

# **De Unix kernel voor leken**

(Een verhaal in 3 afleveringen)

Marc Seutter

Subfaculteit Informatica  
Katholieke Universiteit Nijmegen

1 juni 2002

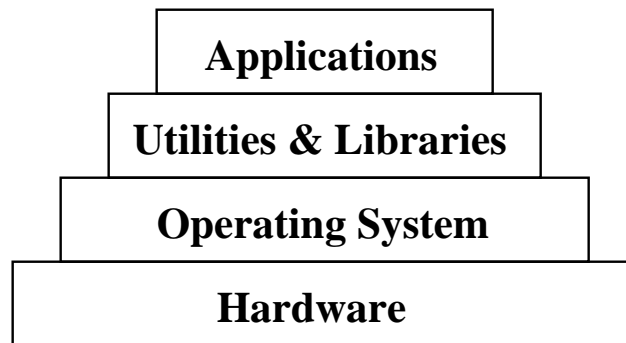
5 oktober 2002

14 december 2002

# Wat is een Operating System?

Een definitie:

- Een **Operating System (Bedrijfsysteem)** is een programma dat de executie van applicatie programma's controleert en dat optreedt als een interface tussen computer gebruikers en de computer hardware (Stallings).



Nog een definitie:

- Een **Operating System** is een programma (of verzameling programma's) dat de gebruiker(s) een bruikbare abstractie van de hardware geeft, zó dat de computer resources zo eerlijk en efficiënt mogelijk gebruikt worden.

Hoort de inhoud van `/lib` nou wel of niet bij het OS?

# Wat eis je van een Unix kernel?

- Eerlijke verdeling van computer resources.
- Veilige afscherming tussen verschillende programma's en gebruikers.
- Abstractie van de hardware.
- Robuustheid.
- Betrouwbaarheid.
- Goede implementatie van virtueel geheugen.
- Mechanismen voor netwerkverbindingen.
- .....

# Een klein beetje historie

1969 Dennis Ritchie en Ken Thompson programmeren Unix versie 1 op een PDP7.

1976 Unix versie 6 op een PDP11/34, /45  
Toevoeging van hardware MMU support.

1978 De eerste “commerciële” versie Unix, versie 7, komt uit. Toevoeging van environment variabelen.

1979 3BSD. De eerste Berkeley VAX UNIX  
Toevoeging van demand paged virtueel geheugen.

1983 4.2BSD.  
Toevoeging van de Internet networking protocollen.

1984 SunOS  
Toevoeging van Network File System (NFS, NIS).

1988 De IEEE POSIX.1 standaard.

1991 Linus Torvalds schrijft de eerste versie van Linux.

1993 4.4BSD.

# User en Kernel mode

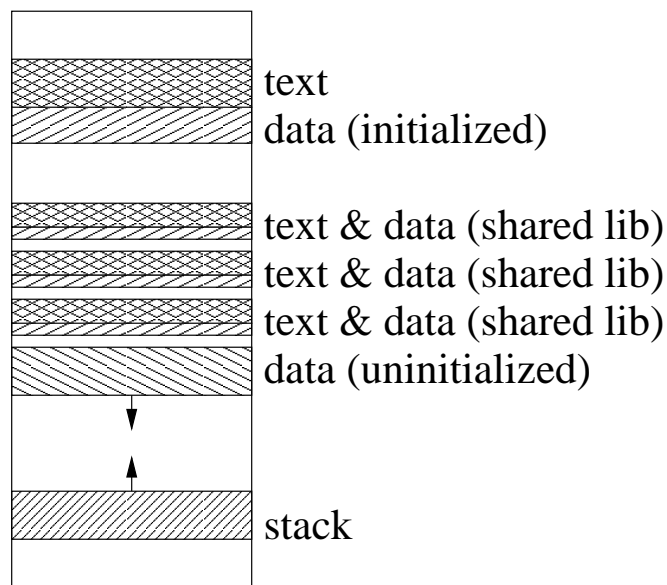
- Unix verwacht dat de processor hardware tenminste twee (fysieke) modi van executie heeft nl **User** en **Kernel** mode. Typisch wordt de mode met een of twee bitten in het status register van de processor aangegeven.
- In kernel mode mag de processor “alles”.  
In user mode alles wat “niet gevaarlijk” is.
- Alleen in kernel mode mag de processor direct aan hardware (I/O) registers komen. Dit geldt i.h.v. voor de memory management registers, waarmee o.a. de afscherming tussen verschillende programma's (inclusief die tussen user programma's en de kernel) geregeld wordt.
- Als een programma in user mode draait, mag het alleen op een gecontroleerde manier in kernel mode overgaan. Bij zo'n transitie naar kernel mode wordt voldoende informatie bewaard op een eigen *kernel stack* om eventueel later naar het user programma terug te kunnen keren.

