent dat een shed_fifo proces alleen onderbroken kan worden doordat een proces met een hogere prioriteit runnable wordt, doordat het proces zichzelf aan het einde van zijn active run queue zet of doordat het proces wacht op I/O. Wordt een shed_fifo proces onderbroken door een proces met hogere prioriteit, dan blijft dit proces vooraan in de active run queue staan en zal het processortijd krijgen zodra alle processen met hogere prioriteit geblokkeerd zijn (in de wait queue staan). Met de system call sched_setscheduler() of sched_setparam() wordt het betreffende proces vooraan in de active run queue gezet, mits dat proces runnable is. Dit betekent dat dit proces een proces met dezelfde prioriteit kan onderbreken.

- **SCHED_RR: round-robin scheduling** (real-time)
  Het enige verschil met sched_fifo processen is dat nu wel timeslicing optreedt. Na het verstrijken van zijn timeslice wordt het proces aan het eind van de active run queue van zijn prioriteit geplaatst.
  timeslice = (140 – statische prioriteit) x 20 ms

- **SCHED_DEADLINE: real-time scheduler Earliest Deadline First (EDF)**
  Deze methode is geïmplementeerd sinds de Linux kernel v3.14 en maakt het mogelijk real-time gedrag te realiseren door gebruik te maken van deadlines. Zie http://www.evidence.eu.com/sched_deadline.html voor meer informatie en het dictaat Real-time Scheduling voor EDF.

Voor niet-real-time systemen zijn de volgende schedulers beschikbaar:

- **SCHED_OTHER: default time-sharing scheduling**
  Hierbij is de statische prioriteit altijd 0. Bij deze vorm van time-sharing wordt gebruik gemaakt van de dynamische prioriteit. Deze ligt tussen 100 en 139 en is afhankelijk van de nice-waarde en van het processorgebruik.

- **SCHED_BATCH: scheduling batch processen**
  Deze methode lijkt sterk op die van sched_other met dit verschil dat de scheduler altijd aanneemt dat de processen CPU-gebonden zijn en derhalve een lagere prioriteit krijgen.

- **SCHED_IDLE: scheduling very low priority processen**
  Deze methode lijkt sterk op die van sched_other en sched_batch maar hier heeft de nice-waarde geen effect. De betreffende processen hebben een prioriteit die nog lager is dan processen met een nice-waarde van +19.

5.4.8 System calls voor de Linux scheduler

Tot slot bespreken we enkele system calls. De returnwaarde van deze system calls is 0 als de functie correct is uitgevoerd en -1 als er een fout is opgetreden (errno bevat dan nadere informatie omtrent de fout), tenzij uitdrukkelijk anders wordt vermeld. Om deze system calls te kunnen gebruiken, moet de volgende #include worden opgenomen:

```
#include <shed.h>

int sched_setscheduler (pid_t pid, int policy,
                        const struct sched_param *param);
int sched_getscheduler (pid_t pid);
int sched_setparam (pid_t pid, const struct sched_param *param);
int sched_getparam (pid_t pid, struct sched_param *param);
int sched_get_priority_min (int policy);
int sched_get_priority_max (int policy);
int sched_yield(void);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

De functie **sched_setscheduler()** wordt gebruikt om voor het proces met de proces identifier pid (zie §6.3) een van de in §5.4.8 genoemde schedulermethoden (policies) in te stellen. De parameters die worden meegegeven, hangen af van de schedulermethode die gekozen wordt:

- Met **sched_priority** (in de struct sched_param, een argument van de functie sched_setscheduler()) wordt de statische prioriteit ingesteld.
- Met **sched_getscheduler()** wordt van het proces met de identifier pid de schedulermethode opgehaald (als returnwaarde).
- Met **sched_setparam()** worden de parameters van het met pid aangegeven proces overeenkomstig de betreffende schedulermethode ingesteld.
- Met **sched_getparam()** worden de parameters van het met pid aangegeven proces opgehaald.

- Met **sched_get_priority_min()** en **sched_get_priority_max()** wordt de toegestane minimale, resp. maximale waarde van de prioriteit opgehaald (als returnwaarde) die bij de opgegeven policy hoort.
- Met **sched_yield()** staat het proces vrijwillig de processor af aan een andere taak en zet zichzelf aan het einde van zijn active wait queue.
- Met **pid=0** wordt het aanroepende proces bedoeld.

Zie **`man 2 <functienaam>`** voor meer informatie over deze functies.

5.5 Prioriteiten en nice-waarden

5.5.1 Prioriteiten en de run queue

De meeste moderne Linux schedulers gebruiken **priority-based pre-emptive scheduling**. Hierbij worden processen met een hogere **prioriteit vaker running** gemaakt, en kunnen ze ook een **langere timeslice** krijgen. Hierdoor krijgen processen met een hogere prioriteit meer processortijd toebedeeld dan processen met een lagere prioriteit. Aan het eind van iedere timeslice bepaalt de scheduler aan de hand van de prioriteiten van de verschillende processen in de run queue welke van hen vervolgens running wordt. Bij Linux wordt gebruik gemaakt van **nice-waarden** om de prioriteiten in te stellen.

De Linux kernel maakt vanaf versie 2.2 onderscheid in 139 prioriteiten: 1 t/m 99 zijn de zogenaamde **real-time task priorities** en 100 t/m 139 de **user task priorities**. Hoe lager de waarde, hoe hoger de prioriteit!

5.5.2 Nice-waarden

In Linux krijgen taken een zogenaamde **nice-waarde** (default 0). Hoe hoger deze nice-waarde is, des te "aardiger" is deze taak voor de andere taken, oftewel des te lager is de prioriteit van deze taak. De nice-waarde ligt tussen –20 en +19. Een normale gebruiker kan een taak alleen een positieve nice-waarde geven (met het commando **`nice`**) en de prioriteit alleen verlagen (met **`renice`**). Alleen de superuser kan taken een negatieve nice-waarde geven of de prioriteit van een taak verhogen. Zie **`man nice`**, **`man renice`** en **`man 2 nice`** voor meer informatie over de commando's en system call.

De user tasks zijn de taken met de nice-waarden -20 tot +19 (default 0). Voor deze taken geldt: **prioriteit = 120 + nice-waarde**. Er is een aparte run queue voor elke procesprioriteit.

5.5.3 Statische en dynamische prioriteit en de grootte van timeslices

De scheduling methoden sched_other en sched_batch (zie §5.4.7) maken gebruik van een **dynamische prioriteit**. Deze hangt af van een **statische prioriteit** en een **bonus**:

dynamische prioriteit = statische prioriteit – bonus + 5

met een minimale waarde van 100 en een maximale waarde van 139 en waarbij

statische prioriteit = 120 + nice-waarde

Verder krijgt een proces een **bonus**, die afhangt van de **average sleep time** (AST):

average sleep time (ms)	**bonus**
0 ≤ avg. sleep time < 100	0
100 ≤ avg. sleep time < 200	1
200 ≤ avg. sleep time < 300	2
300 ≤ avg. sleep time < 400	3
400 ≤ avg. sleep time < 500	4
500 ≤ avg. sleep time < 600	5
600 ≤ avg. sleep time < 700	6
700 ≤ avg. sleep time < 800	7
800 ≤ avg. sleep time < 900	8
900 ≤ avg. sleep time < 1000	9
avg. sleep time ≥ 1000	10

Er geldt dus: $bonus = \min\left(floor\left[\frac{AST}{100\ ms}\right], 10\right)$.

Dit betekent dat de prioriteit van de conventionele (niet real-time) processen met maximaal 5 kan worden verhoogd of verlaagd.

De **grootte van de timeslice**, ook wel **(base) time quantum** genoemd, hangt af van de statische prioriteit:

base time quantum = (140 – statische prioriteit) x 20 ms, als stat. prioriteit < 120

base time quantum = (140 – statische prioriteit) x 5 ms, als stat. prioriteit ≥ 120

Voorbeelden van het base time quantum als functie van de statische prioriteit:

statische prioriteit	**nice-waarde**	**base time quantum**
100	–20	800 ms
110	–10	600 ms
119	– 1	420 ms
120	0	100 ms
130	+10	50 ms
139	+19	5 ms

Een proces wordt als **interactief** (dus als niet-CPU-gebonden) opgevat als

dynamische prioriteit ≤ 3 x statische prioriteit/4 + 28,

dus als

statische prioriteit – bonus + 5 ≤ 3 x statische prioriteit/4 + 28,

ofwel

bonus – 5 ≥ statische prioriteit/4 – 28.

5.5.4 System calls voor prioriteiten en nice-waardes

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prty);
int nice (int increment);
```

- Met **getpriority()** en **setpriority()** wordt de prioriteit opgehaald, resp. ingesteld. Hierin wordt met which aangegeven wat er wordt bedoeld, PRIO_PROCESS, PRIO_PGRP,of PRIO_USER en met who wordt aangegeven wie hierbij hoort, de proces identifier, de identifier van de procesgroep of de identifier van de user. Met who=0 wordt het aanroepend proces, de procesgroep van het aanroepend proces, resp. de user van het aanroepend proces bedoeld.
- Met **prty** wordt de gewenste nice-waarde (-20 tot +19) aangegeven. De default waarde is 0. Alleen de superuser mag negatieve nice-waarden instellen of nice-waarden verlagen.
- De functie **getpriority()** geeft als returnwaarde de hoogste prioriteit (dus de laagste waarde) terug van de betreffende groep processen.
- Met de functie **nice()** kan de nice-waarde van het aanroepend proces met **`increment`** worden verhoogd, of, in het geval van de superuser, verlaagd.

5.6 Single- en multithreaded operating systems

Er zijn operating systems die het ook mogelijk maken dat delen van één taak concurrent kunnen worden uitgevoerd. Dit worden **threads** genoemd. De verschillende threads die behoren bij één taak hebben alle toegang tot de gemeenschappelijke globale variabelen van deze taak. Net zoals er een **task switch** bestaat tussen taken, bestaat er een **thread switch** tussen threads. Deze laatste switch is korter in uitvoeringstijd dan een switch tussen taken. De verschillende threads behoren bij dezelfde context (bij de taak behorende code, functies, data en andere resources). Er hoeft dus veel minder statusinformatie opgeslagen en uitgewisseld te worden dan bij een task switch.

5.7 Event driven scheduling voor real-time toepassingen

Real-time operating systems gebruiken geen **timeslice driven scheduling** maar standaard een **event driven scheduling** algoritme. Een event is hier dus een externe (tijdloze) gebeurtenis die via een interrupt en bijbehorende interrupt handling rou-

tine aan het systeem wordt kenbaar gemaakt. De interrupt handling routine zorgt ervoor dat een bepaalde taak running kan worden. De interrupt handling routine wordt geïnstalleerd door een driver waarnaar de taak een pad heeft gecreëerd. Indien de taak uit het betreffende pad gaat lezen, wordt deze taak sleeping als er geen data beschikbaar zijn. Op basis van een interrupt wordt aangegeven dat er data zijn en wordt de bijbehorende taak active gemaakt. Op een gegeven moment wordt deze taak running.

Bij **event driven scheduling** bepaalt de prioriteit van een taak volledig het verkrijgen van processortijd. Dit wil zeggen dat de taak met de hoogste prioriteit altijd de processor krijgt en dat de volgende taak pas running wordt bij beëindiging van de running taak of als deze zichzelf opschort (suspend) door sleeping of waiting te worden. Er is dus hier op geen enkele manier meer sprake van een timeslice!

Het voordeel van event driven scheduling is dat taken niet onderbroken worden door een task switch. Dit voorkomt ongewenste overhead die het behalen van deadlines in gevaar brengt. Uiteraard moeten we ervoor zorgen dat taken niet in een oneindige lus terechtkomen, omdat andere taken met een lagere prioriteit dan nooit meer aan bod komen. Belangrijke eis voor **event driven scheduling**: taken dienen een zo klein mogelijke en zo voorspelbaar mogelijke executietijd te hebben!

Meer over scheduling wordt besproken in het dictaat 'Real-time scheduling'.

5.8 Tuning van schedulingparameters voor real-time systemen

Taken die eerder de processor toegewezen moeten krijgen dan andere taken, moeten een hogere prioriteit krijgen dan de minder belangrijke taken. Het bepalen van de juiste waarden voor de eerder genoemde schedulingparameters bij gegeven real-time specificaties is een moeilijke zaak. In de praktijk wordt door proberen (trial & error) een mogelijke oplossing gevonden. Men noemt dit **real-time application tuning**.

Een alternatieve oplossing vormt het simuleren van het gedrag. Men moet echter wel een goed model hebben van het systeem en de optredende events uit de omgeving. Indien voor een aantal taken de juiste prioriteit is verkregen, leidt het toevoegen van een of meer taken vaak tot het **geheel herhalen** van de genoemde tuning procedure. Een en ander vormt een aanzienlijke kostenpost bij het verder aanpassen en/of uitbreiden van de real-time applicatie.

5.9 Starten van concurrent uitvoering van taken via de shell

Als we een taak willen uitvoeren, dan typen we op de terminal de naam van de uit te voeren module. Het is mogelijk om ook meerdere taken via de shell concurrent op te starten. Een voorbeeld van het concurrent starten van taken:

```
task1 &    # task starts in the background
task2 &    # task starts in the background
task3      # task starts in the foreground
```

De taken **task1** en **task2** moeten wel op de achtergrond uitgevoerd kunnen worden en geen terminal I/O plegen, anders is het niet goed meer mogelijk om via de shell commando's in te typen! De voorgrondtaak is de taak die invoer van toetsenbord en/of muis verwerkt.
We geven de namen van de taken met daarachter een '&'. Voorbeeld:

task1 & task2 & task3

De taken **task1** en **task2** worden op de achtergrond uitgevoerd en **task3** draait op de voorgrond. De shell komt pas terug na afloop van **task3**. De shell is dus in feite ook een taak binnen het operating system. Door achter **task3** geen '&' te plaatsen blijft de shell wachten totdat **task3** exit heeft gedaan. Wordt de '&' er wel achter geplaatst, dan komt de shell direct met de prompt terug. De shell komt dus direct op de voorgrond want ook **task3** wordt dan op de achtergrond uitgevoerd.

Indien we een taak een lagere prioriteit willen geven dan de default prioriteit (nice-waarde 0), dan is dit via de Linux command line te doen met het commando **nice**:

nice -5 task4

Proces **task4** krijgt nu nice-waarde 5.

Met het commando

nice --10 task4

zou **task4** als nice-waarde -10 krijgen, maar zoals gezegd kan een normale gebruiker de prioriteit van een taak alleen positief maken.

Van een (lange) runnende taak kan de nice-waarde worden aangepast met de commando's **ps** en **renice**:

```
task4 &              # task4 runs in the background with nice value 0
ps –l                # show processes; find PID of task4
renice 5 <PID>       # task4 gets nice-value 5
renice -5 <PID>      # only root can do this…
```

Let wel dat deze prioriteiten bij Unix en Linux alleen betrekking hebben op de beginwaarden. De prioriteiten worden door de kernel aangepast zodra een taak erg veel processortijd gebruikt. De prioriteit van dat proces wordt lager, al verandert de nice-waarde niet. Zodra het proces zich weer 'netjes' gedraagt, dus niet veel processortijd gebruikt, krijgt het weer een hogere prioriteit. De gebruiker heeft geen invloed op dit dynamische prioriteitsschema.

5.10 Starten van sequentiële uitvoering van taken via de shell

Taken zijn als volgt sequentieel op te starten via de shell:

task1 ; task2 ; task3

Pas als **task1** beëindigd is, zal **task2** gestart worden. Hetzelfde geldt voor **task3**. Deze wordt pas gestart na afloop van **task2**. M.b.v. de **;** kunnen we meerdere commando's op één regel opgeven. De commando's worden nu na elkaar uitgevoerd.

Meerdere taken kunnen ook conditioneel worden uitgevoerd met behulp van een logische AND (**&&**). Een volgende taak start dan alleen wanneer de vorige taak succesvol is afgesloten:

task1 && task2 && task3

Het al dan niet conditioneel sequentieel starten van taken is handig wanneer verschillende handelingen vaak achter elkaar worden uitgevoerd. Met bijvoorbeeld

make program && ./program

wordt het programma program gecompileerd en vervolgens uitgevoerd, maar alleen wanneer het compileren gelukt is.

Met het logische OR (||) kan een tweede taak worden uitgevoerd wanneer de eerste niet is geslaagd:

make program || echo "Compilation failed"

5.11 Shell commando's door taken laten uitvoeren

Met behulp van de C-functie **system()** is het mogelijk om in een programma een commando aan te bieden aan de shell. Het commando is een string zoals we die ook aan de shell hadden kunnen aanbieden via de command line. We kunnen dit nu ook doen in een programma. De bij het programma behorende taak wordt waiting en het commando zoals opgegeven in de invoerstring wordt eerst volledig uitgevoerd door de beschikbare shell. Na uitvoering van het commando door de shell wordt de exit code teruggegeven als returnwaarde van de functie **system()** en de taak gaat verder met de uitvoering van zijn code.

Voorbeeld:

system ("echo hello");

5.12 Daemons

Taken die op de achtergrond zinvolle diensten uitvoeren die door andere taken of meerdere gebruikers van nut kunnen zijn, worden **daemons** genoemd. De meeste daemons worden gestart bij het opkomen van het systeem.

Voorbeelden: daemons ten behoeve van de verzorging van email, netwerkcommunicatie en aansturing van gemeenschappelijke printers over een netwerk. Daemons voeren in het algemeen zelf geen directe communicatie met een terminal.

5.13 Deterministisch en non-deterministisch gedrag

In een multitasking systeem wordt een aantal taken quasi-parallel uitgevoerd. Een taak is op zichzelf een stuk sequentieel uit te voeren code. Bij een gegeven invoer moet de uitgevoerde taak altijd tot hetzelfde resultaat komen. Men noemt dit **deterministisch gedrag**, met andere woorden de uitvoer is volledig voorspelbaar aan de hand van de invoer en het betreffende algoritme.

Bij het quasi-parallel uitvoeren van taken ontstaat er een onderlinge verweving van de uitvoering van instructies van de verschillende taken door de processor. Om een en ander concreet toe te lichten volgt hier een sterk vereenvoudigd voorbeeld. Er zijn twee taken T1 en T2. Taak T1 bestaat uit de twee CPU instructies a en b. Taak T2 bestaat uit de CPU instructies p en q.

CPU instructies zijn atomair, dit wil zeggen dat ze niet onderbroken kunnen worden. Dit geldt ook bij het optreden van interrupts.

Taken:

```
T1:   a b
T2:   p q
```

De processor kan nu één van de volgende instructiereeksen uitvoeren:

```
a b p q   of

a p b q   of

a p q b   of

p a b q   of

.......
```

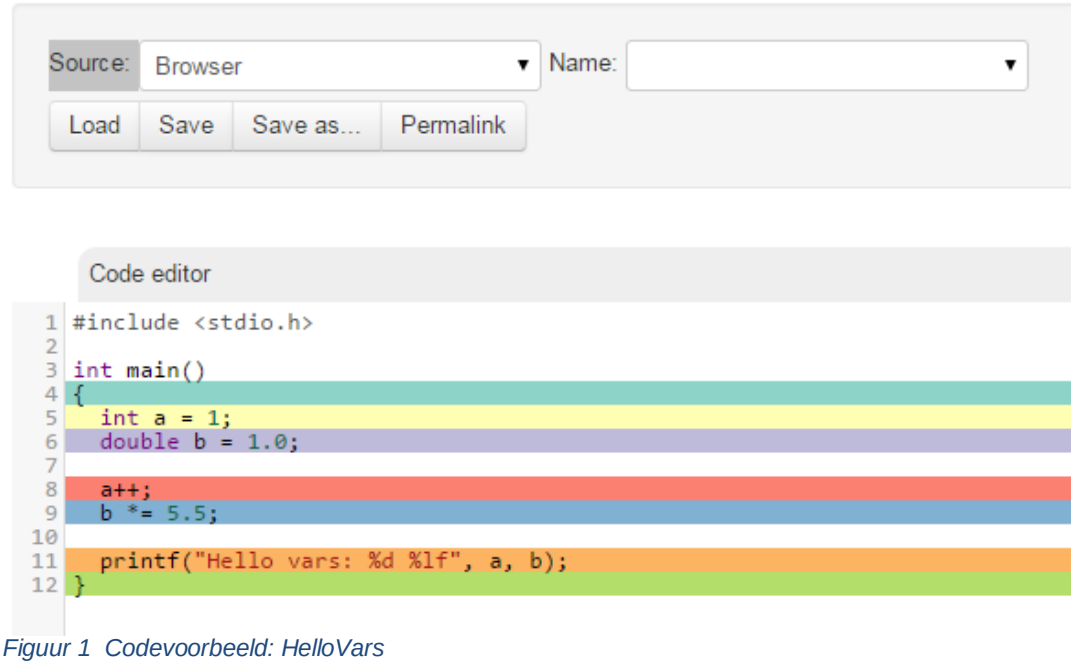
Elke mogelijke instructiereeks wordt een **trace** of **execution path** genoemd. Binnen een trace blijft het sequentiële karakter van elke *taak* behouden! De volgende trace kan dus niet:

```
a q p b
```

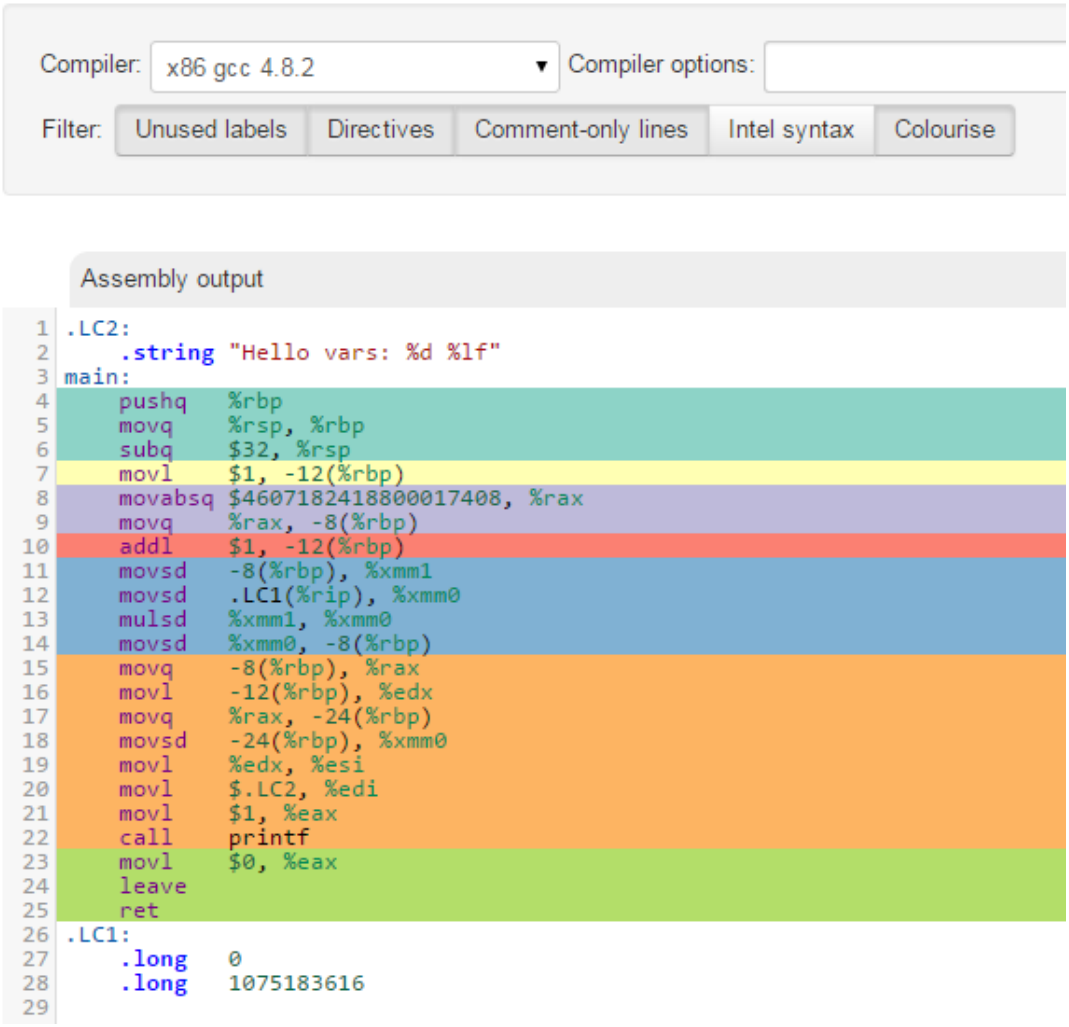
Welke trace gevolgd wordt, is niet voorspelbaar en hoeft niet reproduceerbaar te zijn als we de taken opnieuw opstarten. Hier is sprake van **non-deterministisch gedrag**. Dit non-deterministische gedrag op microniveau kan invloed hebben op het macroscopische gedrag van het systeem (input en output via de interfaces en randapparatuur) als we geen bijzondere maatregelen treffen in de code.

Statements in een C programma zijn dus niet atomair omdat ze bijna altijd resulteren in meer dan één CPU-instructie. Door te kijken naar het resultaat van het compileren

van een programma is dit zichtbaar te maken. Aan een compiler kunnen we vragen de assembler code te laten zien. De kleuren in de C-statements in Figuur 1 hieronder komen overeen met die in de (niet-geoptimaliseerde) assemblycode in Figuur 2 (bron: Interactive Compiler: https://gcc.godbolt.org). Merk op dat het vermenigvuldigen van de variabele b met 5,5 **niet atomair** is, en dus kan worden onderbroken!



Figuur 1 Codevoorbeeld: HelloVars



Figuur 2 Assembly voor HelloVars

We mogen bij het ontwerp van de verschillende taken niet uitgaan van een bepaalde (te verwachten) trace. Indien het resultaat van de uitvoering van enkele taken afhankelijk is van een bepaalde trace, dan zijn deze resultaten in principe niet voorspelbaar. Dit levert onbetrouwbaar werkende systemen op. Indien er een fout optreedt die alleen kan optreden ten gevolge van het doorlopen van één of meer bepaalde traces, dan spreekt men van een **transient error**. Het optreden van een transient error wordt dus bepaald door het moment van het optreden van de timeslice interrupt en de daarop volgende task switch. Transient errors kunnen juist optreden als verschillende taken opereren op gemeenschappelijke data. We moeten dan bijzondere maatregelen nemen.

Het begrip **race-conditie** of **data race** wordt ook vaak gebruikt: dit is de situatie waarin verschillende taken (of threads) gemeenschappelijke data gebruiken voor lezen en schrijven. Het uiteindelijke resultaat (bewerking gemeenschappelijke data) hangt of van de timing van de uitvoering van de lees- en schrijfoperaties (dus ook het moment van optreden van de task- of thread switches). Dit kan bijvoorbeeld betekenen dat de uitvoering van een programma leidt tot niet te verklaren resultaten en dat de uitvoer van een programma kan verschillen per uitvoering. Het operating system moet dus faciliteiten bieden (system calls) die we in ons programma moeten opnemen om dit soort problemen te kunnen voorkomen.

Stel ten behoeve van een magazijnadministratie kan via een aantal terminals de voorraad van artikelen worden opgevraagd en kunnen eventuele bestellingen van de voorraad worden afgeboekt. De volgende activiteiten vinden plaats in een multitasking multi-user systeem:

- Vrijwel gelijktijdig komen twee verzoeken binnen voor afboeking van hetzelfde artikel P. Van artikel P zijn er 5 in voorraad voordat deze bestellingen binnenkomen.
  {aantal P = 5}

- Via terminal A worden er van artikel P 4 besteld. Echter voordat deze 4 zijn afgeschreven, komt een timeslice interrupt en komt er een bestelling via terminal B binnen. Hier worden er van artikel P 3 besteld.
  {aantal P = 5 - 3 = 2}

- Na enige tijd wordt de bestelling van terminal A verder afgewerkt.
  {aantal P = 5 - 4 = 1}

De voorraadadministratie geeft nu aan dat er van artikel P nog 1 in voorraad is en dit terwijl in werkelijkheid een tekort van 2 stuks is ontstaan (som van de bestellingen is 4 + 3 = 7, voorraad 5 - 7 = -2). Het resultaat is dus afhankelijk van het moment van optreden van de task switch. Het kan dus best voorkomen dat de applicatie op bepaalde momenten de juiste uitvoer levert, maar dat is dus niet altijd gegarandeerd.

Om dit soort problemen te vermijden moet het lezen, modificeren en daarna weer terugschrijven van gemeenschappelijke data een **niet te onderbreken actie** of **atomaire actie** zijn. Een dergelijke actie wordt ook wel een **kritieke actie** genoemd. Dit geldt alleen voor taken die op dezelfde gemeenschappelijke data opereren. Een kritieke actie mag dus best onderbroken worden door de uitvoering van code die niets met de gemeenschappelijke data te maken heeft. Om dergelijke problemen het hoofd te kunnen bieden zijn **semaforen** bedacht (zie hoofdstuk 10) om de correcte werking in de code af te kunnen dwingen.

5.14 Opgaven

1. Kan een taak die zich in de sleeping toestand bevindt, zichzelf running maken?

2. Wanneer is de task switch time afhankelijk van het aantal taken in de active queue?

3. Geef in een figuur duidelijk aan hoe de task switch plaatsvindt zoals geschetst in de C-functie **TaskSwitch()**.
   Geef per programmaregel aan waar de pointers naar wijzen.

4. Geef in pseudo-code een ontwerp van de interrupt service routine die de timeslice interrupt afhandelt. Ga hier alleen uit van de running en active toestand.

6 Command-line parameters, processen en threads

6.1 Intertaakcommunicatie met command-line parameters

Communicatie tussen taken is een essentiële functie die door een operating system mogelijk wordt gemaakt. Bij het opstarten van een taak vanaf de command line kunnen we aan deze taak diverse parameters meegeven, bv.:

prog hello 25 x

In dit geval geeft de shell deze parameters door aan de taak prog. Als de source code van prog een C-programma is, kan dit parametergebied (de command line) binnen dit programma worden uitgelezen met behulp van de command line argumenten **argc**, **argv** en **envp**.

De taak prog kan de betreffende parameters bereiken, mits die in zijn header in de juiste volgorde zijn opgesomd. De taak prog is in feite ook een functie, namelijk main():

```
/************************************************ prog.c */
int main (int argc, char *argv[], char *envp[]) {
    ----
    return 0;
}
```

- Parameter **argc** geeft het aantal argumenten weer, in dit voorbeeld 4, immers de naam van het programma wordt ook als argument doorgegeven en dus meegeteld.
- Parameter **argv** is het beginadres van een array van pointers naar characters. De opeenvolgende pointers van deze array wijzen naar de opeenvolgende parameters. Hierbij is argv[0] de naam van het programma en bevatten de argv-elementen 1 t/m argc-1 (in dit voorbeeld 1-3) de argumenten (zie fig. 6.1).
- Parameter **envp** is eveneens een beginadres van een array van pointers naar characters, maar deze pointers wijzen naar diverse zogenaamde **environment variables**. Deze bevatten de naam van de home directory, de naam van de shell, de user name en veel meer.
- Beide arrays worden afgesloten met de NULL pointer.

In het programma zijn de command-line parameters als volgt te bereiken:

```
int iAantal = argc;                    /* iAantal wordt 4  */
char cKar1 = *(argv[0] + 3);           /* cKar1 wordt 'g'  */
char cKar2 = argv[1][4];               /* cKar2 wordt 'o'  */
printf ("Arg2 = %s\n", argv[2]);
                /* De string "Arg2 = 25" wordt afgedrukt* /
```

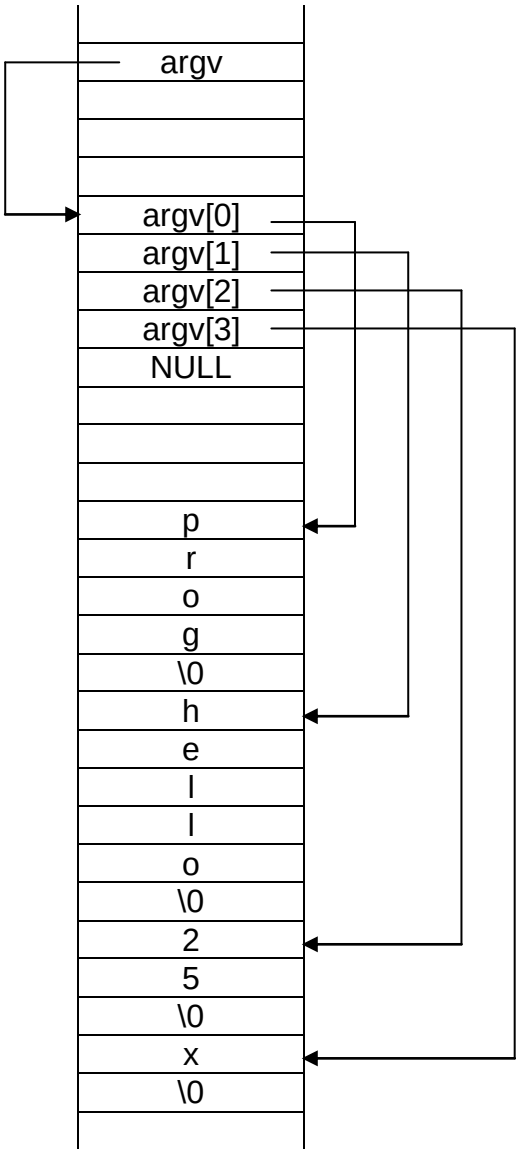


Fig. 6.1 Command line parameters van "prog hello 25 x"

6.2 Returnparameters van system calls, errno en perror()

Als een system call een fout ontdekt, wordt gewoonlijk de waarde **-1** als **returnwaarde** aan het aanroepend programma teruggegeven en krijgt de globale variabele **errno** een waarde die het type fout aangeeft. De gedefinieerde foutmeldingen zijn te vinden via **`man 3 errno`** en de file **`errno.h`**.