# Interrupts, traps en system calls

- Er zijn drie manieren om van user mode in kernel mode te gaan:
  - Exceptions (Page Fault, Div by 0, Privilege,...)
  - Interrupts (I/O, clock)
  - Traps (Software interrupts)
- System calls zijn in feite één specifieke trap b.v. voor de Intel INT 0x80, die geparametriseerd wordt met het nummer van de syscall en eventuele extra parameters. Programma's in user mode gebruiken system calls om specifieke diensten aan de kernel te vragen.
- De lijst van beschikbare system calls is vanaf het begin gestandaardiseerd. Bijvoorbeeld: Linux kent ±180 system calls. Onder de eerste 80 zijn er 50 die ook al in de V7 kernel aanwezig waren, zoals *exit*, *fork*, *read*, *write*, etc. De overige 80 zijn vooral latere toevoegingen.

# Processen

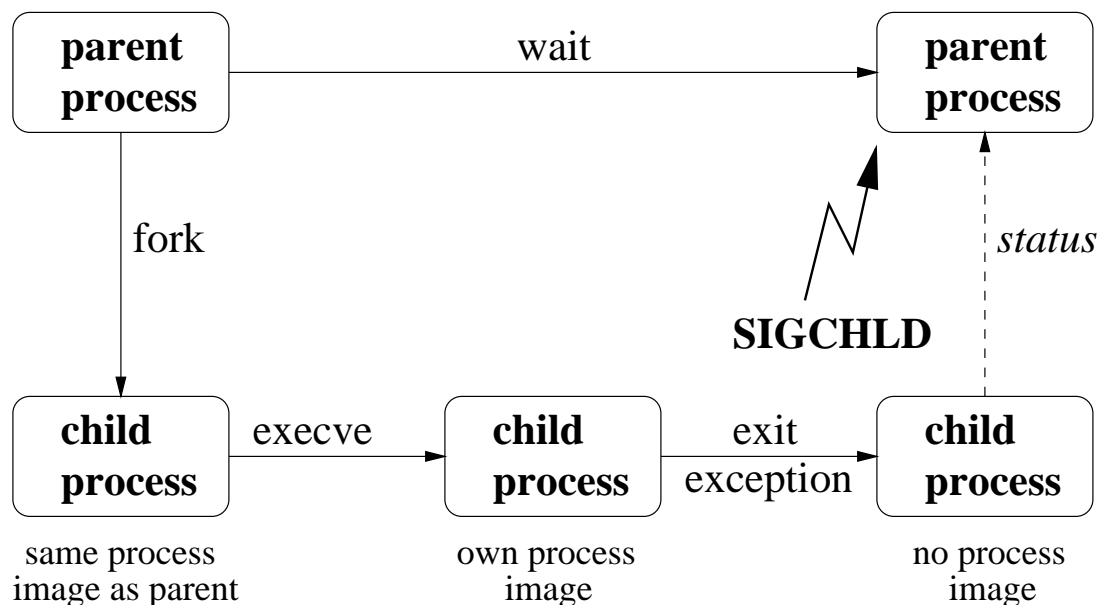
- Een **proces** is een programma in executie.
- Processen hebben een eigenaar en groep.
- De meeste processen executeren in user mode, al zullen ze zo nu en dan de kernel met een system call ingaan. Er zijn echter een aantal processen die door de kernel zelf gestart worden en permanent in kernel mode draaien (b.v. *kupdate*, *kflushd*, etc.).
- De kernel geeft een proces in user mode een virtueel geheugen image dat er min of meer altijd hetzelfde uitziet:



# Process Mgmt system calls

- Processen worden geboren met *fork*. Hierbij krijgen ze hetzelfde geheugen image als het ouderproces. *Beiden* returnen vanuit *fork*.
- Processen geven zichzelf een nieuw proces image met *execve*.
- Na een tijdje gaan ze dood, hetzij door een exception, hetzij met de *exit* system call.
- Met *wait* wacht het ouderproces op de dood van het kind. Hij krijgt dan de finale status van het kind terwijl het kind door de kernel begraven wordt.

In een plaatje:



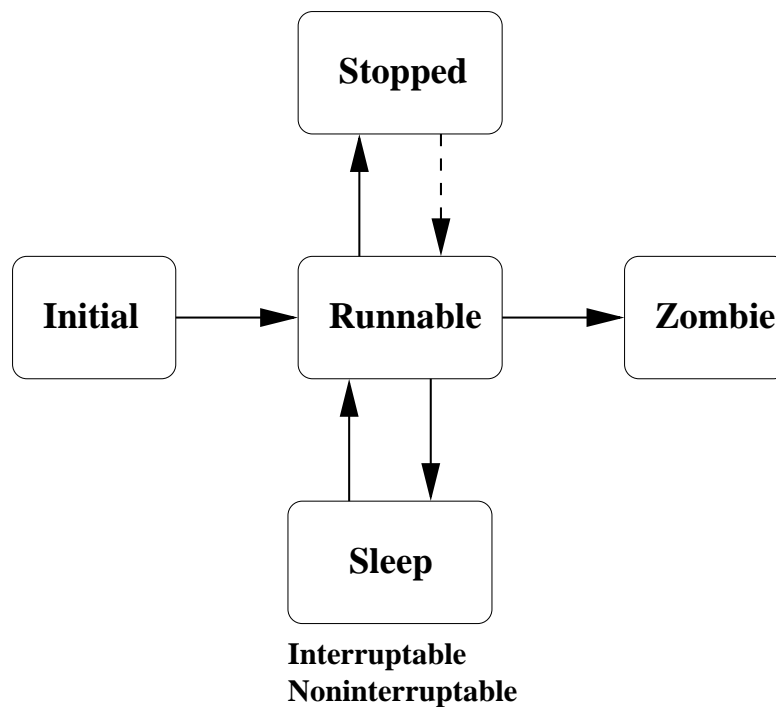
# Signals

- Signals zijn een software analogon van interrupts.
- De kernel kan altijd een signal naar een proces sturen als gevolg van een bepaalde gebeurtenis b.v. het doodgaan van een kindproces (SIGCHLD).
- User processen sturen met de *kill* system call een signal naar een ander proces (De kernel checkt of het wel mag...).
- Per signal kent de kernel een default actie (Negeren, termineren, dan wel termineren met een core dump).
- Een proces kan een user handler voor een specifiek signal aangeven die opgeroepen moet worden wanneer dit signal voor dit proces optreedt. Bij oproep bepaalt de handler of hij het proces afbreekt dan wel doorstart.
- Sommige signalen kunnen niet op deze manier opgevangen worden (SIGKILL, SIGSTOP, etc.).
- De kernel checkt op signals als een proces de kernel verlaat.

# Proces administratie

Per proces heeft de kernel het een en ander bij te houden:

- De process state:



- Het process ID.
- Het (effective, real en saved) user en group ID.
- De registers en memory management informatie.
- De basis prioriteit van het proces en het aantal over zijnde tikken van de klok in de huidige epoche.
- en nog veel meer...

# De Linux scheduler

- De dynamische prioriteit van een proces is de som van de basis prioriteit en het aantal over zijnde tikken van de klok in de huidige epoche.
- Elke 10 ms (1 tik) wordt een timer interrupt gegenereerd. De clock interrupt handler werkt hierop de globale kernel en de administratie van het lopende proces bij. De dynamische prioriteit van een proces zakt dus met elke tik.
- Als de dynamische prioriteit van een proces zover is gezakt dat een ander runnable proces hogere prioriteit heeft vindt er een *geforceerde* proces switch plaats.
- Als een proces wakker wordt (b.v. omdat I/O klaar is) en zijn prioriteit is dan hoger dan die van het lopende proces zal de kernel van proces switchen bij het verlaten van de kernel.
- Bij een *fork* krijgen ouder en kind de helft van de over zijnde tikken van de ouder.