Als de system call met succes wordt uitgevoerd, wordt errno nooit gewijzigd. De waarde van errno blijft dus gelijk zolang er geen andere fout optreedt. De variabele errno mag dus alleen worden geraadpleegd na het optreden van een fout!

De system call **perror()** drukt de foutboodschap horende bij de waarde van errno af naar de standard error file descriptor, meestal het display (zie hoofdstuk 8). Als het foutnummer niet wordt herkend, wordt de boodschap "Unknown error: " afgedrukt, gevolgd door de decimale waarde van errno.

Het prototype van deze functie (die gedeclareerd is in stdio.h) is

void perror (const char *pcFoutboodschap);

Zie **`man 3 perror`** voor details.

6.3 Processen creëren met fork() en exec()

Zie §12.4 van het boek *Linux for programmers and users* voor achtergrondinformatie.

6.3.1 Parent-child relatie, fork() en wait()

Linux biedt taken de mogelijkheid zelf nieuwe taken te creëren. Men duidt dit aan met een **parent-child relatie**. In principe erft de child alle gegevens van zijn parent. Er wordt gebruik gemaakt van hetzelfde text segment (codegebied) en user data segment (datagebied). Wel krijgt de child een eigen procesidentificatie en een andere parent proces ID (die van zijn parent). De parent van een child heeft zelf immers een ander proces als parent. Een child erft dus onder andere alle variabelen en open file descriptors van zijn parent. Zodra de child (of de parent) een variabele wijzigt, wordt er eerst een kopie van het datagebied gemaakt t.b.v. de child. Dit betekent dat parent en child niet van elkaars gewijzigde data gebruik kunnen maken. Is dat wel nodig, dan moet er van zogenaamd shared memory gebruik worden gemaakt (zie hoofdstuk 8). Parent en child kunnen elkaar ook informatie toezenden via pipes (zie hoofdstuk 7) en door middel van signals (zie hoofdstuk 9). De child wordt in de active queue gezet en werkt dus parallel met zijn parent en eventuele andere processen.

Het afsplitsen van een child taak vanuit een parent taak wordt uitgevoerd met de system call **fork()**. Het functieprototype luidt:

pid_t fork (void);

Elk proces krijgt een eigen unieke procesidentificatie, de **process ID**.

Bij het commando fork() splitst het proces zich in twee identieke processen. Het bestaande proces dat de fork() uitvoerde, is het parent process en het nieuwe proces is het child process. De parent kan opnieuw children afsplitsen en ook kan de child op zijn beurt weer children afsplitsen. Elke nieuwe child krijgt van de kernel een uniek process ID.

Parent en child voeren vanaf de fork() beide dezelfde code uit. Nadat de system call fork() is uitgevoerd, gaat zowel het parent proces als het child proces verder met de instructie die op de fork() call volgt. Alleen krijgt de child 0 als returnwaarde van de system call fork(), terwijl de parent als returnwaarde de process ID van de child ontvangt. De parent en de child zijn twee concurrent processen.

Als de returnwaarde van fork() -1 is, is de afsplitsing van een nieuwe taak niet gelukt. In dat geval is er geen child proces en wordt er een foutcode in de globale variabele **errno** gezet (zie **`man 3 errno`**). Het kan bv. zijn dat er te weinig vrij geheugen is.

Fig. 6.2 laat zien wat het effect van fork() is.

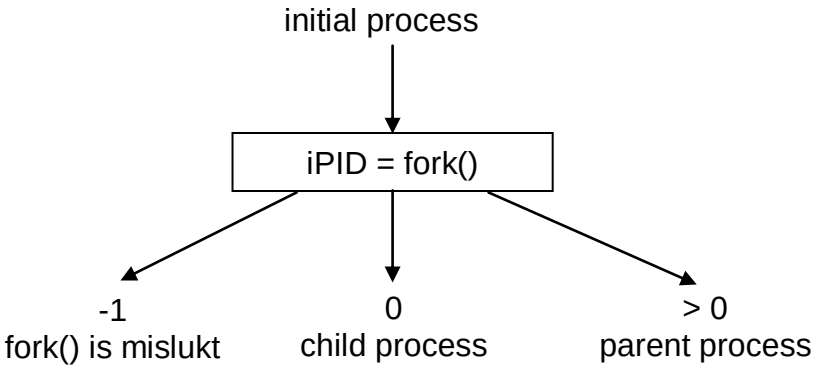


Fig. 6.2 Het uitvoeren van de system call fork()

Voorbeeld 6.3.1: het forken van een proces

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main (void)
{
    printf ("PID van initieel proces: %d\n", (int) getpid ());
                      /*tot hier bestaat alleen het initiële proces*/
    switch (fork ())
                         /*zowel parent als child voeren switch uit*/
    {
        case -1:                                      /*initiële proces*/
            printf ("Helaas is de fork mislukt!\n");
            break;
        case 0:                                                 /*child*/
            printf ("Child: My PID = %d\n", (int) getpid ());
            printf ("Child: My parent's PID = %d\n",
                     (int) getppid ());
```

```
            break;
        default:                                              /*parent*/
            printf ("Parent: My PID = %d\n", (int) getpid ());
            wait (NULL);
            break;
    }                          /*vanaf hier zowel parent als child*/
    printf ("Proces met PID %d stopt nu.\n", (int) getpid ());
    return 0;
}
```

Het resultaat van het bovenstaande programma ziet er bv. als volgt uit:

```
PID van initieel proces: 3806
Child: My PID = 3807
Child: My parent's PID = 3806
Proces met PID 3807 stopt nu.
Parent: My PID = 3806
Proces met PID 3806 stopt nu.
```

We zien in deze situatie dat het initiële proces als PID 3806 krijgt en de child 3807. De parent van de child heeft als PID 3806. Dit is indedaad de PID van het initiële proces. De child stopt nadat deze zijn gegevens heeft afgedrukt. Nu krijgt de parent weer processortijd en drukt zijn PID (natuurlijk weer 3806) af en stopt vervolgens.

M.b.v. de system call getpid() kan een proces zijn eigen PID opvragen. Met de system call getppid() vraagt een proces de PID van zijn parent op.

pid_t getpid (void);
pid_t getppid (void);

Een parent proces wacht doorgaans op het stoppen van een childproces om de exit status van de child op te vragen. Dit gebeurt met een van de system calls

pid_t wait (int *piStatus);
pid_t waitpid (pid_t iPID, int *piStatus, int iOptions);

In beide gevallen wordt gewacht op het einde van het child proces als dat nog niet is gestopt. De returnwaarde is de PID van de child die gestopt is of -1 in geval van een fout. De foutcode is dan te vinden in de globale variabele **errno**. De variabele piStatus is een pointer naar de locatie waar de exit status van de child wordt opgeslagen (door de kernel). Als deze pointer NULL is, wordt de exit status van de child niet opgeslagen.

iPID geeft aan op het einde van op welke child wordt gewacht.

Met iOptions kunnen opties worden meegegeven met de system call, bv. om aan te geven dat de parent niet gaat wachten als er nog geen child is beëindigd. We gaan hier niet verder op in.

De mogelijke waarden en betekenis van iPID:

< -1	wacht op een child met PGID = -iPID
-1	wacht op een child met willekeurige PID

0 wacht op een child met PGID = PGID van aanroepend proces
> 0 wacht op een child met PID = iPID

Zie §3.26.7 en §12.5.11 van het boek *Linux for programmers and users* voor meer informatie over Group ID's.

De exit status van een proces blijft bewaard totdat de parent deze informatie leest (via een wait system call). Als een child beëindigd wordt, voordat de parent een wait gedaan heeft, wordt de child een zogenaamd **zombie** proces (zie bijlage 2). Alle resources van de child zijn vrijgegeven behalve de procesdatastructuur waarin zijn exit status staat (zie ook §12.4.4 van het boek *Linux for programmers and users*).

Voorbeeld 6.3.2: parent met twee children en een grandchild

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main (void)
{
    printf ("PID van initieel proces: %d\n", (int) getpid ());
                    /*tot hier bestaat alleen het initiële proces*/
    switch (fork ())
                     /*zowel parent als child 1 voeren switch uit*/
    {
        case -1:                                    /*initiële proces*/
            printf ("Helaas is de fork mislukt!\n");
            break;
        case 0:                                                /*child 1*/
            printf ("Child 1: My PID = %d\n", (int) getpid ());
            printf ("Child 1: My parent's PID = %d\n",
                     (int) getppid ());
            switch (fork ())
                  /*zowel child 1 als grandchild voeren switch uit*/
            {
                case -1:                                       /*child 1*/
                    printf ("Fork tbv grandchild is mislukt!\n");
                    break;
                case 0:                                  /*grandchild*/
                    printf ("Grandchild: My PID = %d\n",
                             (int) getpid ());
                    printf ("Grandchild: My parent's PID = %d\n",
                             (int) getppid ());
                    break;
                default:                                       /*child 1*/
                    printf ("Child 1 na fork: My PID = %d\n",
                             (int) getpid ());
                    printf ("Grandchild met PID %d is gestopt\n",
                             wait (NULL));
                    break;
            }
            break;
        default:                                                /*parent*/
```

```
            printf ("\nParent na 1e fork: My PID = %d\n",
                    (int) getpid ());
            switch (fork ())
                     /*zowel parent als child 2 voeren switch uit*/
            {
                case -1:                                    /*parent*/
                    printf ("Tweede fork is mislukt!\n");
                    break;
                case 0:                                    /*child 2*/
                    printf ("Child 2: My PID = %d\n",
                             (int) getpid ());
                    printf ("Child 2: My parent's PID = %d\n",
                             (int) getppid ());
                    break;
                default:                                    /*parent*/
                    printf ("\nParent na 2e fork: My PID = %d\n",
                             (int) getpid ());
                    printf ("Child met PID %d gestopt\n",
                             wait (NULL));
                    printf ("Child met PID %d gestopt\n",
                             wait (NULL));
                    break;
            }
            break;
    }
        /*vanaf hier zowel parent, child 1, child 2 als grandchild*/
    printf ("Proces met PID %d stopt nu.\n", (int) getpid ());
    return 0;
}
```

Het resultaat van het bovenstaande programma ziet er bv. als volgt uit:

```
PID van initieel process: 3771
Child 1: My PID = 3772
Child 1: My parent's PID = 3771
Grandchild: My PID = 3773
Grandchild: My parent's PID = 3772
Proces met PID 3773 stopt nu.
Child 1 na fork: My PID = 3772
Grandchild met PID 3773 is gestopt
Proces met PID 3772 stopt nu.

Parent na 1e fork: My PID = 3771
Child 2: My PID = 3774
Child 2: My parent's PID = 3771
Proces met PID 3774 stopt nu.

Parent na 2e fork: My PID = 3771
Child met PID 3772 gestopt
Child met PID 3774 gestopt
Proces met PID 3771 stopt nu.
```

Het initiële proces krijgt hier PID 3771. We zien dat bij een fork() de child meteen processortijd krijgt. Child 1 krijgt als PID 3772 (en zijn parent heeft PID 3771 en is dus gelijk aan die van het initële proces). De grandchild krijgt PID 3773 en heeft child 1als pa-

rent (met PID 3772). De grandchild stopt na het afdrukken van zijn gegevens en het vermelden dat hij nu stopt (PID 3773). Hij voert hiervoor de laatste printf() van het programma uit. Child 1 gaat verder met het nogmaals afdrukken van zijn PID (3772) na de fork(). Child 1 wacht nu op het einde van zijn child, de grandchild (PID 3773) en geeft aan dat hij zelf nu ook stopt (PID 3772). Dit gebeurt weer met de laatste printf() van het programma.

Nu gaat de parent weer verder met het afdrukken van zijn PID (3771) na de 1e fork(). Hierna doet de parent een tweede fork() t.b.v. child 2. Child 2 krijgt nu processortijd en drukt zijn PID (3774) af en die van zijn parent (3771). Hierna geeft hij aan dat hij (PID 3774) nu stopt, weer met de laatste printf() van het programma.

De parent gaat verder met het afdrukken van zijn PID (3771) na de 2e fork() en gaat wachten op het einde van zijn 2 children. Zoals we al gezien hebben is child 1 het eerst gestopt en deze PID (3772) wordt dan ook het eerst afgedrukt, gevolgd door de PID (3774) van child 2. Tot slot geeft ook de parent aan dat hij nu stopt (PID 3771).

De laatste printf() van het programma wordt dus inderdaad 4 keer uitgevoerd, namelijk door de grandchild, child 1, child 2 en de parent.

6.3.2 Return versus exit()

Een proces kan zijn exit status aan de kernel doorgeven met de system call

void exit (int iStatus);

of met

return iStatus;

Normaal is iStatus = 0 (EXIT_SUCCESS) als het proces zonder fouten wordt afgesloten en iStatus = 1 (EXIT_FAILURE) als er wel een fout is opgetreden. iStatus is een 8-bits getal (0 … 255).

Bij gebruik in functies is er echter een verschil tussen exit() en return: met behulp van return wordt vanuit de functie teruggekeerd naar het aanroepend proces, dus één niveau hoger. Bij gebruik van exit() wordt het main proces beëindigd van waaruit de functie is aangeroepen, ongeacht hoe diep deze functieaanroep zich in het proces bevindt. Zie bijlage 3 voor nadere demonstratie van deze effecten.

Bij het gebruik van exit() moet de header file stdlib.h aan het programma worden toegevoegd (zie **`man 3 exit`**).

6.3.3 Nieuwe taak starten met exec()

De enige manier om in Linux een ander proces dan een kopie van zichzelf op te starten is het gebruik van de system call exec(). Bij exec() wordt echter niet meer teruggekeerd naar het oorspronkelijke proces. Nu worden het programmasegment en het

datasegment van de oorspronkelijke child overschreven met het programmasegment en het datasegment van het programma waarvan de (pointer naar de) naam als parameter wordt meegegeven met de system call exec(). Er kunnen nog meer parameters worden meegegeven die in het nieuwe programma op dezelfde wijze toegankelijk zijn als bij de command line parameters.

De objectcode voor de child moet reeds bestaan. Indien de objectcode-module van de child nog niet in het werkgeheugen staat, wordt deze van het externe geheugen geladen.

Linux kent 6 leden van exec()-familie van functies (zie **`man 3 exec`**):

```
int execl     (const char *pathName, const char *arg0,
               const char *arg1, const char *arg2, …,
               const char *argn, (char *) NULL);

int execlp    (const char *cmdName, const char *arg0,
               const char *arg1, const char *arg2, …,
               const char *argn, (char *) NULL);

int execle    (const char *pathName, const char *arg0,
               const char *arg1, const char *arg2, …,
               const char *argn, (char *) NULL,
               char * const envp[ ]);

int execv     (const char *pathName, char * const argv[ ]);

int execvp    (const char *cmdName, char * const argv[ ]);

int execve    (const char *pathName, char * const argv[ ],
               char * const envp[ ]);
```

De letter **l** staat voor het opgeven van een argument **l**ist, **v** voor het opgeven van een arg**v**[ ]-array, **p** voor het gebruik van de $**P**ATH environment variable en **e** voor het opgeven van een **e**nvp[ ]-array in de call.

- **pathName** is de pointer naar het volledige pad van het commando dat moet worden uitgevoerd. Hierna volgen de pointers argi (i = 0 tot n) naar de parameters die aan het commando meegegeven worden, afgesloten met een NULL pointer. Let wel dat arg0 wijst naar de naam van het commando zelf!

- **cmdName** is de pointer naar de naam van het commando dat moet worden uitgevoerd. Als de naam geen slash (/) bevat, wordt de naam in het standaard zoekpad gezocht.

- **argv** is het beginadres van de array van pointers naar command line parameters. Deze array moet met een NULL pointer zijn afgesloten.

- **envp** is het beginadres van de array van pointers naar environment parameters. Deze pointers wijzen naar strings. De array moet afgesloten zijn met een

NULL pointer. De exec() system calls die geen envp parameter bevatten, starten het nieuwe proces op met dezelfde environment als het huidige proces.

- De **returnwaarde** is -1 als er een fout is opgetreden. De globale variabele errno krijgt een waarde die aangeeft om welke fout het gaat. Dit gebeurt alleen als de system call exec() terugkeert. Als er geen fout optreedt, wordt immers niet teruggekeerd in het parentprogramma.

Voorbeeld 6.3.3: communicatie tussen parent en child via command-line parameters

```
/* parent.c ************************************************/
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main (void)
{
    const char acToChild[] = "Groeten van parent!";
    switch (fork ())
    {
        case -1:
            printf ("Helaas is de fork mislukt!\n");
            break;
        case 0:                                          /*child*/
            execl ("./child", "child", acToChild,
                   (char *)NULL);
            printf ("Opstarten van child is mislukt!\n");
            break;
        default:                                        /*parent*/
            (void) wait (NULL);
    }
    return 0;
}

/* child.c *************************************************/
#include <stdio.h>

int main (int argc, char *argv[])
{
    printf ("Ontvangen van parent:\n");
    printf ("\tcommand line parameter 0: %s\n", argv[0]);
    printf ("\tcommand line parameter 1: %s\n", argv[1]);
    return 0;
}
```

Het resultaat ziet er als volgt uit:

```
Ontvangen van parent:
    command line parameter 0: child
    command line parameter 1: Groeten van parent!
```

We zien dat de child inderdaad de beide command line parameters afdrukt.

6.4 Real-time eisen en forken van taken

Het forken van children brengt een **overhead** met zich mee. Dit kan betekenen dat het forken veel tijd kost tijdens de uitvoering van een programma, wat het halen van de gestelde deadlines in gevaar kan brengen. Voor real-time toepassingen moet men dus altijd letten op de benodigde tijd voor het uitvoeren van de fork-functie en het laden van de codemodule van de child in het werkgeheugen. Belangrijke tijdwinst is te behalen door de child-modules al in een **initialisatiefase** in het werkgeheugen te laden voordat de hoofdtaak wordt gestart. Dit kan bijvoorbeeld door een taak bij opstarten al te forken en vervolgens te suspenden, zodat deze mbv. een signal kan worden gewekt zodra deze nodig is. Deze aanpak draagt ertoe bij dat het systeem zich meer voorspelbaar gedraagt.

Het van tevoren forken van alle benodigde taken kan in een systeem met een **klein intern geheugen** ertoe leiden dat deze taken niet allemaal in het geheugen passen. We moeten dan zorgvuldig een keuze maken van taken die we van tevoren laden en op welke momenten ze verwijderd moeten worden om ruimte te maken voor andere taken. Uiteraard dienen we ervoor te zorgen dat taken die een zeer kritieke functie moeten uitvoeren, altijd kunnen beschikken over voldoende geheugenruimte.

6.5 Multithreading in Linux

Bij het gebruik van multithreading moet bij het linken de optie -pthread (POSIX threads) worden gebruikt! Zie `man 7 pthreads` voor achtergrondinformatie.

Een **thread of execution** (executiepad) is het kleinste programmaonderdeel dat door een operating system kan worden gescheduled. Zo resulteert een fork() in twee afzonderlijke taken (processen) die parallel worden uitgevoerd. Veelal wordt de fork() gevolgd door een exec() call waarbij de programmacode van de child wordt vervangen door code van een ander programma. Dit kost veel tijd, omdat de programmacode van disk moet worden geladen. Ook de task switch kost veel tijd, omdat de hele processorcontext van de ene taak moet worden gered en van de andere hersteld. Zodra de data worden veranderd, moet ook het datagebied (t.b.v. de child) worden gekopieerd. Al met al kan een fork/exec/task switch-procedure een **grote overhead** opleveren.

Net als bij veel andere operating systems kan er ook bij Linux binnen één proces **parallellisme** worden bewerkstelligd. Dit gebeurt door binnen één proces meerdere **threads** executiepaden parallel (in timesharing) uit te voeren. Men spreekt hierbij van **multithreading**. In een thread wordt een **C-functie** uitgevoerd. De functie main() wordt wel aangeduid als de **main thread**. De main thread kan andere threads creëren, deze worden aangeduid als **worker threads**.

Threads maken gebruik van hetzelfde geheugengebied (zowel voor de programmacode als voor de globale data), file descriptors, signals en signal handlers. Ze krijgen een eigen program counter, stack, stack pointer en thread identifier. Het aanmaken van een thread kost weinig tijd en een thread switch evenmin. Er hoeven immers geen data- of programmagebieden te worden aangemaakt en bij een thread switch hoeft slechts weinig te worden gered en hersteld.

Een voordeel van multithreading is dat het proces sneller op een multiprocessorsysteem kan worden uitgevoerd. Een ander voordeel is dat threads geblokkeerd worden als ze wachten op (relatief trage) I/O. Andere threads kunnen dan gewoon verder werken, zodat niet het hele proces wordt geblokkeerd. Een nadeel kan zijn dat het programma complexer wordt.

In Linux wordt met de call **pthread_create()** een thread aangemaakt:

```
int  pthread_create (pthread_t *pThreadID,
                     const pthread_attr_t *pAttributes,
                     void *(*RoutineAdres)(void*),
                     void *pRoutineArgument);
```

- De **returnwaarde** is 0 als de thread succesvol is aangemaakt. Bij een fout wordt een foutcode (errno) teruggegeven. De thread bestaat dan niet.
- Op adres **pThreadID** wordt de thread ID opgeslagen. Het datatype pthread_t is in feite een unsigned long integer.
- Het adres **pAttributes** geeft aan op welke locatie de attributes staan t.b.v. de nieuwe thread. Wordt hier NULL ingevoerd, dan krijgt de thread de default attributes.
- **RoutineAdres** is het beginadres van de functie die als thread zal worden uitgevoerd. Als de threadroutine eindigt met een returnwaarde, is het effect alsof er een impliciete system call pthread_exit() wordt uitgevoerd die de returnwaarde van de threadroutine als inputparameter (= adres van de exit status) gebruikt.
- De pointer **pRoutineArgument** is de enige parameter van de threadroutine. Dit is **altijd een void pointer**, en zal dus in veel gevallen moeten worden getypecast. In de threadroutine moet het argument dan weer worden teruggecast van `void*` naar het oorspronkelijke type.
- Merk op dat pthread_create() een thread **aanmaakt**, maar **niet uitvoert**. Wanneer een thread wordt gestart hangt af van de scheduler!

Het prototype van de system call pthread_exit() is:

```
void  pthread_exit (void *pExitStatus);
```

Deze functie beëindigt de thread die deze call uitvoert. Het adres pExitStatus wordt ter beschikking gesteld aan de functie pthread_join(), mits deze natuurlijk succesvol wordt uitgevoerd.

```
int  pthread_join (pthread_t ThreadID, void **pAdresExitStatus);
```

Deze functie zorgt ervoor dat het proces wacht op het einde van de thread met het ID **ThreadID**. Om de statusinformatie (de returnwaarde) van de betreffende thread te kunnen gebruiken, moet aan pthread_join() als tweede parameter het adres **pAdresExitStatus** worden meegegeven van de locatie waar die returnwaarde (zijnde een pointer) kan worden opgeslagen. Via die pointer is de werkelijk exitstatus van de thread vervolgens te bereiken. Wordt de exitstatus niet gebruikt, dan kan als tweede inputparameter de pointer NULL worden opgegeven.

Een thread stopt door de betreffende routine te laten eindigen met return, exit() of pthread_exit() of als het proces stopt dat de thread heeft aangemaakt. Let wel dat exit() het hele proces afbreekt, inclusief alle threads! Met de call pthread_exit() wordt alleen de betreffende thread afgebroken.

De functie pthread_join() is dus nodig om te voorkomen dat de threads worden afgebroken als het hoofdprogramma stopt terwijl de threads nog niet zijn voltooid.

In het volgende voorbeeld worden twee threads aangemaakt. Elke thread drukt de informatie af die hij van het aanroepend proces meekrijgt.

Voorbeeld 6.5.1: twee threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *Threadroutine (void *pArgAdr);

int main (void) {
    pthread_t ThreadID_A;
    pthread_t ThreadID_B;
    const char ThreadArgA[] = "Ik ben thread A.";
    const char ThreadArgB[] = "Ik ben thread B.";

    if (pthread_create (&ThreadID_A, NULL, Threadroutine,
                        (void*) ThreadArgA)) {
        fprintf (stderr, "Geen thread A\n");
        exit (EXIT_FAILURE);
    }
    if (pthread_create (&ThreadID_B, NULL, Threadroutine,
                        (void*) ThreadArgB)) {
        fprintf (stderr, "Geen thread B\n");
        exit (EXIT_FAILURE);
    }

    printf ("Thread A heeft ID %lu\n", ThreadID_A);
    printf ("Thread B heeft ID %lu\n", ThreadID_B);
    pthread_join (ThreadID_A, NULL);
    pthread_join (ThreadID_B, NULL);

    return 0;
}

void *Threadroutine (void *pArgAdr) {
    printf ("%s\n", (char *) pArgAdr);
    return NULL;
}
```

Merk op dat in de pthread_create() calls het **argument van de threadfunctie** moet worden getypecast naar **`void*`**. In de threadfunctie wordt het argument weer teruggecast naar het oorspronkelijke type (**`char*`**) om te kunnen worden afgedrukt.

Het resultaat van het bovenstaande programma ziet er bijvoorbeeld als volgt uit:

```
Thread A heeft ID 140464516097792
Thread B heeft ID 140464507705088
Ik ben thread A.
Ik ben thread B.
```

Ook al wordt Thread A eerder gecreëerd dan Thread B dan betekent dit niet dat Thread A eerder dan Thread B wordt uitgevoerd. Dit bepaalt de scheduler! Het resultaat kan er dus ook als volgt uitzien:

```
Thread A heeft ID 140316429874944
Thread B heeft ID 140316421482240
Ik ben thread B.
Ik ben thread A.
```

Bij herhaald uitvoeren kan het resultaat verschillend kan zijn; het gedrag is **niet-deterministisch**.

Een ander opvallend detail is het gebruik van printf() naar stdout als gemeenschappelijke resource in de twee threads. Zie paragraaf 6.6.

In het volgende voorbeeld creëren we drie threads. Elke thread drukt 10 keer een X, Y, resp. Z af. De verschillende wachttijden zorgen ervoor dat de threads voldoende lang duren om elkaar te onderbreken en bovendien een verschillende executietijd hebben. Twee threads geven op een verschillende wijze hun statusinformatie terug. We willen hiermee nagaan of de returnwaarde (de exit status) van de thread in deze twee situaties via thread_exit() en thread_join() wordt teruggegeven aan het aanroepend proces. Dat blijkt het geval te zijn (zie het resultaat van het programma).

Voorbeeld 6.5.2: drie threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

void mSleep (int iMS);
void *ThreadX (void *pArgX);
void *ThreadY (void *pArgY);
void *ThreadZ (void *pArgZ);

int iStatX, iStatY, iStatZ;

int main (void) {
    pthread_t threadID_X, threadID_Y, threadID_Z;
    char cL1 = 'X', cL2 = 'Y', cL3 = 'Z';
    int *piExX, *piExY, *piExZ;
```

```
    (void) pthread_create(&threadID_X, NULL, ThreadX, (void*) &cL1);
    (void) pthread_create(&threadID_Y, NULL, ThreadY, (void*) &cL2);
    (void) pthread_create(&threadID_Z, NULL, ThreadZ, (void*) &cL3);

    pthread_join (threadID_X, (void*) &piExX);
    pthread_join (threadID_Y, (void*) &piExY);
    pthread_join (threadID_Z, (void*) &piExZ);

    printf ("\nStatus van thread X = %d", *piExX);
    printf ("\nStatus van thread Y = %d", *piExY);
    printf ("\nStatus van thread Z = %d\n", *piExZ);

    return 0;
}

void *ThreadX (void *pArgX) {
    int i;
    iStatX = 21;
    for (i = 0; i < 10; i++) {
        write(1, (char *) pArgX, 1);
        mSleep (13);
    }
    return &iStatX;
}

void *ThreadY (void *pArgY) {
    int i;
    iStatY = 22;
    for (i = 0; i < 10; i++) {
        write(1, (char *) pArgY, 1);
        mSleep (27);
    }
    return &iStatY;
}

void *ThreadZ (void *pArgZ) {
    int i;
    iStatZ = 23;
    for (i = 0; i < 10; i++) {
        write(1, (char *) pArgZ, 1);
        mSleep (53);
    }
    pthread_exit (&iStatZ);
}

void mSleep (int iMS) {
    struct timespec tWachtTijd = {0,0};

    tWachtTijd.tv_sec = 0;                /*wachttijd in s */
    tWachtTijd.tv_nsec = iMS * 1000000;   /*wachttijd in ns*/
    nanosleep (&tWachtTijd, NULL);
}
```

Het resultaat van het programma kan er als volgt uitzien:

```
XYZXXYXZXYXYXXZYXXYZYYZYYZZZZZ
Status van thread X = 21
Status van thread Y = 22
Status van thread Z = 23
```

De volgorde van de karakters kan per uitvoering verschillen.

Merk op dat de **returnwaarden van de threadfuncties** worden teruggeleverd aan de routine die de thread opstartte via de pthread_join() call. Het type van het argument in pthread_join() is **`void*`**, en dus moet ook het type van de threadfunctie **`void`** zijn (een functie is een pointer naar het beginadres van die functie).

Het lijkt nogal omslachtig om de uitvoerparameter van een thread via een pointer te laten overdragen als de thread ook direct bij de globale variabelen van het proces kan. Het kan voorkomen dat de parameters die de thread overdraagt een variabele omvang hebben, en dat de benodigde geheugenruimte hiervoor dus door het proces dynamisch wordt geallocceerd. De functie malloc() levert het beginadres op van het toegewezen geheugengebied met de gewenste grootte. Dit beginadres moet dan overgedragen worden aan de thread die dit datagebied moet vullen.

De uitvoerparameters mogen niet lokaal in de thread worden gedeclareerd, omdat deze parameters niet meer bestaan zodra de thread wordt verlaten. Het locale geheugen van de thread is dan immers weer vrijgegeven.

Wordt de instructie **`return &iStatY`** vervangen door **`exit (iStatY)`**, dan wordt het resultaat van het programma als volgt:

```
XYZXXYXZXYXYXXZYXXYZYYZYYZ
```

Hieruit zien we dat de routine exit() inderdaad het complete proces beëindigt, immers thread Z heeft zijn taak nog niet voltooid. Er zijn slechts 7 i.p.v. 10 Z's afgedrukt. Ook worden de printf()-routines niet uitgevoerd, want tijdens het wachten op het einde van thread Y, beëindigt deze thread het hele proces.

Het komt vaak voor dat threads kritieke acties uitvoeren die elkaar wederzijds uitsluiten of dat threads moeten worden gesynchroniseerd. In dat geval moeten we **semaforen** gebruiken. Zie hiervoor Hoofdstuk 10.

6.6 Reentrant en threadsafe functies

Indien we in verschillende threads dezelfde functie aanroepen, moet deze functie daarvoor geschikt zijn. Indien in een thread de functie func() in uitvoering is, dan moet een andere thread ook deze functie kunnen starten. Deze functie moet **reentrant** zijn. We moeten dan het gebruik van lokale static variabelen en het teruggeven van pointerwaarden naar deze lokale static variabelen voorkomen, omdat deze in elke thread hetzelfde zijn (=gemeenschappelijk). Tevens moet een functie geen andere non-reentrant functies aanroepen om zelf reentrant te blijven.