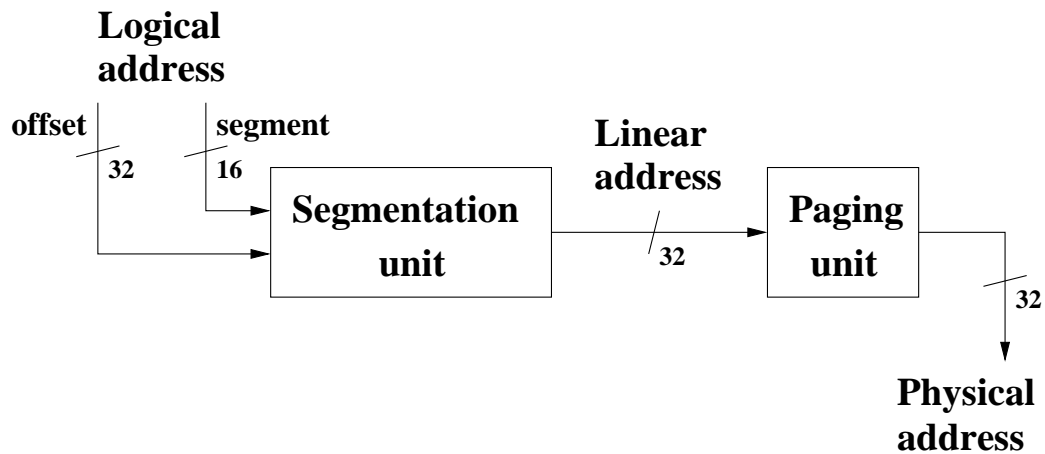
## Scheduling 2

- Als een proces zijn totale tijdsquantum verbruikt heeft komt hij in deze epoche niet meer aan de beurt.
- De crux van de scheduler:  
Als alle runnable processen hun tijdsquantum verbruikt hebben, begint een nieuwe epoche. *Alle, niet alleen de runnable* processen krijgen een nieuw tijdsquantum gelijk aan het halve oude tijdsquantum plus de basis prioriteit.
- Linux prefereert dus interactieve processen boven rekenprocessen.
- Linux gebruikt 20 tikken als basis prioriteit. Met de *nice* system call kan de basis prioriteit van een proces aangepast worden.

De Linux scheduler heeft als voordeel dat hij simpel is. Het grote nadeel van de Linux scheduler is zijn slechte schaalbaarheid.

# Virtueel geheugen

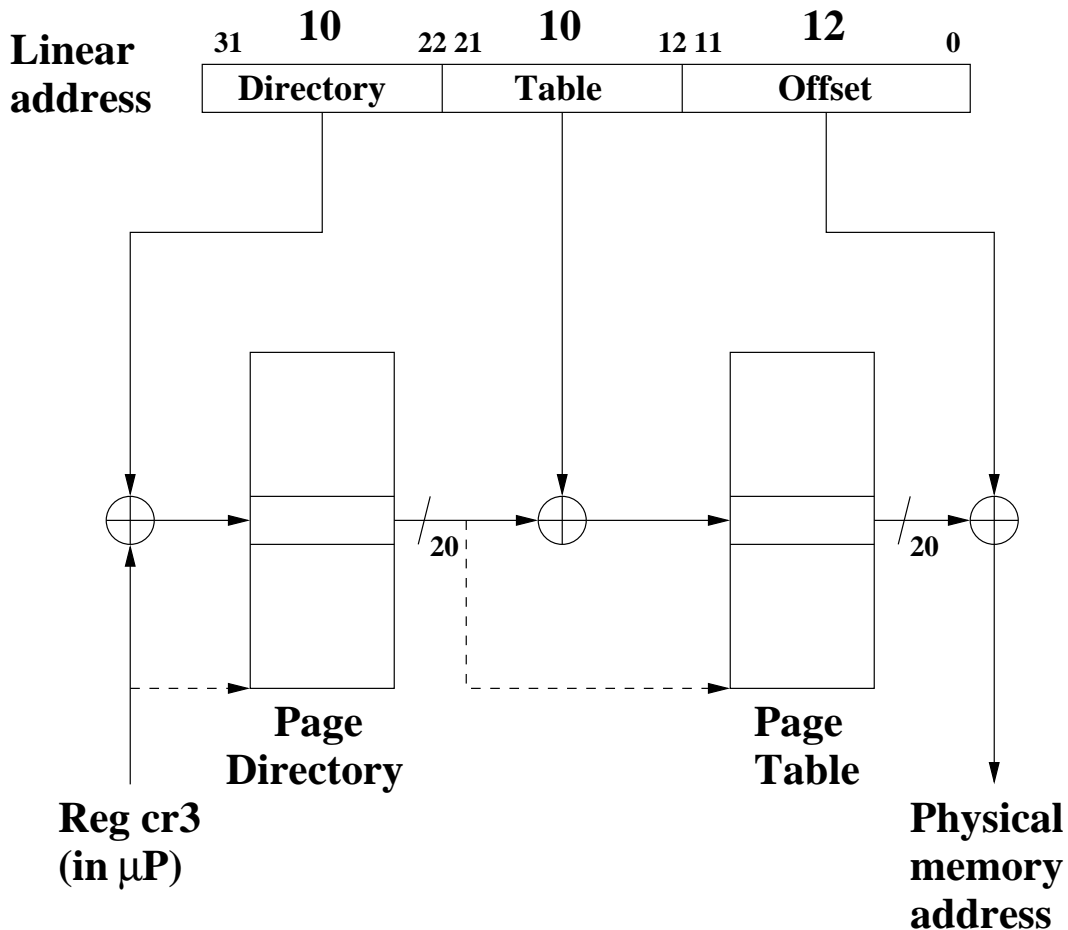
Virtueel geheugen werd op de Intel vanaf de 386 mogelijk dmv. de *protected mode*. In protected mode ondergaat een adres 2 vertaalslagen voordat het uit de processor komt:



Linux programmeert de segmentation unit zodanig dat deze meestal de inkomende offset direct als lineair adres kopiëert. Er zijn echter bepaalde applicaties b.v. WINE, die wel (beperkt en gecontroleerd) van de segmentation unit gebruik maken.

# Paging hardware

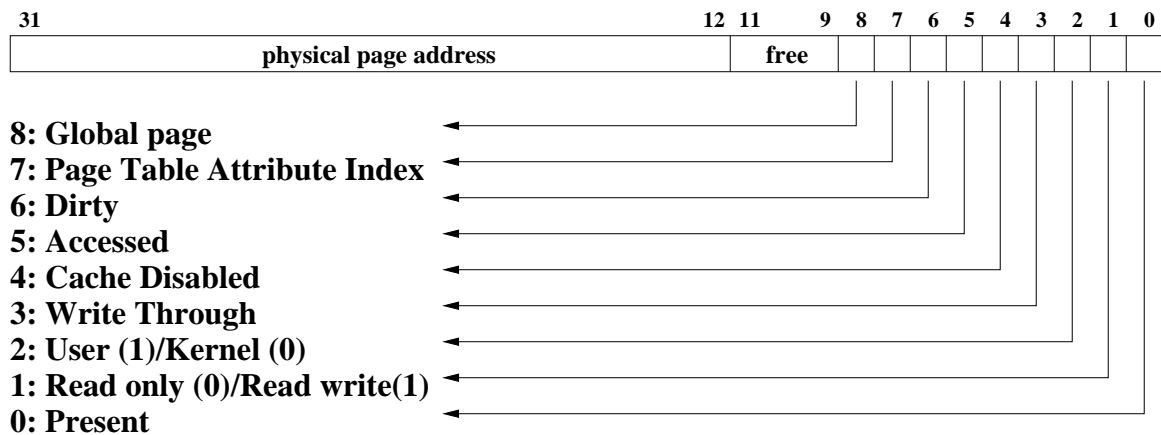
De paging unit doet het echte vertaalwerk:



- Een page bestaat een aaneengesloten stuk fysiek geheugen van 4KB. Een page table of directory zijn precies een page groot.
- De kernel zorgt ervoor dat CR3 altijd geladen is met het fysieke adres van de page directory horend bij het huidige proces.

# Page Table Entries (PTEs)

Een entry in een page table of een page directory ziet er als volgt uit:



- Een entry in de page directory of page table bestaat dus uit de bovenste 20 bits van een fysiek adres plus bijbehorende status informatie.
- Een page table beschrijft dus precies 1024 pages (4MB). Een page directory beschrijft de complete geheugen map van één process.
- Paging kan gebruikt worden om pagina's te delen tussen verschillende processen (Shared libraries).
- Merk op dat een proces pagina's kan bevatten die alleen in kernel mode zichtbaar zijn. I.h.b. de code van de Linux kernel wordt bij een switch naar kernel mode zichtbaar bovenin de image van het process (meestal vanaf 0xc0000000).

# Page Faults

- Als de processor een adres accesst, waar hij volgens de bijbehorende pte geen recht toe heeft, treedt een *page fault* exception op.
- Bij een page fault wordt voldoende informatie op de kernel stack van het proces bewaard om bij een return uit kernel mode de instructie die de page fault veroorzaakte te herstarten.
- De page fault handler beslist op basis van het fault adres wat te doen:
  - **SEGV**  
Het adres was echt fout: de handler stuurt een SEGV signaal om het proces om zeep te helpen.

## **Page in**

De geaccessde pagina stond niet in het geheugen maar kan vanuit een file of de swap opgehaald worden. Na het binnenhalen wordt het proces doorgestart (demand paging).

## - **Copy on Write**

De geaccessde pagina was read only gemarkeerd omdat dit proces hem deelde met andere processen. Van de pagina wordt een privé kopie gemaakt, de pte wordt hiermee bijgewerkt en het proces doorgestart.

# Regions

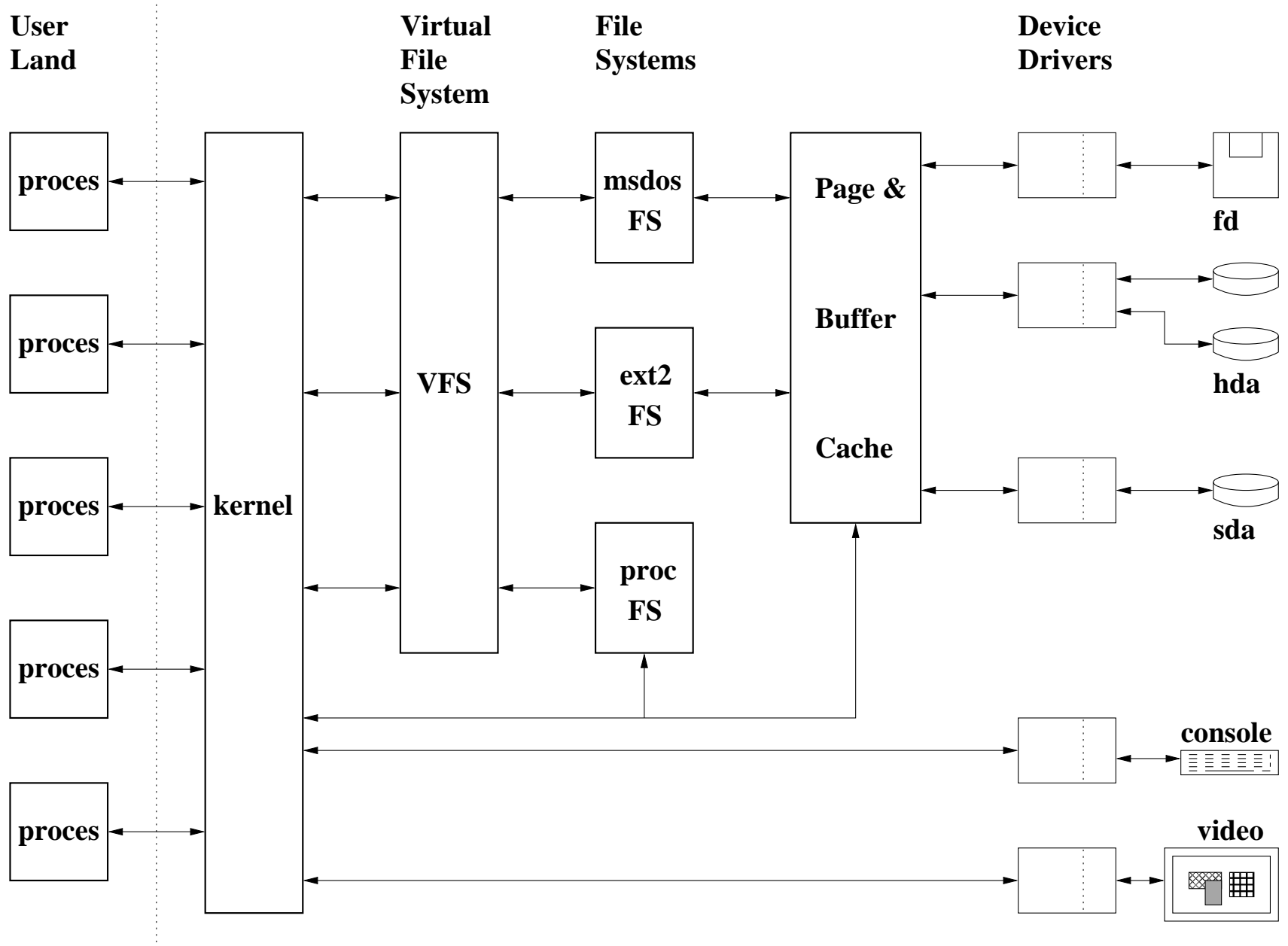
- Per proces houdt de kernel een memory map bij. Zo'n map bestaat uit een aantal *regions* die elk 1 of meer pagina's beslaan. Deze memory map kun je in `/proc/<pid>/maps` bekijken.
- Bij een region wordt een gezamenlijke protectie en locatie voor page in / page out bijgehouden.
- Alleen het region dat hoort bij de user mode stack wordt dynamisch door de kernel aangepast. Voor alle andere geldt beleefd vragen...
- Met de `mmap` system call kan een proces een region aan zijn memory map toevoegen. Afhankelijk van de parameters wordt zo'n region hetzij met een file, hetzij met een stuk uit de swap space (anonymous mapping) geassocieerd.
- Met de `execve` system call wordt de complete memory map van een proces vervangen. De kernel creëert een compleet nieuwe set regions die horen bij de nieuwe executable, de eventuele bijbehorende dynamic loader en shared binaries. Het feitelijk binnenhalen van de nieuwe code wordt overigens aan het paging mechanisme overgelaten.
- Met de `brk` system call kan een proces zijn beschikbare data ruimte verkleinen of vergroten.