In multithreaded applicaties moeten we vaststellen of er geen **gemeenschappelijk resources** door de functie func() intern in verschillende threads worden gebruikt. Denk bijvoorbeeld aan het gebruik van eenzelfde file die intern in de functie geopend wordt of het gebruik van globale variabelen. Indien we door uitvoering van een functie in meerdere threads gemeenschappelijke resources creëren of gebruik maken van gemeenschappelijke variabelen, moeten we in de functie kunnen zien dat er gebruik is gemaakt van **mutexes** (zie Hoofdstuk 10). Indien de functie func() gegarandeerd correct blijft werken in meerdere threads dan noemt men deze functie **threadsafe**. Een functie die niet threadsafe is kan alleen maar gebruikt worden in één thread. Het verschil tussen threadsafety en reentrance is dat een reentrant functie veilig kan worden afgebroken en vervolgens opnieuw worden aangeroepen (reentered).

Pas op met het gebruik van functies uit de standaard C-bibliotheek die in verschillende threads op gemeenschappelijke resources of variabelen opereren: deze zijn niet gegarandeerd reentrant of threadsafe! Controleer dat met de C programming reference manual van de specifieke compiler. Voorbeeld: printf() en malloc() zijn conform de POSIX standaard threadsafe maar non-reentrant.

Indien we zelf functies maken voor multithreaded toepassing moeten we attent zijn op het zelf maken van reentrant en threadsafe functies om correct gedrag te kunnen garanderen.

6.7 Opgaven

1. Schrijf een parent programma ouder.c en twee child programma's kind1.c en kind2.c.
   De parent geeft het procesidentificatienummer (process ID) van kind via de command line parameters door aan child 2.
   De parent drukt zijn PID af en die van zijn children.
   Kind1 drukt zijn PID af en die van zijn parent.
   Kind2 drukt de PID's af van zichzelf, zijn parent en kind1.
   Laat hierbij duidelijk zien welk proces (parent, kind1 en kind2) welke gegevens afdrukt!

2. Schrijf een programma prog.c dat zijn PID afdrukt en vervolgens twee threads afsplitst. Elke thread drukt zijn threadID en zijn procesID af.
   Laat hierbij duidelijk zien welke thread (prog, thread1 en thread2) welke gegevens afdrukt!

7 Intertaakcommunicatie met pipes

De meest eenvoudige manier om data tussen taken uit te wisselen, is via files. De ene taak kan in de file schrijven en de andere taak kan uit diezelfde file lezen. Hierbij kan zeer veel data worden uitgewisseld. Wel moeten er speciale maatregelen worden getroffen om de taken op elkaar te synchroniseren. De leestaak kan natuurlijk pas data lezen als de schrijftaak van tevoren data heeft opgeborgen in die file. Ook kan de schrijftaak alleen data aan het einde van de file toevoegen, omdat anders de leestaak de data misschien nog niet gelezen heeft. Hierdoor kunnen files erg groot worden. Deze problemen kunnen door het gebruik van pipes eenvoudig worden opgelost. Zie ook §12.6.1 en §13.7.2 van het boek *Linux for programmers and users*.

7.1 Pipes

Een **pipe** is queue, een FIFO-datastructuur in RAM (FIFO = First In, First Out). Een FIFO-datastructuur is een cyclische buffer die beheerd kan worden door twee pointers, een invoer pointer en een uitvoer pointer. Een pipe is dus niet willekeurig toegankelijk en na het lezen van data is deze data niet meer beschikbaar. (Dit geldt ook voor een stack. Alleen is een stack een LIFO-datastructuur, Last In, First Out).
Pipes worden gebruikt voor het overbrengen van data tussen taken (sequential data transfer).

In fig. 7.1 zijn vier verschillende communicatiestructuren weergegeven:

1) taak A heeft een FIFO-datastructuur nodig; het is dus mogelijk dat een taak zowel leest als schrijft in dezelfde pipe
2) dataoverdracht in één richting van taak A naar taak B
3) dataoverdracht in twee richtingen tussen de taken A en B; een pipe kan maar voor één communicatierichting gebruikt worden; voor bidirectionele communicatie zijn dus twee pipes noodzakelijk
4) de twee taken A en B schrijven in dezelfde pipe; één taak, taak C, leest in die pipe
5) pipe voor communicatie tussen parent en child. Dit kan worden verkregen door een pipe te creëren en vervolgens een child te forken. In veel gevallen zullen vervolgens een lees- en een schrijfkant worden gesloten met close().

Taken die data produceren worden **producers** genoemd; taken die data opnemen worden **consumers** genoemd.

Een pipe heeft een vaste grootte. Bij oudere versies van Linux is dit 4096 bytes, vanaf Linux 2.6.11 is dit **64 kb**. Is de **pipe vol** en wil een producer data aan de pipe toevoegen, dan komt deze producer default automatisch in de **blocked toestand**. De producer wacht dan totdat een consumer data uit de pipe heeft gelezen en gaat dan weer verder met schrijven. Als een consumer uit een **lege pipe** wil lezen, wordt deze consumer default automatisch in de **blocked state** gezet en kan pas verdergaan met lezen, nadat een producer data heeft opgeborgen in de pipe. Zo worden producer en consumer automatisch **gesynchroniseerd**. Ook bij het openen van een pipe wordt er gesynchronyseerd.

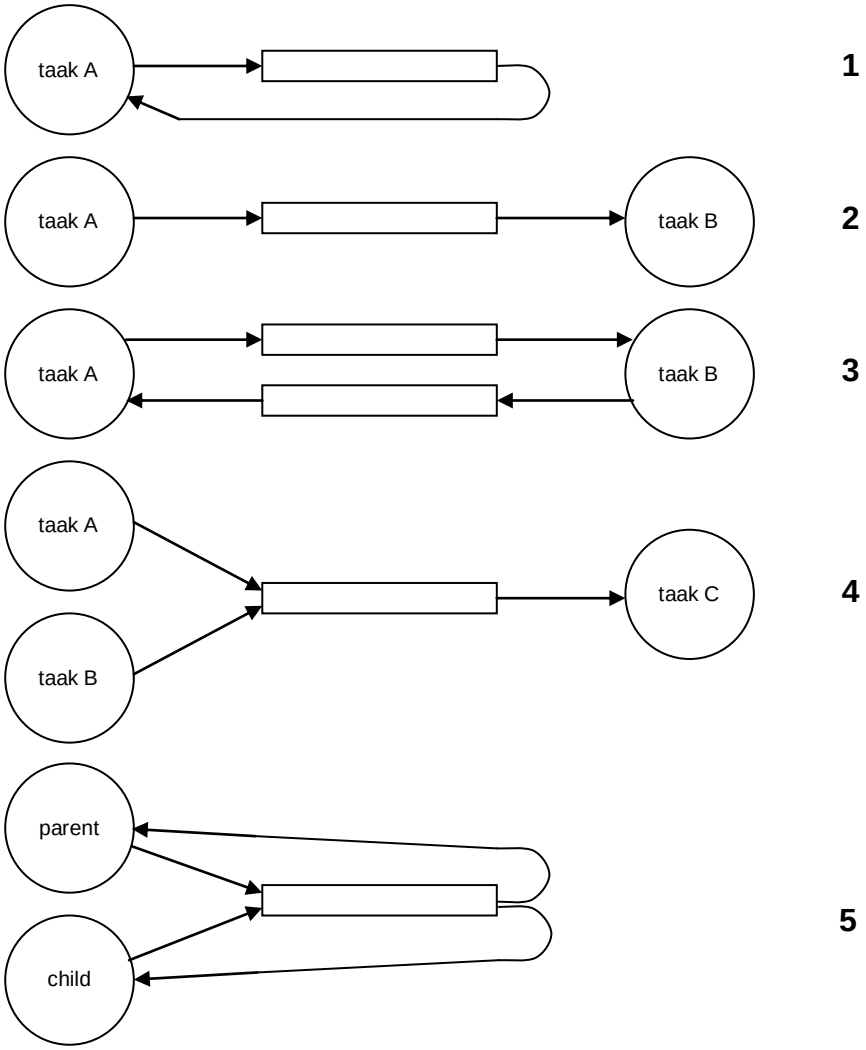


Fig. 7.1 Toepassingen van pipes

7.2 Unnamed pipes

Een pipe wordt gecreëerd met de system call **pipe()**. Het prototype van deze system call luidt:

int pipe (int fd[2]);

De **invoerparameter** `fd[]` is een pointer naar een array van 2 integers. De pipe-functie zet twee file descriptors in deze array. De **uitvoerparameter** is een error code: -1 betekent dat er een fout is opgetreden bij het uitvoeren van de system call, bij 0 is de pipe aangemaakt. Zie `man 2 pipe` voor meer informatie (en `pipe2()` voor extra opties).

Omdat een pipe twee einden heeft, één om in te schrijven en één om uit te lezen, heeft men toegang tot een pipe via twee file descriptors: **fd[0]** is de file descriptor van de **leeskant**, dus de **uitvoerkant** van de pipe en **fd[1]** is die van de **schrijfkant**, de **invoerzijde** van pipe.
Merk op dat er geen padnaam wordt opgegeven bij de system call pipe(). Men spreekt daarom ook wel van **unnamed pipes**. Dit houdt in dat geen enkele andere

bestaande taak toegang heeft tot de gecreëerde pipe. Alleen taken met een zelfde voorouder hebben toegang tot de pipe.

De system calls fork() en exec() zorgen ervoor dat kopieën van de file descriptors van de parent beschikbaar zijn voor de children. Elke taak heeft zijn eigen file descriptor tabel.

Elke taak krijgt bij het creëren drie file descriptors voor I/O ter beschikking:

0 = standard input device	**stdin**
1 = standard output device	**stdout**
2 = standard error output device	**stderr**

De standaard file descriptors vormen dus eigenlijk een standaard communicatie interface voor alle taken. Als default input device wordt het toetsenbord gebruikt en als default output device en default error output device wordt het scherm gebruikt. Deze paden hebben geen naam maar alleen een nummer (file descriptor). Het zijn **unnamed pipes** en deze zijn niet te vinden in een directory.

Het is mogelijk nieuwe unnamed pipes te creëren met de system call pipe(). Linux zal steeds de eerste vrije file descriptor uitgeven in volgorde van laag naar hoog. Een file descriptor is taakgebonden. Zo kan voor taak A file descriptor 5 verwijzen naar bijvoorbeeld een geopend pad naar een printer terwijl voor taak B file descriptor 5 gebruikt kan worden voor verwijzing naar een geopende file op schijf. In Linux kan elke taak maximaal 256 file descriptors tegelijkertijd in gebruik hebben. De toegewezen file descriptors worden bijgehouden in het system data segment van het betreffende proces. Iedere kant van een pipe kan afzonderlijk worden afgesloten met de functie close().

Fig. 7.2 toont de relatie tussen de file descriptors en de bijbehorende "files" van een taak die achtereenvolgens een pipe en een file opent.

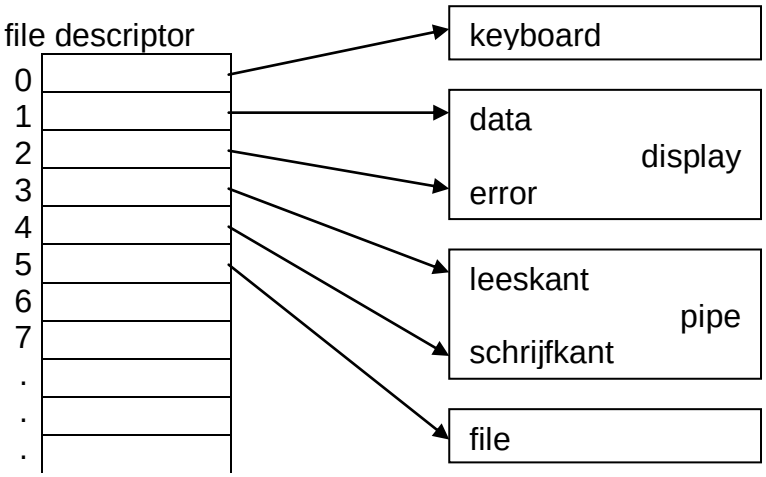


Fig. 7.2 Relatie tussen file descriptors en de bijbehorende "files"

7.3 Redirectioneren van I/O

M.b.v. de system call **dup()** wordt een kopie gemaakt van een file descriptor. De originele file descriptor en de kopie wijzen naar dezelfde "file".
Het functieprototype ziet er als volgt uit:

int dup (int fd);

- De **returnwaarde** is gelijk aan de nieuwe file descriptor die verwijst naar dezelfde file description als waar de file descriptor **fd** naar verwijst. Als er een fout optreedt, is de returnwaarde -1 en bevat errno een foutcode.

Als de taak horende bij de file descriptor tabel van fig. 7.2 de instructie dup(1) uitvoert, wijst file descriptor 6 eveneens naar het standard output device (zie fig. 7.3).
We laten nu een aantal voorbeelden zien van het gebruik van redirectioneren.

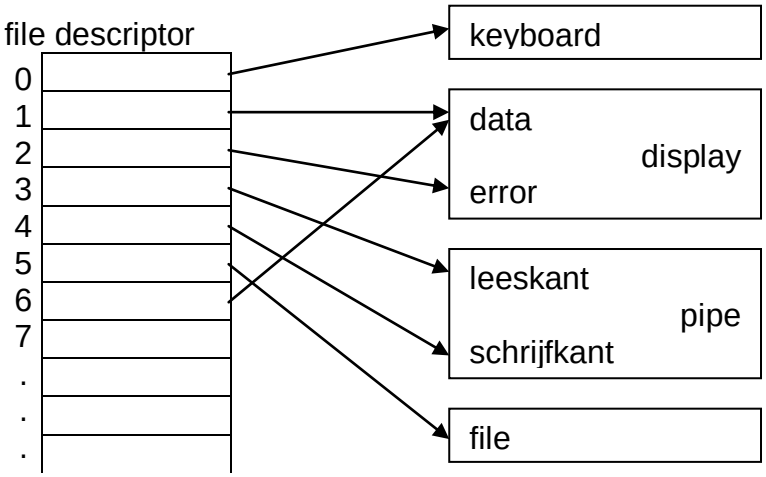


Fig. 7.3 Het resultaat van dup (1)

Voorbeeld 7.3.1 (zie fig. 7.4)
Een parent taak is opgestart met een keyboard en scherm (respectievelijk standard in en standard out). Dit programma heeft ten behoeve van een bepaalde applicatie de inhoud van een directory nodig, bijvoorbeeld om te onderzoeken of een file al dan niet aanwezig is. Het commando (de taak) ls levert de inhoud van de directory.
Fig. 7.5 en 7.6 tonen de pseudo-codes van het parent- en child-programma.

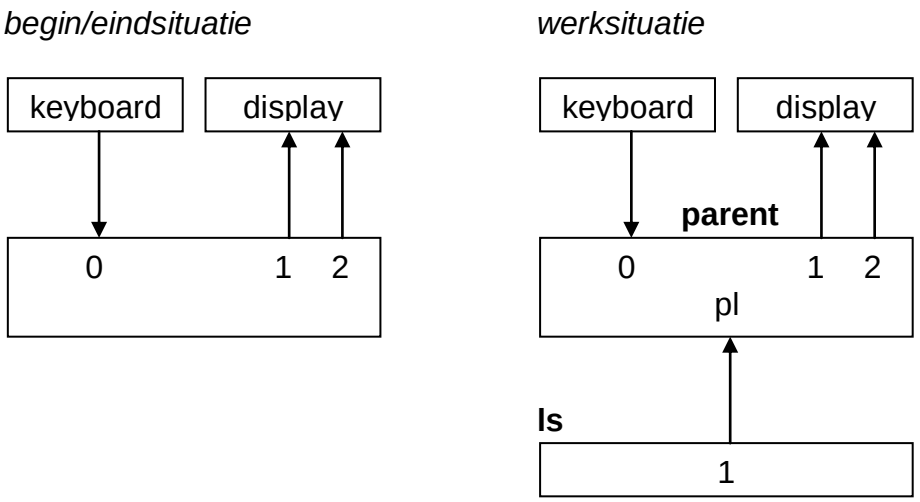


Fig. 7.4 Redirectioneren, voorbeeld 7.3.1

	file descriptors met betekenis
{beginsituatie}	
creëer pipe p	3 = leeskant pl, 4 = schrijfkant ps
fork child t.b.v. taak ls	
sluit schrijfkant ps	
{werksituatie}	
wacht op einde child	
sluit leeskant pl	
{eindsituatie}	

Fig. 7.5 Pseudo-code van de parent van voorbeeld 7.3.1

{beginsituatie}
sluit leeskant pl
sluit stdout
dupliceer schrijfkant ps
sluit schrijfkant ps
execute taak ls
{werksituatie}
:
{eindsituatie} (wordt bereikt als ls klaar is;
file descriptor tabel van ls is opgeheven)

Fig. 7.6 Pseudo-code van de child van voorbeeld 7.3.1

Voorbeeld 7.3.2

Een parent start twee child-taken op: TaakA en TaakB. De communicatie via unnamed pipes wordt weergegeven in fig. 7.7 en de pseudo-code in fig. 7.8 en 7.9.

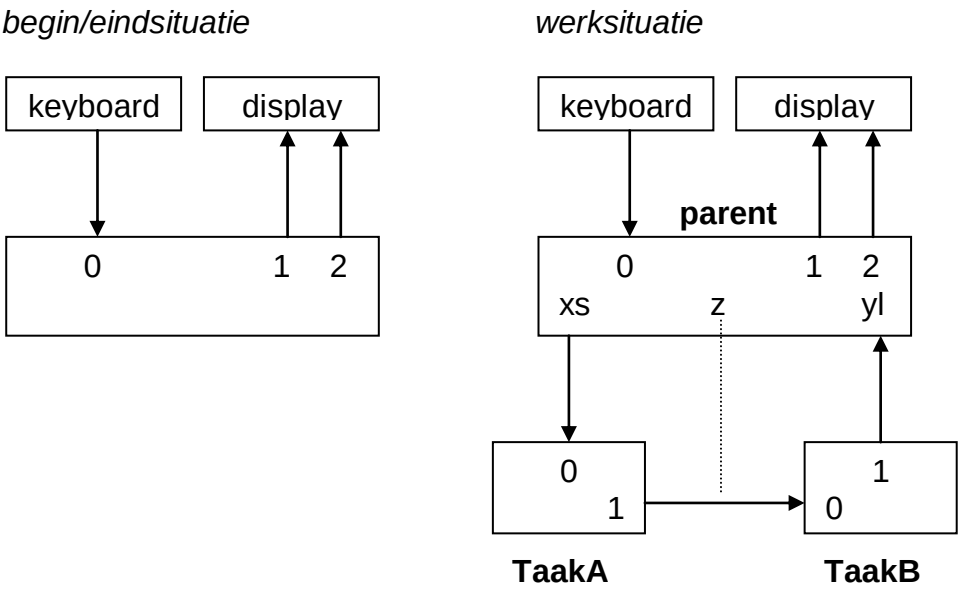


Fig. 7.7 Redirectioneren, voorbeeld 7.3.2

	file descriptors met betekenis
{beginsituatie}	
creëer pipe x	3 = leeskant xl, 4 = schrijfkant xs
creëer pipe y	5 = leeskant yl, 6 = schrijfkant ys
creëer pipe z	7 = leeskant zl, 8 = schrijfkant zs
fork child t.b.v. TaakA	
fork child t.b.v. TaakB	
sluit xl	
sluit ys	
sluit zl	
sluit zs	
{werksituatie}	
wacht op einde van TaakA	
sluit schrijfkant xs	
wacht op einde van TaakB	
sluit leeskant yl	
{eindsituatie}	

Fig. 7.8 Pseudo-code van de parent van voorbeeld 7.3.2

TaakA

{beginsituatie}
sluit stdin
dupliceer xl
sluit xl
sluit xs
sluit stdout
dupliceer zs
sluit zs
sluit zl
sluit yl
sluit ys
execute TaakA
{werksituatie}
:
{eindsituatie} (wordt bereikt als TaakA klaar is; file descriptor tabel van TaakA is opgeheven}

TaakB

{beginsituatie}
sluit stdin
dupliceer zl
sluit zl
sluit zs
sluit stdout
dupliceer ys
sluit ys
sluit yl
sluit xl
sluit xs
execute TaakB
{werksituatie}
:
{eindsituatie} (wordt bereikt als TaakB klaar is; file descriptor tabel van TaakB is opgeheven}

Fig. 7.9 Pseudo-codes van de children van voorbeeld 7.3.2

Voorbeeld 7.3.3
Hieronder staat het programma met communicatiestructuur 2) van fig. 7.1. De parent stuurt een boodschap via een unnamed pipe naar de child. De child leest deze boodschap uit de pipe en drukt deze af op het display.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main (void) {
     char aiBuffer[10];
     int  fdPijp[2];

     if (pipe (fdPijp) == -1) {
          printf ("Helaas is de pijp mislukt!\n");
          exit (1);
     }

     switch (fork ()) {
          case -1:
               printf ("Helaas is de fork mislukt!\n");
               break;
          case 0:                                          /*child*/
               close (fdPijp[1]);
               read (fdPijp[0], aiBuffer, 6);
               close (fdPijp[0]);
               printf ("Ontvangen van parent via pipe: %s\n",
                        aiBuffer);
               break;
```

```
        default:                                                /*parent*/
            close (fdPijp[0]);
            write (fdPijp[1], "Hallo", 6);
            close (fdPijp[1]);
            wait (NULL);
    }
    return 0;
}
```

Het resultaat van dit programma ziet er als volgt uit:

```
Ontvangen van parent via pipe: Hallo
```

Bij het redirectioneren van pipes moet altijd goed worden gelet op de **volgorde** van de system calls close() en dup(). Het meest eenvoudig is om dit steeds direct achter elkaar te doen:

```
close (fd1);
dup   (fd2);
```

heeft tot gevolg dat fd1 en fd2 beide naar de file description verwijzen als waar fd2 naar verwees voordat deze instructies werden uitgevoerd. Merk op dat dit alleen correct is als fd1 op dat moment de laagste vrije file descriptor is! Hierna wordt meestal de oorspronkelijke file descriptor vrijgegeven:

```
close (fd2);
```

Nu verwijst alleen fd1 nog naar de oorspronkelijke file description die bij fd2 hoorde.

Deze system calls close() en dup() kunnen worden vervangen door

```
dup2 (fd2, fd1);
```

Het prototype van deze system call is

```
int  dup2 (int fdOud, int fdNieuw);
```

Nu komt de kopie van `fdOud` in ieder geval in `fdNieuw`, ook al zijn er lagere file descriptors vrij! Als fdNieuw al geopend is, wordt deze eerst gesloten. Daarna wordt fdNieuw geopend en refereert naar dezelfde file description als fdOud. Als dit succes heeft, is de returnwaarde gelijk aan fdNieuw. Bij een fout is de returnwaarde -1 en staat de foutcode in errno.

Het is overigens wel veiliger om fdNieuw toch eerst te sluiten, omdat bij dup2() eventuele foutmeldingen bij het sluiten van fdNieuw verloren gaan. Zie hoofdstuk 2 van de man pages voor de dup(), dup2() en close()-functies voor meer details.

7.4 System calls t.b.v. files en pipes

De volgende system calls kunnen worden gebruikt na het opnemen van de preprocessorinstructie **#include <unistd.h>**:

```
ssize_t  read   (int iFiledescr, void *Buf, size_t iNrBytes);
ssize_t  write  (int iFiledescr, void *Buf, size_t iNrBytes);
int      close  (int iFiledescr);
```

- Als er een fout optreedt, is de returnwaarde -1. Anders is de returnwaarde bij read() en write() het werkelijk aantal gelezen, resp. geschreven bytes. close() geeft returnwaarde 0 bij succes.
- Invoerparameter iFiledescr is de filedescriptor van de file die benaderd moet worden.
- Buf is het beginadres van de buffer waaruit de bytes gelezen worden, resp. waarin de bytes geschreven worden.
- Verder is iNrBytes het aantal bytes dat er gelezen, resp. geschreven moet worden.

Bij pipes worden de benodigde file descriptors verkregen met de system call pipe(), zoals we gezien hebben in §7.2. Bij files en device drivers wordt de file descriptor als returnparameter geleverd door de system call open():

```
int  open (const char *pcPadnaam, int iOflag, ... );
```

- De returnwaarde is de laagste vrije file descriptor. Als er een fout optreedt is de returnwaarde -1. De foutcode bevindt zich dan in errno.
- Er wordt een file description aangemaakt die aan de file refereert waarvan de padnaam aangewezen wordt door de pointer pcPadnaam. De file descriptor verwijst naar deze file description.
- Invoerparameter iOflag is de bitsgewijze or-functie van een aantal flags. Één van de flags O_RDONLY, O_WRONLY en O_RDWR moet in ieder geval worden gebruikt. Hiermee wordt immers aangegeven of de file wordt geopend voor alleen lezen, alleen schrijven of zowel lezen als schrijven.
- Andere flags zijn bv.

O_APPEND	om data aan het eind van de file toe te voegen (normaal wordt aan het begin van de file gestart met lezen en schrijven, zodat eventueel bestaande data overschreven wordt).
O_CREAT	Als de file bestaat, wordt de bestaande file descriptor teruggegeven. Bestaat de file niet, dan wordt deze gecreëerd en wordt er een nieuwe file description met bijbheorende file descriptor aangemaakt.
O_EXCL	Als deze flag tegelijk met O_CREAT wordt gebruikt, wordt er file aangemaakt als deze nog niet bestaat. Bestaat de file al, dan wordt deze niet geopend en wordt -1teruggegeven.

Hierbij zijn de volgende preprocessorinstructies nodig:

```
#include  <sys/stat.h>
#include  <fcntl.h>
```

7.5 Named pipes of FIFO's

Omdat het I/O system van Linux sterk modulair en zo hardwareonafhankelijk als mogelijk is gerealiseerd, kunnen zogenaamde **named pipes** vrijwel net zo benaderd worden als files op een extern geheugenmedium. Een named pipe kent net als een file een naam en is te vinden in een directory.

Vanaf de command line kan een named pipe (FIFO file) worden gecreëerd met een van de commando's:

mknod filenaam p

mkfifo filenaam

Hierna kunnen de permissies eventueel worden gewijzigd met het commando

chmod modebits filenaam

De volgorde van de modebits is rwe (read write execute) voor achtereenvolgens de user, de group en others. Met

chmod 0660 filenaam

wordt filenaam toegankelijk voor lezen en schrijven door de user en de group. De eerste 0 geeft aan dat het volgende getal in octale representatie is gegeven.

De toegankelijkheid kan ook meteen worden bepaald:

mknod -m 0660 filenaam p

Voor meer informatie, zie Hoofdstuk 1 van de man pages van de mknod, mkfifo en chmod commando's.

Vanuit een programma kan een FIFO file worden gecreëerd met de system call

int mknod (char *pathname, mode_t mode, dev_t dev);

Merk op dat **pathname** zowel de directory als file name bevat. Hierin is **mode** een combinatie van het file type (in dit geval S_IFIFO) en de permissiebits. Creëren we een FIFO file waar alleen eigen user programma's in mogen lezen en schrijven, dan is de mode S_IFIFO | S_IRUSR | S_IWUSR. De variabele **dev** wordt gebruikt bij het creëren van speciale **device files** (**special files**, zoals FIFO's), en moet in dit geval 0 zijn (zie ook §12.3.23 van het boek *Linux for programmers and users*). Include hierbij **<sys/stat.h>** (zie **`man 2 mknod`**).

Er kan ook gebruik worden gemaakt van de system call

int mkfifo (const char *pathname, mode_t mode);

Hierbij horen de files **<sys/types.h>** en **<sys/stat.h>** (zie **`man 3 mkfifo`**).

De returnwaardes van deze system calls zijn 0 bij succes en -1 bij een fout.
Een fifo kan net als een file worden verwijderd met het Shell commando

rm fifonaam

of met de system call (zie **`man 2 unlink`** en §8.1)

int unlink (const char *pcFifonaam);

Elke willekeurige andere taak kan deze pipe met een open() system call openen (mits deze taak de rechten hiertoe heeft). Er is dus bv. geen parent-child-relatie nodig tussen de taken.

In een pipe kunnen verschillende datatypen achtereenvolgens geschreven worden. Dit betekent dan wel dat er weer in de juiste volgorde met de juiste datatypen gelezen moet worden!

Een named pipe is dus slechts bytegeoriënteerd, met andere woorden er wordt geen run-time type checking toegepast. Het is niet mogelijk aan een pipe te 'vragen' welk datatype er in staat.

Een named pipe wordt pas uit de directory verwijderd als alle links van de taken die hem gebruiken, opgeheven zijn en als de pipe leeg is. Een named pipe blijft dus bestaan als er nog data in zit maar er geen enkele taak meer is die deze pipe gebruikt.

Er wordt geen foutmelding gegeven als een named pipe gecreëerd wordt en gevuld wordt met data terwijl er géén consumenten zijn. Er wordt ook geen foutmelding gegeven als een named pipe nog data bevat en de bijbehorende producerende en consumerende taken zijn beëindigd. De named pipe blijft dan bestaan en wordt niet gedealloceerd.

7.6 Opgaven

1. Op welke wijze kan een consumer taak nagaan of er data in een pipe aanwezig is in het geval er meerdere pipes zijn?

2. Schrijf in pseudo-code het programma om de werksituatie van fig. 7.10 te creëren.

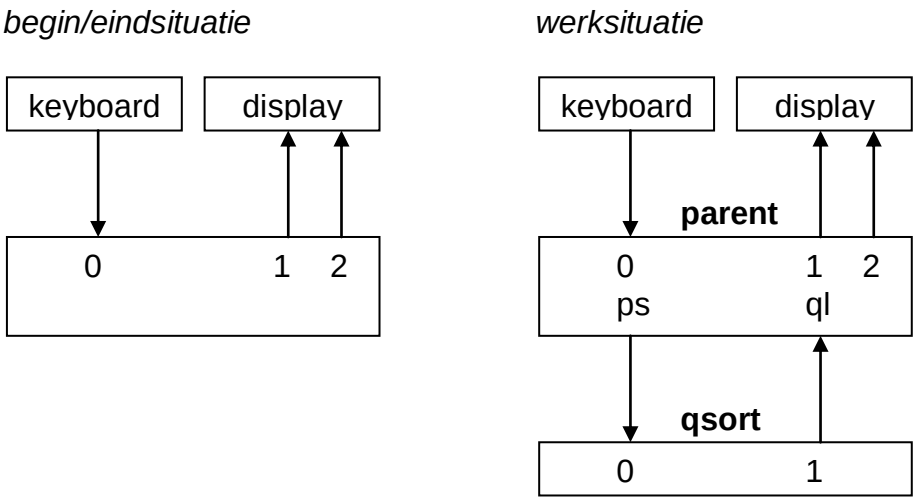


Fig. 7.10 Opgave 2, redirectioneren

8 Intertaakcommunicatie met shared memory

In veel gevallen moet een aantal taken een gemeenschappelijk stuk random access geheugen tot zijn beschikking hebben. In Linux wordt hiervoor **shared memory** gebruikt (zie fig. 8.1).

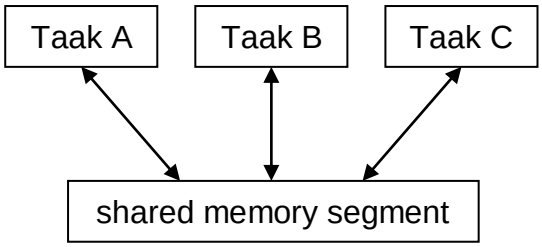


Fig. 8.1 Drie taken met een gemeenschappelijk geheugensegment

Een gemeenschappelijk geheugensegment is gelijkwaardig aan globaal geheugen voor twee of meer taken. Toepassing van geheugensegmenten maakt bovendien **security en error checking** mogelijk voor taken die een gemeenschappelijk datagebied nodig hebben.

De verschillen met pipes zijn de volgende:

- Een datamodule is willekeurig toegankelijk (random access). Je hoeft dus niet te beginnen bij de eerste byte, en kunt heen en weer springen.
- Bij het uitlezen van een datamodule blijft de inhoud behouden.
- Er treedt geen automatische synchronisatie op tussen de taken die gebruikmaken van het gemeenschappelijk geheugen.
- Er is geen parent-child-relatie nodig.

8.1 Shared memory in Linux

Om van een gemeenschappelijk geheugen gebruik te kunnen maken, moet het natuurlijk eerst gecreëerd worden. Dit gebeurt met de system call shared memory get:

```
int  shmget (key_t key, int iSize, int iFlags);
```

De **returnwaarde** van shmget() is een **memory segment ID** die dezelfde functie heeft als de file descriptor bij de open() call. Echter de memory segment ID kan gebruikt worden door elk willekeurig proces, terwijl een file descriptor alleen kan worden gebruikt door de parent en zijn child processen en de child processen daarvan, etc.

Is de returnwaarde -1, dan is het creëren van het geheugensegment mislukt. Hierin is **key** een numerieke waarde die een soortgelijke functie heeft als een filenaam. Met **iSize** wordt aangegeven hoeveel bytes er nodig zijn. (Dit wordt naar boven afgerond op een geheel aantal geheugenpagina's.)

Ten slotte is **iFlags** een set geORde (d.w.z. verbonden met | (OR)) maskerbits die de toegangspermissies en de creatievlaggen aangeven. Het is dus een combinatie van de tweede en derde parameter bij een open() system call. De vlag **IPC_CREAT** geeft aan dat het shared memory met de opgegeven key alleen wordt gecreëerd als het nog niet bestaat. Als het shared memory segment wel bestaat, wordt de memory segment ID van het bestaande segment teruggegeven. Wordt tegelijk met IPC_CREAT de vlag **IPC_EXCL** meegegeven, dan veroorzaakt dit een foutconditie als het shared memory al bestaat.

Een eenvoudige manier om een aantal processen (zonder parent-child-relatie) gebruik te laten maken van shared memory is een key af te spreken. Elk proces gebruikt die key in zijn shmget() call met de vlag IPC_CREAT, maar zonder de vlag IPC_EXCL. Het gevaar van deze methode is echter dat een ander proces die key al gebruikt en dus de verkeerde shared memory ID terugkrijgt van de shmget() call. Dit probleem is te vermijden door één proces de shmget() call te laten uitvoeren en hierbij **IPC_PRIVATE** als key te gebruiken. Nu wordt er een nieuw memory segment aangemaakt. De bijbehorende shared memory ID is vervolgens bekend bij alle (nieuwe) child processen. Ook kan het betreffende proces dit shared memory ID doorgeven aan alle processen die er gebruik van moeten maken. Een parent-child-relatie is in dit geval dus nodig voor het doorgeven van de key.

Als een shared memory segment eenmaal bestaat, kan een proces toegang krijgen met de system call shared memory attach:

void * shmat (int iShmID, char *pcShmAddr, int iFlags);

Hierin is de **returnwaarde** een pointer naar het shared memory segment; **iShmID** is de shared memory identifier die verkregen is met shmget(). **pcShmAddr** is het geheugenadres van de plaats waar je het shared memory wil hebben. Meestal wil je dat niet zelf opgeven, maar overlaten aan de kernel. In dat geval vul je hier NULL in. Met de flag **SHM_RDONLY** kun je aangeven dat je alleen wil lezen in plaats van lezen en schrijven (iFlags = 0).

Een proces dat het shared memory niet meer nodig heeft, kan zich hiervan losmaken met de system call shared memory detach:

int shmdt (char *pcShmAddr);

Het geheugensegment blijft hierbij echter beschikbaar voor andere processen. Om het geheugensegment op te heffen, moet gebruik worden gemaakt van de system call shared memory control:

int shmctl (int iShmID, int iCmd, struct shmid_ds *Buf);

Als commando **iCmd** moet in dit geval **IPC_RMID** worden gebruikt. Vanaf de command line kan het geheugensegment worden verwijderd met het commando **`ipcrm -m <shmID>.`**

De system call shmctl() kan ook worden gebruikt om de shared memory struct te kopiëren in een struct via de pointer Buf (met **IPC_STAT**). Omgekeerd kan de struct

worden gekopieerd in de shared memory struct (met **IPC_SET**). We gaan hier verder niet op in.

Vanaf de command line kunnen shared memory segmenten worden bewerkt met de volgende commando's. Zie hun man pages voor meer details:

```
ipcs              # toon IPC status
ipcmk <options>   # maak een IPC resource aan
ipcrm <options>   # verwijder IPC resource
```

Voorbeeld 8.1.1
In dit voorbeeld laten we een zendend proces data in het shared memory schrijven en het ontvangend proces via een FIFO meedelen dat de data gelezen kan worden. Hiertoe schrijft het zendend proces het shared memory ID in de FIFO.
Het ontvangend proces leest de data uit het shared memory en drukt de data op het display af.