# Geheugengebrek

- De kernel gebruikt een eigen allocator om het fysieke geheugen te beheren. Deze allocator zorgt er voor dat er altijd een kleine reserve aan vrije pagina's voor kritische kernel delen overblijft.
- Als het aantal beschikbare vrije pagina's beneden een zekere drempelwaarde dreigt te komen zal het systeem proberen pagina's vrij te maken. Deze drempelwaarde wordt ingesteld bij het booten.
- De kernel zoekt nu onder de in gebruik zijnde pagina's naar pagina's waarvan het Accessed bit in de pte op 0 staat. In dit geval is de pagina sinds de laatste zoekactie niet meer aangeraakt en is de kans groot dat hij dat binnenkort ook niet wordt (Localiteitsbeginsel).
- Pagina's waarvan het Dirty bit op 0 staat zijn gemakkelijk: de kernel mag de pagina gewoon vergeten. Deze zijn dus meteen vrij te maken.
- Pagina's die wel beschreven zijn, moeten naar de swap space weggeschreven worden. Zodra de bijbehorende I/O gedaan is, kan de pagina hergebruikt worden.
- Ongedeelde pagina's zijn te prefereren boven gedeelde pagina's voor een page out.

# I/O devices

- I/O devices worden met speciale IO-instructies aangesproken of dmv. *memory mapped I/O*, d.w.z. dat een aantal fysieke geheugenadressen voor het I/O device gereserveerd zijn.
- Meestal moet voor het starten van I/O een aantal registers in de I/O controller geschreven worden. Dit is nogal specifiek voor de soort hardware.
- Sommige snelle I/O devices kunnen data rechtstreeks van en naar het geheugen transporteren dmv. DMA (*Direct Memory Access*).
- Bij fouten tijdens transport of bij voltooiing van transport kan het I/O device een interrupt naar de processor sturen.
- I/O devices kunnen met een opdracht vaak meer dan 1 byte verplaatsen. Een typische I/O opdracht zou kunnen zijn: lees 1K bytes van block 104890 van SCSI disk 1 en schrijf die naar locatie 0x690ab0 in het (fysieke) geheugen. Voor een COM poort zou de opdracht kunnen zijn: lees 1 byte.
- Processen en de kernel benaderen een I/O device vrijwel nooit rechtstreeks maar altijd via een device driver.



# Device drivers

- Device drivers komen in 2 soorten nl. de *character* en de *block* device drivers.
- Een block device driver transporteert alleen hele blokken van bv. 1K, waarbij met de (page en) buffer cache wordt gecoördineerd om I/O operaties te optimaliseren. Hierbij vindt I/O alleen plaats van en naar I/O buffers in de kernel. Block device interfaces zijn typisch bedoeld voor disken.

Merk hierbij op dat de werking van de (page en) buffer cache onafhankelijk is van de organisatie van de data op disk (daar is de filesystem code voor).

- Een character device driver transporteert data zonder intern te bufferen, dus vaak direct naar proces geheugen. Bovendien kan een character device interface arbitraire (of hooguit hardware afhankelijke) hoeveelheden data transporteren. Character device interfaces worden gebruikt voor terminals, printers, systeem pseudo devices (`/dev/null`), e.d.
- Bij een aantal Unix versies (bij Linux pas vanaf 2.4) is het mogelijk dat een device via zowel de block als de character interface benaderd wordt (De character versie noemen we dan vaak het *raw* device).

# Major en minor device nrs

- Device drivers worden geïdentificeerd met hun type, *major* en *minor* device nummer. Wat voorbeeldjes uit `/dev`:

```
crw-r----- 1 2 /dev/kmem
crw-rw-rw- 1 3 /dev/null
crw-rw---- 4 64 /dev/ttyS0
brw-rw---- 3 4 /dev/hda4
brw-rw---- 8 1 /dev/sda1
brw-rw---- 8 3 /dev/sda3
crw-rw---- 14 0 /dev/mixer
crw-rw---- 14 2 /dev/midi00
```

Het type en major device nummer worden door de kernel gebruikt om de goede device driver te vinden. Het minor device nummer wordt gebruikt om het goede device te identificeren dat door deze device driver wordt aangestuurd.

- Elke device driver stelt een bepaalde vaste set operaties ter beschikking aan hoger gelegen lagen in de kernel.

# Inodes

- Elke filesystem biedt aan het VFS informatie over een file in de vorm van een inode (*index node*).
- Een inode bevat onder meer:
  - type en access mode  
4 bit soort, 12 bit protectie
  - user id
  - file lengte
  - tijd van laatste access, laatste inode verandering en laatste inhoud verandering.
  - group id
  - reference count
  - block count
  - file flags
- Van zo'n inode bestaat op disk een filesystem afhankelijke afspiegeling. Per filesystem bevat de inode ook specifieke informatie b.v. voor het ext2fs:
  - disk blok nummers van de eerste 12 blokken van een file
  - een indirect disk blok nummer
  - een dubbel indirect disk blok nummer
  - een triple indirect disk blok nummer

# Inodes en directories

- Directories worden door de meeste filesystemen gezien als een speciaal soort file waarvan de inhoud een lijst paren (filename, inode nr) is.
- MERK OP dat er dus geen filenaam in de inode staat!!
- Het is dus heel goed mogelijk dat één inode meerdere namen heeft. Het reference count veld in de inode houdt dit aantal bij.
- Per file systeem worden een aantal functies aan het VFS ter beschikking gesteld om directories en inodes te manipuleren.
- De file systeem afhankelijke informatie in de inode wordt vaak misbruikt om allerlei andere informatie in te bewaren, b.v. voor het ext2fs:
  - Het major en minor device nr van een device file worden in de ruimte van het eerste blok nummer bewaard.
  - Een symlink waarbij de link minder dan 60 bytes nodig heeft wordt eveneens in de bloknummer ruimte bewaard. Bij langere symlinks neem je een indirect disk blok...

# File handling

- Per proces houdt de kernel een set file objecten bij, geïdentificeerd met hun *file descriptors*. Bij een file object houdt de kernel o.m. de geassocieerde directory entry, inode en de current file pointer bij.
- Bij elk proces houdt de kernel de current working directory bij. Met de **chdir** system call is die te veranderen.
- Bij een **fork** erft het kind de open file descriptors van zijn ouder. Een kind van de shell zal dus typisch 0 (stdin), 1 (stdout) en 2 (stderr) van de shell erven.
- Met de **open** system call wordt voor de laagst vrije file descriptor een file object geïntialiseerd. Met **close** wordt een file afgesloten.
- Met de **lseek** system call kunnen we de file pointer verzetten. Met **read** wordt uit een file gelezen; met **write** wordt geschreven.
- Met de **dup** en **dup2** system calls kunnen we het file object horend bij een descriptor dupliceren naar een andere.
- Met de **pipe** system call worden één read en één write file object gecreëerd plus een gemeenschappelijk page frame waarin gebufferd wordt.