```
// send.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
  int iShmID, iFD_FIFO;
  char *pcShmAddr;


  // Allocate shared memory:
  iShmID = shmget(IPC_PRIVATE, 8192, IPC_CREAT | 0644);
    // permissions: u+rw, go+r - see man 2 chmod
  if(iShmID == -1) {
    printf("Sender:  Shared memory not created.\n");
    exit(1);
  }

  // Attach shared memory to local character pointer:
  pcShmAddr = shmat(iShmID, NULL, 0);
  if(pcShmAddr == (char *) -1) {
    printf("Sender:  Shared memory not attached.\n");
    exit(1);
  }
  // Create message:
  strcpy(pcShmAddr, "Just a trivial message.");
  printf("Sender:  Message written to shared memory\n");

  // Create named pipe:
  if(mkfifo("FIFO", S_IFIFO|0666) == -1) {  // 0666: u/g/o r/w
    printf("Sender:  FIFO not created.\n");
```

```
    exit(1);
  }

  // Open pipe:
  if((iFD_FIFO = open("FIFO", O_WRONLY)) == -1) {
    printf("Sender:  FIFO not opened.\n");
    exit(1);
  }

  // Write shmID to pipe:
  write(iFD_FIFO, &iShmID, sizeof(iShmID));

  // Close pipe:
  close(iFD_FIFO);

  // Remove pipe:
  if(unlink("FIFO") == -1) {
    printf("Sender:  FIFO not removed.\n");
    exit(1);
  }

  // Detach shared memory:
  if(shmdt(pcShmAddr) == -1) {
    printf("Sender:  Shared memory not detached.\n");
    exit(1);
  }

  // Flag shared memory for removal:
  if(shmctl(iShmID, IPC_RMID, NULL) == -1) {
    printf("Sender:  Shared memory will not be removed.\n");
    exit(1);
  }

  return 0;
}


// receive.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
  int iShmID, iFD_fifo;
  char *pcShmAddr, acText[51];

  // Open existing named pipe:
  if((iFD_fifo = open("FIFO", O_RDONLY)) == -1) {
    printf("Receiver:  FIFO not opened.\n");
    exit(1);
  }
```

```
  // Read memory ID from pipe:
  read(iFD_fifo, &iShmID, sizeof(iShmID));

  // Close pipe:
  close(iFD_fifo);


  // Attach shared memory:
  pcShmAddr = shmat(iShmID, NULL, 0);
  if(pcShmAddr == (char *) -1) {
    printf("Receiver:  Shared memory not attached.\n");
    exit(1);
  }

  // Copy the contents of the shared memory:
  strcpy(acText, pcShmAddr);

  // Detach the shared memory:
  if(shmdt(pcShmAddr) == -1) {
    printf("Receiver:  Shared memory not detached.\n");
    exit(1);
  }

  // Print message:
  printf("\nMessage according to receiver: %s\n\n", acText);

  return 0;
}
```

Het resultaat van deze programma's ziet er als volgt uit:

```
Message according to receiver: Just a trivial message.
```

Let erop dat proces send eerst wordt opgestart en pas daarna receive:

./send & sleep 1 & ./receive

Dit is nodig om eerst de pipe FIFO aan te maken. Proces receive kan immers deze pipe niet openen als hij niet bestaat. In dat geval wordt proces receive beëindigd.

De pipe FIFO moet door proces send (of receive) worden verwijderd om te voorkomen dat proces receive wordt beëindigd bij het creëren van de FIFO als die al bestaat. Dit gebeurt met de system call unlink() (zie ook **`man unlink`** en §7.5):

int unlink (const char *pcPadnaam);

- Deze system call verwijdert een link naar een file, fifo e.d. Dit houdt in dat de zogenaamde link count met 1 wordt verlaagd. De file wordt door unlink() verwijderd als de link count 0 is geworden. De link count krijgt bij het creëren van de file de beginwaarde 0. Met de system call open() wordt de link count met 1 verhoogd.

- De invoerparameter **pcPadnaam** bevat het adres van de string die de padnaam van de gewenste file bevat.
- De **returnparameter** is 0 als unlink() gelukt is en -1 als er een fout is opgetreden. De foutcode staat dan in errno.

Het is belangrijk dat de pipe en het shared memory in dezelfde taak worden verwijderd (beide in send, of beide in receive).

8.2 Opgaven

1. Geef twee praktische voorbeelden van het gebruik van datamodules. Geef hierbij een schets van de inhoud en de functie van de betrokken taken.

2. Als meerdere taken gebruikmaken van dezelfde datamodule, kunnen er dan transient errors ontstaan? Geef een voorbeeld.

9 Intertaakcommunicatie met signals

9.1 Het zenden en ontvangen van signals

In een multitasking omgeving lijken taken elk op een eigen processor te worden uitgevoerd. Die processoren zijn er in werkelijkheid niet (er is maar één fysieke processor in het systeem aanwezig). Het zijn in feite dus virtuele processoren. De parallelle taken moeten soms op elkaar wachten, voordat ze verder kunnen. Er moet synchronisatie tussen de taken plaatsvinden. Dit kan eenvoudig geschieden met behulp van **signals**. De ene virtuele processor stuurt een signal aan de andere virtuele processor. In werkelijkheid gaat het hier niet om een elektrisch signaal dat via een draad van de ene virtuele processor naar de andere wordt gezonden maar om een virtueel signaal dat van het ene proces via de kernel aan het andere proces wordt doorgegeven. De kernel fungeert als interface (interface module) tussen de taken die met elkaar communiceren (zie fig. 9.1). Zie voor signals ook **`man 7 signal`** en §12.5 van het boek *Linux for programmers and users*.

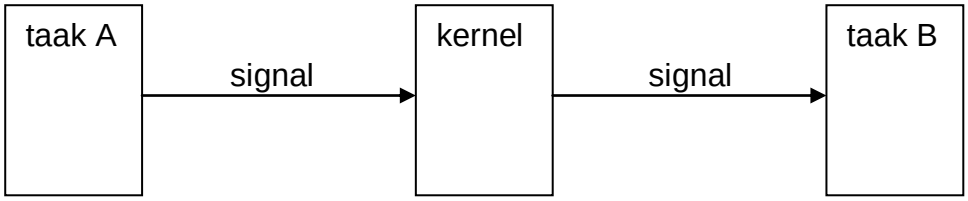


Fig. 9.1 Communicatie tussen taken met behulp van signals

9.1.1 Wachten op een signal met pause()

Een proces kan wachten op een signal m.b.v. de system call

int pause (void);

De **returnwaarde** is 1 en errno is EINTR, waarmee is aangeduid dat er een singal ontvangen is. Een en ander is gedeclareerd in de file **unistd.h** en toegelicht in **`man 2 pause`**. De pause()-functie lijkt op de wait()-functie, waar gewacht wordt op een signal van een child process (zie §6.3.1).

9.1.2 Zenden van een signal met kill()

Om een signaal naar een proces te kunnen sturen, moet het process ID (PID) van dat proces bekend zijn. De zendende taak (taak A) stuurt een signal naar de kernel met behulp van het shellcommando

kill -<signaalnummer> <PID>

of met de system call

int kill (pid_t pid, int iSignr);

De kernel stuurt het signal **iSignr** vervolgens door aan het proces (taak B) met de opgegeven process ID (**pid**). Dit gebeurt doordat de kernel de zogenaamde **signal-handling routine** (kortweg **signal handler** of **SHR** genoemd) van de ontvangende taak (taak B) opstart. De returnwaarde van kill() is 0 bij succes en anders -1.

9.1.3 Een signal-handling routine installeren m.b.v. signal()

Om een gebruiker-gedefinieerde actie uit te voeren wanneer een bepaald signal wordt ontvangen, moet de ontvangende taak van tevoren een signal handler hebben geïnstalleerd. Dit kan met behulp van de functie signal():

void (*signal (int signum, void (*handler) (int))) (int);

Dit is een complex prototype: zowel de input als de output is een pointer naar een functie. Merk op dat een functie in feite een pointer is naar het beginadres van die functie, zodat een functie en een functiepointer in de praktijk hetzelfde betekenen. De reden dat in- en output een functiepointer is, is dat als input de nieuwe SH-functie moet worden opgegeven en als output de oude (default) SH-functie wordt teruggegeven om deze eventueel later te kunnen herstellen.

Om het prototype van een SHR toe te lichten geven we in het kader hieronder uitleg bij een aantal voorbeelden van functieprototypes.

double fun1 (int, char);

Dit **functieprototype** toont dat **fun1** een int en een char als invoerparameters heeft en een double als returnparameter.

double fun2 (int, char *);

Dit **functieprototype** geeft aan dat **fun2** een int en een pointer naar een char als invoerparameters heeft en een double als returnparameter.

double fun3 (int, char[ ]);

Dit **functieprototype** laat zien dat **fun3** een int en het beginadres van een array van chars als invoerparameters heeft en een double als returnparameter.

double * fun4 (int, char);

Dit **functieprototype** toont dat fun4 heeft een int en een char als invoerparameters en een pointer naar een double als returnparameter.

double (*fun5) (int, char);

De regel hierboven is geen functieprototype, maar een **pointer naar een functie**. Hiermee wordt het mogelijk een functienaam als variabele te gebruiken, bv. fun5 = werkA en fun5 = werkB. Aangezien **fun5** een int en een char als invoerparameters heeft en een double als returnparameter, moeten de functies werkA en werkB dit ook hebben: double werkA (int, char); en double werkB (int, char);

double (*fun6[3]) (int, char);

fun6 is het **beginadres van een array van 3 pointers naar functies** (dus geen functieprototype) met een int en een char als invoerparameters en een double als returnparameter. Dit geeft de mogelijkheid om meerdere variabele functies tegelijkertijd toe te passen. Voorbeeld: fun6[0] = doX, fun6[1] = doY en fun6[2] = doZ. De functies doX, doY en doZ moeten gedeclareerd zijn als functies met een int en een char als invoerparameters en een double als returnparameter: double doX (int, char); etc.

double fun7 (int, double, double (*fp) (double));

fun7 heeft een int, een double en een **pointer naar functie** als invoerparameters en een double als returnparameter. De functie waar deze functiepointer **fp** naar wijst, moet ook een double als invoerparameter en een double als returnparameter hebben. Dit geeft de mogelijkheid om op te geven welke andere (variabele) functie in de aangeroepen functie wordt gebruikt.

Voorbeeld:

```
double fun7 (int iP, double dQ, double (*f) (double)) {
    return iP + f(dQ);
}
```

In een programma wordt fun7 als volgt aangeroepen:

double dX = fun7 (23, 15.3, cos);

Het derde argument (cos) is de naam van de functie, maar dus ook de pointer naar die functie. Het adres van die pointer wordt doorgegeven. In dit voorbeeld krijgt dX de waarde 23 + cos(15.3).

We keren nu terug naar het prototype van de functie signal() en zullen laten zien dat op deze manier een pointer naar een functie als returnparameter wordt gegeven. Hiertoe hakken we het prototype in stukjes:

```
void  (*signal (int iSignum, void (*handler) (int))) (int);
```

Op deze wijze kunnen we het prototype eenvoudiger opschrijven als:

```
void (*oldhandler) (int);
void newhandler (int);
oldhandler = signal (iSignum, newhandler);
```

Hierin is **oldhandler** het adres van de vorige signal handler en **newhandler** het adres van de nieuwe signal handler die geïnstalleerd gaat worden. Dit gebeurt voor het signaalnummer **iSignum**. Als een proces een signaal stuurt met dat nummer, zal de kernel de functie newhandler() van het ontvangende proces opstarten en deze functie het signaalnummer iSignum doorgeven als invoerparameter. De verdere communicatie tussen de signal handler en de ontvangende taak gaat via globale variabelen.

Signals kunnen taken uit de sleeping of waiting toestand halen. Taken die werken met 32-bits woordbreedten hebben 32 signals. Taken met een woordbreedte van 64 bits hebben 64 signals.

Tijdens de timeslice van de zendende taak wordt gesprongen naar de **signal handling routine** (**SHR**), ook wel aangeduid als **signal intercept routine** van de ontvangende taak. Een SHR is een gewone **C-functie**. Het **adres** van deze C-functie en de **waarde van het signal** moeten wel van tevoren door de ontvangende taak met behulp van de system call signal() aan de kernel bekend gemaakt zijn.

Normaliter moet een taak de process ID van de ontvangende taak kennen om een signal naar die taak te zenden. Hebben zender en ontvanger een parent child relatie, dan kan de process ID eenvoudig worden opgevraagd met behulp van system calls. Is dit niet het geval, dan moet de ontvangende taak eerst aan de zendende taak zijn process ID bekendmaken, bv. door deze bv. via een named pipe (FIFO) naar de zendende taak te sturen.

Via het toetsenbord kunnen ook signals naar de foreground taak worden gestuurd met bepaalde CTRL-toetsen, zoals <**Ctrl-C**> (SIGINT - interrupt) en <**Ctrl-Z**> (SIGTSTP – terminal stop = suspend). De signals SIGKILL en SIGSTOP kunnen niet worden ondervangen door een SHR. Zie man 7 signal of kill –l voor een lijst met signals en hun nummers op jouw systeem en het artikel *Efficient use of the Bash shell* voor meer informatie.

Voorbeeld 9.1.1
Een proces wordt default beëindigd als het een SIGINT signal ontvangt. Als de gebruiker <Ctrl-C> intypt zendt de terminal driver het signal SIGINT naar het proces dat in de voorgrond draait. Een proces kan echter zelf een signal handler hebben geïnstalleerd voor het signal SIGINT (en ook voor andere signals) en er een andere actie aan koppelen.

Het onderstaande programma installeert een signal handler voor SIGINT die telt hoe vaak SIGINT wordt ontvangen. Hierbij wordt de pointer naar de oude signal handler bewaard. Vervolgens leest het programma karakters van het toetsenbord totdat '\n' (enter) wordt gelezen. Dan wordt het aantal keren afgedrukt dat <Ctrl-C> is ingedrukt (ofwel het aantal keren dat SIGINT is ontvangen). De oude signal handler voor het

signal SIGINT wordt hersteld om het proces weer te kunnen stoppen met <Ctrl-C> en het programma komt in een “oneindige” wacht lus.

```
// Voorbeeld 9.1.1
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t iTeller = 0;
void (*old_handler) (int);
void ctrl_c (int);

int main (void) {
    old_handler = signal (SIGINT, ctrl_c);
    while (getchar() != '\n')
    {
        ;
    }
    printf ("Er is %d keer CTRL-C ingedrukt.\n", iTeller);
    (void) signal (SIGINT, old_handler);
    while (1)
    {
        ;
    }

    return 0;
}

void ctrl_c (int iSignaalnummmer) {
    iTeller++;
}
```

Starten we het bovenstaande programma op en drukken we bv. 5 keer op <Ctrl-C>, dan wordt iTeller 5 keer verhoogd door de signal handler ctrl_c() en krijgt de waarde 5. Merk op dat iTeller globaal gedeclareerd moet zijn en de qualifier **volatile** moet meekrijgen om aan de optimiser aan te geven dat de waarden van buitenaf veranderd kan worden (bv. door een hardware-event). Het type sig_atomic_t geeft aan dat de variabele **atomair** is. Dit type is in Linux een int.

Drukken we vervolgens op <Enter>, dan wordt de eerste while lus verlaten. Het aantal keren dat <Ctrl-C> is ingedrukt, wordt afgedrukt op het display en de oude signal handler wordt hersteld. Het proces komt nu in een oneindige lus terecht. Het proces wordt nu afgebroken door nogmaals op <Ctrl-C> te drukken.

De uitvoer van dit proces ziet er als volgt uit:

```
Er is 5 keer CTRL-C ingedrukt.
```

9.1.4 Een signal-handling routine installeren m.b.v. sigaction()

Het vorige voorbeeld met een SHR is verouderd. De volgende aanpak is overeenkomstig een meer recente **POSIX** standaard (zie **`man 2 sigaction`** voor meer details):

```
// Voorbeeld 9.1.2:
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t count = 0;
void ctrl_c1 (int sig );
void ctrl_c2 (int sig, siginfo_t *si, void *context);

int main(void){
   struct sigaction act;
   struct sigaction act_previous;

   memset(&act, '\0', sizeof(act));
   memset(&act_previous, '\0', sizeof(act_previous));

   act.sa_handler = ctrl_c1;
   act.sa_flags = 0;
   //act.sa_sigaction = ctrl_c2;
   //act.sa_flags = SA_SIGINFO;
   sigemptyset(&act.sa_mask);

   sigaction(SIGINT, &act, &act_previous);
   while (getchar() != '\n') {
      ;
   }
   printf ("CTRL-C %d times pressed\n", count);
   sigaction(SIGINT, &act_previous, NULL);

   while(1) {
      sleep(1);
   }

   return 0;
}

void ctrl_c1(int sig){
   count++;
}

void ctrl_c2(int sig, siginfo_t *si, void *context){
   count++;
}
```

De **functie sigaction()** maakt gebruik van de **struct sigaction** om de SHR te definieren. Er zijn twee mogelijkheden: gebruik het element **.sa_handler** om de SHR aan te geven en zet **.sa_flags=0**. In dat geval heeft de SHR één dummyvariabele, en is

zijn prototype hetzelfde als in het geval van signal(). We tonen deze mogelijkheid met de SHR **ctrl_c1()** in bovenstaand voorbeeld. De tweede optie is het gebruik van .sa_sigaction voor de definitie van de SHR en zet .sa_flags = SA_SIGINFO. Dit gebruikt een ander prototype voor de SHR, met drie dummyvariabelen (waarvan de laatste weinig wordt gebruikt). Dit gebruik is aangegeven met de SHR ctrl_c2(). De regels zijn hier uitgecommentarieerd omdat op sommige systemen slechts .sa_handler of .sa_sigaction mag worden gedefinieerd. Sigaction() installeert alleen een SHR wanneer de tweede variabele non-NULL is. Het element **.sa_mask** geeft aan of er signals gemaskeerd moeten worden tijdens het uitvoeren van de SHR (zie §9.2.2). In dit voorbeeld worden geen signals gemaskeerd.

9.2 Het schrijven van signal-handling routines

9.2.1 Signal-driven applications

We kunnen twee uiterste strategieën onderscheiden voor de invulling van SHR's voor de realisatie van **signal-driven** applicaties:

- De eerste strategie stelt voor een SHR zo kort als mogelijk te houden, want de ontvangende taak krijgt processortijd vóór zijn beurt en dit moet geminimaliseerd worden.
- De tweede strategie stelt voor in de SHR zo mogelijk bijna alle code van de taak te plaatsen. Deze laatste strategie geeft de bij de SHR behorende taak een meer event-driven karakter. Alle bij een ontvangen signal behorende activiteiten worden eerst volledig uitgevoerd voordat een volgend signal aan de beurt komt (signals worden **gemaskeerd** en in een queue geplaatst bij de uitvoering van een SHR).

De verwerking van signals geschiedt in **dezelfde volgorde** als waarin de signals naar de ontvangende taak zijn verzonden. Deze garantie moet een real-time operating system geven omdat dit noodzakelijk is voor real-time besturingsapplicaties waarin de volgorde van activiteiten essentieel is voor het correcte verloop van de activiteiten.

Een probleem treedt op als een taak een signal stuurt, terwijl de ontvangende taak zijn **signal handler nog niet heeft geïnstalleerd**. Dit betekent dat de ontvangende taak de **default SHR uitvoert**.

9.2.2 Het maskeren van signals

In bepaalde gevallen moeten er meerdere variabelen (bijvoorbeeld in een struct) door de SHR na ontvangst van een specifiek signal van waarde worden veranderd. Stel dat het hoofdprogramma deze data gebruikt. Tijdens het verwerken van een deel van de gemeenschappelijke data wordt de SHR uitgevoerd (asynchrone activiteit) en worden de verschillende variabelen gemuteerd. Het hoofdprogramma heeft een deel van deze verzameling data met de oude waarden al verwerkt. Na het optreden van de aanroep van de SHR gaat het programma verder met het resterende deel van deze data dat net gemuteerd is! Er kan sprake zijn van een bepaalde logi-

sche samenhang tussen de data die nu verstoord zijn na uitvoering van de SHR. De verwerkte data zijn dan niet meer consistent. Dit betekent dat we tijdens de verwerking van de verschillende dataeenheden geen signals mogen verwerken. Met andere woorden, we moeten de signals **maskeren** totdat alle bij elkaar behorende data zijn gemuteerd. We voorkomen met maskeren het optreden van **ongewenste race-condities** of **transient errors**.

Het voorafgaande geldt niet alleen voor SHR's maar ook voor **interrupt handling routines** (IHR's). Ook IHR's worden op asynchrone wijze ten opzichte van het hoofdprogramma uitgevoerd. Pas dus op als er gemeenschappelijke datastructuren worden gemuteerd door meerdere instructies van de processor. Meerdere instructies zijn niet als atomair te beschouwen, omdat meerdere instructies door interrupts zijn te onderbreken. In bepaalde gevallen is het dus noodzakelijk voor bepaalde delen van de code interrupts te maskeren. Uiteraard moeten we deze maskeeractiviteit zoveel mogelijk vermijden.

Het voorkomen van de geschetste problemen in het geval dat er zowel door asynchrone activiteiten (code-uitvoering) als door een hoofdprogramma op gemeenschappelijke datastructuren wordt geopereerd, wordt ook wel aangeduid als **async safe programmeren**.

9.2.3 Eisen aan signal-handling routines

Belangrijke eisen betreffende de invulling van de **SHR code**:

- **Minimaliseer** de hoeveelheid **gemeenschappelijke data** tussen de SHR en het hoofdprogramma. Maak het muteren van de data atomair, of maskeer het signal tijdens die bewerkingen.
- **Vermijd** in het algemeen **I/O-functies** in de SHR. Dit kan leiden tot vervelende problemen. De I/O-functies uit de C-bibliotheek maskeren geen signals (zijn dus niet **async safe**). Indien in de SHR high-level standaard I/O-functies (die gebruikmaken van file pointers van het type FILE*) worden gebruikt, dan moeten de high-level I/O-functies in het hoofdprogramma gemaskeerd worden!
- Het voorafgaande probleem is onder andere te vermijden door in het gehele programma (inclusief SHR) **alleen low-level I/O-functies** (die gebruikmaken van paden) te gebruiken. Eventueel is het mogelijk low-level I/O-functies toe te passen in de SHR en high-level I/O-functies in het hoofdprogramma.
- Gebruik **geen floating-pointberekeningen** in een SHR.
- Stuur pas een signal **nadat** de bestemmingstaak zijn **SHR** heeft **geïnstalleerd**, anders kan dit fataal zijn voor de bestemmingstaak: de bestemmingstaak zou immers kunnen worden gekilled. Let vooral op het probleem dat optreedt als een parent een child forkt en direct daarna een signal stuurt aan de desbetreffende child. Als de child task nog niet running is geweest, heeft deze taak nog niet zijn SHR kunnen installeren en kan deze worden gekilled.

9.3 Alarms

In real-time applicaties hebben we vaak behoefte aan signals die een taak op bepaalde momenten naar zichzelf stuurt, **timerfuncties**. Deze signals worden **alarms** genoemd. Deze alarms kunnen de volgende eigenschap hebben voor wat betreft het moment van optreden:

- op een specifiek tijdstip (kalender trigger)
- na een bepaalde tijd – zie **`man 2 alarm`**
- periodiek (cyclisch) – zie **`man 2 setitimer`**

Een alarm kan bv. dienst doen als een watchdog timer. Dit is de mogelijkheid om een time-out te genereren als bepaalde zaken na een ingestelde tijd niet hebben plaatsgevonden. Denk bijvoorbeeld aan het wachten op de binnenkomst van data. Zo kunnen we dus **hanging states** tegen gaan.

9.4 Opgaven

1. Hoe kunnen de process ID's van twee child processen aan elkaar doorgegeven worden, zodat deze over en weer signals naar elkaar kunnen zenden?

2. Is het mogelijk dat meerdere signals gelijktijdig binnenkomen?

3. In een real-time systeem zijn drie taken bezig om elk een sensorsignaal te controleren. Deze taken zijn sleeping als er niets gebeurt. De sensor units leveren een hardware interrupt aan de drivers als het sensorsignaal een bepaalde waarde overschrijdt. De taak die daardoor running wordt (hoge prioriteit, preemption), stuurt een signal naar een monitoring taak. De monitoring taak geeft het totale aantal overschrijdingen per sensor weer. De signal waarden zijn 21 (sensor1), 22 (sensor2) en 23 (sensor3).

    a. Schrijf de SHR van de monitoring taak en geef duidelijk aan wat de globale variabelen zijn.
    b. Schrijf de monitoring taak.

10 Semaforen

In de voorafgaande hoofdstukken is op verschillende plaatsen gewezen op **transient errors**. Vooral als we de manipulatie van gemeenschappelijke data of devices door meerdere taken willen synchroniseren, zullen we bijzondere eisen moeten stellen die transient errors voorkomen. **Concurrency** kan dus heel vervelende problemen geven als we daar niet op de juiste manier mee omgaan. Indien we concurrency problemen niet principieel kunnen voorkomen, zullen we nooit betrouwbaar werkende software kunnen maken. Er moet dus een mechanisme zijn dat door het operating system wordt uitgevoerd en niet onderbroken kan worden door het optreden van een task switch. Operating systems ondersteunen daarvoor het zogenaamde semaforenmechanisme. De ontwikkelaar moet semaforen in de programma's opnemen. Achtergrondinformatie over semaforen is te vinden via **`man 7 sem_overview`**.

10.1 Concurrency problemen

Het onderstaande voorbeeld schetst een toepassing waarin een transient error naar voren komt. Het voorbeeld is een herhaling van het gegeven voorbeeld in §5.12 bij de behandeling van **transient errors**.

Stel dat een administratie wordt uitgevoerd op een computer. Ten behoeve van een magazijnadministratie kan de voorraad artikelen via een aantal terminals worden opgevraagd en kunnen eventuele bestellingen van de voorraad worden afgeboekt in de administratie op een hard disk (gemeenschappelijke data/device).

De volgende activiteiten in het multitasking multi-user systeem vinden plaats:

- Vrijwel gelijktijdig komen twee verzoeken binnen voor afboeking van hetzelfde artikel P. Van artikel P zijn er vijf in voorraad voordat deze bestellingen binnengekomen zijn. **{aantal P==5}**

- Via terminal A worden er van artikel P vier besteld, echter voordat deze vier zijn afgeschreven (= mutatie op de hard disk) komt de timeslice interrupt.

- Een tweede bestelling wordt via terminal B afgehandeld. De nog niet gemuteerde data op de hard disk wordt geraadpleegd. Er worden van artikel P drie besteld. De mutatie op de hard disk wordt uitgevoerd. **{aantal P=5-3=2}**

- Na enige tijd wordt de bestelling op terminal A verder afgewerkt. De resterende voorraad wordt berekend (gelijk aan 1) en dit wordt over de mutatie, door terminal B uitgevoerd, heen geschreven. **{aantal P=5-4=1}**

De voorraadadministratie geeft nu aan dat er van artikel P nog 1 in voorraad is (op de hard disk) en dit terwijl in werkelijkheid een tekort van 2 stuks is ontstaan. De consistentie van gegevens is in gevaar gekomen.

Om dit soort problemen te vermijden moet het lezen, modificeren en daarna weer terugschrijven, **Read-Modify-Write actie**, van gemeenschappelijke data een niet te

onderbreken actie of **atomaire actie** zijn voor elke taak die op dezelfde gemeenschappelijke data opereert.

Zogenaamde **semaforen** bieden de oplossing van het geschetste **concurrency** probleem bij het benaderen van gemeenschappelijke data. Deze problematiek speelt een belangrijke rol bij het gebruik van **databases** waar meerdere gebruikers (= taken) op hetzelfde moment transacties uitvoeren op gemeenschappelijke data.

In real-time systemen kan niet alleen data gemeenschappelijk zijn maar devices kunnen ook door verschillende taken gemeenschappelijk gebruikt worden. Een device is te beschouwen als een hoeveelheid data in de registers van de hardware van het device. We kunnen in een applicatie naar elk device een pad openen via de bijbehorende driver. Het creëren van paden naar hetzelfde device door verschillende taken is alleen mogelijk als het device **sharable** is. Dus ook hier kan concurrency problemen opleveren. Het is aan te bevelen een device maar door één taak te laten benaderen.

10.2 Record locking

Ook bij gemeenschappelijk gebruik van files kunnen problemen optreden. Stel taak A heeft een file voor update geopend. Na het lezen van een blok data door taak A, zal taak A eigenaar zijn van dit blok totdat opnieuw door taak A naar deze file is geschreven of gelezen. Taak B gaat van dezelfde file (records) gebruik maken. De file is dus een **shared file**. Dit geeft problemen die bekend zijn bij het gebruik van gemeenschappelijke resources. We moeten er op een of andere manier voor zorgen dat we records kunnen blokkeren voor andere taken: **record locking**.

Scenario **zonder record locking**:

- taak A leest de eerste 100 bytes van file X
- taak B leest de eerste 100 bytes van file X
- taak A schrijft belangrijke data naar dit blok terug
- taak B schrijft de ingelezen data onveranderd terug

Hierdoor is de update van taak A ongedaan gemaakt!

Scenario **met record locking**:

- taak A voert een record locking procedure uit op de eerste 100 bytes, totdat deze ge-update zijn
- taak B moet zolang wachten totdat taak_A de record locking heeft opgeheven

Problemen veroorzaakt door record locking:

- Door een lock langer vast te houden dan strikt noodzakelijk, kunnen andere taken onnodig lang geblokkeerd worden. Dit wordt aangeduid met **lockout** of **starvation** ('uithongeren').
- Er kan **deadlock** optreden (ook wel **deadly embrace** genoemd, zie H12).

10.3 Semaforen en basisoperaties op semaforen

Een **semafoor** (letterlijk vertaald: seinpaal) is een datastructuur die gebruikt wordt voor synchronisatie tussen taken en die de eerder geschetste concurrency problemen kan oplossen. Een **semafoor** is in feite niets anders dan een teller (integer variabele >= 0) en een bijbehorende wait queue die door het operating system worden bijgehouden.

We onderscheiden **vier basisoperaties** op semaforen:

1. de **declaratie** van een semafoor (eenmalig)
2. de **initialisatie** van een semafoor (eenmalig)
3. de **Passeeroperatie**, ook wel **P**-operatie of **Wait**-operatie (**Lock**; **-1**)
4. de **Verhoogoperatie**, ook wel **V**-operatie of Signal-operatie (**Post**; **Unlock**; **+1**)

We zullen bij de theorie van semaforen gebruikmaken van de pseudo-code. Een applicatie in C is te complex om de principes toe te lichten.

- De (globale) semafoorvariabele sem wordt gedeclareerd met:

  SEMAPHORE sem

  Het type SEMAPHORE geeft aan dat het om een variabele gaat die bij de kernel moet worden opgegeven om te bewaken. De waarde **sem=0** betekent dat de semafoor **gelocked** is; **sem>0** betekent **unlocked**.

- De semafoorvariabele sem krijgt de beginwaarde n met:

  SEMINIT (sem, n)

  Daar de kernel de initialisatie zal uitvoeren, kan niet volstaan worden met de eenvoudige notatie tussen {} om aan te geven wat SEMINIT() doet. Hier is **n** een integer expressie >= 0 (meestal een constante van het type integer).

- De passeeroperatie op een semafoor:

```
P (sem)   {
              IF sem = 0 THEN wait (sem);
              sem = sem - 1;
          }
```

  Bij de uitvoering van een P-operatie door een taak op een semafoor zal eerst geëvalueerd worden of het semafoorgetal nul is. Zo ja, dan wordt deze taak in een wachtrij gezet voor die semafoor (mogelijkheden zijn bijvoorbeeld een FIFO of een priority queue). Zo nee of na bevrijding uit de wait queue zal het semafoorgetal met 1 verlaagd worden. Hierbij moet het evalueren of sem > 0 is en het verlagen van het semafoorgetal een ondeelbare actie zijn.

- De verhoogoperatie op een semafoor:

```
V (sem)  {
             sem = sem + 1;
             signal (sem);
         }
```

Ook hier is het verhogen van sem een ondeelbare lees-modificeer-schrijf actie. Een eventuele wachtende taak voor **sem** zal uit de wait queue bevrijd worden.

10.4 Wederzijdse uitsluiting en mutexen

Om te voorkomen dat meer taken tegelijk dezelfde hulpbron (resource, data) kunnen gebruiken, wordt gebruikgemaakt van zogenaamde **binaire semaforen**. We spreken in dit geval van **wederzijdse uitsluiting** (mutual exclusion). Het gebruik van de hulpbron noemen we een **kritieke actie (KA)**. De kritieke actie wordt bewaakt door gebruik te maken van een speciale semafoor genaamd **mutex**.

Mutex staat voor **MUTual EXclusion** en is een binary semafoor met speciale eigenschappen. Zo kan alleen het proces dat een mutex heeft gelocked die mutex ook weer unlocken, dwz. **voor het locken en unlocken van een mutex is hetzelfde PID nodig**. Waar semaforen in het algemeen worden gebruikt voor synchronisatie, regelen mutexen **exclusieve toegang**. In een door mutexen bewaakt stuk code mogen meerdere threads naar binnen, maar steeds slechts een tegelijk. Je kunt een mutex een beetje vergelijken met het **slot op een toilet** – meerdere mensen mogen het gebruiken, maar niet tegelijkertijd – het gebruik is exclusief. Alleen de persoon die het slot heeft dichtgedraaid kan het ook weer openen. Het programma in pseudo-code hieronder geeft een voorbeeld van het gebruik van mutexen.

```
SEMAPHORE mutex
```

hoofdprogramma:

```
SEMINIT (mutex,1)                          {mutex = 1}
splits 3 cyclische concurrent child processen af:
taak, taak, taak
wacht op overlijden van de 3 children
```

taak:

```
DOE
    ...
    ...
    P (mutex)
    KA           {kritieke actie}
    V (mutex)
    ...
    ...
ZOLANG voorwaarde geldt
```

Als we nu even beschouwen dat er drie processen worden uitgevoerd met een identieke code, dan kan er slechts één tegelijk met zijn kritieke actie bezig zijn. Zodra namelijk een taak voor het eerst de P-operatie uitvoert, passeert deze zonder wachten dit punt en zal **mutex** 0 worden (locked).

Vervolgens gaat deze taak zijn kritieke actie uitvoeren. Als nu een van de andere twee taken de P-operatie uitvoert, dan wordt deze taak in de wait queue gezet omdat **mutex** immers 0 was. Ook de laatste taak kan op deze manier in de wait queue terechtkomen. Zodra de eerste taak met zijn kritieke actie klaar is, voert deze de V-operatie uit waardoor een wachtende taak voor **mutex** een signal ontvangt en dus uit de wait queue bevrijd wordt en vervolgens zijn kritieke actie zal gaan uitvoeren. Ondertussen is **mutex** achtereenvolgens 1 (unlocked) en weer 0 (locked) geworden. Waren er geen wachtende taken dan zal de V-operatie alleen het verhogen van **mutex** tot gevolg hebben. Omdat de in het voorafgaande probleem toegepaste semaforen alleen de waarden 0 of 1 kunnen aannemen, worden deze aangeduid als **binaire semaforen**. In §10.10 gaan we verder in op het gebruik van mutexen.

10.5 Synchronisatie

Als een taak moet wachten op acties van een andere taak, spreken we van **synchronisatie**. Dit doet zich bijvoorbeeld voor als het eindproduct van de ene taak de grondstof voor de andere taak levert. We zullen dit aan de hand van een voorbeeldprogramma nader bestuderen. Hierbij wordt gebruik gemaakt van twee taken, namelijk een **producer** die een product (data) aanmaakt en een **consumer** die dit product (data) gebruikt.

De productievoorraad wordt bijgehouden met de semafoor **syn** (synchronisatie). Het zal duidelijk zijn dat de consument moet wachten als er geen voorraad is (syn = 0). Het hoofdprogramma zal de producent en de consument opstarten en de semafoor **syn** op 0 initialiseren (geen beginvoorraad). Om het parallelle gedrag van de producent en de consument wat meer te accentueren zijn beide codes naast elkaar in plaats van onder elkaar geschreven.

```
   SEMAPHORE syn
```
hoofdprogramma:

```
   SEMINIT (syn, 0)                  {syn = 0}
   splits af de concurrent child processen:
   producent, consument
   wacht op het beëindigen van de twee child processen
```

producent:

```
   ...
   ...
   DOE
       ...
       produce
       V (syn);
       ...
   ZOLANG produceer
   ...
```

consument:

```
   ...
   ...
   DOE
       ...
       P (syn)
       consume
       ...
   ZOLANG consumeer
   ...
```

Het zal duidelijk zijn dat dit programma niet volledig is. Zo is niet aangegeven waar de producent zijn producten opslaat en waar de consument deze producten vandaan haalt. Men zou hiervoor een lijststructuur kunnen gebruiken.

De consument zal moeten wachten als de productievoorraad leeg is totdat een signaal wordt ontvangen van de producent. Als er voldoende voorraad is, zal de consument bij het uitvoeren van de P-operatie alleen **syn** met 1 verlagen. Een probleem vormt nog de overproductie. Er is namelijk geen begrenzing aan de bovenkant.

10.6 Buffergebruik en semaforen

Vaak doet zich bij het **producent-consument probleem** ook een opslagprobleem voor. Het aantal producten (de voorraad) wordt dan ook aan de bovenkant begrensd door de grootte van het magazijn (buffer of pool). Voor de opslag van de voorraad zal nu gebruikgemaakt worden van een zogenaamde **cyclische buffer**. Voor een cyclische buffer worden twee indices **IN** en **UIT** (of twee pointers) gebruikt. De IN-index wijst naar de bufferplaats waar een producent zijn product kwijt kan (eerste vrije bufferplaats). De UIT-index wijst naar de bufferplaats waar een consument een product kan ophalen. In fig. 10.1 wordt de actuele voorraad aangegeven door middel van arcering. Een probleem is hoe men kan constateren of de buffer vol of leeg is. De buffer is vol als de IN-index de UIT-index inhaalt. De buffer is leeg als de UIT-index de IN-index inhaalt.

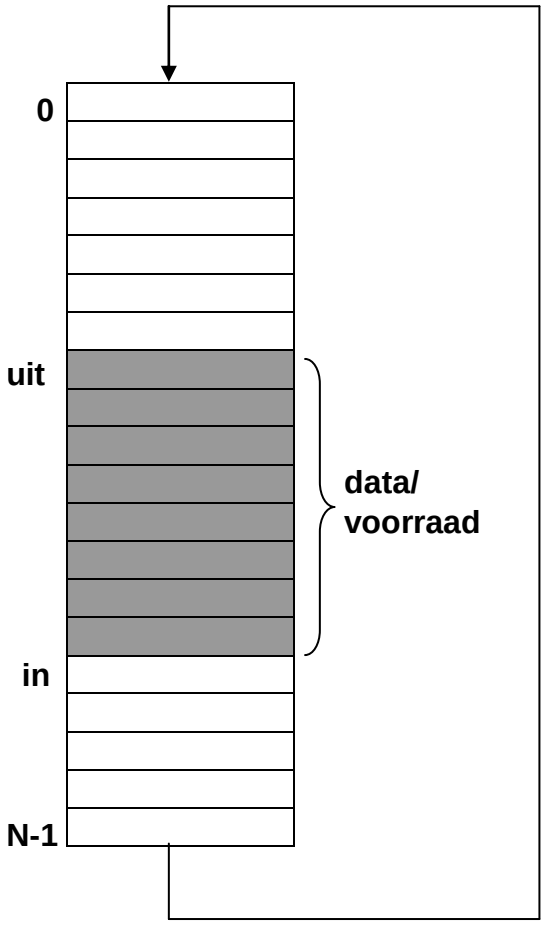


Fig. 10.1 Cyclische buffer met n elementen

Naast de indices IN en OUT gaan we ook nog twee semaforen gebruiken. De ene noemen we BEZET omdat deze het aantal door producten bezette plaatsen aangeeft. De andere semafoor noemen we VRIJ omdat deze het aantal vrije plaatsen in de buffer aangeeft. Normaal wordt BEZET op nul geïnitialiseerd (geen beginvoorraad). In dat geval moet VRIJ op N worden geïnitialiseerd.

```
N = 12   {aantal bufferplaatsen}
SEMAPHORE bezet, vrij
```

hoofdprogramma:

```
in = 0
uit = 0
SEMINIT (bezet,0)
SEMINIT (vrij,N)
splits af de concurrent child processen:
producent, consument
wacht op het overlijden van de twee child processen
```

producent:

```
...
DOE
   produce product p
   P (vrij)  // -1
   buffer[in] = p
   in = (in+1) modulo N
   V (bezet)  // +1
ZOLANG produceer
...
```

consument:

```
...
DOE
   P (bezet)  // -1
   q = buffer[uit]
   uit = (uit+1) modulo N
   V (vrij)  // +1
   consume product q
ZOLANG consumeer
...
```

Semaforen die waarden kunnen aannemen groter dan 1, worden **tellende semaforen** (counting semaphores) genoemd.

10.7 Verkeerd gebruik van semaforen kan leiden tot deadlocks

Stel er zijn twee taken, T1 en T2, die elk van vier beschikbare non-shareable resources er drie gelijktijdig nodig hebben om hun taak te kunnen uitvoeren. We komen in principe dus twee resources te kort om beide taken concurrent te laten werken. We gebruiken een tellende semafoor om het aantal nog beschikbare resources aan te geven.

```
SEMAPHORE teller
```

hoofdprogramma:

```
SEMINIT (teller, 4)
splitst af de concurrent child processen: T1, T2
wacht op het beëindigen van de twee child-processen
```

T1:

```
....
....
P (teller)
....
....
P (teller)
....
P (teller)
....
```

T2:

```
....
P (teller)
....
....
....
P (teller)
P (teller)
....
....
```

Stel dat T1 running is en in de beschikbare tijd in de timeslice worden alle drie de verlagingen van de semafoor uitgevoerd. T1 kan dan zijn werkzaamheden starten. Als T2 na de task switch running wordt, kan deze de semafoor maximaal nog maar één maal met 1 verlagen (als T1 nog geen resource heeft vrijgegeven). Daarna wordt T2 wachtend in de bij de semafoor behorende queue. Pas als T1 minimaal 1 keer de semafoor heeft verhoogd, kan T2 verdergaan. Als T2 zijn derde resource wil reserveren, zal hij opnieuw moeten wachten totdat T1 nogmaals de semafoor heeft verhoogd. Dan kan T2 verdergaan met het uitvoeren van de taak waarvoor de drie resources nodig zijn.

Tot zover lijkt alles in orde. Maar stel dat de task switch in T1 komt nadat de semafoor twee maal en niet drie maal verlaagd is. Daarna komt T2 aan de beurt en deze kan in de toegewezen tijd de semafoor maximaal met 2 verlagen (niet lager want er zijn nog maar twee resources beschikbaar). Nu zijn alle resources aangevraagd. T1 blijft wachten en T2 blijft wachten. Beide hebben nu geen enkele mogelijkheid een resource vrij te geven. Deze situatie betekent dat beide taken zijn vastgelopen en er niet meer uit kunnen komen. Dit wordt een **deadlock**situatie genoemd.

Het voorkomen van de geschetste deadlock situatie is als volgt: Verlaag de semafoor niet drie maal gescheiden met 1 maar **verlaag de waarde in één keer met 3**. Indien er minder dan 3 resources beschikbaar zijn dan wordt de taak in de queue voor de bijbehorende semafoor geplaatst. Het testen en verlagen van de semafoor is een **atomaire actie**, met andere woorden een ondeelbare actie (= niet te onderbreken door een task switch).

Samenvattend kan gesteld worden dat indien een taak meerdere met andere taken gemeenschappelijk resources nodig heeft, deze in één keer moeten worden aangevraagd om deadlock te voorkomen. Het vrijgeven kan in principe één voor één gebeuren. Het is natuurlijk verstandig om een gemeenschappelijke resource zo kort mogelijk bezet te houden. Dit voorkomt dat andere taken onnodig (lang) moeten wachten.

10.8 Semaforen en multitasking

Bij het gebruik van semaforen moet de optie **-lpthread** bij het linken worden gebruikt!

10.8.1 System calls voor unnamed semaphores

De system calls voor het gebruik van unnamed semaphores ziet er als volgt uit.

```
#include  <semaphore.h>

sem_t  mySem;

int  sem_init (sem_t *pSem, int iShared, unsigned int uiBeginwaarde);
```

iShared = 0: bij process threads, waarbij mySem globaal moet zijn

iShared ≠ 0: bij verschillende processen, waarbij mySem in shared memory moet staan

Men spreekt hier van een zogenaamde **unnamed semaphore**. Voorbeeld:

```
sem_init (&mySem, 1, 1);
sem_wait (&mySem);          /* P-actie/lock/-1 */
sem_post (&mySem);          /* V-actie/lock/+1 */
```

Het initialiseren van een semafoor die al geïnitialiseerd is, leidt tot ongedefinieerd gedrag! De returnwaarde van deze functies is 0 bij succes en -1 als er een fout is opgetreden.

```
int  sem_destroy (sem_t *pSem);
```

vernietigt alleen een semafoor die door sem_init() is geïnitialiseerd. Het gaat hierbij dus om een unnamed semaphore. Als er processen door deze semafoor geblokkeerd zijn of als processen deze semafoor willen gebruiken, leidt dit tot ongedefinieerd gedrag. Wel kan de semafoor opnieuw worden geïnitialiseerd met sem_init().

10.8.2 System calls voor named semaphores

Naast unnamed semaphores bestaan er ook **named semaphores**. Dit type semafoor is in het **hele systeem bekend**. De naam van een named semaphore begint altijd met een slash (/). Een bestaande named semaphore kan als volgt worden geopend:

```
sem_t *  sem_open (const char *pcSemNaam, int iFlag);
```

Indien een named semaphore niet bestaat kan deze bij openen worden aangemaakt met **iFlag = O_CREAT**; hierbij zijn twee extra parameters nodig:

sem_t * sem_open (const char *pcSemNaam, int iFlag,
mode_t permissies, unsigned int uiBeginwaarde);

Voorbeeld:

sem_t *pMySem;
pMySem = sem_open("/mysem", O_CREAT, 0640, 1)

Als **iFlag = O_CREAT | O_EXCL** wordt de semafoor alleen aangemaakt en geopend als hij niet bestaat. De returnwaarde is een pointer naar de semafoor of de foutmelding SEM_FAILED.

int sem_close (sem_t *pSem);

Hiermee wordt de named semaphore gesloten.

int sem_unlink (const char *pcSemNaam);

verwijdert de semafoornaam en de bijbehorende named semaphore. Indien sem_unlink() niet wordt aangeroepen blijft de semafoor bestaan zolang het systeem draait. Voor de P-actie en V-actie worden net als bij unnamed semaphores de system calls

sem_wait (&mySem); /* P-actie/lock/-1 */
sem_post (&mySem); /* V-actie/lock/+1 */

gebruikt. Merk op dat sem_open() en sem_unlink() de **naam van de semafoor** gebruiken, terwijl de andere functies de **semafoor-ID** (pSem in dit voorbeeld) nemen.

10.8.3 Codevoorbeelden

Eerst laten we een programma zien dat **niet werkt**. Een parent en een child process voeren beide een kritieke actie KA uit, die 10 ms duurt (m.b.v. mSleep() uit § 6.5). Na het uitvoeren van de kritieke actie wacht de parent 30 ms en de child 13 ms. De tijden zijn zo gekozen dat het effect goed zichtbaar is en de kritieke actie van het ene process een grote kans loopt door het andere proces te worden onderbroken. Het juiste gebruik van semaforen zou dit moeten voorkomen. In de programma's zijn de controles op het correct uitvoeren van de system calls weggelaten vanwege het overzichtelijk weergeven van de essentie.

```
// Voorbeeld 10.8.1: unnamed semaphore, niet werkend! (zie Fig 10.2)
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <wait.h>
#include <time.h>

#define MAXLOOP 10
```

```
int main (void) {
     int i = 0;
     sem_t mySem;
     sem_init (&mySem, 1, 1);
     switch (fork()) {
          case -1: /*error*/
               printf ("Geen child.\n");
               break;
          case 0: /*Child process*/
               for (i = 0; i < MAXLOOP; i++) {
                    sem_wait (&mySem);
                    printf ("Child start KA\n");
                    mSleep (10);
                    printf ("Child stopt KA\n");
                    sem_post (&mySem);
                    mSleep (13);
               }
               break;

         default: /*Parent process*/
               for (i = 0; i < MAXLOOP; i++) {
                    sem_wait (&mySem);
                    printf ("Parent start KA\n");
                    mSleep (10);
                    printf ("Parent stopt KA\n");
                    sem_post (&mySem);
                    mSleep (30);
          }
               wait (NULL);
               sem_destroy (&mySem);
          break;
     }
     return 0;
}
```

Het resultaat van het bovenstaande programma is te zien in fig. 10.2. Weliswaar erft de child alle gegevens van de parent, maar hij krijgt een eigen datagebied toegewezen waarin alle variabelen worden gekopieerd. Pointers wijzen naar dit nieuwe datagebied. Dit betekent dat ook de semafoor wordt gekopieerd en dat wijzigingen van de semafoor door het ene proces niet door het andere proces worden gezien.

Fig. 10.3 toont de output van het onderstaande programma waarbij de semafoor correct werkt en ervoor zorgt dat de kritieke actie niet onderbroken wordt. Hierbij is het probleem opgelost door de semafoor in shared memory te plaatsen.

Fig. 10.2 KA wordt onderbroken

Fig. 10.3 KA wordt niet onderbroken

```
// Voorbeeld 10.8.2: unnamed semaphore, werkend (zie Fig. 10.3)
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <wait.h>
#include <time.h>

#define MAXLOOP 10

int main (void) {
    int iShmID;
    sem_t *pmySem;
    int i = 0;
    iShmID = shmget (IPC_PRIVATE, sizeof (sem_t),
                     IPC_CREAT | 0600);
    pmySem = shmat (iShmID, NULL, 0);
    sem_init (pmySem, 1, 1);
    switch (fork()) {
        case -1: /*error*/
            printf ("Geen child.\n");
```

```
            break;
        case 0: /*Child process*/
            for (i = 0; i < MAXLOOP; i++) {
                sem_wait (pmySem);
                printf ("Child start KA\n");
               mSleep (10);
               printf ("Child stopt KA\n");
               sem_post (pmySem);
                mSleep (13);
           }
            shmdt (pmySem);
           break;
        default: /*Parent process*/
           for (i = 0; i < MAXLOOP; i++) {
               sem_wait (pmySem);
               printf ("Parent start KA\n");
               mSleep (10);
               printf ("Parent stopt KA\n");
               sem_post (pmySem);
                mSleep (30);
           }
            shmdt (pmySem);
            wait (NULL);
            shmctl (iShmID, IPC_RMID, 0);
            sem_destroy (pmySem);
           break;
    }
    return 0;
}
```

Het voorbeeld hieronder toont een (correcte) toepassing van een **named semaphore**:

```
// Voorbeeld 10.8.3: named semaphore
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <wait.h>
#include <time.h>

#define MAXLOOP 10

int main() {
    int i = 0;
    sem_t *pmySem;
    pmySem = sem_open ("/sem_mySem", O_CREAT, 0600, 1);
    switch (fork()) {
        case -1: /*error*/
            printf ("Geen child.\n");
            break;
        case 0: /*Child process*/
            for (i = 0; i < MAXLOOP; i++) {
```

```
                sem_wait (pmySem);
                printf ("Child start KA\n");
               mSleep (10);
               printf ("Child stopt KA\n");
               sem_post (pmySem);
                mSleep (13);
            }
             sem_close (pmySem);
            break;
        default: /*Parent process*/
            for (i = 0; i < MAXLOOP; i++) {
             sem_wait (pmySem);
             printf ("Parent start KA\n");
             mSleep (10);
             printf ("Parent stopt KA\n");
             sem_post (pmySem);
                 mSleep (30);
            }
             sem_close (pmySem);
             wait (NULL);
             sem_unlink ("/sem_mySem");
            break;
    }
    return 0;
}
```

Het gebruik van **named semaphores** werkt ook voor processen die geen relatie met elkaar hebben. Een van de taken creëert de semafoor en voorziet die van een beginwaarde. De andere taken wachten totdat de semafoor bestaat of totdat ze een teken via een fifo ontvangen dat de semafoor is gecreëerd.

```
// Example 10.8.4a: named semaphore, unrelated processes, sync
through sem_open()
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

#define MAXLOOP 10

int main() {
    int i = 0;
    sem_t *pmySem;
    pmySem = sem_open ("/sem_mySem", O_CREAT, 0600, 1);
    for(i = 0; i < MAXLOOP; i++) {
         sem_wait (pmySem);
         printf ("Task1 starts CA\n");
        mSleep (10);
        printf ("Task1 stops CA\n");
        sem_post (pmySem);
         mSleep (13);
    }
```

```
    sem_close (pmySem);
    sem_unlink ("/sem_mySem");

    return 0;
}


// Example 10.8.4b - run with 10.8.4a
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

#define MAXLOOP 10

int main() {
    int i = 0;
    sem_t *pmySem;
    while ((pmySem = sem_open ("/sem_mySem", 0)) == SEM_FAILED) {
        printf ("NomySem in Task2\n");
        mSleep (50);
    }
    for(i = 0; i < MAXLOOP; i++) {
        sem_wait (pmySem);
        printf ("Task2 starts CA\n");
        mSleep (10);
        printf ("Task2 stops CA\n");
        sem_post (pmySem);
        mSleep (30);
    }
    sem_close (pmySem);

    return 0;
}
```

Het uitvoeren van de twee taken in dit voorbeeld (na compileren) gaat als volgt (probeer ook de andere volgorde en het weglaten van de ampersand):

```
$ ./ex_4b &
$ ./ex_4a
```

Een semafoor kan natuurlijk ook worden gebruikt voor het **synchroniseren** van processen. Hierbij moet de beginwaarde van de semafoor 0 zijn.

```
// Example 10.8.5a: named semaphores, unrelated processes
// Use sem_post() and sem_wait() to determine the order of execution
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
```

```
int main() {
     int i = 0;
     sem_t *psync;

     psync = sem_open ("/sem_sync", O_CREAT, 0600, 0);
     printf ("Process A should run first\n");
     sem_post (psync);
     sem_close (psync);
     return 0;
}

// Example 10.8.5b: run with 10.8.5a
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

int main() {
     int i = 0;
     sem_t *psync;

     while ((psync = sem_open ("/sem_sync", 0)) == SEM_FAILED) {
          printf ("No sync at task B\n");
          mSleep (50);
     }
     sem_wait (psync);
     printf ("Process B should follow A\n");
     sem_close (psync);
     sem_unlink ("/sem_sync");

     return 0;
}
```

Het uitvoeren van de twee taken in dit voorbeeld (na compileren) gaat als volgt (probeer ook de andere volgorde en het weglaten van de ampersand):

```
$ ./ex_5b &
$ ./ex_5a
```

10.9 Semaforen en multithreading

Semaforen kunnen natuurlijk ook worden toegepast bij **threading**. In het volgende voorbeeld is de ene thread **producer** van data en de andere thread is **consumer**. Met de delay in de producer zorgen we ervoor dat producer zijn data opslaat in de buffer nadat de consumer de informatie uit de buffer gelezen heeft. Zonder gebruik te maken van de semafoor **synchr** resulteert dit in het lezen van de verkeerde data door de consumer. Hij wacht immers niet op het schrijven van de producer. Het resultaat is dat de oude (ongeldige) data worden afgedrukt:

```
Uitvoer van consument: 0123456789
```

We zorgen ervoor dat de synchronisatie wordt uitgevoerd met een semafoor (synchr). Nu worden wel de correcte data gelezen en afgedrukt:

```
Uitvoer van consument: abcdefghijklmnopqrstuvwxyz
```

```
// Voorbeeld 10.9.1
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>

void *Prod (void *pArgP);
void *Cons (void *pArgC);

char acBuffer[30] = "0123456789";
sem_t synchr;

int main (void) {
    pthread_t threadProd, threadCons;
    sem_init (&synchr, 0, 0);

    (void) pthread_create (&threadProd, NULL, Prod, NULL);
    (void) pthread_create (&threadCons, NULL, Cons, NULL);
    (void) pthread_join (threadProd, NULL);
    (void) pthread_join (threadCons, NULL);

    sem_destroy (&synchr);

    return 0;
}

void *Prod (void *pArgP) {
    char cL;
    int i = 0;
    mSleep (30);
    for (cL = 'a'; cL <= 'z'; cL++, i++) {
        acBuffer[i] = cL;
    }
    acBuffer[i] = '\0';
    sem_post (&synchr);
    return NULL;
}

void *Cons (void *pArgC) {
    sem_wait (&synchr);
    printf ("Uitvoer van consument: %s\n", acBuffer);
    return NULL;
}
```

Willen we dat de producer meerdere keren data produceert, die telkens worden gelezen door de consumer, dan hebben we twee semaforen nodig. De semafoor ProdMag (beginwaarde 1) geeft aan dat de producer mag produceren. Deze moet beginnen en daarna telkens wachten totdat de consumer aangeeft dat hij klaar is met con-

sumeren. De consumer maakt ProdMag telkens 1 na het consumeren. ConsMag (beginwaarde 0) zorgt ervoor dat de consumer telkens pas kan beginnen nadat de producer geproduceerd heeft. De producer maakt hiertoe ConsMag telkens 1 na het produceren. De regels voor lock/KA/unlock staan steeds in for loops.

```
// Voorbeeld 10.9.2: producer en consumer
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>

void *Prod (void *pArgP);
void *Cons (void *pArgC);

char acBuffer[30] = "xxx";
sem_t ProdMag, ConsMag;

int main (void) {
    pthread_t threadProd, threadCons;
    sem_init (&ProdMag, 0, 1);
    sem_init (&ConsMag, 0, 0);

    (void) pthread_create (&threadProd, NULL, Prod, NULL);
    (void) pthread_create (&threadCons, NULL, Cons, NULL);
    (void) pthread_join (threadProd, NULL);
    (void) pthread_join (threadCons, NULL);
    sem_destroy (&ProdMag);
    sem_destroy (&ConsMag);

    return 0;
}

void *Prod (void *pArgP) {
    char cL = '1';
    int i, j, iMax = 9;
    for (i = 0; i < 5; i++, iMax--) {
        mSleep (12);
        sem_wait (&ProdMag);
        for (j = 0; j < iMax; j++) {
            acBuffer[j] = cL + j;  // Produce
        }
        acBuffer[j] = '\0';
        sem_post (&ConsMag);
        cL = cL + 16;
    }
    return NULL;
}

void *Cons (void *pArgC) {
    int i;
    for (i = 0; i < 5; i++)
    {
        sem_wait (&ConsMag);
```

```
        printf("Uitvoer van consument: %s\n", acBuffer);  // Consume
        sem_post (&ProdMag);
    }
    return NULL;
}
```

De uitvoer van dit programma is:

```
Uitvoer van consument: 123456789
Uitvoer van consument: ABCDEFGH
Uitvoer van consument: QRSTUVW
Uitvoer van consument: abcdef
Uitvoer van consument: qrstu
```

Zonder gebruik van de semaforen zou de uitvoer als volgt zijn:

```
Uitvoer van consument: xxx
Uitvoer van consument: xxx
Uitvoer van consument: xxx
Uitvoer van consument: xxx
Uitvoer van consument: xxx
```

10.10 Condition variabelen en multithreading

Condition variabelen (CVs) worden gebruikt in combinatie met **mutexen** om de samenwerking van threads te synchroniseren op basis van een conditie (logische voorwaarde) gerelateerd aan de gedeelde data. Denk bijvoorbeeld aan de conditie of een queue leeg of vol is. Zonder een condition variabele moet er gebruik worden gemaakt van **polling**. Dit heeft als nadeel dat de processor onnodig veel belast wordt met “niets doen” (busy waiting). Let erop dat deze voorwaarde te toetsen is met een variabele die niet de condition variabele is! Een condition variabele is dus geen gewone variabele waaraan we toekenningen kunnen doen. Een condition variabele moet altijd gebruikt worden in samenwerking met een mutex. Alle operaties op een condition variabele moeten gedaan worden als deze **mutex gelockt** is.

De system call **pthread_cond_wait()** unlockt de mutex en laat vervolgens de thread slapen totdat de bijbehorende conditional variable (CV) wordt gesignalled. Bij het ontvangen van dat signal ontwaakt de thread, wordt de mutex gelocked en returnt de functie, waarna het programma verder loopt. Zie **`man pthread_cond_timedwait`** voor meer details.

De system call **pthread_cond_signal()** stuurt het signal dat bij de CV hoort. Dit wekt een (van de) slapende threads op die op dit signal wachten (als die bestaan). Voordat deze functie wordt aangeroepen moet de mutex die hoort bij de CV zijn gelocked door de aanroepende functie. Deze functie houdt de mutex; de ontwakende thread kan pas veder wanneer deze is vrijgegeven.

De system call **pthread_cond_broadcast()** werkt als pthread_cond_signal(), maar stuurt een signal naar **alle slapende threads** die wachten op het signaal bij deze

CV. Alle drie de system calls moeten altijd omringd worden door mutex lock/unlock calls!

```
// Voorbeeld 10.10.1
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

volatile int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit(void) {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);  // Signal a condition: this wakes up
                              //        1 thread waiting on the CV
    pthread_mutex_unlock(&m);
}

void *child(void *arg) {  /* thread function */
    printf("child\n");
    thr_exit();  // Local function
    return NULL;
}

void thr_join(void) {
    pthread_mutex_lock(&m);
    while (done == 0) {
        pthread_cond_wait(&c, &m);  // man pthread_cond_timedwait
    }
    pthread_mutex_unlock(&m);
}

int main() {
    printf("parent: begin\n");
    pthread_t p;
    if (pthread_create(&p, NULL, child, NULL) != 0) {
        fprintf(stderr, "Cannot create thread");
        exit(EXIT_FAILURE);
    }
    thr_join();  // Local function
    printf("parent: end\n");

    return 0;
}
```

10.11 Opgaven

1. Beschouw het volgende concurrency probleem. Er zijn twee taken T1 en T2. Er wordt gelezen en geschreven in de globale variabelen A, B en Som, waarbij de beginwaarden van A en B resp. 5 en 10.

Ga na wat de mogelijke uitkomsten zijn voor de data A, B en Som. Ga er vanuit dat elke actie die genoemd is, atomair is. Neem op allerlei momenten een task switch aan en ga na wat de waarde is geworden van de verschillende variabelen na beëindiging van beide taken. Dit probleem wordt het **inconsistant analysis** probleem genoemd.

T1:
```
Lees A uit file
A = A + 1
schrijf A in file
Lees B uit file
B = B + 1
schrijf B in file
```

T2:
```
Som = 0
lees A uit file
Som = Som + A
lees B uit file
Som = Som + B
schrijf Som in file
```

2. Wat is de grootste en de kleinste waarde die **mutex** kan aannemen in het getoonde programma betreffende wederzijdse uitsluiting in §10.4?

3. Een semafoor voor wederzijdse uitsluiting wordt ook wel **binaire semafoor** genoemd. Wat is de reden dat ook voor deze naam gekozen kan worden?

4. Wat zijn de kleinste en grootste waarde van **bezet**, **vrij**, **in** en **uit** in het behandelde programma in §10.6?

5. Het volgende probleem moet met inzet van een semafoor en de bijbehorende operaties opgelost worden: *er mag niet met een volle mond gepraat worden*. We beschouwen twee taken: **eten** en **praten**. Neem de procedures **eten** en **praten** over en plaats daarin op een of meer plekken (....) de juiste semafooroperatie (P, V of geen operatie) om te voorkomen dat er gelijktijdig gegeten en gepraat wordt. Geef tevens in het hoofdprogramma de juiste initialisatie van de semafoor **mond**.

```
SEMAPHORE mond
```

hoofdprogramma:

```
SEMINIT (mond, ...)       {<---- semafoor initialiseren}
splits af de concurrent child processen:
praten eten
wacht op het overlijden van de twee child processen
```

praten:
```
---
DOE
   ...
   kletsen
   ...
   ...
ZOLANG info
---
```

eten:
```
---
DOE
   ...
   neem_hap_en_kauw
   slik_door
   ...
ZOLANG trek
---
```

6. Drie cyclische processen (bevatten oneindige lus) moeten om de beurt processortijd krijgen. Volgorde: TaakA, TaakB, TaakC, TaakA, TaakB, TaakC, TaakA, … enz.

   Daarvoor gebruiken we drie semaforen: **beurtA**, **beurtB** en **beurtC**.
   Neem de processen **TaakA, TaakB** en **TaakC** over en plaats daarin op een of meer plekken (....) de juiste semafooroperatie (P of V) of geen operatie om te zorgen dat de taken om de beurt aan de orde komen.

   Geef tevens in het hoofdprogramma de juiste initialisatie van de semaforen **beurtA, beurtB en beurtC**.

```
SEMAPHORE beurtA, beurtB, beurtC
```

hoofdprogramma:

```
SEMINIT (beurtA, ...)     { <---- semaforen initialiseren }
SEMINIT (beurtB, ...)
SEMINIT (beurtC, ...)
splits af de concurrent child processen:
TaakA, TaakB, TaakC
wacht op het beëindigen van de drie child processen
```

TaakA:

```
---
DOE
    ....
    kritieke_actie
    ....
ZOLANG bezig
---
```

TaakB:

```
---
DOE
    ....
    kritieke_actie
    ....
ZOLANG actief
---
```

TaakC:

```
---
DOE
    ....
    kritieke_actie
    ....
ZOLANG doorgaan
---
```

7. Geef de C-code voor het gebruik van twee semaforen zodanig dat taak X het eerst wordt uitgevoerd, daarna Y, dan weer X enz.

11 Deadlocks

11.1 Inleiding

In voorgaande hoofdstukken hebben we gezien hoe **multitasking en multithreading** kunnen leiden tot **data races**, die vervolgens met behulp van **semaforen** kunnen worden voorkomen. Onjuist gebruik van semaforen en mutexen bij het regelen van de toegang tot resources voor verschillende taken kan leiden tot het 'vastlopen' van de uitvoering van de betrokken taken. Dit wordt een **deadlock** genoemd. Een eenvoudig voorbeeld van een deadlock is het ‘ABBA-deadlock’. Om het probleem van deadlocks verder duidelijk te maken wordt er in de literatuur vaak een klassiek voorbeeld gegeven van een deadlockprobleem: 'the dining philosophers problem'. Deze algemene versie gaat nog niet in op real-time eisen. De variant die daar wel op in gaat wordt 'the dying, dining philosophers problem' genoemd.

11.1.1 Het ABBA-deadlock

Het **ABBA-deadlock**, ook wel **deadly embrace** genoemd, is een eenvoudig voorbeeld om het verschijnsel deadlock duidelijk te maken. Stel, er zijn twee taken (of threads) T_1 en T_2 en twee resources, A en B, beveiligd door mutexen. Beide taken hebben beide resources tegelijk nodig om te kunnen runnen. Als T_1 mutex A aanvraagt, en T_2 mutex B, en vervolgens wacht T_1 op B en T_2 op A (A-B-B-A), dan is een ABBA-deadlock ontstaan. Elk van de taken heeft één mutex gelocked en wacht op de andere mutex. De twee taken hebben elkaar dus in een ‘dodelijke omhelzing’, waaruit ze niet kunnen ontsnappen, en het systeem komt tot stilstand. Een simpele algemene oplossing ligt in het aanvragen van de mutexen in een vaste volgorde. Dit vergt goede planning en documentatie bij het ontwikkelen van het system, in het bijzonder bij het nesten van mutexen. Een betere oplossing voor dit simpele specifieke geval is het locken van de twee mutexen **atomair** te maken.

11.1.2 The dining philosophers problem

Vijf filosofen hebben als levenstaak filosoferen en eten. Ze doen altijd slechts één van de twee. De filosofen kunnen een kamer betreden en aanzitten aan een ronde tafel met vijf stoelen. In het midden van de tabel staat een pan met rijst. Op tafel liggen verder vijf eetstokjes. Tussen elke twee filosofen ligt één eetstokje! De filosofen communiceren niet met elkaar. Voor het eten zijn twee eetstokjes nodig. Een filosoof kan per handeling maar één eetstokje pakken! Indien er een stokje beschikbaar is, wordt dit gepakt en kan niet meer worden teruggelegd. Als een filosoof eet, dan eet deze net zolang tot de honger over is. Als een filosoof klaar is met eten, worden beide stokjes neergelegd en begint hij weer te filosoferen. Mogelijk kunnen dan anderen weer gaan eten.

Indien op gegeven moment alle filosofen maar één stokje in de hand hebben, is er een deadlock-situatie opgetreden. Geen enkele filosoof komt meer aan het eten, want zij kunnen niet door overleg een stokje neerleggen.

We kunnen door middel van het toepassen van een semafoor voorkomen dat twee buren willen gaan eten. Ook in dat geval is de kans op optreden van deadlock echter niet te voorkomen.

Hier volgen enkele suggesties voor de oplossing van het genoemde probleem, het voorkomen van een **deadlock**:

- **Oplossing 1**
  Zorg dat er één chop stick meer is dan er filosofen zijn (**meer resources**). Dit voorkomt deadlock doordat er altijd wel ergens twee eetstokjes beschikbaar zijn zodat er een filosoof kan eten. Er kan wel **starvation** optreden, dat wil zeggen, er wordt uiteindelijk wel gegeten, maar mogelijk te laat (waarmee een realtime-eis wordt geschonden). Een algemene techniek om starvation te voorkomen is om de **age of requests for chopsticks** bij te houden. Als een aanvraag een te hoge age krijgt, dan worden aanvragen met een lagere age opgehouden totdat de aanvraag met de hoogste age kan worden uitgevoerd. Met deze aanpak zal elke filosoof uiteindelijk eten.

- **Oplossing 2**
  Een filosoof mag alleen twee stokjes in één enkele handeling pakken (**atomaire actie**). Het is dus niet meer toegestaan om een enkel eetstokje te pakken. Dit voorkomt deadlock, maar hier dienen ook maatregelen genomen te worden die starvation voorkomen.

- **Oplossing 3**
  De filosofen maken gebruik van een protocol (**beter ontwerp**). Bijvoorbeeld: elke filosoof krijgt een volgnummer. De filosofen met het even nummer mogen alleen eerst het rechter en dan pas het linker stokje oppakken. Voor de filosofen met een oneven nummer geldt het omgekeerde, eerst links dan rechts oppakken. Deze aanpak vermijdt deadlock maar niet starvation.

11.1.3 The dying, dining philosophers problem

Willen we het voorafgaande probleem real-time gaan maken, dan moeten we deadlines invoeren. We kunnen bijvoorbeeld stellen dat er voor elke filosoof een maximale tijd gesteld moet zijn waarvóór het eten moet zijn gestart. Indien een filosoof niet gestart is voor haar of zijn deadline, dan is er sprake van verhongering en de filosoof gaat dood.

De genoemde oplossingen bij het niet-realtime-probleem in de vorige paragraaf voldoen niet. De filosofen zullen uiteindelijk eten, maar deadlines zijn niet gegarandeerd.

De oplossing ligt in het theoretische vlak (hier wordt wetenschappelijk onderzoek naar gedaan). Elk moment dat er een nieuwe filosoof aan tafel verschijnt (= wil gaan eten), wordt een administratie geraadpleegd die heeft geregistreerd hoeveel eetstokjes (resources) er bezet zijn en hoeveel eettijd elke filosoof al heeft verbruikt. Uit deze gegevens worden maximaal **n!** volgorden berekend (’n faculteit’; n = alle etende filosofen inclusief de nieuwe). Er wordt geëvalueerd of de vereiste deadlines gehaald

kunnen worden (check for feasability). Onderzoek richt zich op het ontwikkelen van **heuristieken** om te voorkomen dat we worst case **n!** volgorden moeten evalueren.

11.2 Eisen voor het optreden van deadlock

Deadlock treedt dan en slechts dan op als er aan alle vier de voorwaarden hieronder op één moment voldaan is (de Coffman conditions: **voldoende en noodzakelijke voorwaarden**):

1. **wederzijdse uitsluiting (mutual exclusion)**
   De resource kan maar door één taak gebruikt worden (bijvoorbeeld een printer).

2. **bezet-houden-en-wachten (hold and wait)**
   Er moet minimaal één taak zijn die een resource bezet houdt en wacht op additionele resources die bezet worden gehouden door andere taken. Dit houdt in dat deze taak de benodigde resources achter elkaar tracht te reserveren, dus niet tegelijk (**incremental reservation**).

3. **voortijdige ontneming is niet mogelijk (no preemption)**
   Een resource kan alleen vrijgegeven worden als een taak ermee klaar is.

4. **wachten-in-een-kring (circulair wait)**
   De taken staan op elkaar te wachten in een cyclische keten.

Bijvoorbeeld er zijn **n** taken, die als volgt op elkaar staan te wachten:

```
p1    wacht op een resource die gebruikt wordt door p2
p2    wacht op een resource die gebruikt wordt door p3
p3    wacht op een resource die gebruikt wordt door p4
..    ..........................................  ..
pn-1  wacht op een resource die gebruikt wordt door pn
pn    wacht op een resource die gebruikt wordt door p1
```

Een **deadlock** kan voorkomen worden als steeds aan één of meer van de genoemde voorwaarden niet is voldaan.

11.3 De toewijzingsgraaf

Voor het onderzoek naar deadlock maken we gebruik van een zogenaamde **toewijzingsgraaf (resource-allocation graph, RAG)**. De toewijzingsgraaf is een grafische voorstelling waarin duidelijk wordt gemaakt welke taken welke systeemelementen (resources) bezet houden en aanvragen. In bepaalde gevallen zijn systeemelementen meervoudig uitgevoerd en niet van elkaar te onderscheiden. Een taak kan dan willekeurig één van de meervoudige systeemelementen kiezen omdat ze identiek zijn uitgevoerd.

Voor de volgende voorbeelden gelden in elk geval de deadlock eisen 1 (wederzijdse uitsluiting) en 3 (voortijdige ontneming is niet mogelijk).

- **Voorbeeld 1** (zie fig. 11.1)

  Er zijn vier systeemelementen (resources): R1, R2 (dubbel uitgevoerd) en R3.
  Er zijn drie taken: T1, T2 en T3.

taak	**toegewezen resource**	**aangevraagde resource**
T1	R2	R1
T2	R1 R2	R3
T3	R3	-

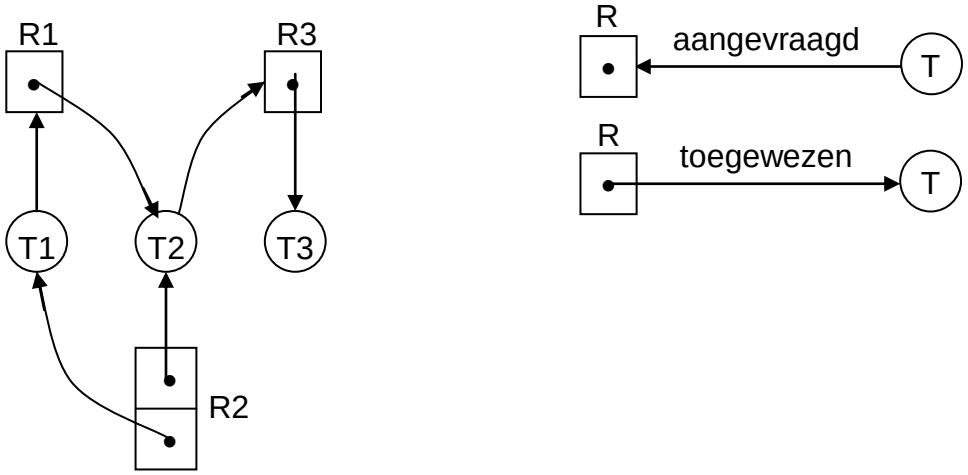


Fig. 11.1 Toewijzingsgraaf van voorbeeld 1

De voorafgaande tabel is nu weer te geven als een **graaf**. De systeemelementen geven we weer als vierkanten of rechthoeken. De taken geven we weer met cirkels. Indien een taak een systeemelement heeft toegewezen gekregen, tekenen we een pijl vanuit het midden (zwaartepunt) van het systeemelement naar de rand van de cirkel die de bijbehorende taak voorstelt. Indien een taak een systeemelement aanvraagt, wordt een pijl getekend van de rand van de cirkel naar de buitenrand van het geclaimde systeemelement. Het is essentieel deze conventie voor het maken van een **gerichte graaf** strikt te volgen. We kunnen dan zien of er sprake is van een lus van op elkaar wachtende taken (cyclische keten, deadlock eis 4). In veel gevallen wordt de graaf het meest **overzichtelijk** wanneer de taken meer in het midden van de figuur staan, en de resources meer naar buiten (zie de voorbeelden in dit hoofdstuk).

Indien we de pijlen in fig. 11.1 gaan volgen, zien we geen lus. Met andere woorden er kan hier geen deadlock optreden. De taken kunnen als volgt worden uitgevoerd:

T3 gebruikt R3 en R3 is na beëindiging van T3 beschikbaar.
T2 kan nu zijn taak uitvoeren en na beëindiging van T2 staan R1, R2 en R3 weer ter beschikking voor andere taken.
T1 heeft al R2 en krijgt daarbij ook R1 en kan dus zijn taak uitvoeren.

- **Voorbeeld 2** (zie fig. 11.2)

  In dit voorbeeld gaan we uit van voorbeeld 1, maar nu vraagt T3 ook een systeemelement R2 aan.

taak	**toegewezen resource**	**aangevraagde resource**
T1	R2	R1
T2	R1 R2	R3
T3	R3	R2

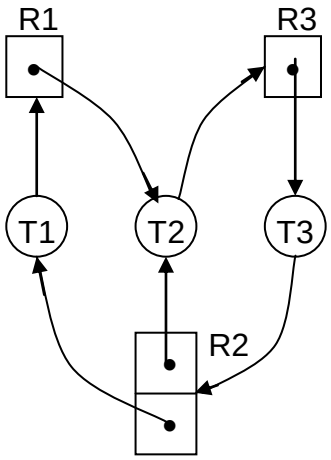


Fig. 11.2 Toewijzingsgraaf van voorbeeld 2

In de toewijzingsgraaf (zie fig. 11.2) kunnen we nu twee lussen (cyclische ketens) zien:

T1 - R1 - T2 - R3 - T3 - R2 - T1
T2 - R3 - T3 - R2 - T2

Er is nu sprake van deadlock, want alle vier de deadlockvoorwaarden gelden op hetzelfde moment.

In het volgende voorbeeld laten we zien dat er sprake is van een lus, maar dat er geen sprake is van deadlock.

- **Voorbeeld 3** (zie fig. 11.3)

  Er zijn vier systeemelementen (resources): R1 (dubbel uitgevoerd) en R2 (dubbel uitgevoerd).

  Er zijn vier taken: T1, T2, T3 en T4.

taak	toegewezen resource	aangevraagde resource
T1	R2	R1
T2	R1	-
T3	R1	R2
T4	R2	-

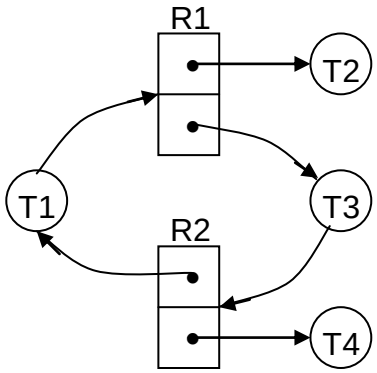


Fig. 11.3 Toewijzingsgraaf van voorbeeld 3

De volgende lus is in de toewijzingsgraaf (zie fig. 11.3) van voorbeeld 3 aanwezig:

T1 - R1 - T3 - R2 - T1

De vierde eis voor deadlock geldt hier. Maar er is geen deadlock omdat niet aan alle vier de deadlock eisen op hetzelfde moment kan worden voldaan. Taak T4 is niet wachtend op een additionele resource en geeft na enige tijd systeemelement R2 vrij, als hij met deze resource klaar is.

Indien T4 het systeemelement R2 en T2 S1 **onnodig te lang vasthouden** (door bijvoorbeeld een programmeerfout), kunnen de andere taken niets uitvoeren. Dit onbepaalde wachten als er geen sprake is van deadlock, wordt **starvation** (uithongeren) genoemd.

11.4 Voorkomen van deadlocks

Zorg ervoor dat **vanaf de ontwerpfase** van een systeem of code rekening wordt gehouden met locking in een multithreaded omgeving. Houd hierbij de volgende regels aan:

- **toets je code** aan de voorwaarden die kunnen leiden tot een deadlock;
- gebruik een **toewijzingsgraaf** voor de allocatie van resources;
- let op de **volgorde** waarin semaforen worden aangevraagd bij nesten;
- een **complex ontwerp** vergroot de risico's op een deadlock;
- een **asymmetrische of acyclische beginsituatie** kan helpen om deadlocks te voorkomen.

Het onjuist gebruiken van semaforen geeft in de praktijk vaak aanleiding tot het optreden van deadlock. In de tool **valgrind** vinden we **helgrind** ‘a thread-error detector’, die synchronisatieproblemen in C, C++ en Fortranprogramma’s kan detecteren die de POSIX pthreads threading primitives gebruiken.

11.5 Opgaven

1. Kan een **read-only file** als gemeenschappelijke resource deadlock veroorzaken?

2. Voor deadlock zijn vier voorwaarden genoemd. Hoe zijn deze concreet van toepassing op het genoemde 'dining philosophers problem'?

3. Teken voor het geschetste 'dining philosophers problem' de toewijzingsgraaf voor het moment dat er sprake is van deadlock.

4. Teken de toewijzingsgraaf voor de volgende situatie:

   - Er zijn 3 systeemelementen (resources) R1 en R2 (dubbel uitgevoerd).
   - Er zijn 4 taken T1, T2, T3 en T4.
   - T1 vraagt R2 aan, heeft R1 toegewezen gekregen.
     T2 vraagt R1 aan, heeft R2 toegewezen gekregen.
     T3 vraagt R1 aan, heeft R2 toegewezen gekregen.
     T4 vraagt R1 aan.

   Geef aan of er sprake is van **deadlock** of niet en waarom. Als er geen sprake is van deadlock geef dan aan in welke volgorde de taken kunnen worden uitgevoerd.

12 Geheugenbeheer

12.1 Inleiding

Op een computer draaien tegenwoordig vele programma's tegelijk en programma's worden steeds groter. Dit betekent dat de ruimte in het werkgeheugen te klein kan zijn om alle processen tegelijk en volledig te bevatten. De programmeur weet niet welk deel van het werkgeheugen vrij is als zijn programma t.z.t. zal worden geladen. Een belangrijke taak van het operating system is daarom het **beheer van het geheugen**. Daarvoor zijn afspraken nodig over de structurering of indeling van het geheugen. Er moet een administratie worden bijgehouden van waar data en code staan en welke ruimte beschikbaar is. Tevens moeten verschillende taken en eventueel verschillende gebruikers elkaar niet verstoren. Daarnaast moet integriteit van de toegewezen geheugengebieden gewaarborgd zijn.

12.1.1 Soorten geheugen

Geheugen kan worden verdeeld in:

1. **Registers**
   On-chip zeer snel geheugen (accestijd in de orde van 1 ns), waarvan enkele tientallen aanwezig zijn met een breedte van 32 of 64 bits.
2. **Cache**
   Meestal on-chip zeer snel geheugen (SRAM, accesstijd in de orde van 1 tot 10 ns) voor instructies en data; bevindt zich tussen de microprocessor en het werkgeheugen; bevat 32 kBytes tot 8 MByte
3. **Main memory (werkgeheugen)**
   On-board geheugen (DRAM, accesstijd in de orde van 10 ns) met een omvang van 1 tot 10 GByte
4. **Extern geheugen**
   Off-board geheugen:
   - hard disk: 100 GByte tot 1 TByte met omwentelingssnelheden tot 7200 rpm (revolutions per minute), seek times van ongeveer 5 ms en datasnelheden in de grootte-orde van 1 Gbps;
     De seek time is de tijd die nodig is om de lees/schrijfkop boven de juiste track (spoor) te zetten. Vervolgens moet de juiste sector nog onder de lees/schrijfkop verschijnen. Dit duurt gemiddeld een halve omwenteling. Bij 7200 rpm is dit dus 0,5x60/7200 s = 4 ms. Het duurt dus ongeveer 9 ms voordat met het lezen/ schrijven van de data kan worden begonnen. Dit gebeurt met een snelheid van ongeveer 1 Gbps ofwel met ongeveer 100 MByte/s. Met data transfer rate (datasnelheid) wordt bedoeld de snelheid van de data tussen de disk en het geheugenbuffer in de disk drive.
   - CD-ROM, CD-RW: 700 MByte optische schijf; 0,1 – 10 MByte/s (bij 200 – 2000 rpm)
   - DVD: 4 – 30 GByte optische schijf; 1 – 20 MByte/s
   - blu-ray disk: 25 – 100 GByte optische schijf; 5 – 50 MByte/s (tot 10.000 rpm)
   - USB memory stick: 32 GByte; 30 MByte/s

Het externe werkgeheugen is weliswaar erg snel, maar er moet de vertraging van de signalen over de **bus** worden bijgeteld (in tegenstelling tot bij het on-chip cache). Deze bedraagt **enkele tot een tiental ns**, afhankelijk van de belasting van de bus. De processor haalt zijn instructies en data zoveel mogelijk uit het cache op. Is een instructie of data daar niet aanwezig, dan wordt het cache aangevuld met (een blok) data uit het werkgeheugen en vervolgens door de processor opgehaald.

De kans dat de gegevens in het cache staan, wordt de **hit rate** H genoemd.
Stel dat de toegangstijd tot het cache T_C bedraagt en die tot het werkgeheugen T_W, dan geldt voor de gemiddelde toegangstijd T_A (statistisch karakter!):

$$T_A = H \cdot T_C + (1 - H) \cdot (T_C + T_W) = T_C + (1 - H) \cdot T_W$$

Voor T_C = 4 ns, T_W = 15 ns en H = 0,9 is $T_A = 4 + 0{,}1 \cdot 15 = 5{,}5$ ns.

Voor caches van 32 kByte blijkt de hit rate ongeveer 0,9 te zijn; bij caches van 256 kByte is deze 0,999 (en wordt Tc = 4,015 ns in bovenstaand voorbeeld).

Het beheren van caches en de bijbehorende adresvertalingen wordt gedaan door speciale hardware, een **memory management unit** (**MMU**).

De grote off-board geheugens zijn relatief traag, bij disks vooral t.g.v. de seek time en de rotation delay. On-board halfgeleidergeheugens kunnen erg groot zijn en hebben hoge snelheden. Bij een 32-bits on-board bus worden snelheden in de orde van 400 MByte/s gehaald.

12.1.2 Technieken voor geheugenbeheer

Een operating system voert het **geheugenbeheer** (**memory management**) uit. Het bepaalt waar en wanneer bepaalde hoeveelheden data en programma's in het werkgeheugen geladen worden. Daarvoor wordt een administratie in de vorm van tabellen bijgehouden van welke plekken in het geheugen bezet of vrij zijn.

Zoals we in §5.2 gezien hebben, bestaat een proces uit **segmenten**: een codesegment, een user-datasegment en een system-datasegment (met de proces descriptor). Het datasegment is verder onder te verdelen in een datagebied voor de globale variabelen en een stack voor de lokale variabelen en terugkeeradressen bij functiegebruik.

We onderscheiden een zevental memory-managementtechnieken:

1. **Fixed partitioning**
   Het hoofdgeheugen wordt verdeeld in een aantal **partities** met vaste grootte. Partities zijn groot (MBs) in vergelijking met page frames (zie hieronder). Deze grootte kan voor alle partities gelijk zijn of niet. Een proces kan in een partitie worden geplaatst als die partitie groter of gelijk is aan de benodigde ruimte voor het proces. Deze werkwijze is eenvoudig en levert **weinig overhead** op. Wel is het **aantal processen beperkt** door het aantal partities en wordt het geheugen inefficiënt

gebruikt ten gevolge van **interne fragmentatie**. Interne fragmentatie treedt op als de partitie niet helemaal is gevuld, wat meestal het geval zal zijn (zie §12.2).

2. **Dynamic partitioning**
   In dit geval wordt bij het laden van een proces een **partitie** aangemaakt die precies de grootte van het proces is. Hierbij treedt er geen interne fragmentatie op. Er treedt echter wel **externe fragmentatie**, doordat processen na beëindiging ook weer uit het geheugen worden verwijderd (zie §12.3). Hierdoor komt er ruimte vrij tussen de partities die te klein kan worden om te gebruiken voor processen. Dit vindt plaats t.g.v. het laden en verwijderen van processen. Hierdoor kan er op een gegeven moment erg veel vrije geheugenruimte ontstaan, die echter niet meer bruikbaar is. In dat geval moet het werkgeheugen worden gedefragmenteerd (men spreekt hier wel van **garbage collection**). De processen worden in het geheugen opgeschoven, zodat er weer een aaneengesloten vrije geheugenruimte ontstaat. Dit zal herhaaldelijk moeten geschieden, hetgeen veel processortijd kost. Dynamic partitioning levert hierdoor dus een inefficiënt gebruik van de processor op, ook al geschiedt dit met behulp van DMA (direct memory access).

3. **Simple paging**
   Het hoofdgeheugen wordt hierbij verdeeld in **frames** van dezelfde grootte. Elk proces wordt verdeeld in **pages** die dezelfde grootte hebben als de frames. Een page is de kleinste adresseerbare eenheid die door de memory-management unit (MMU) kan worden beheerd. Op **32-bitsystemen** is dit **4 kb** (en met 10^6 pages een maximaal geheugen van 4 Gb), op **64-bitsystemen** is dat **8 kb** (en met 2×10^{15} pages een maximaal geheugen van 16 exabyte). Als een proces wordt geladen in het hoofdgeheugen, worden alle pagina's van dat proces geladen in vrije frames. Deze frames hoeven **niet aaneengesloten** in het geheugen te zitten. Er treedt geen externe fragmentatie op. Wel treedt er enige **interne fragmentatie** op, omdat het laatste frame meestal niet helemaal gevuld zal zijn.

4. **Simple segmentation**
   In deze situatie wordt elk proces verdeeld in een aantal **segmenten** (code, data, stack, etc.). Als een proces in het werkgeheugen wordt geladen, worden alle segmenten van dat proces als **dynamische partities** in het geheugen gezet. Deze partities hoeven niet aaneengesloten in het geheugen te komen. Er treedt hierbij geen interne fragmentatie op, wel **externe fragmentatie**. Het geheugengebruik is **efficiënter** dan dynamic partitioning, omdat een proces wordt verdeeld in meerdere en dus kleinere segmenten.

5. **Virtual memory paging**
   Het verschil met simple paging is dat nu **niet alle pagina's** van een proces in het geheugen geplaatst hoeven te worden. Pagina's die later nodig zijn, worden dan automatisch geladen. Er treedt geen externe fragmentatie op (alleen enige **interne fragmentatie**). Er kunnen **meer processen** parallel draaien dan bij simple paging en simple segmentation. Het is wel complexer en dus treedt er meer **overhead** op.

6. **Virtual memory segmentation**
   Het verschil met simple segmentation is dat **niet alle segmenten** van een proces geladen hoeven te worden. Segmenten die later nodig zijn, worden dan automatisch geladen. Het heeft dus zin om meer kleinere segmenten te definiëren, bv. door functies in aparte segmenten onder te brengen. Dit kan ook gedaan worden met verschillende data arrays. Dit maakt het mogelijk om deze functies en arrays door meerdere processen te laten gebruiken, wat geheugenruimte bespaart. Er kunnen dan **meer processen** tegelijk geladen worden dan bij simple segmentation. Er treedt geen interne fragmentatie op (wel **externe fragmentatie**, zij het minder dan bij simple segmentation). De **overhead** is wel groter ten gevolge van een complexer geheugenmanagement.

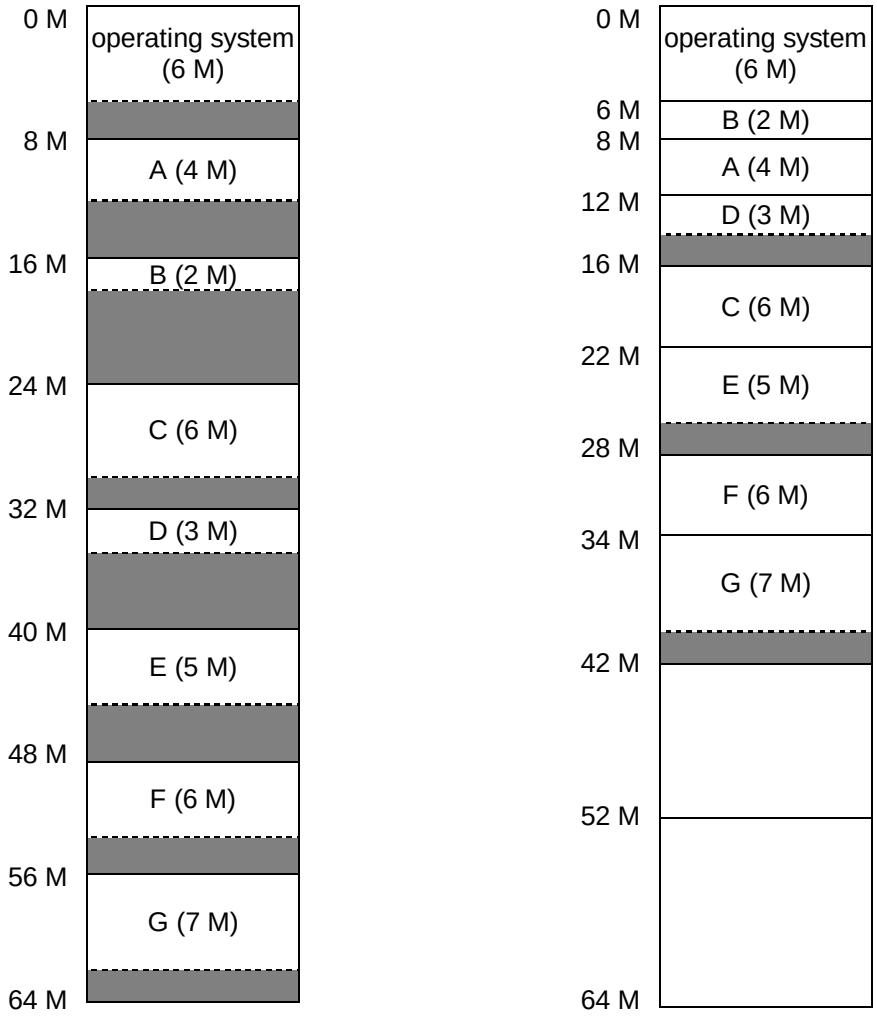
7. **Virtual memory segmented paging**
   Dit is een combinatie van **paging en segmentation**. Een proces wordt verdeeld in segmenten die op hun beurt weer worden verdeeld in pages. Hiermee worden de voordelen van segmentation en paging gecombineerd.

12.2 Fixed partitioning en interne fragmentatie

Stel het operating system neemt 6 MByte in beslag. Laden we achtereenvolgens de processen A, B, C, D, E, F en G in het geheugen bestaande uit 8 partities van elk 8 MByte. In fig. 12.1.a zien we een aanzienlijke **interne fragmentatie** optreden van in totaal 25 MByte aan "vrije" geheugenruimte, die echter niet bruikbaar is. Plaatsen we dezelfde processen in een geheugen met ongelijke partities van resp. 6, 2, 4, 4, 6, 6, 6, 8, 10 en 12 MByte (fig. 12.1.b), dan zien we dat de interne fragmentatie slechts 3 MByte beslaat en dat er nog vrije partities zijn van 10 MByte en van 12 MByte. We moeten hierbij wel opmerken dat we 2 partities meer hebben gemaakt.

Als een proces in het geheugen moet worden geladen terwijl er geen vrije partities meer zijn, kunnen processen die noch running noch active zijn uit het geheugen worden verwijderd. Als een programma te groot is voor een partitie, moet de programmeur dit verdelen in kleinere modules die niet gelijktijdig nodig zijn. Als een module nodig is, moet het programma deze module laden en over een niet meer nodige module in de gebruikte partitie plaatsen (**overlay**).

De interne fragmentatie is bij partities met gelijke grootte extreem groot, omdat sommige processen veel minder geheugen nodig hebben. Bij gebruik van partities met verschillende grootte worden de problemen met overlays en interne fragmentatie gedeeltelijk opgelost. Ook het gebruik van **verschillende lagen van allocatiemethoden** (van grof naar fijner) kan interne fragmentatie reduceren.



a. vaste partities met gelijke grootte b. vaste partities met ongelijke grootte

Fig. 12.1 Interne fragmentatie bij fixed partitioning

12.3 Dynamic partitioning en externe fragmentatie

Bij dynamic partitioning ontstaat **externe fragmentatie** (zie fig. 12.2). Stel we laden achtereenvolgens de processen A t/m J. Er zijn dan nog 5 MByte aan vrij geheugen over.

Vervolgens worden de processen A, E, G en H verwijderd en wordt proces K (9 M) geladen. Er ontstaat nu fig. 12.2.b (ga dit na). Er is nu totaal 20 MByte aan vrije geheugenruimte die echter zwaar gefragmenteerd is. Als er nu een proces ter grootte van bv. 7 MByte geladen moet worden, lukt dat niet. Er zal dan eerst gedefragmenteerd moeten worden. Dit kan alleen als het proces kan worden verplaatst zonder dat de geheugenadressen ongeldig worden. Met andere woorden, de adressen moeten worden aangepast tijdens het uitvoeren van het proces. Dit kan geschieden door een memory management unit die de **logische adressen** vertaalt in **fysieke adressen**.

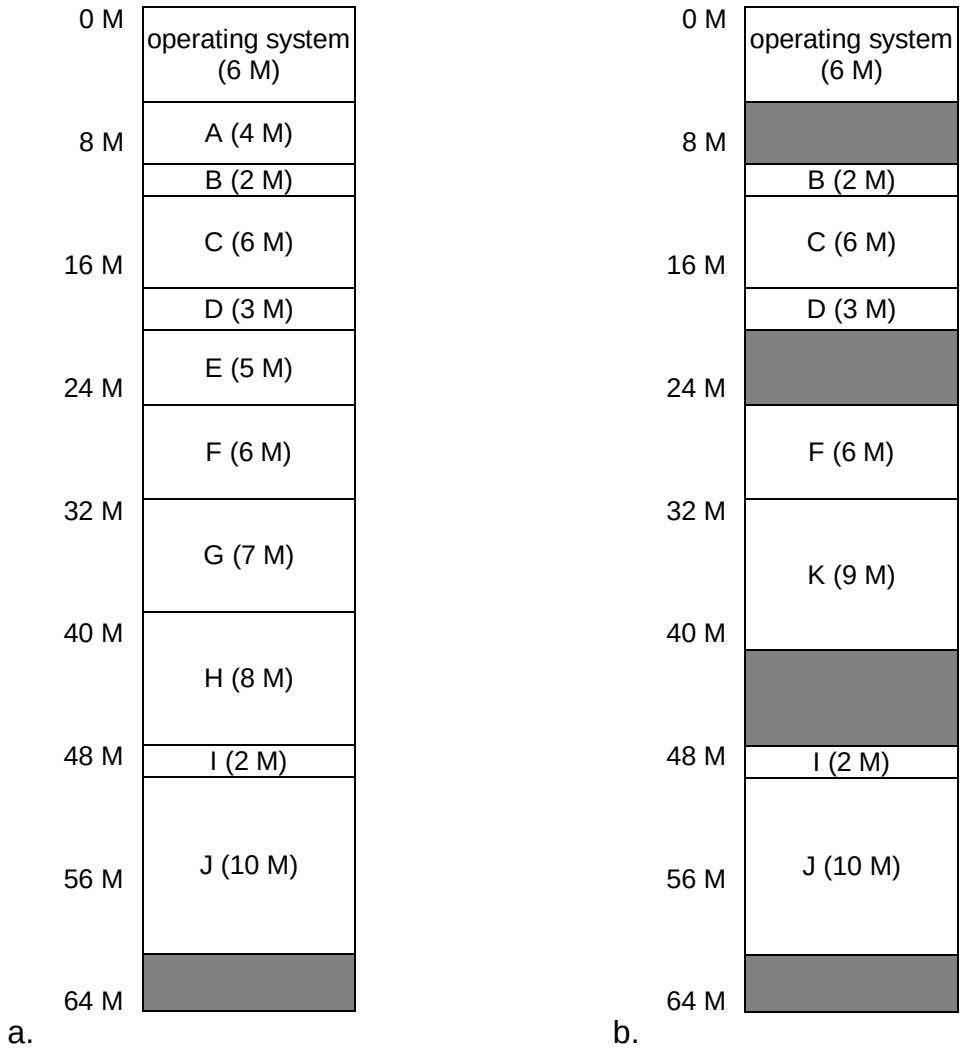


Fig. 12.2 Externe fragmentatie bij dynamic partitioning

Logische adressen zijn referenties die alleen binnen het programma gelden en die door compiler en linker worden berekend **t.o.v. het begin van het programma**. Logische adressen zijn dus onafhankelijk van de fysieke plaats in het geheugen waar het programma geladen zal worden. De vertaling van logisch adressen naar **fysieke adressen** kan in dit geval eenvoudig geschieden door het logisch adres te verhogen met het **beginadres van de partitie** waarin het proces zich bevindt.

Let wel dat het defragmenteren regelmatig moet gebeuren en dat dit veel tijd kost wat tot inefficiënt processorgebruik leidt. Het is dus zaak zoveel mogelijk te voorkomen dat het geheugen gefragmenteerd raakt door slim het geheugen toe te wijzen.

12.3.1 Methoden van geheugentoewijzing

Het laden van een proces in het geheugen kan bij **dynamic partitioning** op verschillende manieren geschieden:

- **First fit**
  Vanaf het begin van het geheugen wordt de **eerste partitie** gebruikt die voldoende groot is. Deze methode werkt eenvoudig, maar levert ernstige fragmentatie op. Aan het begin van het geheugen ontstaan al snel veel kleine fragmenten die telkens moeten worden overgeslagen als er weer een proces wordt geladen (dat immers niet past). Hierdoor wordt het proces van toewijzing vrij traag.

- **Next fit**
  Hierbij wordt domweg de **eerstvolgende** voldoende grote vrije ruimte gebruikt **achter de laatst toegewezen** ruimte. Deze methode is erg snel en de fragmenten worden meer over het hele geheugen verspreid. Hierdoor wordt al snel de grote vrije ruimte aan het eind van het geheugen opgedeeld in kleine stukken, zodat er geen grote aaneengesloten vrije ruimte meer over blijft. Dit levert problemen voor het laden van grote processen.

- **Best fit**
  De **partitie waarin de minste ruimte overblijft** na laden van het proces, wordt gebruikt. Hierbij is de fragmentatie relatief klein, maar het kost vrij veel tijd om de meest geschikte partitie op te zoeken. Bovendien zullen de fragmenten die overblijven meestal zo klein zijn dat ze niet meer te gebruiken zijn. Best fit werkt vaak efficiënt, maar de benodigde overhead is afhankelijk van de geschiedenis. Hierdoor is deze **overhead lastig te voorspellen**, wat ongunstig is voor realtime-systemen.

- **Worst fit**
  Hierbij wordt juist die **partitie** gekozen **waarin de meeste vrije ruimte overblijft**. Hiermee wordt bereikt dat deze overblijvende vrije fragmenten voldoende groot zijn voor andere processen. Ook deze methode kost vrij veel tijd. Verder gaan hiermee de grote vrije geheugenruimten snel verloren, zodat grote processen niet meer geladen kunnen worden.

- **Buddy allocation**
  Deze allocatiestrategie heeft een veel beter **voorspelbaar executietijdgedrag** dan first-fit allocation, wat gunstig is voor realtime-systemen. Bij elke geheugenaanvraag wordt het beschikbare geheugen net zo lang **in tweeën gedeeld**, totdat er net voldoende overblijft om data en/of code te plaatsen. De bij halvering **ontstane helften zijn elkaars buddy**. De vrije buddy kan toegewezen worden aan een nieuw proces. Deze buddy wordt niet samengevoegd met andere vrije geheugenblokken. Zodra een geheugenblok niet meer nodig is, wordt dit teruggegeven aan de free memory pool. Er wordt dan gekeken of zijn buddy ook vrij is. In dat geval worden de **twee blokken weer samengevoegd**. Als de buddy van dit grotere vrije blok vrij is, worden deze twee blokken ook weer samengevoegd, enz.
  Er is alleen sprake van stukken geheugen die een grootte hebben van een macht van 2. Buddy allocation leidt eerder tot geheugenfragmentatie dan first-fit allocatie! Stel dat we 512K + 2 bytes nodig hebben. Dit betekent dat we met buddy allocation altijd 1024K nodig hebben. We verliezen nogal wat geheugencapaciteit in het toegewezen blok (interne fragmentatie).

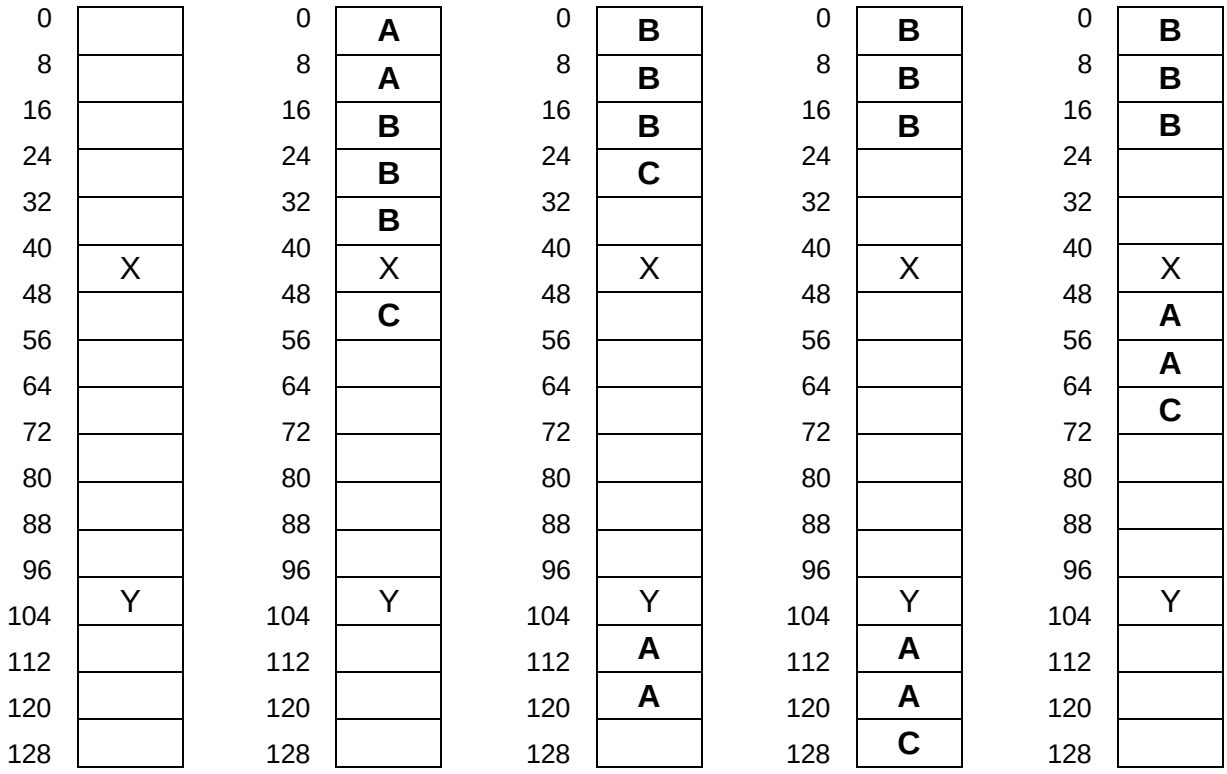
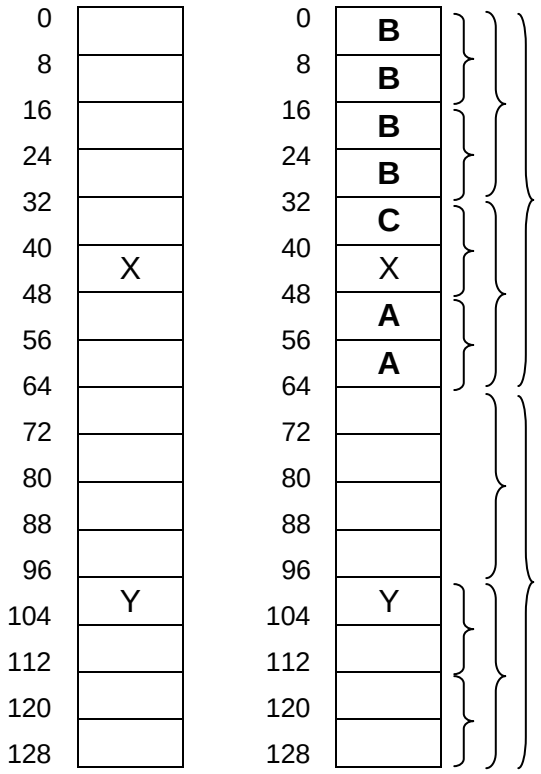


Fig. 12.3 Methoden van geheugentoewijzing (grootten in kByte)

Voorbeeld van first, next, best en worst fit
Het werkgeheugen van een computer bestaat uit 16 pagina's van elk 8 kByte. Er wordt gebruikgemaakt van gepagineerde segmentering (zie §12.6.3). Er zijn gedurende enige tijd processen in het geheugen geladen en ook weer verwijderd. Op een bepaald moment staat proces **X (6 kByte)** in pagina 6 en proces **Y (7 kByte)** in pagina 13 (zie fig. 12.3.a). Proces Y is het proces dat het laatst in het geheugen is geladen. Achtereenvolgens worden nu proces **A (13 kByte)**, proces **B (22 kByte)** en proces **C (8 kByte)** geladen. Dit geschiedt achtereenvolgens m.b.v. first, next, best en worst fit.

We zien dat first fit in dit voorbeeld nog niet zo slecht is. Dit levert de grootste vrije geheugenblokken op (van 5 en 3 aaneengesloten pagina's). In de praktijk blijkt best fit vaak goed te voldoen, al is de overhead niet te voorspellen.

Voorbeeld van buddy allocation
Hierbij gaan we uit van hetzelfde voorbeeld als bij fig. 12.3. In fig. 12.4 geven we de buddies met accolades aan. De kleinste buddies zijn in de gegeven situatie 8 kByte groot (de grootte van een page) en het hele geheugen 128 kByte. De enige vrije buddy met de omvang van 8 kByte in fig. 12.4b begint op adres 104 k.



a. initieel b. buddy allocation

Fig. 12.4 Buddy allocation (grootten in kByte)

Merk op dat proces B nu 4 pagina's omvat (de grootte van de kleinste vrije buddy), omdat een buddy niet uit 3 pagina's (= 24 kByte) kan bestaan. Buddies hebben immers een grootte van 2^i bytes. Merk verder op dat A ook in de laatste vrije buddy van 16 kByte past (vanaf adres 112 k). Echter dan wordt het geheugen meer gefragmenteerd. Zou proces Y in dat geval uit het geheugen verwijderd worden, dan zou de vrijgekomen buddy alleen met de volgende buddy van 8 kByte samengevoegd kunnen worden tot een buddy van 16 kByte. Door A in de 4^{de} buddy te plaatsen, zal één grote buddy van 64 kByte ontstaan, zodra Y wordt vrijgegeven. Een voorbeeld van het laden en verwijderen van processen m.b.v. buddy allocation is ook te vinden op de pagina *Buddy memory allocation* van de Engelstalige Wikipedia.

12.4 Simple paging

Net als bij partioning moet ook hier het **hele programma** in het geheugen worden geladen. De verschillen met partitioning zijn dat het programma **over meerdere pagina's verdeeld** kan worden, dat deze pagina's veel kleiner zijn dan partities en dat de programmapagina's niet aaneengesloten in het geheugen hoeven te staan. Hierdoor treedt er geen externe fragmentatie op. De pagina's zijn relatief klein van omvang zodat de interne fragmentatie (die bovendien alleen in de laatste pagina kan optreden) klein is. Een programma (op disk) wordt verdeeld in pagina's met dezelfde grootte als de frames (in het werkgeheugen).

12.4.1 Adresvertaling m.b.v. een page table

Met **adresvertaling** wordt het **logische adres** van een geheugenblok in de **software** vertaald naar het **fysieke adres** van het geheugen op de **hardware**. Bij paging bestaat het logische adres uit een paginanummer, gevolgd door een **offset**. Deze offset is het relatieve adres t.o.v. het begin van de betreffende pagina. Het logische adres wordt bij simple paging vertaald in een fysiek adres door de offset **samen te voegen** met het nummer van het frame waarin de betreffende pagina geladen is. Het **paginanummer** van het *logische* adres wordt dus **vervangen door** het **framenummer** van het *fysieke* geheugen.

Dit framenummer is te vinden in de **page table** van het betreffende proces (zie fig. 12.5). Een page table is in feite een array met de paginanummers als indices en de framenummers als array-elementen. Pagina's en frames hebben natuurlijk dezelfde grootte, bv. 4 of 8 kByte. De grootte van de offset moet bekend zijn voor de adresvertaling.

In het voorbeeld van fig. 12.5 is het logische adres **0x002426**. De offset is **12 bits**, d.w.z. de laatste **drie** hexadecimale karakters. Een hexadecimaal karakter vertegenwoordigt immers 4 bits ($2^4 = 16$). De **offset** bedraagt dus **0x426** en wat overblijft (ook 12 bits) is het **paginanummer**: **0x002 = 2**. We kijken vervolgens in **rij 2** van de **paginatabel** en vinden het **framenummer 0x00C**. Wanneer we het paginanummer van het logische adres **vervangen** door het framenummer van het fysieke adres (met dezelfde offset; page en frame hebben immers dezelfde grootte) vinden we het 24-bits **fysieke adres 0x00C426**.

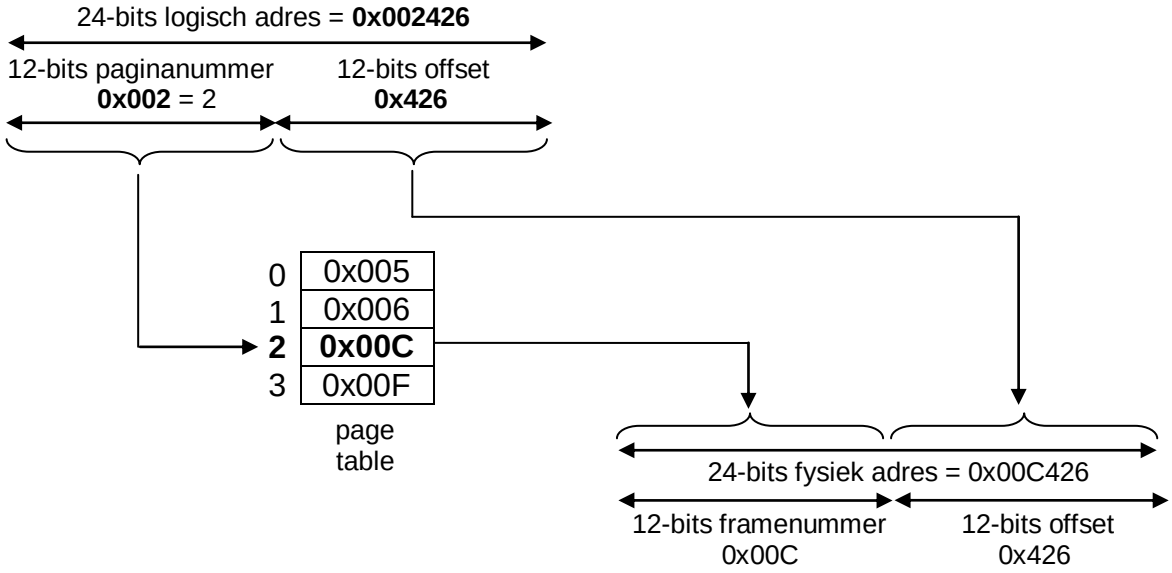


Fig. 12.5 Adresvertaling via een paginatabel voor een proces bestaande uit 4 pagina's, met een offset van 12 bits

12.5 Simple segmentation

Het programma wordt nu verdeeld in een aantal **segmenten**. Deze kunnen een verschillende grootte hebben. Net als bij dynamic partitioning moeten **alle segmenten van een proces in het werkgeheugen geladen worden** (tenzij gebruik wordt gemaakt van overlays). Het verschil is echter dat de segmenten niet aaneengesloten in het werkgeheugen hoeven te staan. De externe fragmentatie is echter kleiner doordat de segmenten kleiner kunnen zijn dan de partities.

12.5.1 Adresvertaling m.b.v. een segment table

De programmeur moet zelf zijn programma in segmenten onderverdelen en er is geen eenvoudige relatie tussen de logische adressen en de fysieke adressen. De vertaling van logische adressen naar fysieke adressen geschiedt via een segment tabel. Voor elk proces moet er een **segment table** worden aangemaakt. Omdat niet alle segmenten even groot zijn, bevat de segmenttabel niet alleen het **beginadres** van het fysieke geheugensegment maar ook zijn **lengte**. Zodoende kan worden gecontroleerd of er niet buiten het segment wordt geadresseerd. Het logisch adres bestaat uit een **segmentnummer** en een **offset** (zie fig. 12.6). Het **segmentnummer** wordt gebruikt om in de **segment table** het fysieke **beginadres** van het segment op te zoeken. Dat beginadres wordt vervolgens **opgeteld** bij de offset om het **fysieke adres** van het geheugenblok te verkrijgen.

In het voorbeeld in fig. 12.6 is het **logisch adres** wederom 24 bits lang, maar is de **offset 16-bits**, dus vier hexadecimale karakters: **0x2426**. Het **segmentnummer 0x00 = 0** blijft over, en dus zoeken we in de segmenttabel het fysieke **beginadres** op: **0x1C46**. **Optellen** van het beginadres en de offset geeft het **fysieke adres**: 0x1C46 + 0x2426 = **0x00406C**.

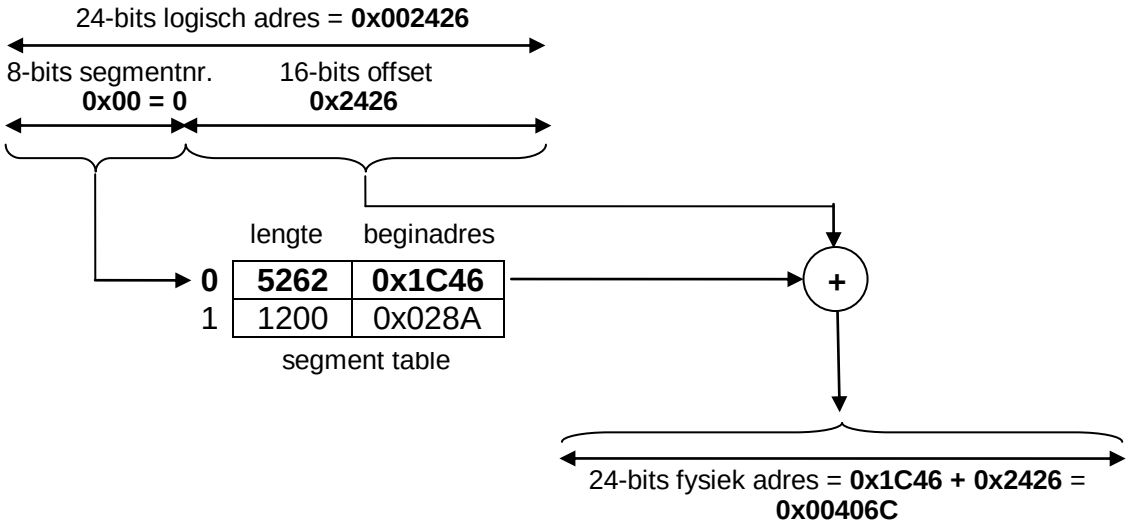


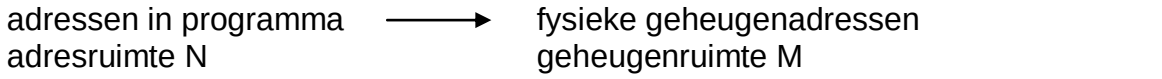
Fig. 12.6 Adresvertaling via een segmenttabel voor een proces bestaande uit 2 segmenten en een 16-bits offset

12.6 Virtueel geheugen

We spreken van **virtueel geheugen** als het secundaire geheugen (bv. disk) kan worden geadresseerd alsof het deel uitmaakt van het fysieke geheugen. Hierdoor is het mogelijk een programma uit te voeren dat groter is dan het werkgeheugen. De delen van het programma en data die niet in het werkgeheugen passen, worden bewaard op disk. Op het moment dat de code moet worden uitgevoerd, zorgt het geheugenbeheer ervoor dat delen van het programma en data van disk in het beschikbare werkgeheugen worden geladen. Er vindt dus uitwisseling van code en data plaats tussen werk- en achtergrondgeheugen (swapping). Deze gehele aanpak noemt men **virtueel geheugenbeheer** (**virtual memory management**). Een adresvertaler stelt de programmeur in staat gebruik te maken van een reeks programma-adressen die totaal verschillend kunnen zijn van de beschikbare geheugenplaatsen. De programmeur 'ziet' een virtueel geheugen.

Het operating system zorgt ervoor dat programma-adressen zoals de programmeur deze ziet op een of andere manier worden afgebeeld op het fysieke geheugen.

Er vindt een vertaling plaats van **programma-adressen** naar **fysieke adressen** door een **adresvertaler**:



Ten behoeve van het geheugenbeheer beschikt een operating system over een groot aantal software functies. Voor het opvoeren van de snelheid zijn er in de hardware allerlei oplossingen bedacht om de beheerfuncties acceptabel snel te kunnen uitvoeren. Dergelijke hardware wordt een **memory management unit** (**MMU**) genoemd.

Als er een nieuw proces wordt gestart, laadt het operating system een deel (het begin) van het programma en dat deel van de data waaraan wordt gerefereerd, in het werkgeheugen. Zodra de processor een logisch adres tegen komt dat niet in het werkgeheugen voorkomt, wordt er een interrupt gegenereerd die een adresfout aangeeft. Het betreffende proces wordt door het operating system geblokkeerd en het benodigde programmadeel wordt in het geheugen geladen. Intussen kan een ander proces van de processor gebruik maken (disk acces is immers traag). Zodra het juiste programmadeel in het werkgeheugen is gezet, wordt het proces in de ready state geplaatst en krijgt t.z.t. weer processortijd.

Om bij te houden of een pagina of segment al dan niet in het geheugen voorkomt, wordt de page table, resp. segment table uitgebreid met een **present bit**.

Het voordeel van het toepassen van virtueel geheugen is dat er meer processen in het werkgeheugen kunnen plaatsnemen dan zonder virtueel geheugen. De processen hoeven immers niet meer in hun geheel te worden geladen. Om dezelfde reden kunnen de processen groter zijn dan het werkgeheugen.

Als er een stuk programma in het geheugen moet worden geladen terwijl er onvoldoende vrije ruimte is, moet er een stuk werkgeheugen teruggeschreven worden op disk om geheugen vrij te maken. Terugschrijven op disk hoeft natuurlijk alleen te ge-

beuren als de betreffende page of segment is gewijzigd. Hiertoe is er in de paginatabel, resp. segmenttabel een **modify bit** opgenomen.

Als er een stuk wordt verwijderd dat even later weer nodig is, treedt er extra vertraging op. Als dit vaak gebeurt doordat er telkens de 'verkeerde' onderdelen worden verwijderd en op disk geschreven, kan het systeem zeer traag worden. Men spreekt van **thrashing**.

Het zal duidelijk zijn dat het terugschrijven van delen van het werkgeheugen op disk weloverwogen dient te gebeuren. Optimaal zou zijn als die pagina's of segmenten op disk worden gezet die het langst niet meer nodig zijn. Dit is niet uitvoerbaar omdat het operating system dan zou moeten weten wat er in de toekomst allemaal gaat plaatsvinden.

Het verwijderen van pagina's of segmenten kan bv. met de volgende strategieën (policies) geschieden:

- **Optimaal**
  Hierbij wordt dat frame verwijderd dat het **langst niet gebruikt zal gaan worden**. Natuurlijk is deze methode niet praktisch omdat het operating system niet weet wat er in de toekomst gaat gebeuren. Dit kan worden gebruikt als referentie bij benchmarking.

- **Minst recent gebruikt**, dus langst niet gebruikt **(least recently used, LRU**)
  Het frame dat het **langst niet gebruikt is**, wordt vervangen. Dit frame wordt waarschijnlijk niet gebruikt in de nabije toekomst, omdat instructies achter elkaar in het geheugen staan en data staan veelal achter elkaar in arrays. Deze methode vergt tijd om bij te houden hoe lang een frame niet meer gebruikt is.

- **First-in-first-out (FIFO**)
  Het frame dat het **eerst geladen** is, wordt ook weer het **eerst verwijderd**. Dit is een eenvoudige en weinig tijd kostende methode. Een pointer wijst in een circulair buffer aan welk frame als eerste moet worden verwijderd. Het frame dat wordt toegevoegd wordt op de plaats van het verwijderde frame geplaatst en de pointer wordt één plaats teruggezet.

- **Clock**
  Bij deze methode worden de frames in een **circulair buffer** opgenomen. Iedere keer wanneer een frame gebruikt wordt, wordt aan zijn **use bit** de waarde 1 toegekend. Een pointer wijst aan welk frame als volgende verwijderd moet worden. Bevat dat frame echter een use bit met de waarde 1, dan wordt dat frame niet verwijderd. Wel wordt zijn use bit 0 gemaakt en de pointer wordt één plaats verder gezet. Dit gaat door totdat een frame met use bit 0 wordt gevonden. Dat frame wordt vervangen door het frame dat geladen moet worden en de pointer wordt één plaats verder gezet. Een geset use bit tracht te voorkomen dat het frame verwijderd wordt wanneer het recent nog gebruikt werd.

- **Minst gebruikt (LU, least used)**
  Het frame dat het minst gebruikt is, wordt verwijderd. Deze methode kost wat meer tijd dan FIFO. Elke keer dat een frame wordt gebruikt, wordt een teller met één verhoogd. Bij het verwijderen van het frame, moet worden gezocht naar het frame met de laagste tellerstand.

- **Prioriteit**
  Het frame van het proces met de laagste prioriteit wordt verwijderd.

De overhead van clock is kleiner dan bij LRU, maar groter dan bij FIFO. De performance blijkt uit meetgegevens tussen LRU en FIFO in te liggen. LRU benadert in de praktijk het meest het optimale resultaat.

Stel we kunnen 3 frames gebruiken en achtereenvolgens hebben we de volgende frames nodig om een proces te kunnen uitvoeren: 1 2 1 3 4 1 2 5 2 3 4 5 2 4 5. Fig. 12.7 laat zien hoeveel **framefouten** (F) bij de diverse methoden optreden. Een frame fout treedt op als er een frame gebruikt moet worden, dat niet in het geheugen staat.

	1	2	1	3	4	1	2	5	2	3	4	5	2	4	5		**F**
Optimaal	1	1	1	1	1	1	1	5	5	5	5	5	5	5	5		
		2	2	2	2	2	2	2	2	3	3	3	2	2	2		
				3	4	4	4	4	4	4	4	4	4	4	4		
					F			**F**		**F**			**F**				**4**
LRU	1	1	1	1	1	1	1	1	1	3	3	3	2	2	2		
		2	2	2	4	4	4	5	5	5	4	4	4	4	4		
				3	3	3	2	2	2	2	2	5	5	5	5		
					F		**F**	**F**		**F**	**F**	**F**	**F**				**7**
FIFO	1	1	1	1	4	4	4	5	5	5	5	5	2	2	2		
		2	2	2	2	1	1	1	1	3	3	3	3	3	5		
				3	3	3	2	2	2	2	4	4	4	4	4		
					F	**F**	**F**	**F**		**F**	**F**		**F**		**F**		**8**
Clock	1*	1*	1*	►1*	4*	4*	►4*	5*	5*	5*	►5	►5*	5*	5*	5*		
	►	2*	2*	2*	►2	1*	1*	►1	►1	3*	3	3	2*	2*	2*		
		►	►	3*	3	►3	2*	2	2*	►2*	4*	4*	►4*	►4*	►4*		
					F	**F**	**F**	**F**		**F**	**F**		**F**				**7**
LU	1	1	1	1	1	1	1	1	1	1	1	1	1	1	5		
		2	2	2	4	4	2	2	2	2	2	2	2	2	2		
				3	3	3	3	5	5	3	4	5	5	4	4		
					F		**F**	**F**		**F**	**F**	**F**		**F**	**F**		**8**

F=frame fout; *=geset use bit; ►=pointer

Fig. 12.7 Diverse methoden voor het verwijderen van frames en het aantal framefouten

12.6.1 Virtual-memory paging

Net als bij simple paging heeft elk proces zijn eigen page table. Nu hoeven echter **niet alle pagina's in het geheugen** te staan. Wel zijn er een **present en een modify bit** nodig in de page table. Het virtuele paginanummer is de index van de paginatabel. Hier staat het fysieke paginanummer met de P- en M-bit (en overige control

bits, bv. om aan te geven wie toegang tot de betreffende pagina hebben). Virtuele offset en fysieke offset zijn gelijk.

Bij de voorgaande methoden kon het probleem optreden dat het proces **meer geheugen** nodig heeft (bij dynamische geheugenallocatie, bij een groeiende stack of bij data-invoer). Bij virtual memory paging wordt er in zo'n situatie gewoon een **extra pagina gealloceerd**.

12.6.2 Virtual memory segmentation

Ook hier hoeven **niet alle segmenten in het geheugen** aanwezig te zijn. Bij simple segmentation moest dat wel. Bij virtual memory segmentation kunnen datagebieden probleemloos groeien. Is het gealloceerde segment te klein, dan wordt het gedealloceerd en er wordt een nieuw, groter segment gealloceerd. Ook hoeven niet alle segmenten opnieuw te worden gecompileerd en gelinked als er een enkel segment wordt gewijzigd. Ook kunnen segmenten door meerdere processen gedeeld worden, zonder hen meermaals in het geheugen te laden. Aan elk segment afzonderlijk kunnen toegangsrechten worden toegekend.

Net als bij simple segmentation is ook hier per proces een segment table nodig. Hieraan is toegevoegd een present bit en een modify bit. Ook kunnen er control bits aanwezig zijn, bv. om de toegang tot een segment te regelen of om aan te geven of een segment gedeeld mag worden door andere processen.

12.6.3 Virtual memory segmented paging

Bij **segmented paging** (gepagineerde segmentering) wordt een programma verdeeld in segmenten en vervolgens wordt elk segment weer verdeeld in pagina's. Op deze manier worden de voordelen van paginering en segmentering gecombineerd. Externe fragmentatie wordt opgeheven door de paginering en de interne fragmentatie blijft beperkt door de relatief kleine pagina's.

Per proces is er nu **een segmenttabel** en een **aantal paginatabellen** (één per segment). Het segmentnummer is weer de index van de segmenttabel. Hierin staat het beginadres van de betreffende paginatabel. Het paginanummer is op zijn beurt de index in deze paginatabel waar het beginadres van de gezochte pagina staat. De offset tot slot is de offset van de betreffende pagina (zie fig. 12.8).

De segmenttabel bevat weer de diverse control bits (bv. de toegangsprotectiebits en share bit) en de segmentlengte. Deze is nu uitgedrukt in aantal pagina's (i.p.v. in aantal bytes). De paginatabellen bevatten de present en modify bits.

logisch adres

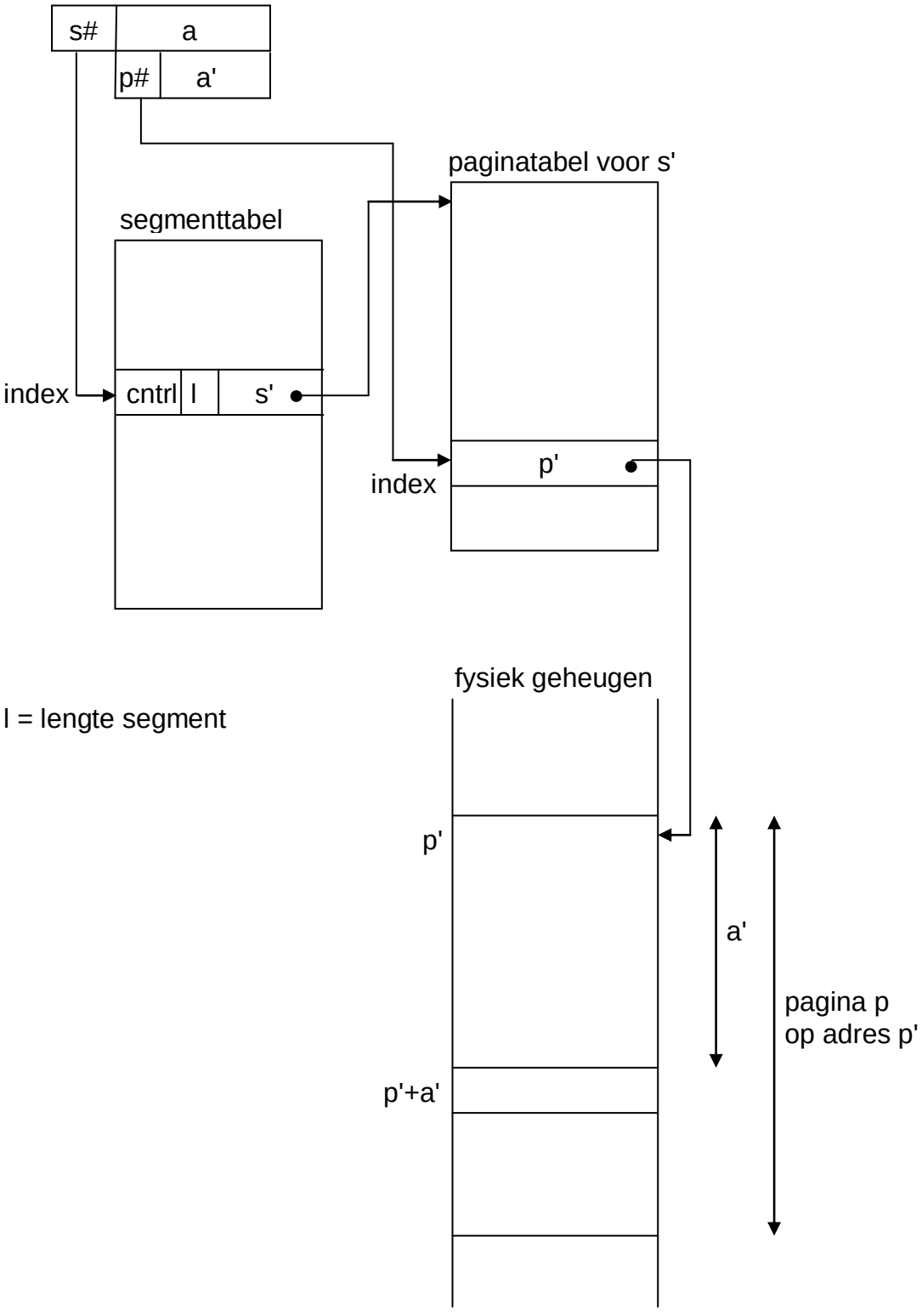


Fig. 12.8 Adresvertaling bij gepagineerde segmentering

12.7 Memory management en protectie

In een multitasking omgeving en zeker in een multi-user omgeving is het noodzakelijk dat elke taak alleen mag adresseren binnen geheugengebieden die aan die taak zijn toegewezen. Het eenvoudigst is de bovengenoemde protectie te realiseren indien de toegewezen geheugenruimte aaneengesloten is. De controle en adresberekening zouden softwarematig kunnen plaatsvinden met behulp van het volgende eenvoudige algoritme:

```
if (voldoet(user#, attr) & (offset < lengte))
    fysiek_adres = base + offset;
else
    error;
```

Indien een hoge verwerkingssnelheid wordt vereist en tevens hoge eisen aan de protectie wordt gesteld, is een hardware MMU (Memory Management Unit) onontbeerlijk. De segmenttabel wordt dan in registers van de MMU aangebracht en error detectie en fysieke adresberekening wordt volledig hardwarematig uitgevoerd (zie fig. 12.9).

Ook virtueel memory management kan nu efficiënt worden ondersteund in samenwerking met een **DMAC** (Direct Memory Access Controller).

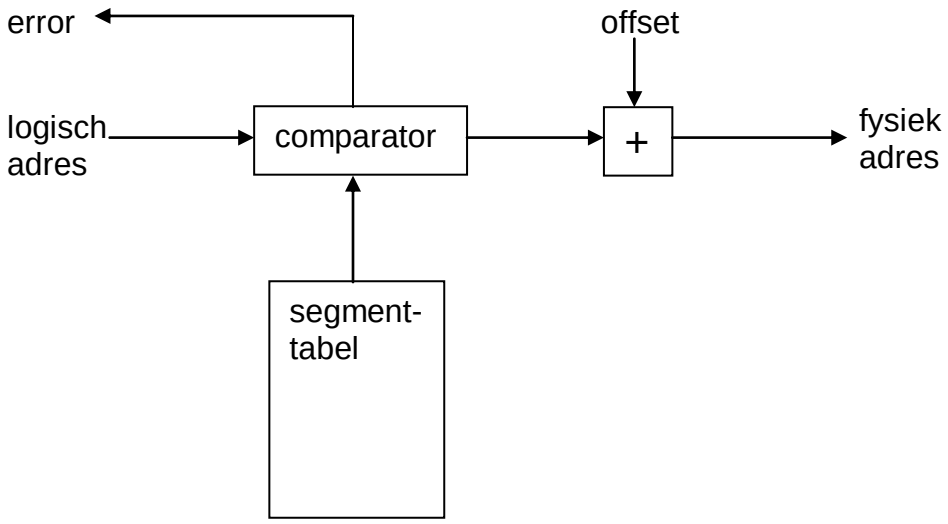


Fig. 12.8 Protectie bij adresvertaling

12.8 Opgaven

1. Een real-time systeem moet zich voorspelbaar (predictable) gedragen. Als nu taken allerlei modules van schijf laden, mogen we dan nog spreken van een voorspelbaar systeem? Welke twee problemen komen we tegen? Wat is hier tegen te doen?

2. Ontwerp de tabellen voor een **gepagineerde segmentatiestructuur**, bepaald door onderstaande eisen:
   paginagrootte = 512 words
   maximum segmentgrootte = 65.536 words,
   de processor genereert een 32-bit adres
   geheugenprotectie: read-only of execute-only
   privileges: 1 niveau voor de user en 3 niveaus voor het operating system

3. Een systeem maakt gebruik van de buddy allocation geheugen management strategie. Stel er is een 1M geheugenblok beschikbaar en er wordt 1k geheugen aangevraagd.
   Geef weer in welke geheugenblokken het 1M blok wordt opgedeeld.
   Wat wordt na afloop van de toekenning van 1k de grootte van de interne geheugenfragmentatie?

4. Idem als 3, maar nu worden er 1034 (= 1k+10) bytes aangevraagd.

5. Idem als 3, maar nu worden er 4 bytes meer dan 512 kByte aangevraagd. Welk nadeel komt hier duidelijk naar voren?

Bijlage 1 Simulatie scheduler

In deze bijlage wordt een ANSI-C-programma getoond dat het mogelijk maakt een scheduler te simuleren van een operating system.

Het voorbeeld dient gezien te worden als een simulatieprogramma en niet als een geschikte implementatie van een echte scheduler. Er wordt namelijk fysiek geschoven met process descriptors in het geheugen. Er wordt niet gewerkt met dynamische allocatie van geheugen en pointers voor het beheer van de active queue. De in het simulatieprogramma aanwezige process descriptor bevat de volgende data in een C-structure: de naam van de taak (1 karakter), de prioriteit en de age.

Scheduling vindt plaats op basis van prioriteiten en aging. Na afloop van de timeslice worden de ages van de taken in de active queue met 1 verhoogd. Daarna wordt de process descriptor van de taak die running is geweest, op basis van de waarde van zijn prioriteit in de active queue geplaatst. De taak met de hoogste age wordt daarna running.

Het programma **schedsim** is niet voorzien van invoertesten. Er wordt niet gecontroleerd op correcte invoer. Dit wordt geheel aan de gebruiker overgelaten. Het programma is modulair opgebouwd. Het hoofdprogramma ziet er als volgt uit:

```
/************************************************************ schedsim.c */
/* Simulator: prioritized scheduling                                      */
/* Compiler: ANSI-C compliant compiler                                    */
/************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include "pqueue.h"
#include "procdesc.h"

#define PROCESSNAME(i)  ((char)('A' + i))

void InputParams (int *piNumberOfTasks, int *piNumberOfSchedSteps);
void InputQueue (ProcDescr Queue[], const int iNumberOfTasks);
void ShowResult (const ProcDescr Queue[], const int iNumberOfTasks);
void Schedule (ProcDescr Queue[], const int iNumberOfTasks,
               const int iNumberOfSchedSteps);

int main (void) {
    int         iNumberOfTasks,
                iNumberOfSchedSteps;
    ProcDescr *pProcDescr;

    InputParams (&iNumberOfTasks, &iNumberOfSchedSteps);
    pProcDescr = (ProcDescr *) malloc (sizeof (ProcDescr)
                 * iNumberOfTasks);
    if (pProcDescr == NULL) {
        puts ("\n\tNot enough memory availlable\n");
        exit (1); /*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
    }
    InputQueue (pProcDescr, iNumberOfTasks);
    Schedule (pProcDescr, iNumberOfTasks, iNumberOfSchedSteps);
    free (pProcDescr);
```

```
    return 0;
}   /* main() */
/*------------------------------------------------------------ functions */

void InputParams (int *piNumberOfTasks, int *piNumberOfSchedSteps) {
    puts ("------------------------------- Scheduler simulation");
    printf ("Enter number of tasks: ");
    scanf ("%d", piNumberOfTasks);
    printf ("Enter number of scheduling steps: ");
    scanf ("%d", piNumberOfSchedSteps);
    puts ("");
}   /* InputParams() */


void InputQueue (ProcDescr Queue[], const int iNumberOfTasks) {
    const int iMaxIndex = iNumberOfTasks - 1;

    int       i;
    ProcDescr Input;

    printf ("Tasks in Active queue: ");
    for (i = 0; i < iMaxIndex; i++) {
        printf ("%c ", PROCESSNAME(i));
    }
    printf (" Running task: %c\n\n", PROCESSNAME(iMaxIndex));
    for (i = 0; i < iMaxIndex; i++) {
        Input.cName = PROCESSNAME(i);
        printf ("Enter priority task %c in Active queue: ", Input.cName);
        scanf ("%d", &Input.uiPriority);
        printf ("Enter age      task %c in Active queue: ", Input.cName);
        scanf ("%d", &Input.uiAge);
        PQ_Put (Queue, &Input);
    }
    Queue[iMaxIndex].cName = PROCESSNAME(iMaxIndex);
    printf ("Enter priority task %c         Running: ",
            Queue[iMaxIndex].cName);
    scanf ("%d", &Queue[iMaxIndex].uiPriority);
    Queue[iMaxIndex].uiAge = Queue[iMaxIndex].uiPriority;
    puts ("");
}   /* InputQueue() */


void ShowResult (const ProcDescr Queue[], const int iNumberOfTasks) {
    const int iMaxIndex = iNumberOfTasks - 1;

    int       i;

    printf ("Active Queue: ");
    for (i = 0; i < iMaxIndex; i++) {
        printf ("%c %4d, ", Queue[i].cName, Queue[i].uiAge);
    }
    printf (" Running: ");
    printf ("%c %4d\n", Queue[iMaxIndex].cName,
            Queue[iMaxIndex].uiPriority);
}   /* ShowResult() */


void Schedule (ProcDescr Queue[], const int iNumberOfTasks,
               const int iNumberOfSchedSteps) {
    int i;
```

```
    for (i = 0; i < iNumberOfSchedSteps; i++) {
        ShowResult (Queue, iNumberOfTasks);
        PQ_Schedule (Queue, iNumberOfTasks);
    }
}   /* Schedule() */

/************************************************** eof  schedsim.c */
```

De priority queue waarmee de active queue is geïmplementeerd, is te vinden in de module **pqueue.c**:

```
/************************************************************ pqueue.c */
/* Module: priority queue implemented as an array (type: ProcDescr*)    */
/************************************************************************/

#include "pqueue.h"

static int PQ_LessThen (const ProcDescr* pPD1, const ProcDescr* pPD2);
static void PQ_ShiftRight (ProcDescr Queue[], int iStart, int iEnd);

void PQ_Put (ProcDescr Queue[], const ProcDescr *pIn) {
    static int iNotSorted = 0;

    if (iNotSorted == 0) {
        Queue[iNotSorted++] = *pIn;
    } else {
        int iSorted = 0;

        while(iSorted < iNotSorted && PQ_LessThen (&Queue[iSorted], pIn)) {
            iSorted++;
        }
        PQ_ShiftRight (Queue, iSorted, iNotSorted - 1);
        Queue[iSorted] = *pIn;
        iNotSorted++;
    }
}   /* PQ_Put() */


void PQ_Schedule (ProcDescr Queue[], const int iSize) {
    int       i, iSorted;
    ProcDescr Temp;

    for (i = 0; i < iSize - 1; i++) {   /* aging */
        (Queue[i].uiAge)++;
    }
    Temp = Queue[iSize - 1];
    iSorted = 0;
    while (iSorted < iSize - 1 && PQ_LessThen (&Queue[iSorted], &Temp)) {
        iSorted++;
    }
    PQ_ShiftRight (Queue, iSorted, iSize - 1);
    Queue[iSorted] = Temp;
    Queue[iSize - 1].uiAge = Queue[iSize - 1].uiPriority;
}   /* PQ_Schedule() */


/*------------------------------------------------- private functions */

static int PQ_LessThen (const ProcDescr* pPD1, const ProcDescr* pPD2) {
    int iTest = 0;
```

```
    if (pPD1->uiAge < pPD2->uiAge) {
        iTest = 1;
    } else {
        if (pPD1->uiAge > pPD2->uiAge
            || (pPD1->uiAge == pPD2->uiAge
                && pPD2->uiAge == pPD2->uiPriority)) {
            iTest = 0;
        }
    }

    return iTest;
}   /* PQ_LessThen() */


static void PQ_ShiftRight (ProcDescr Queue[], int iStart, int iEnd) {
    while (iEnd >= iStart) {
        Queue[iEnd + 1] = Queue[iEnd];
        iEnd--;
    }
}   /* PQ_ShiftRight() */

/************************************************************ eof  pqueue.c */
```

De interface file voor de module **pqueue.c** is als volgt:

```
/************************************************************* pqueue.h */

#ifndef _PQUEUE_H
#define _PQUEUE_H

#include "procdesc.h"

void PQ_Put (ProcDescr Queue[], const ProcDescr *pIn);
void PQ_Schedule (ProcDescr Queue[], const int iSize);

#endif  /* not _PQUEUE_H */

/************************************************************ eof  pqueue.h */
```

De header file **procdesc.h** wordt door **schedsim.c** en **pqueue.c** gebruikt:

```
/************************************************************ procdesc.h */

#ifndef _PROCDESCR_H
#define _PROCDESCR_H

                                    /* struct template: process descriptor */
typedef struct {
            char         cName;
            unsigned int uiPriority;
            unsigned int uiAge;
        } ProcDescr;

#endif  /* not _PROCDESCR_H */

/************************************************************ eof  procdesc.h */
```

Voor een eerste implementatie van de priority queue is voor het sorteren van de inhoud van deze queue gebruikgemaakt van de standaard ANSI-C functie **qsort()**,

maar deze verplaatste ook elementen in de queue die eenzelfde waarde hebben. Dit maakt **qsort()** ongeschikt voor sorteren in een priority queue zoals toegepast voor een scheduler.

Voorbeeld van een simulatie-opdracht:

```
-------------------------------- Scheduler simulation
Enter number of tasks: 3
Enter number of scheduling steps: 15
Tasks in Active queue: A B  Running task: C

Enter priority task A in Active queue: 100
Enter age      task A in Active queue: 102
Enter priority task B in Active queue: 101
Enter age      task B in Active queue: 103
Enter priority task C         Running: 104

Active Queue: A  102, B  103,  Running: C  104
Active Queue: A  103, C  104,  Running: B  101
Active Queue: B  101, A  104,  Running: C  104
Active Queue: B  102, C  104,  Running: A  100
Active Queue: A  100, B  103,  Running: C  104
Active Queue: A  101, C  104,  Running: B  101
Active Queue: B  101, A  102,  Running: C  104
Active Queue: B  102, A  103,  Running: C  104
Active Queue: B  103, C  104,  Running: A  100
Active Queue: A  100, B  104,  Running: C  104
Active Queue: A  101, C  104,  Running: B  101
Active Queue: B  101, A  102,  Running: C  104
Active Queue: B  102, A  103,  Running: C  104
Active Queue: B  103, C  104,  Running: A  100
Active Queue: A  100, B  104,  Running: C  104
```

Bijlage 2 Zombie proces

We behandelen hier een voorbeeld van een **zombie** proces. Kopieer de file zombie-test.zip van Scholar. Deze file bevat de files die in deze bijlage getoond worden.
In het leerboek wordt vermeld dat een child proces in de zombie state terechtkomt als het eindigt terwijl het parent proces nog geen wait() heeft gedaan voor dat proces. Om in de gelegenheid te zijn de toestand van de processen in de verschillende situaties te bestuderen, ontwerpen we een parent proces startzombie en een child proces kind.
Er worden wachttijden als command line parameters meegegeven:

./startzombie <wachttijd1> <wachttijd2> <wachttijdchild>

Het proces startzombie voert de volgende acties uit:

- Start m.b.v. een exec-call het child proces wacht op met als command line parameter <wachttijdchild>.
- Druk de boodschap "Proces kind is gestart." af.
- Wacht <wachttijd1> seconden.
- Druk de boodschap "Parent wacht op einde kind." af
- Voer de system call wait(NULL) uit.
- Druk de boodschap "Parent ontdekt: kind is gestopt." af.
- Wacht <wachttijd2> seconden.
- Druk de boodschap "Parent stopt" af.

Het proces kind voert de volgende acties uit:

- Wacht <wachttijdchild> seconden.
- Druk de boodschap "Kind stopt" afdrukt.

Zie hieronder de betreffende files.

```
/*************************************************************
* File:           startzombie.c
* Versie:         1.0
* Datum:          31-01-2012
* Ontwerper:      R.B.A. Elsinghorst, J.G. Rouland
* Beschrijving:   Start proces wacht dat zombieproces kan worden.
*                 Er wordt van uitgegaan dat de command line
*                 parameters correct zijn ingevoerd!
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int iAantal, char *argv[]) {
     unsigned int uiGetal;
     switch (fork ()) {
          case -1:
               printf ("fork mislukt!\n");
               exit (1);
```

```
            break;
        case 0:
            execl ("./kind", " kind", argv[3], (char *) NULL);
            printf ("exec voor childproces kind is mislukt\n");
            exit (1);
            break;
        default:
            printf ("Proces kind is gestart.\n");
            uiGetal = (unsigned int) strtoul (argv[1], NULL, 10);
            sleep (uiGetal);
            printf ("Parent wacht op einde kind.\n");
            wait (NULL);
            printf ("Parent ontdekt: kind is gestopt.\n");
            uiGetal = strtoul (argv[2], NULL, 10);
            sleep (uiGetal);
            printf ("Parent stopt.\n");
    }
    return 0;
}

/************************************************************
* File:           kind.c
* Versie:         1.0
* Datum:          31-01-2012
* Ontwerper:      R.B.A. Elsinghorst, J.G. Rouland
* Beschrijving:   Wacht de via command line parameter opgegeven
*                 wachttijd (in s).
************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int iAantal, char *argv[]) {
    unsigned int uiGetal;
    uiGetal = strtoul (argv[1], NULL, 10);
    sleep (uiGetal);
    printf("Kind stopt.\n");
    return 0;
}
```

Compileer en link beide programma's:

gcc -Wall -o startzombie startzombie.c
gcc -Wall -o kind kind.c

Open twee terminalvensters in Linux en start in het ene venster proces startzombie:

./startzombie 15 8 10

Bekijk in het andere venster de activiteiten en toestanden van de processen door op de juiste momenten het volgende commando te geven:

ps -a

Een zombie proces wordt aangeduid met <defunct>.

Ontwerp het programma nozombie.c dat hetzelfde doet als zombie.c met dit verschil dat nozombie.c begint met het uitvoeren van de system call

signal (SIGCHLD, SIG_IGN);

Hiermee zal (volgens het leerboek) de kernel op de hoogte worden gesteld van het feit dat er geen zombies moeten worden gecreëerd van de children van het parent proces.
Compileer en link het programma:

gcc -Wall -o nozombie nozombie.c

Start proces nozombie als volgt op:

./nozombie 15 8 10

en kijk in een tweede terminalvenster mee met wat er gebeurt.

Bijlage 3 Exit status, return en exit()

In deze bijlage wordt het vershil tussen returen en exit() getoond en tevens wordt aangegeven hoe de exit status bij het aanroepend programma wordt afgeleverd. We ontwerpen hiertoe een parent programma statustest.c en een child programma kind.c.

Het parent programma statustest.c start (m.b.v. een exec-call) het child proces stop en geeft zijn command line argumenten aan deze child door. Vervolgens wacht de parent op het stoppen van de child en drukt op het display zowel de hexadecimale als de decimale waarde af van de exit status die hij via de functie wait() ontvangt. Ook wordt de exit status afgedrukt na gebruikmaking van de macro WEXITSTATUS.

Het parent proces wordt als volgt gestart:

./statustest <letter> <getal>

met als command line argumenten
- < letter> die de waarde 'r' of 'e' kan aannemen om aan te geven of de child zijn exit status als returnwaarde of via een exit call moet doorgeven aan de parent
- <getal> een integer die de child als exit status moet teruggeven aan de parent

```
/*********************************************************
* File:           statustest.c
* Versie:         1.0
* Datum:          29-01-2012
* Ontwerper:      R.B.A. Elsinghorst, J.G. Rouland
* Beschrijving:   Toont de exit status en het verschil tussen
*                 return en exit().
*                 Er wordt van uitgegaan dat de command line
*                 parameters correct zijn ingevoerd!
*********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int iAantal, char *argv[]) {
     int iStatus;
     switch (fork()) {
          case -1:
               printf ("fork mislukt!\n");
               exit (1);
               break;
          case 0:
               execl ("./stop", "stop", argv[1], argv[2],
                       (char *) NULL);
               printf ("exec voor childproces mislukt\n");
               exit (1);
               break;
          default:
               wait (&iStatus);
               printf ("Terugontvangen status van proces stop:\n");
```

```
            printf ("Status = 0x%x = %d\n", iStatus, iStatus);
            printf ("M.b.v. WEXITSTATUS(iStatus) wordt dit: ");
            printf ("0x%x = %d\n", WEXITSTATUS (iStatus),
                     WEXITSTATUS (iStatus));
            printf ("en met (char) WEXITSTATUS(iStatus) wordt het ");
            printf ("decimaal %d\n", (char) WEXITSTATUS (iStatus));
    }
    return 0;
}
```

Het child proces stop voert de achtereenvolgende activiteiten uit:

- drukt het getal af dat van de parent via de command line parameters is ontvangen
- roept de functie plus1() aan en geeft deze functie de stopletter ('r' of 'e') en het ontvangen getal als invoerargumenten mee
- verhoogt de returnwaarde van plus1() met 1
- geeft deze waarde via return door aan het parent proces statustest

Het prototype van de functie plus1() is als volgt:

```
    int plus1 (char cStopmode, int iInvoer)
```

Deze functie verhoogt de (tweede) invoerparameter met 1 en geeft die waarde terug afhankelijk van de stopmode:

return <getal>; als <letter> = 'r'
exit (<getal>); als <letter> = 'e'

```
/*********************************************************
* File:           stop.c
* Versie:         1.0
* Datum:          29-01-2012
* Ontwerper:      R.B.A. Elsinghorst, J.G. Rouland
* Beschrijving:   Child proces stop wordt gestart door
*                 statustest en stopt met return of exit().
*********************************************************/

#include <stdio.h>
#include <stdlib.h>

int plus1 (char cStopmode, int iInvoer);

int main (int iAantal, char * argv[]) {
    int iStatus;
    iStatus = (int) strtol (argv[2], NULL, 10);
    printf ("De door stop ontvangen waarde is: %d\n", iStatus);
    iStatus = plus1 (*argv[1], iStatus);
    printf ("Deze waarde is door functie plus1 met 1 verhoogd,\n");
    printf ("wordt door proces stop nogmaals met 1 verhoogd\n");
    printf ("en wordt als status teruggegeven aan statustest.\n");
    iStatus++;
    return iStatus;
}

int plus1 (char cStopmode, int iInvoer) {
    iInvoer++;
    if (cStopmode == 'r') {
```

```
        return iInvoer;
    } else {                    // cStopmode == 'e'
        exit (iInvoer);
    }
}
```

Kopieer de file statustest.zip van Scholar. Deze file bevat de files die in deze bijlage getoond worden.

Ga na wat er gebeurt als het proces statustest als volgt wordt opgestart:

```
./statustest  r  -3
./statustest  r  -2
./statustest  r  0
./statustest  r  258
./statustest  e  -3
./statustest  e  -2
./statustest  e  0
./statustest  e  258
```

Als gekozen wordt voor de stopmode 'r', zien we dat het ingevoerde getal met 2 wordt verhoogd, eerst door de functie plus1() en daarna nog een keer door het proces stopt zelf. Verder zien we dat de exitstatus 8 bitposities naar links is geschoven. Met de macro WEXITSTATUS (exit status) wordt de 8-bits waarde van exit status bepaald. Aangeraden wordt om altijd deze macro te gebruiken voor het verkrijgen van de juiste exit status. Mocht de exit status in een volgende release van Linux op een andere manier worden opgeslagen, dan zal de macro WEXITSTATUS hierop worden aangepast, zodat op deze wijze altijd de juiste informatie wordt verkregen. WEXITSTATUS geeft de minstwaardige 8-bits van de exit status als integer terug. Dit betekent dat het getal 258+2 = 260 als exit status de waarde 4 oplevert, omdat de exit status slechts één byte groot is (dus maximaal 255).

Ook zien we dat het getal -3+2 = -1= 0xFF de exitstatus 0x0000FF = 255 oplevert. Om de echte exit status te krijgen, moeten we de waarde van WEXITSTATUS typecasten naar char.

Kiezen we de stopmode 'e', dan wordt het ingevoerde getal met 1 verhoogd. Dit komt doordat met de exit call in de functie plus1() niet wordt teruggekeerd naar het aanroepend programma, maar dat het proces (stop), waarin de exit() wordt uitgevoerd, onmiddellijk wordt beëindigd.

Index